Mitigating Denial-of-Service Flooding Attacks with Source Authentication

by

Xin Liu

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Xiaowei Yang, Supervisor

_____
Bruce M. Maggs

_____
Jeffrey S. Chase

_____
Romit Roy Choudhury

_____
Michael K. Reiter

Dissertation submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in the Department of Computer Science
in the Graduate School of Duke University
2012

Abstract

Mitigating Denial-of-Service Flooding Attacks with Source Authentication

by

Xin Liu

Department of Computer Science
Duke University

Date: _____

Approved:

_____
Xiaowei Yang, Supervisor

_____
Bruce M. Maggs

_____
Jeffrey S. Chase

_____
Romit Roy Choudhury

_____
Michael K. Reiter

An abstract of a dissertation submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science in the Graduate School of Duke University
2012

Abstract

Denial-of-Service (DoS) flooding attacks have become a serious threat to the reliability of the Internet. For instance, a report published by Arbor Networks reveals that the largest DoS flooding attack observed in 2010 reaches 100Gbps in attack traffic volume. The defense against DoS flooding attacks is significantly complicated by the fact that the Internet lacks accountability at the network layer: it is very difficult, if not impossible, for the receiver of an IP packet to associate the packet with its real sender, as the sender is free to craft any part of the packet.

This dissertation proposes to mitigate DoS flooding attacks with a two-step process: first to establish accountability at the network layer, and second to utilize the accountability to efficiently and scalably mitigate the attacks. It proposes Passport, a source authentication system that enables any router forwarding a packet to cryptographically verify the source Autonomous System (AS) of the packet. Passport uses symmetric key cryptography to enable high-speed verification and piggy-backs its key exchange into the inter-domain routing system for efficiency and independence from non-routing infrastructures.

On top of Passport, this dissertation proposes NetFence, a DoS flooding attack mitigation system that provides two levels of protection against the attacks: if a victim can receive and identify the attack traffic, it can throttle the attack traffic close to the attack sources; otherwise, the attack traffic cannot be eliminated, but it would not be able to consume more than the attack sources' fair shares of the capacity

of any bottleneck link. NetFence achieves its goals by putting unforgeable congestion policing feedback into each packet. The feedback allows bottleneck routers to convey congestion information back to the access routers that police the traffic accordingly. A destination host can throttle unwanted traffic by not returning the feedback to the source host.

We have implemented prototypes of Passport and NetFence in both ns-2 simulator and Linux. We have also implement a prototype of Passport on a NetFPGA board. Our evaluation of the prototypes as well as our security and theoretical analysis demonstrate that both Passport and NetFence are practical for high-speed router implementation and could mitigate a wider range of attacks in a more scalable way compared to previous work.

Contents

List of Tables

List of Figures

List of Abbreviations and Symbols

Abbreviations

AS	Autonomous System
AIMD	Additive Increase and Multiplicative Decrease
DoS	Denial of Service
MAC	Message Authentication Code or Media Access Control
RED	Random Early Detection

Acknowledgements

It has been a long journey for me to pursuit the PhD, and I am fortunate to have Xiaowei Yang as my adviser. I learned from her the basic skills for doing scientific research and, more importantly, how to tackle a complex problem from different view points and with divide-and-conquer. I am also grateful to the invaluable insight and guidance she provided to me when my research got stuck.

I would like to thank my PhD committee: Bruce Maggs, Jeff Chase, Michael Reiter and Romit Roy Choudhury for giving me insightful feedback about my work, especially through their questions during the preliminary and defense exams. The feedback has helped me to improve the quality of this dissertation. I would also like to thank Stanislaw Jarecki at University of California, Irvine for getting me started on cryptography and helping me in picking the correct MAC functions used in this dissertation, and Yong Xia at NEC Labs China for helping me to mathematically prove several properties of the system I designed.

Lab mates and friends at school are also an essential part of my PhD study. I enjoyed discussing various research ideas with Ang Li, Michael Sirivianos, Yanbin Lu, Mishari I. Almishari, Xin Wu, Qiang Cao, and many other friends I have at University of California, Irvine and Duke University. I am also grateful to the emotional support they gave me at times when I felt disheartened during my PhD study.

Last but not least, I am fortunate to have Licong He as the girl friend and then wife. To put it simply, I would not survive the graduate school at all without her

support, both emotionally and physically.

1

Introduction

In a Denial-of-Service (DoS) flooding attack, attackers aim to disrupt and prevent legitimate communications by congesting links in a network. The first series of high-profile and well-publicized DoS flooding attacks occurred in 2000, in which attackers disrupted the services of various popular websites including Yahoo, eBay and CNN [31]. Today, DoS flooding attacks have become a serious threat to the reliability of the Internet. A recent report published by Arbor Networks reveals that the largest DoS flooding attack observed in 2010 reaches 100Gbps in attack traffic volume [5]; such an attack can congest almost any link in the current Internet and cause denial of service for legitimate users whose packets traverse the congested link.

Much research has been done to mitigate DoS flooding attacks. Most of the existing work falls into one of three categories: blacklist-based solutions that attempt to block the identified attack traffic before it congests a link [30, 91, 59, 6, 53, 3, 84], whitelist-based solutions that require a sender to first obtain the permission to send [26, 9, 99, 103, 71, 17], and overlay-based solutions that use overlay networks to deliver user traffic in order to reduce the damage of attackers congesting a few links [2, 40, 89, 23]. However, these existing solutions all fall short against powerful

attackers that have the accurate network topology and use a large number of attack sources and malicious routers to launch sophisticated DoS flooding attacks. In this dissertation, we aim to design an effective, scalable and incrementally deployable DoS flooding attack mitigation architecture that can defend against these powerful attackers.

1.1 DoS Mitigation: A Two-Step Approach

We observe that one of the major reasons why DoS flooding attacks have become a difficult problem is that the Internet lacks accountability at the network layer: the source identifier in an IP packet, the source IP address, can be easily spoofed by a source host. The attackers' ability to spoof source addresses significantly complicates the design of an effective and scalable DoS flooding mitigation system, because the lack of reliable source identifiers makes it difficult to locate the attack sources, finger-print the attack traffic, and separate attack traffic from legitimate traffic. If source address spoofing were prevented in the Internet, many of the existing DoS flooding mitigation solutions could be made much simpler and more effective. For instance, with authentic source addresses, blacklist-based approaches can easily describe and filter attack traffic using the attack sources' addresses, and the filters may be deployed close to the attack sources by contacting the source ASes inferred from the source addresses. Even more importantly, authentic source addresses can make a DoS flooding mitigation solution fail-safe: that is, even if a proposal fails to eliminate the attack traffic, it can at least use source address based fair queuing to separate attack traffic from legitimate traffic to a certain degree, so that attackers cannot use a small number of attack sources to disrupt the communications of many legitimate users. In addition, source address authentication alone can eliminate an important type of DoS flooding attack: reflector attack (Section 2.2), because attackers can no longer trigger return traffic sent by legitimate hosts towards spoofed addresses.

The importance of authentic source addresses inspires us to design a complete DoS flooding mitigation system in two steps: eliminate source address spoofing first, and defend against DoS flooding attacks second. As the first step, we have designed a source authentication sub-system called *Passport*. Passport aims to authenticate source addresses at the AS level: it treats an AS as a trust and fate-sharing unit, and authenticates the source of a packet to the granularity of the origin AS. Each AS is free to choose its own intra-domain source authentication scheme. Passport uses efficient symmetric key cryptography and only requires validating source addresses at administrative boundaries (*e.g.*, the boundary between two ASes). It leverages the routing system to efficiently establish and distribute symmetric keys: it piggybacks a Diffie-Hellman key exchange into inter-domain routing announcements. Passport provides most of the security benefits of a digital signature based approach, *i.e.*, it allows each router on a packet's forwarding path to verify the authenticity of the packet's source address; at the same time, it does not require a router to perform computationally expensive public key operations to validate a packet's source address.

With Passport to enforce authentic source addresses, we are able to design a DoS mitigation sub-system called *NetFence*. Different from many existing proposals, NetFence provides two levels of protection against DoS flooding attacks: when a victim can identify the attack traffic, it can also identify the attack sources via the authentic source addresses in the attack packets, and then block the attack traffic near the attack sources; on the other hand, if legitimate hosts and routers cannot distinguish attack traffic from legitimate traffic and therefore cannot take actions to block the attack traffic, NetFence can still guarantee that regardless of how badly congested a bottleneck link becomes, each sender with sufficient demand can get its fair share of the bottleneck bandwidth. The second level of protection limits the damage of attack traffic, *i.e.*, attackers cannot completely starve legitimate users.

The key component in the NetFence design is the unforgeable congestion policing feedback stamped into packets by the network. Access routers use the feedback to police senders' outgoing traffic to limit the congestion in the network and enforces fairness among senders. When a victim identifies an attack source, it can use the congestion policing feedback as capability tokens [4] to block the attack traffic at the attack source's access router. Thanks to the authentic source addresses enforced by Passport, NetFence can use source based fair queuing or rate limiting to easily limit the damage of compromised access routers that do not properly police their senders' outgoing traffic. NetFence is scalable because it only requires routers to perform lightweight symmetric key operations, and it does not require non-access routers to keep per-host or per-flow state.

Combined together, Passport and NetFence form a complete DoS flooding attack mitigation architecture that is effective against a large number of attackers (including both hosts and routers) and various attack strategies, scalable at the Internet scale, and incrementally deployable in the current Internet.

1.2 Contributions

This dissertation contributes a novel DoS flooding attack defense architecture that is effective, scalable, incrementally deployable, and self-contained. It develops a source authentication sub-system Passport, and a DoS flooding mitigation sub-system NetFence that can both eliminate identified attack traffic and guarantee each sender with sufficient bandwidth demand its fair share of the bottleneck bandwidth. It uses analysis, mathematical proof, simulations and testbed experiments to evaluate the performance of Passport and NetFence and demonstrate that they have achieved the design goals and perform better than existing solutions.

1.3 Organization

The rest of this dissertation is organized as follows. In Chapter 2, we present a brief introduction to DoS flooding attacks and summarize the related work. In Chapter 3, we present our design space, including the thread model, the enabling assumptions, and the design goals. In Chapter 4, we present the overall architecture of our design, including an overview of our two sub-systems Passport and NetFence as well as how they interact with each other. Then we present the detailed design of Passport in Chapter 5 and NetFence in Chapter 6. In Chapter 7, we analyze the security of our design. Then we introduce our prototype implementations of Passport and NetFence in Chapter 8 and benchmark the implementations in Chapter 9. In Chapter 10, we present our testbed experiments and ns-2 simulations to evaluate the effectiveness of our design. In Chapter 11, we discuss various issues including some of our design choices, how our design interacts with existing protocols in the Internet, and how Passport may help to mitigate application-level DoS attacks. Finally we conclude in Chapter 12.

This dissertation contains three appendixes. Appendix A presents the convergence analysis of NetFence's algorithm that aims to ensure fair allocation of bottleneck bandwidth. Appendix B presents alternative designs of NetFence that have better performance in multi-bottleneck scenarios but are more complicated. Appendix C shows the pseudo-code for most operations in Passport and NetFence.

2

Background and Related Work

2.1 DoS Flooding Attacks

As the name suggests, a Denial-of-Service (DoS) attack aims to prevent legitimate users from accessing a particular service. In general, there are two types of DoS attacks in the Internet: application-level attacks, and network-level attacks.

An application-level DoS attack aims to exhaust the resources at a particular service so that legitimate users cannot enjoy the service. For instance, a service may need to perform complicated and expensive database operations in order to answer a query from a user; let us suppose it can at most answer x user queries per second. Then in an application-level DoS attack against this service, attackers may submit $10x$ queries per second. If the service chooses to temporarily hold the unprocessed queries, it may eventually run out of memory and crash, preventing legitimate users from further accessing the service; or if the service discards queries it cannot process in time, the query drop rate for both attackers and legitimate users will be at least 90%, meaning the majority of the users still cannot enjoy the service.

A network-level DoS attack, also commonly referred to as a *DoS flooding attack*,

Table 2.1: **Notable and Publicly Known DoS Flooding Attacks**

Year	Description
2000	A sequence of attacks to Yahoo, Buy.com, eBay, CNN, Amazon, ZDNet, E*Trade, Excite [31]
2001	Attack to Microsoft [46]
	A 13-year-old hacker attacked grc.com [33]
2002	Reflector attack to grc.com [32]
	Attack to DNS root servers [62]
2004	Attack to disrupt a competitor [49]
2005	Extortion via DoS to Authorize.net and others [69]
2006	Spammers attacked to BlueSecurity for revenge [76]
	DNS reflector attacks [81]
2007	Attack to Estonia [65]
2008	Attack to Georgia [66]
2010	Attack to WikiLeaks [44]
	Attacks to MasterCard, Visa [95] and PayPal [72] by Wikileaks supporters
	Attack to Myanmar [79]
2011	Attack to Wordpress.com [93]

aims to simply congest particular links in the network using IP packets. That is, attackers flood an excessive amount of IP packets towards certain destinations such that a router in front of a link has to discard packets that should traverse the link. Without any DoS flooding mitigation schemes, the router will discard both attacker packets and legitimate user packets at the same rate; for instance, if the arrival rate of attacker packets is ten times the router's capacity of forwarding packets through the link, the packet drop rate will be over 90%, meaning that more than 90% of legitimate user packets will be dropped. This high packet drop rate can usually prevent legitimate users from obtaining any meaningful service via the congested link, as most transport protocols (*e.g.*, TCP) would fail to transmit application data at usable speeds.

In this dissertation, we focus on mitigating DoS flooding attacks, but only briefly discuss application-level DoS attack defense in Section 11.3. For brevity, we use the term *DoS attack* to refer to *DoS flooding attack* unless otherwise noted.

Over the years, DoS flooding attacks have become a serious threat to the reliability of the Internet. Table 2.1 enumerates some of the notable attacks that have been made public. Some of these attacks are reported to serve various purposes, ranging from hackers' revenge [33, 76], to gaining sales advantages over competitors [49], to extortion [69], and even to achieving political goals [65, 66, 44, 95, 72, 79, 93].

The attackers' power to launch DoS flooding attacks has been dramatically increasing during the past few years. A network operator survey published by Arbor Networks [5] shows that the attack traffic volume in the largest attack observed in 2010 reaches over 100Gbps, a 1000% increase over the largest attack observed in 2005. Partly due to this immense attack power, 68% respondents in this survey consider DoS attacks as their most significant operational threat, and 59% respondents consider DoS attacks as the top security concern for the next 12 months.

There are two major reasons why attackers can send out an excessive amount of attack traffic. One is that the Internet keeps improving with links that have higher capacities; but more importantly, attackers are able to control a large number of compromised computers, usually referred to as *botnet machines*. In March 2007, the number of botnet machines tracked by a single group was estimated to reach 1.2 million [47]. In June 2007, a presentation from Support Intelligence Inc. reported 48 million infected IP addresses observed in a six month period [96]. In September 2007, the estimated size of the Storm botnet alone reached 50 million [45, 87]. These compromised machines give attackers immense power: if each botnet machine sends one full-sized packet (1500 bytes) per second, the aggregated attack traffic from a 10-million-node botnet would exceed 120 Gbps, sufficient to take down almost any link in the current Internet.

2.2 Source Spoofing and its Mitigation

Source spoofing refers to the attack in which attackers fake the source IP addresses of the attack packets. The fact that source IP addresses can be spoofed in the Internet undermines the security and reliability of the Internet in a variety of ways; in particular, it simplifies the task of launching DoS flooding attacks and makes the defense of DoS flooding attacks much more difficult.

Source spoofing empowers attackers with *reflector attack*, in which the attackers send requests with spoofed source addresses to hosts that will send responses to the spoofed sources. The hosts that send out responses are referred to as *reflectors*, and the spoofed sources are usually the victims of the attack. A reflector attack allows the attackers to hide their true locations from the victims; more importantly, when the response to a request is much larger than the request itself, a reflector attack enables the attackers to amplify the attack traffic, making it possible to launch large-volume DoS flooding attacks with a few true attack sources that only send out moderate attack traffic. For instance, reflector attacks in the early 2006 used DNS servers as reflectors to flood the victims with up to 5 Gbps attack traffic, and the amplification ratio reached 63:1 [81].

Source spoofing also complicates measures to mitigate DoS flooding attacks, because source addresses cannot serve as the indicator of attack traffic. For instance, with source spoofing, we cannot simply filter out attack traffic based on the source addresses, because attackers may deliberately spoof the addresses of legitimate hosts, and then the filters would also block the traffic from the legitimate hosts. Similarly, fair queuing based on source addresses cannot effectively limit bandwidth consumed by attack traffic. This situation creates a vicious cycle in deploying automated DoS flooding defense mechanisms [34]: the possibility of spoofing leads to the absence of automated defense mechanisms; the absence of such mechanisms obviates the

need to attack with spoofed sources, which leads back to the lack of deployment of anti-spoofing mechanisms that makes spoofing possible.

There has been much work in preventing source spoofing. The existing solutions can be classified into five categories: three-way handshake, router filters, cryptographic approaches, path marking, and IP traceback.

2.2.1 Three-Way Handshake

A simple solution to prevent source spoofing at end systems is to use three-way handshakes at the beginning of an interaction. If a source host spoofs its IP address, it will be unable to finish a three-way handshake. This solution works well to prevent source spoofing at end systems, but attackers are free to spoof the source address of the first packet of a three-way handshake, and they can launch DoS flooding attacks with these packets. CAT [16] proposes to deploy in-network cookie boxes to proxy the three-way handshakes for a server so that attackers cannot use packets with spoofed source addresses to congest a link downstream of a cookie box, but the links upstream of the first on-path cookie box are still vulnerable to packet flooding with spoofed source addresses.

2.2.2 Router Filters

The solutions in this category aim to detect and filter packets with spoofed sources without modifying the packet headers. The most widely used mechanism in today's Internet is Ingress Filtering [28]: that is, each access router voluntarily discards incoming packets whose source addresses are outside of the known legitimate address range. For instance, if an ISP provides network connectivity for a particular organization whose IP address range is known to be `100.200.0.0/16`, the ISP's router connecting the organization can be configured to discard any packet from the organization whose source address does not fall into `100.200.0.0/16`.

Ingress Filtering is a lightweight solution readily deployable in the Internet. However, it only provides a very limited security benefit: as long as a single access router does not filter its incoming packets, *e.g.*, when the router has not deployed Ingress Filtering or when the router is simply compromised, attackers behind this router can spoof any other addresses in the Internet. In fact, the latest measurement shows that more than 14% of the allocated IP addresses and ASes on the Internet still allow source address spoofing [88] despite the fact that Ingress Filtering has been standardized as an Internet Best Current Practice for over ten years. As a result, when a router in the middle of the network forwards a packet, it has no guarantee that the packet's source address is authentic or even close to the real packet source.

Strict Reverse Path Forwarding [7] can be considered as an enhanced version of Ingress Filtering: it proposes that each router only forwards a packet if the incoming interface matches the outgoing interface for the packet's source address. It provides a better security benefit compared to Ingress Filtering, because every router, including routers in the middle of the network, can now filter packets, and therefore it is no longer sufficient to compromise a single access router in order to launch source spoofing attacks. However, Strict Reverse Path Forwarding does not work well with asymmetric routing, because when asymmetric routing is present, the outgoing interface for a packet's source address may not be the legitimate incoming interface for packets from the source.

Feasible Path Reverse Path Forwarding [7] is an extension to Strict Reverse Path Forwarding in that a packet will not be dropped as long as there exists a possible (not necessarily chosen) route to reach the packet's source address via the packet's incoming interface. It can work with asymmetric routing, but it may still mistakenly discard packets if some feasible routes are filtered due to routing policy restrictions. More importantly, Feasible Path Reverse Path Forwarding does not fully eliminate source spoofing even when every router deploys it, because as long as there is a

feasible route to a source address via a particular interface, attackers behind the interface can spoof the source address. IDPF [25] is based on Feasible Path Reverse Path Forwarding, but it obtains the feasible routes from BGP. It argues that the business relationship between ASes helps to significantly reduce the number of feasible routes and therefore limits an attacker's ability to spoof source addresses, but some attackers are still able to do source spoofing.

Park et al. [70] analyzed the effectiveness of route-based packet filtering in preventing source spoofing. They analyzed an idealized version of route-based packet filtering: that is, every router has the global knowledge of how the other routers choose routes, and then discards packets that do not follow legitimate routes according to the source and destination addresses in the packets. The analysis shows that with the AS topologies from 1997 to 1999, if a relatively small but carefully chosen set of ASes deploy the idealized router-based packet filtering, attackers can only spoof source addresses if they are in 12% of all the ASes. However, [70] does not propose a solution to realize the idealized route-based filtering: it is non-trivial to allow each router obtain the accurate global routing knowledge, especially when routers may be compromised by attackers.

SAVE [50] is a concrete protocol that realizes route-based filtering. In SAVE, a router maintains an incoming table that maps a source address prefix to an incoming interface at the router. A source AS that deploys SAVE periodically sends a source address update for every destination prefix in its routing table. A router uses the incoming interface of an update message to update its incoming table. To ensure prompt update of incoming tables, an AS may also send out source address updates when routing changes. When a router receives a packet, it discards the packet if the incoming interface does not match the one associated with the packet's source address in the incoming table.

SAVE fully realizes route-based filtering, but the source address updates intro-

duce non-trivial message overhead, especially when a routing change occurs and affects a large number of destination prefixes at the same time. In addition, without signing the source address updates using public key cryptography, a single compromised or malicious router can initiate bogus source address updates and allow attackers to spoof arbitrary source addresses; if the source address updates are signed, the processing overhead for signature generation and verification is non-trivial, and the message overhead will be even larger with the additional public key signatures.

Hop-Count Filtering [39] uses a hop-count table indexed by source IP addresses to filter packets that may have spoofed source addresses. A router or server first builds its hop-count table based on TTLs in packets it receives when there are no source spoofing attacks, and then discards a packet if the hop count inferred from the packet's TTL does not match the hop-count table. Hop-Count Filtering cannot fully eliminate source spoofing, because attackers can still spoof source addresses that are further from the victim than themselves.

Like Ingress Filtering, all the solutions in this router filters category share the same weakness that they do not provide verifiability for source addresses in packets. That is, when a router in the middle of the network receives a packet, it cannot verify by itself whether the source address is authentic. Even with full deployment, a router compromise may enable attackers to spoof many source addresses.

2.2.3 Cryptographic Approaches

The solutions in this category insert cryptographic tokens into packet headers to prove the source addresses are authentic. For instance, Spoofing Prevention Method (SPM) [13] inserts a key into every packet. Each key is a shared secret between the source AS and the destination AS; it is inserted by the source AS and verified by the destination AS. An attacker who does not know the right key cannot spoof the source address of the source AS at the destination AS. Each AS has an AS-server

that pushes the keys to all the border routers of the AS and periodically sends a new set of keys to all the AS-servers in other ASes.

SPM allows each destination AS to verify whether a packet's source address is spoofed, and it is lightweight in terms of computational overhead. However, the security benefit it provides is still very limited: transit ASes cannot verify the authenticity of source addresses, and therefore in the context of DoS flooding attack mitigation, attackers can still congest links in the middle of the network with packets that have spoofed source addresses. In addition, the keys in packet headers are plain text; attackers who can eavesdrop the packets can use the observed keys to spoof source addresses. The key distribution in SPM also has non-trivial management cost, because it requires each AS to know the AS-servers of all the other ASes.

In order to allow every router to verify each packet's source address, Perlman [73] lets a sender sign a packet with its private key so that any router on the packet's forwarding path can verify the authenticity of the source address using the sender's public key. A set of nodes trusted by everyone in the network flood all the public keys to every router. Although simple and effective, this approach introduces significant computation, communication and storage overhead. The validation of public key signatures are very computationally expensive, and performing the validation at packet forwarding time significantly limits the packet forwarding throughput that can be achieved at the router. In addition, at the scale of the Internet, the list of public keys of every host that may send out traffic is huge. As a result, distributing the public keys requires a large amount of traffic, and a router needs a large storage space to keep all the public keys.

Similar to Perlman's approach, Host Identity Protocol (HIP) [63] also assigns a public-private key pair to every sender. However, instead of signing the source IP address of each packet, HIP uses the public key itself as the source identifier for upper layer protocols. These HIP source identifiers are intended to authenticate end

systems and establish end-to-end secure communication channels, not to enable the verification of every packet's source address on forwarding routers.

2.2.4 Path Marking

The solutions in this category bypass the source spoofing problem by not using source addresses to identify the packets' real sources. Instead, they let routers stamp the path information into every forwarded packet. If none of the routers are malicious, the path information indicates a packet's true source; however, malicious routers, or even malicious attack hosts behind access routers that do not reset the path information field, may forge at least part of the path information. For instance, AITF [6] lets each border router stamp its IP address into the forwarded packets, and TVA [102] lets each router stamp the local identifier of a request packet's incoming interface into the packet. Pi [98] and StackPi [101] also let each router stamp part of its IP address into the forwarded packets, but different from AITF and TVA, they only use a fixed length field in the packet header and aim to ensure different paths have different identifiers with high probability instead of recording the full paths.

2.2.5 IP Traceback

There have been many proposals that do not aim to prevent source spoofing at packet forwarding time; instead, their goal is to be able to locate the real source of a packet after the packet is forwarded towards the destination. As a result, these proposals cannot eliminate source spoofing themselves. For instance, Savage et al. [80] propose several ways of probabilistically stamping the information about on-path routers into packets so that if an attack host sends out enough packets towards a victim, the victim can reconstruct the path of the attack packets and therefore discover the true location of the attack host regardless of how the attacker spoofs the packets' source addresses. Song et al. [86] propose two packet marking schemes better than those

in [80]: one requires fewer attack packets to reconstruct the attack path, and the other allows authenticating router markings such that a compromised router cannot forge the markings of other routers. Yaar et al. [100] propose FIT, a mechanism that is still based on probabilistic packet marking but can reconstruct the attack path with even fewer attack packets. Snoeren et al. [85] take a different approach: they propose to deploy bloom filters on routers to record every forwarded packet so that the attack path can be reconstructed given a single attack packet.

2.3 DoS Flooding Attack Mitigation

There have been many proposals in mitigating DoS flooding attacks. Most of the proposals fall into one of three categories: blacklist-based, whitelist-based, or overlay-based.

2.3.1 Blacklist-based Approaches

In a blacklist-based approach, end hosts communicate with each other in the same way as they do in the current Internet when there are no DoS flooding attacks; however, at attack time, traffic filters are installed in the network or sometimes even on end hosts to discard the attack traffic. The simplest blacklist-based approach is *null routing* [30]: that is, when the attack traffic is detected as going towards one or a few specific destination addresses, some routers on the forwarding path of the attack traffic are configured to discard all the traffic towards the destinations. This packet discarding configuration is usually done with null routes, *i.e.*, a special type of route whose next hop is the null interface that discards all packets going through it. The null routes are usually installed on routers in the destination AS, because the purpose of this mitigation approach is mostly to avoid wasting the destination AS' bandwidth on the DoS attack traffic. It does not benefit the victims, who are usually the destinations of the DoS attack traffic, because legitimate user traffic towards the

victims is discarded as well.

A more complicated and useful blacklist-based approach is to let traffic towards the DoS attack victims go through dedicated machines that detect and filter the attack traffic. There are a few ways to deploy the dedicated traffic filtering machines [91]: they can be put inline at the victims themselves, or installed at the destination AS, or even deployed on cloud computing platforms. In the latter cases, when a DoS attack occurs, traffic towards the victims is redirected to the filtering machines via either routing or IP encapsulation, and the traffic that passes the filtering is forwarded back to the victims. Although immediately deployable in the current Internet, this filtering machine based approach cannot make the victims immune to DoS flooding attacks: powerful attackers can simply congest the links at the upstream of the filtering machines.

dFence [59] presents a detailed design that follows the filtering machine based approach. It proposes to let the filtering machines process all the traffic to and from a victim; as a result, the filtering machines can deploy more comprehensive DoS mitigation mechanisms. In addition, dFence also discusses how to handle replacement, failure recovery and load balancing of the filtering machines.

AITF [6] is a complete system that installs network filters close to attack sources in order to discard attack traffic as early as possible. To locate attack sources and have an explicit way to describe attack traffic, AITF requires each participating router append its IP address into a *route record* in each forwarded packet. When a destination host is under DoS attack and can distinguish the attack traffic from the legitimate traffic, it can send out a filter request to its access router asking to discard the attack packets containing specific route records. The destination access router then uses three-way handshakes to relay the filtering request to the first router in each route record in the request. Upon receiving the relayed filtering request, a legitimate router further relays the request to the source host. If the source host does

not stop sending the attack traffic, the legitimate router will disconnect the source host. Similarly, when a destination access router relays a filter request to a router but the corresponding attack traffic is not stopped, the destination access router will consider the router as malicious; therefore, it will relay the request to the next router in the route record and ask to block any traffic forwarded by the previous router that cannot stop the attack traffic. This incentivize each router to honor relayed filtering requests.

AITF works well when only end hosts are malicious and the attack traffic only congests links in the destination AS. However, it is vulnerable to several more complicated attacks. First of all, when a router is malicious, it may stamp fake route records when forwarding attack packets; as a result, when a DoS victim receives a packet, the first half of the route record in the packet may be spoofed. When the victim sends out a filtering request to block traffic with the fake route record, the request may eventually reach a malicious router, and the malicious router can simply start using another fake route record. This makes it very slow, if not impossible, to stop the attack traffic from a particular source host. Second, the three-way handshakes used to securely relay filtering requests may be dropped if they need to traverse congested links themselves, and then the destination access router will mistakenly treat legitimate routers as malicious and request to block traffic forwarded by them, causing collateral damage to legitimate traffic. Third, AITF assumes that each router has enough memory to install network filters that block the attack traffic, but this assumption may not hold with a large number of attack sources. When the attackers can exhaust the router memory used to install network filters, some attack traffic will not be stopped. The AITF design partially addresses this router memory exhaustion problem by using a shorter filter expiration time, but then it will allow the attackers to launch effective on-off attacks: the attack sources can be organized into groups that send the attack traffic at different times, and each group

can be frequently reused so that there is always attack traffic that is not discarded by network filters.

StopIt [53] is similar to AITF in terms of goals and the overall filter-based architecture, but it is more effective against strategic attacks. Instead of using route records, it relies on a source authentication system Passport (presented later in this dissertation) to ensure the authenticity of each packet's source address, and then uses the source address to precisely locate the source access router and describe the attack traffic. Once a DoS attack victim identifies an attack source, it sends a filtering request to its access router. The destination access router will relay the request directly to the source AS, and the source AS will eventually deliver the request to the source access router. In this request relay process, the request can be fully authenticated, because the source authentication system Passport also enables each pair of ASes to share a symmetric secret key. As a result, StopIt does not need the three-way handshakes in the AITF design. Since source addresses are authentic, routers can use source-based fair queuing to give filtering requests higher forwarding priority than normal traffic, preventing the requests from being dropped due to the attack traffic. To mitigate filter exhaustion attacks in which attackers attempt to exhaust the router memory for installing network filters, StopIt installs a filter only when necessary, and uses filter aggregation, random filter replacement, and catch-and-punish mechanisms to ensure that even with small router memory, if an attack source receives a filtering request but does not stop the attack traffic, it will be caught and punished. StopIt also uses source-based fair queuing to separate traffic from different ASes; if a source access router is malicious and does not enforce the network filters, the attack traffic will not starve legitimate traffic from other ASes.

AIP [3] uses a simple shut-off protocol to enable a receiver of a packet to signal to the source of the packet that it no longer accepts packets from the source for a certain duration. This shut-off protocol contains only one shut-off message; it can

be this simple because AIP requires each source host be equipped with a smart NIC that can recognize shut-off messages and cannot be tempered by remote attackers who may compromise the host itself. In addition, AIP has a built-in source authentication mechanism, and therefore the receiver of a shut-off message can trust that the message comes from the claimed source host. However, the AIP design does not address the filter exhaustion problem; that it, it assumes that a smart NIC can have enough memory to keep all the installed network filters. This assumption may not hold when a compromised host colludes with a large number of other compromised hosts that keep sending in shut-off messages.

Simon et al. [84] propose a network filter based system similar to StopIt in terms of filter request propagation and filter installation. However, the proposal assumes that only end hosts may be malicious, and therefore it does not have the proper mechanisms to mitigate malicious routers. When this assumption does not hold, *i.e.*, when some routers are malicious, attack traffic going through the malicious routers cannot be stopped.

Blacklist-based DoS mitigation schemes all require setting up the proper blacklists in order to discard the attack traffic. As a result, they share the common weakness that if attackers can manage to avoid triggering the blacklisting operations (*e.g.*, null routing or packet filtering), they will become less effective or even completely broken. In particular, they are vulnerable to *diffusion attacks* in which the attack traffic all traverses a target link but goes to many different destinations. In such an attack, even when the target link is congested, each destination only receives modest attack traffic and therefore will not initiate the blacklisting operations. With a diffusion attack, some DoS mitigation schemes such as AITF will be completely broken, while StopIt will fall back to per-source-host fair queuing to prevent the attack traffic from completely starving the legitimate traffic. Although effective, per-source-host fair queuing is not scalable enough for core routers that may forward packets for millions

of source hosts.

2.3.2 Whitelist-based Approaches

Unlike blacklist-based approaches, whitelist-based DoS mitigation schemes require changing the way end hosts communicate with each other in the current Internet. That is, one host cannot directly send data to another host; instead, a sender has to first obtain the permission to send to a receiver. For instance, in the Visa protocol [26], each organization has an Access Control Server (ACS). When a source host intends to send data to a destination, it has to first obtain two Visa keys, one from the source ACS and one from the destination ACS. Then it uses the two keys to generate two signatures (either MACs or public key signatures, depending on the type of the keys) for each packet it sends to the destination, and the signatures are verified when the packets leave the source organization and ender the destination organization. If the destination does not want to receive traffic from the source, it may direct its ACS not to issue visa keys to the source. However, the Visa protocol does not specify how to protect the requests for Visa keys from being dropped because of DoS attacks. In addition, it implicitly assumes that only end hosts can be malicious, but this assumption may not hold in the DoS mitigation context. The per-packet signature validation is also unscalable: it requires the validation routers to either keep per-connection state, or verify public key signatures for every packet.

Ballani et al. [9] propose to change the communication model in the current Internet from default-on to default-off: that is, by default a destination host is unreachable to any source host. When a destination host wants to receive traffic, it explicitly announces its reachability to the rest of the Internet, similar to how routing information is propagated in the Internet. Its reachability information specifies who are allowed to send traffic to it. When a router forwards a packet, it discards the packet if the reachability information of the destination does not allow traffic from

the source. To avoid overloading the Internet with the reachability announcements, a router may aggregate the reachability information regarding multiple destinations. However, with this aggregation, unwanted traffic can still enter the network and will only be dropped when it reaches routers that do not aggregate the destinations' reachability information, and therefore it may still congest the links it traverses and cause denial of service. In addition, the proposed design does not specify how a source host may contact a destination host to obtain the permission to send traffic to the destination host.

SIFF [99] addresses two key problems for a whitelist-based DoS mitigation scheme: how to enable a source to obtain the permission to send, and how to enable a router to enforce whitelisting in a lightweight way. In the SIFF design, when a source desires to communicate with a destination, it first sends an EXPLORER (EXP) packet to the destination. Each router on the path inserts a marking referred to as *capability* into the packet. The destination returns all the capabilities back to the source as the proof that it is willing to receive traffic from the source. Then the source can send DATA (DAT) packets to the destination in bulk; each DAT packet carries the returned capabilities. Each router on the path verifies that each DAT packet carries the correct capability it has inserted before; DAT packets without the proper capabilities are discarded. Each capability is generated using a keyed hash function that covers information such as the packet's source and destination addresses, which prevents attackers from reusing the capability elsewhere. Routers forward valid DAT packets with a higher priority than EXP packets, and therefore as long as a source host obtains the capabilities, its traffic to the destination will not be affected by EXP packets flooded by attackers. The verification of capabilities in DAT packets is lightweight: it does not require a router to keep per-source state or use excessive computation power (*e.g.*, to verify public key signatures). The major weakness of SIFF is that EXP packets are not protected: attackers can send a large amount of

EXP packets to congest a link, and then legitimate EXP packets will have a high drop rate on the link, preventing legitimate senders from obtaining the permissions to send. In addition, the capabilities in SIFF are too coarse-grained: once an attacker obtains them, it can keep using them to send as much attack traffic as it can generate until a router changes its key for computing capabilities.

TVA [103] shares the same capability-based architecture as SIFF, but it improves upon SIFF in three key aspects. First of all, the packets used to request for capabilities, referred to as *request packets* in TVA, are queued hierarchically based on the router markings they carry. Since an attacker cannot forge router markings stamped by downstream routers, this hierarchical queuing can separate the attack traffic from the legitimate traffic that traverses different paths. In addition, on every link request packets can use up to 5% of of the link capacity. These two mechanisms can significantly reduce the attackers' ability to prevent legitimate users from obtaining capabilities. Second, capabilities in TVA are fine-grained: when a destination returns capabilities to a sender, it can specify how long the capabilities can last and how much traffic can be sent using the capabilities. Third, packets with capabilities, referred to as *data packets* in TVA, are queued based on their destinations. This design prevents a pair of malicious sender and receiver from colluding to congest a link with data packets. The major weakness of TVA is that its use of router marking based hierarchical fair queuing for request packets and destination based fair queuing for data packets requires each router to keep per-host states, which is not scalable enough at the scale of the Internet. When a router does not have enough queues, attack traffic and legitimate traffic may fall into the same queue, causing denial of service for the legitimate users.

Portcullis [71] is another mechanism to protect legitimate users' packets used to request for capabilities. When a sender sends such a packet, it has to attach the proof that it has solved a computation puzzle. The harder the puzzle is, *i.e.*,

the more computational power the sender spends on solving the puzzle, the higher the packet's forwarding priority will be. It has been proved that regardless of the number of attackers, a legitimate sender can always have its capability-requesting packet delivered to the destination as long as it spends enough computational power. To enable routers to verify the computation puzzles without keeping much state, Portcullis requires a trusted third party periodically generate and distribute via DNS a puzzle seed. The Portcullis design has two major drawbacks: it requires a dedicated external infrastructure DNS that has to be reachable to routers even at attack time, and the computational power needed to solve the puzzles may be too expensive to busy servers and legitimate users at attack time.

Flow-cookies [17] is a whitelist-based approach specifically designed to protect TCP traffic. When a sender initializes a TCP connection, it finishes the three-way handshake with a filtering machine in front of the receiver but not with the receiver itself. The filtering machine uses TCP-handoff [92] to pass established TCP connections to the receiver. When the receiver sends a packet back to a sender, the filtering machine inserts a secure flow cookie in the TCP timestamp field. The sender has to echo back the flow cookie in its subsequent packets to the receiver; otherwise, the filtering machine will drop its packets.

2.3.3 Overlay-based Approaches

Overlay-based DoS mitigation schemes use middle boxes in the network to relay traffic from sources to destinations. These middle boxes may have access control policies that can discard attack traffic. For instance, in the Mayday design [2], a receiver under DoS attack is "surrounded" by routers that only allow authenticated traffic to reach the receiver. The routers are not necessarily close to the receiver; it suffices if they cover every forwarding path that may reach the receiver. This prevents senders, both legitimate and malicious, from directly sending traffic to the receiver.

When a sender wants to send traffic to the receiver, it has to first authenticate itself to a node in an overlay network trusted by the receiver, and then send the packets to an overlay node. The packets may traverse multiple overlay nodes and then exit the overlay network at an egress node. The egress node sends the packets to the receiver after inserting authentication information into the packets, which enables the packets to pass the receiver's surrounding routers. Mayday is a generalization of SOS [40], which settles on particular design choices under the same framework. For instance, SOS uses source addresses as the authentication information on routers surrounding a receiver: the routers only allow packets whose sources are valid egress overlay nodes to pass.

Stavrou et al. [89] propose a system that shares the same overall architecture as SOS and Mayday but improves upon them in two aspects. First, instead of sending packets to a particular overlay node, in Stavrou's design a sender spreads its traffic to a pseudo-random sequence of overlay nodes. This prevents attackers from concentrating on congesting the incoming link of a particular overlay node and causing denial of service to a particular set of legitimate users. Second, after a sender authenticates itself to an overlay node, it will receive a ticket that has to be attached to the sender's packets towards the destination. The ticket includes a message encrypted with a symmetric key only known to the overlay nodes; as a result, an overlay node can verify the tickets without keeping much state.

Phalanx [23] presents another system that improves upon Stavrou's design. It does not require an overlay network; instead, it only needs a set of middle boxes that do not communicate with each other. When a sender desires to send to a receiver, it first sends a request packet to one of the middle boxes. It authenticates itself by attaching either a computation puzzle (*e.g.*, a puzzle used in Portcullis [71]) or a pre-distributed cryptographic token to the request packet. A middle box does not forward packets to the receiver; instead, it verifies the legitimacy of the request packet

and then stores the packet locally. It is the receiver's responsibility to periodically pull packets from middle boxes. This design gives a receiver full control over what traffic it will receive. Once the receiver obtains a request packet, it can establish a shared secret with the sender; therefore, the sender and receiver can secretly agree to a pseudo-random sequence of middle boxes. The sender will deliver subsequent packets to the sequence of middle boxes one by one, and the receiver will sequentially pull the packets from the sequence of middle boxes. The sender's subsequent packets contain authentication information signed using the secret it shares with the receiver, and a middle box only accepts packets with the correct authentication information. Since packets stored on middle boxes are always pulled by the receiver, the receiver's surrounding routers can use bloom filters to control what packets can pass them and reach the receiver. That is, whenever the receiver sends a message to pull a packet from a middle box, the filtering router on the path will record the nonce in the message in a bloom filter; then when the pulled packet reaches the filtering router and it has the same nonce, the router can let it pass.

These overlay-based DoS defenses share two common weaknesses. First, they all require overlay nodes keep non-trivial amount of state at the Internet scale. Even in Stavrou's design [89], although an overlay node does not need much state to validate tickets, it still needs to keep enough state to authenticate every sender for every receiver it protects. Second, these approaches do not change the underlying network; as a result, attacks at the underlying network may still be able to cause denial of service. For instance, if attackers can congest the incoming links of a server's surrounding routers, users will be denied the service regardless how the overlay network performs.

2.3.4 Other Approaches

Kreibich et al. [42] propose to use packet symmetry to curtail DoS attack traffic. Their observation is that during a DoS attack, an attacker host sends out a large amount of traffic while receives little incoming traffic. Therefore, if each access router shapes each host's traffic to keep a relatively low ratio of outgoing traffic over incoming traffic, DoS attack traffic will be significantly curtailed. However, the threshold ratio is difficult to set, because legitimate applications may also result in a high ratio of outgoing traffic over incoming traffic.

An ideal DoS mitigation scheme should be able to identify and drop attack traffic as early as possible, but this goal is often difficult to achieve, especially when attack traffic cannot be distinguished from legitimate traffic or those who can identify attack traffic do not ask to stop the attack traffic. A less effective alternative approach is to limit the congestion of the network and ensure fair bandwidth allocation. For instance, CSFQ [90] can enforce per-source-destination-pair fairness on any link without requiring routers keep per-flow state. With CSFQ, an access router monitors the traffic rate from a sender to a receiver and stamps the rate into every packet from the sender to the receiver. Subsequent routers use the stamped rate to calculate the max-min fair share of bandwidth belonging to the source-destination pair, and then enforce the fair share by probabilistically dropping packets. Although CSFQ is effective in enforcing per-source-destination-pair fairness and lightweight, it is vulnerable to diffusion attacks described in Section 2.3.1, because when an attack source sends to multiple destinations, it can obtain more bottleneck bandwidth than a legitimate source that only communicate with a few destinations. In addition, CSFQ relies on trustworthy access routers to correctly stamp traffic rates into packets; if an access router is compromised, it can stamp a very low traffic rate in attack packets and severely congest a link.

Different from CSFQ, pushback [57] enforces a per-interface fairness on bottleneck links. In pushback, each router monitors packet drops on every link. If the packet drop rate on a link passes a threshold, the router starts to identify aggregates of traffic that consume most of the link bandwidth. Each aggregate is described with a destination address prefix. Once the aggregates are identified, the router computes the max-min fair share of outgoing bandwidth for each aggregate, and then uses rate limiters to enforce the calculated fair shares of bandwidth. Based on the incoming traffic on each incoming interface, the router also computes the max-min fair share of incoming rate for each aggregate on each incoming interface, and then propagates these fair share rates to upstreams routers. Each upstream router repeats the bandwidth allocation process and further propagates the computed fair share rates. Similar to CSFQ, pushback is also vulnerable to diffusion attacks: if an attacker can trigger 100 aggregates on a congested link, a victim behind the link can only get at most 1% of the link capacity, significantly reducing the number of legitimate users the victim can serve. In addition, attack traffic and legitimate traffic may be put into the same aggregate, causing denial of service for the legitimate users.

Re-ECN/re-feedback [15, 14] is another congestion policing framework that can enforce per-source-destination-pair fairness. In this scheme, when a packet leaves its sender, it carries the downstream congestion level known to the sender. Each congested router on the packet's forwarding path updates the congestion level, and the final congestion level received by the receiver is returned to the sender to be stamped into the sender's subsequent packets. Re-ECN/re-feedback incentivizes a sender to honestly stamp the correct congestion level by rate-limiting packets with high congestion levels and dropping packets at congested routers where the congestion levels become negative. If a sender sets the congestion level higher than reported, its traffic will be throttled more by its access router; if it sets the congestion level lower than reported, the congestion level will become negative at a certain congested router and

the packet will be discarded. Similar to CSFQ, re-ECN/re-feedback is also vulnerable to diffusion attacks. In addition, in re-ECN, the congestion level in each packet can only be either zero or one. If attackers flood packets with congestion level zero, legitimate senders can only set the congestion level of their packets to one; otherwise, their packets will suffer from high packet loss rate on congested links. However, when they set the congestion level to one, their traffic would be severely throttled by their access routers, causing a severe degradation of the service quality perceived by the legitimate senders.

3

Design Space

In this chapter, we present the threat model we face, the enabling assumptions that make our design possible, and the design goals of our system.

3.1 Threat Model

DoS flooding attacks: This dissertation focuses on mitigating DoS flooding attacks, in which attackers attempt to cause denial of service by congesting links or exhausting resources on routers in the network. Application-level DoS attacks that aim to exploit vulnerabilities or exhaust resources at online services will be briefly discussed in Section 11.3, but their detailed defense mechanisms are outside the scope of this dissertation.

Untrusted hosts and routers: We assume that attackers may control both end hosts and routers when they launch DoS attacks. This assumption prevents us from using simple mechanisms that rely on the trustworthiness of routers. For instance, we can not trust that every access router will discard outgoing packets with spoofed source addresses; if we do, and an access router is compromised, attack sources

behind that router will be able to spoof any source address.

Large scale attacks: We assume that attackers may use a large number of attack sources in a DoS attack, because it has been shown that the number of botnet machines in the Internet is huge [47, 96, 45, 87, 27]. This assumption forces us to minimize the requirement for computation power and storage on core routers in our design, because a core router in the middle of the Internet may potentially face a large volume of attack traffic from a large number of sources.

3.2 Enabling Assumptions

Routers are less likely to be compromised than end hosts: Attackers rarely use their own equipments to directly send out the attack traffic; instead, they usually compromise end hosts and routers by taking advantage of software vulnerabilities and weak passwords, and then send the attack traffic from the compromised machines. Compared to routers, end hosts usually run more complicated software that is more prone to security vulnerabilities and are maintained by less knowledgeable users; therefore, we assume that it is easier for attackers to compromise end hosts than to compromise routers. This assumption allows us to reduce the complexity and improve the scalability of our design by optimizing for the case that attackers only control at most a small number of routers. For instance, we can place traffic policing functionalities at access routers to minimize the amount of state kept on core routers, and deploy efficient source-AS-based fair queuing on core routers such that if an access router is compromised, the attack traffic going through it only harms legitimate traffic originating from the same AS. This design is only acceptable when routers are rarely compromised; if we assume access routers are as easily compromised as end hosts, this design will become largely ineffective.

Line-speed symmetric key cryptography: We assume that symmetric key

cryptographic operations can be supported at line speed even on core routers. Although many current routers do not yet have the proper hardware to achieve this performance, we believe this assumption is reasonable and achievable, because some current commercially available hardware can support AES operations at 40Gbps [37], and even generic CPUs start to have native support for AES (*e.g.*, Intel Westmere CPUs [38]).

3.3 Design Goals

Provable authentic source addresses: We aim to make source addresses provably authentic, *i.e.*, a router forwarding a packet can independently verify the source address without trusting other upstream routers. A perfect scheme would check every packet that arrives at each router and verify that the packet carries the source address of the host that injects it into the network; packets with spoofed addresses would be identified precisely and discarded. However, this perfect scheme is unattainable, and we relax the goals of source authentication to permit more realistic designs.

First, we relax the granularity of source authentication. Our design treats an AS as a trust and fate-sharing unit. That is, it only prevents hosts in one AS from spoofing the addresses of other ASes. As each AS is separately administered, we consider it to be an internal issue for an AS to prevent a malicious host in its network from spoofing the addresses of other hosts in its network. Each AS can use whatever method it prefers to do so. Even if an AS does not deploy any intra-domain source spoofing prevention scheme, the damage of source spoofing among hosts inside this AS can be confined to this AS: for instance, routers can use source AS based fair queuing to isolate traffic from different ASes so that attackers inside an AS can only cause denial of service to legitimate users in the same AS, or a destination host can discard traffic from any host in an AS that does not prevent intra-domain source

spoofing.

Second, we do not verify the freshness of each packet. That is, we do not distinguish between an authentic packet and a replay (a subsequent copy) of the same packet that is injected along the same network path as the authentic packet. This has the downside that a packet may be duplicated at any point along the network path and still be considered originating from the source address. While it would be desirable to weed out duplicates and we do have a solution presented in an early version of the Passport design [54], this requires more processing than what we think belongs to the lowest layer of source authentication. Moreover, it is not clear that duplicate packets are as problematic as spoofed packets. Therefore, we do not make replay prevention one of our goals.

Effective in mitigating DoS attacks: We aim to make our design effective against DoS attacks. More specifically, when DoS victims can identify the attack traffic, we aim to enable them to suppress the attack traffic near the origins. This reduces the chance that the attack traffic causes any damage and prevents the attack traffic from wasting network resources. When DoS victims fail to identify the attack traffic, or attackers collude into sender-receiver pairs to flood the network such that no legitimate entities can distinguish attack traffic from legitimate traffic, we resort to a weaker goal to guarantee a legitimate sender its fair share of network resources. That is, for any link of capacity C shared by N (legitimate and malicious) senders, each sender with sufficient traffic demand should be guaranteed at least $O(\frac{C}{N})$ bandwidth share from that link. This mitigates the effect of large-scale DoS attacks from denial of service to predictable delay of service.

Robust: Our design should be robust against both simple, brute-force attacks and sophisticated ones that attempt to bypass or abuse the defense system itself.

Scalable and lightweight: Our source authentication sub-system has to run on

core routers that forward packets from a large number of sources and at high speeds. Therefore, our design should be lightweight such that *upgraded* routers, *i.e.*, routers with our design enabled, can still forward packets at line speed.

Our DoS mitigation sub-system may face millions of attackers that attempt to congest a single link. To be effective at such a scale, it does not assume that a router always has sufficient resources to warrant per-flow or per-host state management. It aims to keep little or no state in the core network and avoid heavyweight operations such as per-flow or per-host fair queuing in the core network. To facilitate high-speed router implementation, our design aims to incur low communication, computation, and memory overhead.

Incrementally deployable: It has become increasingly difficult to deploy new network-layer systems in the Internet due to the large legacy install base. Therefore, our design should be incrementally deployable without requiring a "flag day" in which everyone deploys at the same time. In addition, our design should incentivize its deployment by providing immediate benefits to early adopters. That is, even a few early adopters should be able to collectively benefit from the deployment, and the benefit should increase as more ASes deploy our design, creating a network effect to drive the deployment.

Self-contained: We aim for a self-contained solution that only depends on routers in the network, not other infrastructures such as trusted host hardware [3] or DNS extensions [71]. Our hypothesis is that extra dependencies increase security risk and may create deployment deadlocks. That is, without the deployment or upgrade of other infrastructures, the design is not effective. Hence, there is little incentive to deploy it, and vice versa.

4

Architecture

Our design of a full DoS defense system consists of two separate sub-systems: Passport for source authentication, and NetFence for DoS mitigation. Next we present the architecture of each sub-system, and then describe the interaction between the two sub-systems.

4.1 Passport Overview

Figure 4.1 shows how Passport works at a high level. When a packet leaves its source AS, the border router stamps one Message Authentication Code (MAC) per AS on the path into its Passport header. Each MAC is computed using a symmetric key shared between the source AS and the AS on the path. These symmetric keys are established via a Diffie-Hellman key exchange piggybacked into the inter-domain routing protocol BGP.

When the packet enters an AS on the path, the border router verifies the corresponding MAC value using the symmetric key shared with the source AS. A correct MAC can only be produced by the source AS that also knows the key. If the MAC verifies, it is sufficient to show that the packet comes from the source AS indicated by

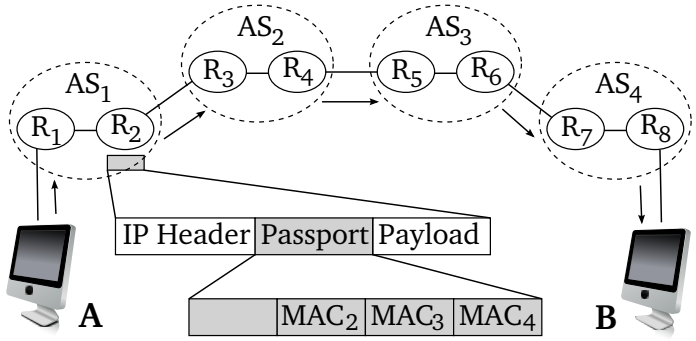


FIGURE 4.1: **Overview of Passport. A border router of a source AS (R_2) stamps source authentication information into the Passport header of an outbound packet. A border router of an intermediate AS or the destination AS (R_3, R_5, or R_7) verifies this information.**

its source address. Otherwise, it is an indication that the source address is spoofed, or there is a temporary routing inconsistency such that the forwarding path perceived by the source AS is different from the real forwarding path. A packet with an invalid MAC is only demoted at an intermediate AS because the invalidity may be due to routing inconsistency, but if the last MAC is invalid, the packet will be discarded at the destination AS. Routers forward demoted traffic in separate queues with limited bandwidth or lower forwarding priority.

4.2 NetFence Overview

On the high level, NetFence resembles a mixture of a capability-based DoS mitigation system (*e.g.*, TVA [103]) and a congestion policing system (*e.g.*, re-ECN [15]). Next we present an overview of the NetFence design.

4.2.1 System Components

NetFence has three types of packets: *request* packets, *regular* packets, and *legacy* packets. The first two, identified by a special protocol number in the IP header, have a shim NetFence header between their IP and upper-layer protocol headers. The NetFence header carries unforgeable congestion policing feedback generated by

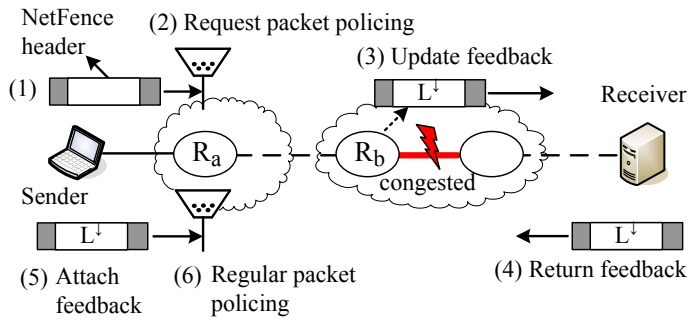


FIGURE 4.2: **Overview of NetFence. Packets carry unspoofable congestion policing feedback stamped by bottleneck routers (R_b in this figure). Access routers (R_a) use the feedback to police senders' traffic, preventing malicious senders from causing severe congestion in the network or gaining unfair shares of bottleneck bandwidth. A destination host can use the congestion policing feedback as capability tokens to suppress any unwanted traffic.**

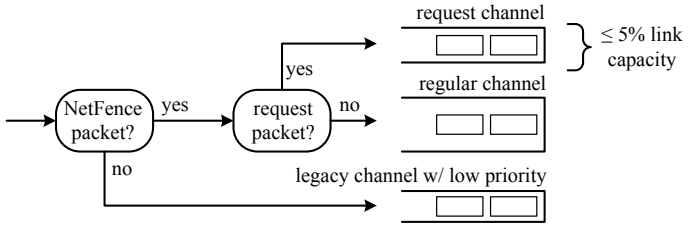


FIGURE 4.3: **Each NetFence router keeps three channels to forward packets.**

the network. A NetFence-ready sender sends request and regular packets, while a non-NetFence sender sends only legacy packets.

Each NetFence router, depicted in Figure 4.3, keeps three channels, one for each of the three packet types discussed above. To motivate end systems to upgrade, the NetFence design gives legacy channel lower forwarding priority than the other two. To prevent request flooding attacks from denying legitimate requests, NetFence has a priority-based backoff mechanism for the request channel. The request channel is also limited to consume no more than a small fraction (5%) of the output link capacity, as in capability-based systems [103, 71].

NetFence places its congestion feedback and traffic policing functions at bottleneck and access routers that are either inside the network or at the trust boundaries

between the network and end systems; it does not place any trusted function at end systems. As shown in Figure 4.2, a NetFence sender starts an end-to-end communication by sending request packets to its NetFence-ready receiver (Step 1). The access router inserts the *nop* feedback (Section 6.1) in the NetFence header of the packet (Step 2). Along the path, a bottleneck router might modify the feedback, in a way similar to TCP ECN [75] (Step 3). After the receiver returns the feedback to the sender (Step 4), the sender can send valid regular packets that contain the feedback (Step 5). In Step 4, two-way protocols like TCP can piggyback the returned feedback in their data packets, while one-way transport protocols such as UDP must send extra, low-rate feedback packets from the receiver back to the sender.

A NetFence router (*e.g.*, R_b in Figure 4.2) periodically examines each outgoing link to decide if an attack is happening at the link. It uses a combination of link load and packet loss rate as the attack indicator (Section 6.3.1). If an attack is detected on a link, a NetFence router starts a monitoring cycle for the link, which lasts until the cycle has existed for an extended period (typically a few hours) after the attack disappears. During a monitoring cycle, a NetFence router stamps the *mon* congestion policing feedback (containing the link ID l, an *action* field, etc.) into the NetFence header of all the passing request and regular packets (Section 6.3.2). The sender's regular packets must include this *mon* feedback to be considered valid, and they will be policed by the access router (Step 6, Section 6.3.3).

An access router maintains one rate limiter for every sender-bottleneck pair to limit a sender's regular packets traversing a bottleneck link. The router uses an Additive Increase and Multiplicative Decrease (AIMD) algorithm to control the rate limit: it keeps the rate limit constant within one predefined control interval (*e.g.*, a few seconds); across control intervals, it either increases the rate limit additively or decreases it multiplicatively, depending on the particular *mon* feedback it receives (Section 6.3.4). We use AIMD to control the rate limit because it has long been

shown to converge onto efficiency and fairness [18]. Other design choices exist; they have different cost-performance tradeoffs, and are discussed in Section 11.2.

When no attack is detected, downstream routers will not modify the *nop* feedback stamped by an access router. When the sender obtains the *nop* feedback and presents it back to the access router in a packet, the packet will not be rate-limited. That is, when no attack happens, NetFence stays in the idle state. The overhead during such idle periods is low: the NetFence header is short (20 bytes, Section 8.2.1), the bottleneck attack detection mechanism only involves a packet counter and a queue sampler, and an access router only needs to validate (not rate limit) and update the NetFence header for each packet. Only when an attack is detected at a bottleneck link, does NetFence activate its policing functions, which adds additional processing overhead at bottleneck and access routers. We show the overhead benchmarking results in Section 9.3.

4.2.2 Unforgeable Congestion Policing Feedback

Congestion policing feedback must be made unforgeable so that malicious nodes cannot evade NetFence's traffic policing functions. NetFence achieves this goal using efficient symmetric key cryptography. An access router inserts a periodically changing secret in a packet's NetFence header. A bottleneck router uses this secret to protect its congestion policing feedback, and then erases the secret. The access router, knowing the secret, can validate the returned feedback. We describe the details of this design in Section 6.4, and discuss how to limit the effect of compromised access routers in Section 6.5.

4.2.3 Congestion Feedback as Capability

If a receiver can identify and desires to bar the attack traffic, NetFence's unspoofable congestion policing feedback also serves as a capability token: the receiver can return

no feedback to a malicious sender. Because the malicious sender cannot forge valid feedback, it cannot send valid regular packets. It can at most flood request packets to the receiver, but an access router will use a priority-based policing scheme to strictly limit a sender's request traffic rate (Section 6.2).

4.2.4 Fair Share Guarantee

With the above-described closed-loop network architecture, we are able to prove (in Appendix A) that NetFence achieves per-sender fairness for single bottleneck scenarios:

***Theorem**: Given G legitimate and B malicious senders sharing a bottleneck link of capacity C, regardless of the attack strategies, any legitimate sender g with sufficient demand eventually obtains a capacity fair share $\frac{\nu_g \rho\ C}{G+B}$, where $0 < \nu_g \leqslant 1$ is a parameter determined by how efficient the sender g's transport protocol (*e.g.*, TCP) utilizes the rate limit allocated to it, and ρ is a parameter close to 1, determined by NetFence's implementation-dependent AIMD and attack detection parameters.*

4.3 Interaction between Passport and NetFence

Passport and NetFence work together to form a complete DoS attack mitigation architecture. NetFence relies on two functionalities provided by Passport:

Authentic source addresses: NetFence makes use of authentic source addresses in two ways. First, as long as a receiver can identity the attack traffic, it can reliably identify the attack sources using the authentic source addresses in attack packets. As a result, it can prevent the identified attack sources from obtaining congestion policing feedback by blacklisting their addresses. Second, a NetFence router identifies the source AS of each packet using the authentic source address, and then performs per-source-AS fair queuing or rate limiting to limit the damage of attack traffic when access routers are compromised and do not police the attack traffic.

Pair-wise symmetric keys between ASes: Passport establishes a symmetric key between any pair of ASes and use them to authenticate source addresses. NetFence also uses these symmetric keys to make congestion policing feedback unforgeable.

It is important to note that Passport and NetFence are only loosely coupled together; they do not necessarily need to be deployed together at the same time. Passport is a generic source authentication system: even without NetFence, it can be independently deployed, and it alone can eliminate reflector attacks and significantly help to mitigate application-level DoS attacks (Section 11.3). NetFence can also work with other source authentication systems; if a source authentication system does not provide the pair-wise symmetric keys between ASes that are needed by NetFence, NetFence itself can do so in the same way as Passport does.

5

Passport: Source Authentication

In this chapter we present the detailed design of Passport, our source authentication sub-system.

5.1 Obtaining Shared Symmetric Keys

Passport uses symmetric key cryptography to authenticate source addresses. This requires each pair of ASes to establish a shared symmetric key prior to validating any packet's source address when forwarding packets. To setup these keys, Passport piggybacks a Diffie-Hellman key exchange [22] into the inter-domain routing system BGP. That is, each AS_i generates a Diffie-Hellman public/private value pair (b_i, r_i), and includes the public value b_i in its prefix announcements. These prefix announcements will reach all other ASes so that they can set up a forwarding entry in their routing tables to reach AS_i. Similarly, AS_i will receive prefix announcements originated from all other ASes. Using the Diffie-Hellman public values included in the prefix announcements, AS_i is able to obtain a shared symmetric key with every other AS_j using a standard Diffie-Hellman construction: $K(i,j) = b_j^{r_i} \; mod \; p = b_i^{r_j} \; mod \; p$, in which p is a system-wide parameter. To reduce Passport's message overhead in

the routing system, if an AS originates multiple address prefixes, it only needs to choose one representative prefix to piggyback its Diffie-Hellman public value.

AS_i may receive a prefix announcement originated by AS_j from multiple neighbors. In this case, AS_i accepts AS_j's public value in the announcement from the next-hop neighbor it chooses to use to reach AS_j. This binds the security of the Diffie-Hellman key exchange to routing security, because the public value of AS_j accepted by AS_i comes from the reverse forwarding path from AS_i to AS_j. If the routing system can eliminate prefix hijacking attacks, *i.e.*, if it can prevent AS_i from forwarding packets through an attacker by rejecting a prefix announcement presented by the attacker, the public value of AS_j accepted by AS_i will not originate from an attacker. AS_i is then able to compute a correct key shared with AS_j, and verify packets from AS_j using that key. Therefore, as long as routing is secure, Passport is secure.

Passport gains additional benefits from distributing the shared symmetric keys within the routing system. First, it can bootstrap the key distribution, which is seemingly a chicken-and-egg problem: keys are needed for packet forwarding, but ASes need to send packets first to negotiate keys. This dependency loop breaks when we distribute keys in the routing system, because the operation of the routing system does not require Passport: as routing packets are exchanged before forwarding state is set up, routing has its own authentication mechanism to validate routing messages without requiring Passport headers, *i.e.*, routers only accept routing messages from known peers. Second, distributing symmetric keys in the routing system is efficient. Each AS only needs to originate one prefix announcement to establish a shared symmetric key with every other AS. Lastly, the key distribution channel can be made highly resilient to DoS flooding attacks, because the key distribution information enjoys the same forwarding priority as routing messages. If routers forward routing messages with the highest priority, Passport's key distribution information is also

forwarded with the highest priority.

5.2 Stamping Passport Headers

Passport uses Message Authentication Codes (MACs), which are efficient to generate and verify, as the inter-domain authentication information. When a border router of an AS receives an outbound packet from an internal interface and can verify that the source address is authentic, it stamps a series of MACs into the packet, each for one AS on the path to the destination. Each MAC is computed using the key the source AS shares with the corresponding AS. The MAC computed for the destination AS covers the source address, the destination address, the IP identifier (IP ID), the packet length, and the first 8 bytes of the packet's payload. For instance, in Figure 4.1, when a packet from host A to host B leaves AS_1, the border router R_2 of AS_1 computes MAC_4 for the destination AS_4 as $MAC_{K(AS_1,AS_4)}(src, dst, len, IPID, payload[0,7])$. The MAC computed for an intermediate AS also includes the previous AS number. For instance, R_2 computes MAC_3 as $MAC_{K(AS_1,AS_3)}(src, dst, len, IPID, payload[0,7], AS_2)$. A router can obtain the AS path information from the inter-domain routing system BGP.

A MAC computation covers the source address to prevent source spoofing. It covers the other fields to detect packets that are sniffed on one path but injected at other network locations. We discuss more on how Passport prevents this type of sniffing and replaying attack in Section 7.1.

A border router only stamps a Passport header for a packet with a valid source address that is within its own address space, and discards the packet otherwise. This step is similar to egress filtering [28], but it is only an optimization. Passport prevents address spoofing even without it. This is because if a router stamps MACs for a source address outside its address space, the MACs will not verify at downstream ASes (as we will see next), wasting an AS's processing power.

5.3 Verifying Passport Headers

When an AS receives a packet from an external interface, it verifies the Passport header using the key it shares with the source AS. The verifying AS uses the source address of the packet to look up the source AS, obtains the shared key, and recomputes the MAC using the shared key and the same packet fields as used by the source AS. An AS can obtain the mapping between a source address and the corresponding source AS from BGP using the `AS_PATH` path attribute.

If the source address is not spoofed, the verifying router is able to locate the correct key, and the re-computed MAC will match the one in the Passport header. This verifies the source AS of the packet. The verifying router erases the MAC value in a packet after the verification to prevent offline cryptanalysis that aims to fake legitimate MACs.

When the MAC does not verify, how the packet is handled depends on the verifying AS. If it is the destination AS, it discards the packet, because the source address must be spoofed. On the other hand, if it is an intermediate AS, it only demotes the packet, because the MAC mismatch may be caused by temporary routing inconsistency instead of address spoofing.

If a packet is demoted, a demotion bit in its Passport header is set, and its IP header is also marked with demotion information to convey the demotion status to legacy ASes (Section 5.6.3). Intermediate ASes forward demoted traffic in a separate queue with limited bandwidth or lower priorities without further verification. A destination AS still verifies the source address of a demoted packet, and discards the packet if the last MAC is incorrect. If the last MAC is valid, the packet is forwarded to its destination host with the demotion mark unchanged. End systems may use these marks to mitigate reflector attacks (Section 7.1.3).

An intermediate AS discards packets received from an external interface that

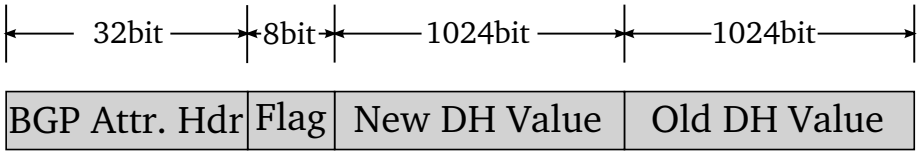


FIGURE 5.1: **Diffie-Hellman key exchange information is encapsulated in a BGP AS path attribute.**

spoof its own addresses. This step precedes Passport header verification, as it does not require verifying a Passport header.

A router in an AS may receive a packet with a Passport header from an internal interface. If the internal interface is a host-to-router interface, *e.g.*, the interface between host A and a router R_1 in Figure 4.1, the router discards the packet, as a host can not generate a valid Passport header. If it is a router-to-router interface, it may assume that the packet has been verified by a border router in its own AS and forward the packet without further verification.

5.4 Re-Keying

An AS may establish new shared keys with other ASes when its old keys have been used for a while, *e.g.*, on the order of a few days or a few weeks, to improve security. To exchange new keys, AS_i originates a prefix update with a new Diffie-Hellman public value, and other ASes will compute new keys shared with the AS using this value.

During the re-keying process, different ASes may use different keys to generate the MACs for AS_i, as the prefix announcements of AS_i will arrive at different ASes asynchronously. To identify different keys, an AS uses an alternating parity bit to mark consecutive Diffie-Hellman public values. When AS_i generates a MAC to be verified by AS_j, it uses the highest-bit in the MAC field to indicate the parity of AS_j's public value, and one bit in the Flags field of a Passport header to indicate the parity of its own public value. These two bits uniquely identify a shared key even

when both ASes re-key simultaneously.

Figure 5.1 shows the key exchange information piggybacked in a BGP prefix announcement. Each prefix announcement carries both the new and old Diffie-Hellman public values in case a remote router crashes when an AS re-keys. The parity of the new value is indicated in the flag field.

5.5 Key Management and Storage

Diffie-Hellman public/private values and the shared symmetric keys are stored together with routing information on routers. If a router reboots and loses those values, it may obtain them from other routers in the same AS, similar to a BGP table transfer after a router reboots.

An AS may configure one router or a route reflector [10] as its key generator. This router will be in charge of generating Diffie-Hellman public/private value pairs and initiating re-keying. Other routers in the same AS can learn the Diffie-Hellman public/private values from the key generator via iBGP.

5.6 Incremental Deployment

One of our major design goals is that our system should be incrementally deployable. Next we present how Passport can be deployed in the presence of various legacy elements in the Internet. We refer to an entity (router or host) that deploys Passport as an *upgraded* entity.

5.6.1 Inter-Operation with Legacy ASes

ASes that adopt Passport may use an optional and transitive path attribute in BGP to distribute their Diffie-Hellman public values as shown in Figure 5.1. Legacy ASes do not process optional and transitive path attributes, but will include them in the routing advertisements they propagate to their neighbors [77]. Therefore, two

upgraded ASes can perform a Diffie-Hellman key exchange even when there are legacy ASes between them.

Passport header is inserted as a shim layer between IP and an upper layer protocol. A source AS stamps MAC values for the upgraded ASes on the path, and legacy ASes do not process the Passport header.

5.6.2 Bump in the Wire

Passport can be deployed as *bump in the wire* without upgrading hosts. When both a source AS and a destination AS have upgraded, the border router at the source AS inserts a Passport header to a packet, and the border router at a destination AS strips off the header.

If a destination AS has not deployed Passport, an upgraded source AS can still use Passport headers so that other upgraded ASes on the path can verify its packets. In this case, the last upgraded AS on the path strips off the Passport header. However, if there is a temporary routing inconsistency, a Passport header may not be stripped off when the packet reaches its destination. In this case, a legacy host in the destination AS will receive a packet with a protocol (Passport) unknown to it and therefore discard the packet.

To solve this problem, Passport uses IP encapsulation. A source AS encapsulates the original IP packet using an outer IP header. The outer IP header uses the same source address as the inner header, and uses the last upgraded AS on the path as the destination address. This address could be a special well-known anycast address of the last upgraded AS. The source AS inserts a Passport header between this outer IP header and the inner IP header. In this encapsulation mode, a MAC in a Passport header covers the source address, both destination addresses in the inner and outer IP headers, the original 8-byte payload, and the IP ID and length fields of the outer header.

When the destination AS in the outer IP header decapsulates a packet, it checks whether the incoming AS is a neighbor to which it announces the destination prefix in the inner header. If not, it discards the packet to prevent a source AS from violating its transit policy.

Modern routers already support encapsulation at line speed because of other needs such as VPNs and IPv4-IPv6 transition. After hosts are upgraded, they can process packets with Passport headers, reducing the need for encapsulation.

5.6.3 Handling Legacy Traffic

In upgraded ASes, legacy and demoted traffic are both unverified traffic, and are treated with the same priority. An upgraded AS may use strict priority queuing to favor verified traffic over unverified traffic, and incentivize legacy ASes to upgrade. Alternatively, it may use two weighted queues to handle verified and unverified traffic, allocating limited bandwidth to unverified traffic. It may set the queue weights according to the ratio of traffic from upgraded ASes and legacy ones to avoid penalizing legacy traffic when there is no spoofing attack.

An upgraded AS will discard or demote legacy traffic if it detects or suspects that the traffic spoofs other upgraded ASes' addresses. The algorithm is as follows:

1. If a legacy packet's source AS and destination AS have both deployed Passport, discard the packet, as it must have a spoofed source address.

2. If a legacy packet's source AS has deployed Passport but the destination AS has not, demote the packet. This is because a source AS will insert a Passport header for a legacy destination if there is any upgraded AS on the path. A legacy packet from an upgraded AS is likely to be spoofed, except during temporary routing inconsistency.

A legacy AS may receive two types of legacy traffic: regular and demoted by other

upgraded ASes. The AS should treat demoted traffic with lower priority, because it is likely to use spoofed source addresses.

Upgraded ASes convey demotion information to legacy ASes via the IP header, such as using the Differentiated Services Code Point (DSCP) [67]. A legacy AS needs to make a configuration change to honor demotion in order to take advantage of Passport. We believe this configuration change can be made at most legacy ASes, because DiffServ is already well supported by commercial routers today, and this change does not require software or hardware upgrade. Besides, if a legacy AS encounters congestion in its network, it has to discard some packets. It's advantageous for the AS to honor a demotion mark and discard packets that are likely to use spoofed source addresses in favor of regular packets.

5.6.4 Additional Issues

Firewall configuration: Since a Passport header is a shim layer between IP and upper-layer protocols, the configuration of some firewalls may need to be updated to allow packets with Passport headers to pass.

MTU discovery: When Passport is deployed in the *bump in the wire* mode, a legacy host may not subtract the Passport header in its MTU discovery process. One solution is for a border router to intercept the ICMP "Fragmentation Needed" message, and subtract the Passport header, a practice used in the deployment of IPv4 to IPv6 [68].

5.7 Adoptability

Passport not only supports incremental deployment, but also provides incentives for ASes to deploy. Once an AS deploys Passport, its traffic traversing at least one deployed, Passport-enabled AS will carry verifiable Passport headers and therefore

enjoy higher forwarding priorities than legacy traffic in the deployed ASes en route. This is true even when the destination AS does not support Passport (Section 5.6.2). In addition, when attackers send traffic to another deployed AS, the attack traffic spoofing this AS's addresses won't be able to reach the destination hosts, as it will be discarded when it enters the deployed destination AS (Section 5.3). As the number of deployed ASes increases, each deployed AS will enjoy a higher benefit as its traffic will have higher forwarding priorities at more locations in the Internet and fewer attackers will be able to spoof its addresses at fewer locations in the Internet, creating a network effect that can further promote the deployment of Passport.

We have compared the adoptability of Passport with several other source address spoofing prevention systems and concluded that Passport has higher adoptability. The details of this comparison can be found in [51].

6

NetFence: DoS Attack Mitigation

In this chapter we present the design details of our DoS mitigation sub-system NetFence. For clarity, we first present the design assuming unforgeable congestion policing feedback and non-compromised routers. We then describe how to make congestion policing feedback unforgeable and how to handle compromised routers. Key notations used to describe the design are summarized in Table 6.1.

6.1 Congestion Policing Feedback

In NetFence, packets carry congestion policing feedback stamped by the network, and the feedback is used by routers to police the packets. There are three types of congestion policing feedback:

- *nop*, indicating no traffic policing action is needed;
- $L^{\downarrow}$, indicating the link L is overloaded, and traffic traversing L should be reduced;
- $L^{\uparrow}$, indicating the link L is underloaded, and more traffic traversing L can be allowed.

Table 6.1: **Key Parameters and Their values in NetFence Implementation**

Name	Value	Meaning
l_1	1 ms	level-1 request packet rate limit
I_{lim}	2 s	Rate limiter control interval length
w	4 s	Feedback expiration time
Δ	12 kbps	Rate limiter additive increase value
δ	0.1	Rate limiter multiplicative decrease ratio
p_{th}	2%	Packet loss rate threshold
Q_{lim}	0.2s $\times$ link capacity	Max queue length
$minthresh$	$0.5\,Q_{lim}$	RED algorithm parameter
$maxthresh$	$0.75\,Q_{lim}$	RED algorithm parameter
w_q	0.1	EWMA weight for average queue length

We refer to $L^{\uparrow}$ and $L^{\downarrow}$ as the *mon* (short for "monitoring") feedback. Each congestion policing feedback includes a timestamp to indicate its freshness.

6.2 Protecting the Request Channel

Without obtaining any congestion policing feedback from the network, attackers may simply flood request packets that legitimately do not carry congestion policing feedback in an attempt to congest downstream links. NetFence mitigates this attack with two mechanisms. First, it limits the request channel on any link to a small fraction (5%) of the link's capacity, as in [103, 71]. This prevents request packets from starving regular packets. Second, it combines packet prioritization and priority-based rate limiting to ensure that a legitimate sender can always successfully transmit a request packet if it waits long enough to send the packet with high priority. This mechanism ensures that a legitimate user can obtain the valid congestion policing feedback needed for sending regular packets. Next we describe this mechanism in detail.

In NetFence, a sender can assign different priority levels to its request packets. Routers, both access routers and non-access routers, forward a level-k packet with a higher priority than lower-level packets, but the sender is limited to send level-k

packets at half of the rate of level-$(k-1)$ packets. An access router installs per-sender and token-based rate limiters to impose this rate limit. That is, when admitting a level-k packet, it removes 2^{k-1} tokens from a request packet rate limiter associated with the sender. Level-0 packets are not rate-limited, but they have the lowest forwarding priority.

This request channel policing algorithm guarantees that at attack time, if a legitimate sender keeps increasing the level of its request packets, it can eventually successfully send a request packet to a receiver regardless of the number of attackers [71]. It holds because the arrival rate of request packets decreases exponentially as their priority level increases. Thus, the arrival rate of high priority request packets (from both attackers and legitimate senders) will eventually be smaller than the request channel capacity, and then they can all be delivered to their destinations.

We note that a sender only needs to send one request packet for each session, not each flow. If the congestion feedback returned from its previous flows have not expired, it can use them to send regular packets for the subsequent flows to the same destination. Thus, during attack times, only a sender's first packet in a session may be delayed due to request channel contention, limiting the damage of request packet flooding attacks. Moreover, a request channel provision of 5% of a link's capacity should be sufficient. It permits about one request packet (88 bytes in our prototype implementation, Section 6.8) per full-sized packet (1500 bytes), while the average end-to-end sessions are longer than one packet.

NetFence does not use computational puzzles as in [71]. This is because computational resources may be scarce [20], especially in busy servers and handheld devices, and distributing the puzzle seed requires using the external DNS infrastructure. In addition, NetFence's design has the flexibility that an access router can configure different token refill rates for different hosts on its subnet. Legitimate servers could be given a higher rate to send more high priority request packets without purchasing

additional CPU power.

When an access router forwards a request packet to the next hop, it stamps the *nop* feedback into the packet, ensuring that even without attacks a receiver can return valid feedback to a sender if it desires to accept the sender's traffic.

6.3 Protecting the Regular Channel

When attack traffic cannot be distinguished from legitimate traffic, *e.g.*, when attackers collude in pairs to send and receive attack traffic among themselves, malicious senders can always obtain valid congestion policing feedback. Then they may flood regular packets in an attempt to congest downstream links. Next we describe how to mitigate this attack.

6.3.1 A Monitoring Cycle

When a router suspects that its outgoing link L is under attack, it starts a monitoring cycle for L. That is, it marks L as in the *mon* state and starts updating the congestion policing feedback in packets that traverse L (Section 6.3.2). Once a sender's access router receives such feedback, it will start rate limiting the sender's regular packets that will traverse the link L (Section 6.3.3).

It is difficult to detect whether L is under an attack, because the attack traffic may be indistinguishable from legitimate traffic. In NetFence, L's router infers an attack based on L's utilization and the loss rate of regular packets. If L is well-provisioned and its normal utilization is low (a common case in practice), it can be considered as under an attack when its average utilization becomes high (*e.g.*, 95%); if L always operates at or near full capacity, its router can infer an attack when the regular packets' average loss rate p exceeds a threshold p_{th}. A link's average utilization and p can be calculated using the standard Exponentially Weighted Moving Average (EWMA) algorithm [29]. The threshold p_{th} is a local policy decision of L's router,

but it should be sufficiently small so that loss-sensitive protocols such as TCP can function well when p is below p_{th}. This is to limit the damage when attackers launch a mild attack and evade the attack detection by keeping p below p_{th}.

When the attack detection is based on the packet loss rate p, a flash crowd may also be considered as an attack. We do not distinguish these two because it is too difficult to do so. As shown by our simulation results (Section 10.2), starting a monitoring cycle for a link does not have much negative impact on a legitimate sender.

It is undesirable to infinitely keep a monitoring cycle due to the added overhead. Thus, a NetFence router terminates a link L's monitoring cycle when L is no longer under attack (*e.g.*, $p < p_{th}$) for a sufficiently long period of time T_b. The router will mark L as in the *nop* state and stop updating the congestion policing feedback in packets traversing L. Similarly, an access router will terminate a rate limiter (src, L) if it has not received any packet with the $L^{\downarrow}$ feedback and the rate limiter has not discarded any packet for T_a seconds.

Routers should set T_a and T_b to be significantly longer (*e.g.*, a few hours) than the time it takes to detect an attack (T_d). This is because attackers may flood the network again after T_a (or T_b) seconds. By increasing the ratio of the monitored period $min(T_a, T_b)$ to the unprotected period T_d, we reduce the network disruption time. Network disruption during an attack detection period cannot be eliminated unless compromised senders are patched up, but we do not assume routers have this ability.

6.3.2 Updating Congestion Policing Feedback

When a link L is in the *mon* state, its router R_b uses the following ordered rules to update the congestion policing feedback in any request or regular packet traversing L:

1. If the packet carries *nop*, stamp $L^{\downarrow}$;

2. Otherwise, if the packet carries $L'^{\downarrow}$ stamped by an upstream link L', do nothing;

3. Otherwise, if L is overloaded, stamp $L^{\downarrow}$;

4. Otherwise, do not update the feedback.

The router R_b never stamps the $L^{\uparrow}$ feedback. As we will see in Section 6.3.3, only an access router stamps $L^{\uparrow}$ when forwarding a packet. If the $L^{\uparrow}$ feedback reaches the receiver of the packet, it indicates that the link L is not overloaded, because otherwise the router R_b would replace the $L^{\uparrow}$ feedback with the $L^{\downarrow}$ feedback.

A packet may cross multiple links in the *mon* state. The access router must ensure that the sender's rate does not exceed its legitimate share at any of these links. The second rule above allows NetFence to achieve this goal, gradually. This is because the first link L_1 on a packet's forwarding path that is both overloaded and in the *mon* state can always stamp the $L_1^{\downarrow}$ feedback, and downstream links will not overwrite it. When the $L_1^{\downarrow}$ feedback is presented to an access router, the router will reduce the sender's rate limit for the link L_1 until L_1 is not overloaded and does not stamp $L_1^{\downarrow}$. This would enable the next link (L_2) on the path that is both in the *mon* state and overloaded to stamp $L_2^{\downarrow}$ into the packets. Gradually, a sender's rate will be limited such that it does not exceed its fair share on any of the on-path links in the *mon* state.

6.3.3 Regular Packet Policing at Access Routers

A sender *src*'s access router polices the sender's regular packets based on the congestion policing feedback in its packets. If a packet carries the *nop* feedback, indicating no downstream links require congestion policing, the packet will not be rate-limited. Otherwise, if it carries $L^{\uparrow}$ or $L^{\downarrow}$, it must pass the rate limiter (src, L).

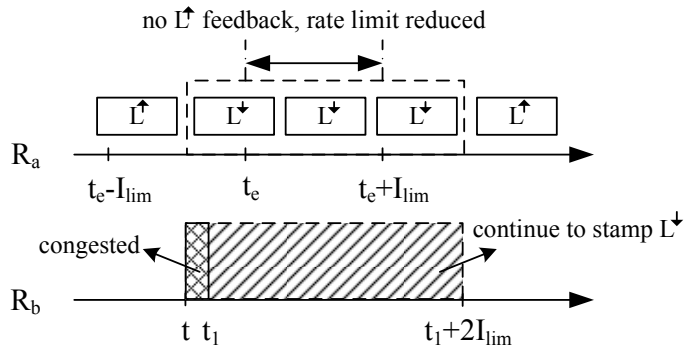


FIGURE 6.1: **Once a router R_b encounters congestion between time $[t, t_1]$, it will continuously stamp the $L^{\downarrow}$ feedback until $t_1 + 2I_{lim}$.**

We implement a rate limiter as a queue whose de-queuing rate is the rate limit, similar to a leaky bucket [94]. We use the queue to absorb traffic bursts so that bursty protocols such as TCP can function well with the rate limiter. We do not use a token bucket because it allows a sender to send short bursts at a speed exceeding its rate limit. Strategic attackers may synchronize their bursts to temporarily congest a link, leading to successful on-off attacks.

When an access router forwards a regular packet to the next hop, it resets the congestion policing feedback. If the old feedback is *nop*, the access router refreshes the timestamp of the feedback. If the old feedback is $L^{\downarrow}$ or $L^{\uparrow}$, the access router resets it to $L^{\uparrow}$. This design reduces the computational overhead at the link L's router, as it does not update a packet's feedback if L is not overloaded.

For simplicity, NetFence uses at most one rate limiter to police a regular packet. One potential problem is that a flow may switch between multiple rate limiters when its bottleneck link changes. We discuss this issue in Section 6.3.5.

6.3.4 Robust Rate Limit Adjustment

The $L^{\uparrow}$ and $L^{\downarrow}$ feedback enables an access router to adjust a rate limiter (src, L)'s rate limit r_{lim} with an AIMD algorithm. A strawman design would decrease r_{lim}

multiplicatively if the link L is overloaded and stamps the $L^{\downarrow}$ feedback, or increase it additively otherwise. However, a malicious sender can manipulate this design by hiding the $L^{\downarrow}$ feedback to prevent its rate limit from decreasing.

To address this problem, we periodically adjust a rate limit, use $L^{\uparrow}$ as a robust signal to increase the rate limit, and ensure that a sender cannot obtain valid $L^{\uparrow}$ feedback for a full control interval if its traffic congests the link L. Let I_{lim} denote the control interval length for rate adjustment on an access router. Suppose a downstream bottleneck router R_b has a link L in the *mon* state. R_b monitors L's congestion status using a load-based algorithm [97] or a loss-based algorithm such as Random Early Detection (RED) [29]. If R_b detects congestion between time t and t_1, it will stamp the $L^{\downarrow}$ feedback into all packets traversing L from time t until two control intervals after t_1: $t_1 + 2I_{lim}$, even if it has considered the link not congested after t_1. This hysteresis ensures that if a sender congests a link L during one control interval, it will only receive the $L^{\downarrow}$ feedback in the following control interval, as shown in Figure 6.1.

For each rate limiter (src, L), the access router R_a keeps two state variables: t_s and $hasIncr$, to track the feedback it has received. The variable t_s records the start time of the rate limiter's current control interval, and $hasIncr$ records whether the rate limiter has seen the $L^{\uparrow}$ feedback with a timestamp newer than t_s. At the end of each control interval, R_a adjusts the rate limiter (src, L)'s rate limit r_{lim} as follows:

1. If $hasIncr$ is true, R_a compares the throughput of the rate limiter with $\frac{1}{2}r_{lim}$. If the former is larger, r_{lim} will be increased by Δ; otherwise, r_{lim} will remain unchanged. Checking the rate limiter's current throughput prevents a malicious sender from inflating its rate limit by sending slowly for a long period of time.

2. Otherwise, R_a will decrease r_{lim} to $(1 - \delta)r_{lim}$.

We discuss how to set the parameters including Δ and δ in Section 6.8.

We now explain why this AIMD algorithm is robust, *i.e.*, a malicious sender cannot gain unfair bandwidth share by hiding the $L^{\downarrow}$ feedback: if a sender has sent a packet when a link L suffers congestion, the sender's rate limit for L will be decreased. Suppose L's router R_b detects congestion and starts stamping the $L^{\downarrow}$ feedback at time t, and let t_e denote the finishing time of an access router's control interval that includes the time t, as shown in Figure 6.1. R_b will stamp the $L^{\downarrow}$ feedback between $[t, t_1 + 2I_{lim}]$. Since $t_e \in [t, t + I_{lim}]$, a sender will only receive the $L^{\downarrow}$ feedback for packets sent during the control interval $[t_e, t_e + I_{lim}]$, because $t_e \geqslant t$ and $t_e + I_{lim} < t_1 + 2I_{lim}$[1]. It can either present the $L^{\downarrow}$ feedback newer than t_e to its access router, or present one older than t_e, or not send a packet. All these actions will cause its rate limit to decrease according to the second rate limit adjustment rule above.

A legitimate sender should always present $L^{\uparrow}$ feedback to its access router as long as the feedback has not expired, even if it has received newer $L^{\downarrow}$ feedback. This design makes a legitimate sender mimic an aggressive sender's strategy and ensures fairness among all senders.

6.3.5 Handling Multiple Bottlenecks

When a flow traverses multiple links in the *mon* state, the flow's access router will instantiate multiple rate limiters for the sender, each corresponding to one of the bottleneck links. For simplicity, the present NetFence design sends a regular packet to only one rate limiter, but it may overly limit a sender's sending rate in some cases. This is because when a sender's packets carry the congestion policing feedback from one of the bottleneck links, all other rate limiters stay idle. The sender's access router will reduce their rate limits, if they are idle for longer than a full control interval,

[1] This inequation also indicates that $2I_{lim}$ is the minimal hysteresis to ensure robustness. If the router R_b stamps the $L^{\downarrow}$ feedback for shorter than $2I_{lim}$ after t_1, an attacker may obtain the $L^{\uparrow}$ feedback in the interval $[t_e, t_e + I_{lim}]$.

as described in Section 6.3.4. Consequently, the idle rate limiters' rate limits may become smaller than a sender's fair share rates at those bottleneck links. When the bottleneck link carried in a sender's packets changes, the sender may obtain less than its fair share bandwidth at the new bottleneck initially, until its rate limit for the new bottleneck converges. If the bottleneck link in the packets changes frequently, it is possible that none of the rate limits converge, giving the sender a throughput smaller than its fair share bandwidth at any of the bottleneck links. In addition, when the bottleneck links' rate limits differ greatly and a sender's packets switch among them frequently, it may be difficult for a transport protocol such as TCP to adjust a flow's sending rate to match the abruptly changing rate limit, further reducing a sender's throughput.

We have considered various solutions to address this problem. One simple solution is to allow a packet to carry all feedback from all the bottleneck links on its path. An access router can then pass the packet through all the on-path rate limiters, each receiving its own feedback and policing the packet independently. This solution requires a longer and variable-length NetFence header. Another one is for an access router to infer the on-path bottleneck links of a packet based on history information and send the packet through all the inferred rate limiters.

We do not include these solutions in the core design for simplicity. The details of these solutions can be found in Appendix B. Simulation results in Section 10.2.2 and Appendix B suggest that NetFence's performance is acceptable. Thus, we consider it a worthy tradeoff to keep the design simple.

6.4 Securing Congestion Policing Feedback

Congestion policing feedback must be unforgeable. Malicious end systems should not be able to forge or tamper with the feedback, and malicious routers should not be able to modify or remove the feedback stamped by other routers. The NetFence

mode	link	action	ts	MAC

FIGURE 6.2: **The key congestion policing feedback fields.**

design uses efficient symmetric key cryptography to achieve these goals.

6.4.1 Feedback Format

A congestion policing feedback consists of five key fields as shown in Figure 6.2: *mode*, *link*, *action*, *ts*, and MAC. When the *mode* field is *nop*, it represents the *nop* feedback. When the *mode* field is *mon*, the *link* field indicates the identifier (an IP address) of the corresponding link L, and the *action* field indicates the detailed feedback: if *action* is *incr* (*decr*), it is the $L^{\uparrow}$ ($L^{\downarrow}$) feedback. The *ts* field records a timestamp, and the MAC field holds a MAC signature that attests to the feedback's integrity.

In addition to the five key fields, a *mon* feedback also includes a field $token_{nop}$. We explain the use of this field later in this section.

6.4.2 Stamping nop Feedback

When an access router stamps the *nop* feedback, it sets the *mode* field to *nop*, *link* to a null identifier $link_{null}$, *action* to *incr*, *ts* to its local time, and uses a time-varying secret key K_a known only to itself to compute the MAC field:

$$token_{nop} = MAC_{K_a}(src, dst, ts, link_{null}, nop) \tag{6.1}$$

The MAC computation covers both the source and destination addresses to prevent an attacker from re-using valid *nop* feedback for a different connection.

6.4.3 Stamping $L^{\uparrow}$ Feedback

When an access router stamps the $L^{\uparrow}$ feedback, the old feedback in the packet must be either $L^{\uparrow}$ or $L^{\downarrow}$ (Section 6.3.3). Therefore, the *mode* field is already *mon*, and the

link field already contains the link identifier L. The router sets *action* to *incr* and *ts* to its local time, and computes the MAC field using the secret key K_a:

$$token_{L\uparrow} = MAC_{K_a}(src, dst, ts, L, mon, incr) \tag{6.2}$$

The router also inserts a $token_{nop}$ as computed in Eq (6.1) into the $token_{nop}$ field.

6.4.4 Stamping $L^{\downarrow}$ Feedback

When a link L's router R_b stamps the $L^{\downarrow}$ feedback, it sets *mode* to *mon*, *link* to L, *action* to *decr*, and computes a new MAC value using a symmetric key K_{ai} shared between its AS and the sender's AS:

$$token_{L\downarrow} = MAC_{K_{ai}}(src, dst, ts, L, mon, decr, token_{nop}) \tag{6.3}$$

The shared symmetric key K_{ai} is established using Passport. The router R_b includes $token_{nop}$ stamped by the sender's access router in its MAC computation, and erases it afterwards to prevent malicious downstream routers from overwriting its feedback. If R_b is an AS internal router that does not speak BGP, it may not know K_{ai}. In this case, R_b can leave the MAC and $token_{nop}$ fields unmodified and let an egress border router of the AS update their values when the packet exits the AS. This design reduces the management overhead to distribute K_{ai} to an internal router R_b.

NetFence recognizes the special encapsulated packets generated by Passport to handle legacy ASes and hosts (Section 5.6). For such an encapsulated packet, the destination address *dst* used in Eq (6.3) is the destination address in the inner IP header, not the one in the outer IP header.

6.4.5 Validating Feedback

When a source access router receives a regular packet, it first validates the packet's congestion policing feedback. If the feedback is invalid, the packet will be treated as

a request packet and subject to per-sender request packet policing.

A feedback is considered invalid if its ts field is more than w seconds older than the access router's local time t_{now}: $|t_{now} - ts| > w$, or if the MAC field has an invalid signature. The MAC field is validated using Eq (6.1) and Eq (6.2) for the *nop* and $L^{\uparrow}$ feedback, respectively. To validate $L^{\downarrow}$ feedback, the access router first re-computes the $token_{nop}$ using Eq (6.1), and then re-computes the MAC using Eq (6.3). The second step requires the access router to identify the link L's AS in order to determine the shared symmetric key K_{ai}. We can use an IP-to-AS mapping tool (*e.g.*, [60]) for this purpose, as the feedback includes the link L's IP address.

6.5 Localizing Damage of Compromised Routers

The NetFence design places enforcement functions that include feedback validation and traffic policing at the edge of the network to be scalable. However, if an access router is compromised, attackers in its subnet or itself may misbehave to congest the network. NetFence addresses this problem by localizing the damage to the compromised AS. If an AS has a compromised router, we consider the AS as compromised, and do not aim to provide guaranteed network access for the AS's legitimate traffic.

A NetFence router can take several approaches to localize the damage of compromised ASes, if its congestion persists after it has started a monitoring cycle, a signal of malfunctioning access routers. One approach is to separate each source AS's traffic into different queues. This requires per-AS queuing. We think the overhead is affordable because the present Internet has only less than 40K ASes in 2010 [11]. We may replace per-AS queuing with per-AS rate limiting and set the rate limits by periodically computing each AS's max-min fair share bandwidth on the congested link as in [57]. Another more scalable approach is to use a heavy-hitter detection algorithm such as RED-PD [58] to detect and throttle high-rate source ASes. A heavy-hitter detection algorithm is suitable in this case because legitimate source

ASes will continuously reduce their senders' traffic as long as they receive the $L^{\downarrow}$ feedback. The detected high-rate ASes are likely to be the compromised ASes that do not slow down their senders.

All these approaches require a router to correctly identify the source AS of a packet. Since Passport has made each packet's source address authentic, we can simply use an IP-to-AS mapping tool to achieve this goal.

6.6 Incremental Deployment

NetFence can be incrementally deployed by end systems and routers. Since the NetFence header is a shim layer between IP and upper layer protocols, legacy applications need not be modified as long as both senders and receivers deploy NetFence to insert and strip off NetFence headers. Legacy routers can ignore the NetFence header and forward packets using the IP header. Routers at congested links and access routers need to be upgraded, but well-provisioned routers that can withstand tens of Gbps attack traffic may not need to upgrade. The deployment can take a bump-in-the-wire approach, by placing inline boxes that implement NetFence's enforcement functions in front of the routers that require upgrading.

Similar to the deployment of Passport, middleboxes such as firewalls need to be configured to permit NetFence traffic. Different from Passport, NetFence requires both the sender and the receiver of a session to deploy in order to provide the protection against DoS attacks for the session.

6.7 Adoptability

NetFence provides deployment incentives to both end systems and ASes. Client systems and their ASes are willing to deploy because their traffic will enjoy higher forwarding priorities in other deployed, NetFence-enabled ASes (Figure 4.3). Server systems and their ASes are willing to deploy because they get protection against DoS

flooding attacks in deployed ASes. Transit ASes are willing to deploy because deployed traffic sources and destinations will prefer their traffic to go through NetFence-enabled ASes in order to get protection against DoS flooding attacks; in addition, a deployed transit AS will have less DoS flooding traffic that wastes its bandwidth because deployed servers will be able to stop some attack traffic at the sources (by not returning congestion policing feedback, as discussed in Section 4.2.3 and Section 6.2). There is also a snow-ball effect: as more end systems and ASes deploy NetFence, each end system or AS, either deployed or legacy, will get higher benefit against DoS flooding attacks from the deployment, thus promoting more legacy systems to deploy NetFence.

6.8 Parameter Settings

Table 6.1 summarizes the main parameters in the NetFence design and their values used in our prototype implementation. The level-1 request packets (l_1) are rate limited at one per 1 ms. The size of a typical request packet is about 88 bytes that includes a 40-byte TCP/IP header and a 48-byte integrated Passport and NetFence header (Figure 8.5, assuming the AS path length is four). We set the control interval I_{lim} to 2 seconds, one order of magnitude larger than a typical RTT on the Internet ($<$ 200ms). This allows an end-to-end congestion control mechanism such as TCP to reach a sender's rate limit during one control interval. We do not further increase I_{lim} because a large control interval would slow down the rate limit convergence.

The rate limit AI parameter Δ can be neither too small nor too large: a small Δ would lead to slow convergence to fairness; a large Δ may result in significant overshoot. We set Δ to 12Kbps because it works well for our targeted fair share rate range: 50Kbps $\sim$ 400Kbps. A legitimate sender may abort a connection if its sending rate is much lower than 50Kbps, and 400Kbps should provide reasonable performance for a legitimate sender during DoS attacks. The rate limit MD parameter δ is set to

0.1, a value much smaller than TCP's MD parameter 0.5. This is because a router may stamp the $L^{\downarrow}$ feedback for two control intervals longer than the congestion period (Section 6.3.4). If we set δ too large, a sender's rate limit may decrease too fast.

We set the attack detection threshold p_{th} to 2%, since at this packet loss rate, a TCP flow with 200ms RTT and 1500B packets can obtain about 500Kbps throughput [61]. We set a link's maximum queue length Q_{lim} to 200ms $\times$ the link's capability. We use a loss-based algorithm RED to detect a link's congestion status.

7

Security Analysis

In this chapter, we analyze the security properties of our design. We present the possible attack strategies and how our design defends against these attacks.

7.1 Passport

7.1.1 Attacker Types

To assist analyzing the Passport design's security properties, we define three types of attackers, each having different capabilities:

- *Compromised Host*: This attacker can inject packets into the network with arbitrary source addresses, but cannot eavesdrop the traffic sent by legitimate sources. It is referred to as a *host attacker*.

- *Compromised Monitor and Host*: This attacker can eavesdrop traffic (sent by legitimate sources) at its network location and replay that traffic from other, compromised host locations in the network. It is referred to as a *monitor attacker*.

- *Compromised Router and Host*: This attacker can eavesdrop traffic (sent by legitimate sources) at its location, and alter or replay that traffic at its location as well as replay it from other, compromised host locations in the network. It is referred to as a *router attacker*. Note that a router attacker is a strong adversary that can cause greater harm than source authentication failures, *e.g.*, directly dropping packets or tampering with packet content.

7.1.2 Source Address Spoofing Attacks

Next we discuss how different types of attackers may attempt to spoof source addresses.

Host attacker: To spoof source addresses, a host attacker may try to break the MAC that is the heart of Passport. But our design uses a standard MAC scheme with 128-bit keys, which is computationally infeasible to break. The attacker might instead try to guess a valid Passport header by sending packets. Since a Passport header has at least a 63-bit destination MAC value (modulo the parity bit of a Diffie-Hellman public value), an attacker would expect to send at least 2^{62} packets to guess one correct, but the time it takes to send those packets would exceed the period for which the crafted Passport header is valid. This is because the symmetric keys distributed via BGP will have advanced in the interim. This attack would also signal a clear anomaly via a large number of invalid Passport headers.

An attacker may try to guess an intermediate 31-bit MAC value by sending packets with small TTLs, and use the IP header and Passport header echoed back in ICMP "TTL Exceeded in Transit" messages to observe whether a guess is demoted. But this again is infeasible because ICMP messages are always rate limited and it would take a long time for the attacker to try enough MAC values.

Monitor attacker: An eavesdropper may observe valid Passport headers but can-

not freely transfer them to another path because a Passport header is bound to one AS path, *i.e.*, its MACs will be found invalid if sent via another AS path. Thus, packets transferred to other paths will be demoted on their paths to destinations, and can not compete for bandwidth with packets with valid MACs.

Router attacker: Packets duplicated by routers on the forwarding path of a source may reach a destination without being demoted. However, in this case, routing is compromised, and Passport's security is bound to routing security. A source host whose packets are duplicated by a router attacker on its forwarding path should choose a different path.

If an attacker compromises an AS, it may use the AS's keys at other, unprotected locations in the network to forge a Passport header that spoofs the AS's addresses. But this only implicates the compromised AS, not other parts of the network. Similarly, a compromised router can spoof packets from other addresses in its AS, but this again implicates the AS and may adversely affect its traffic.

Importantly, even if an AS is compromised, it cannot forge a Passport header that appear to be from another AS. This is because it does not have the secret keys of the other AS. Even colluding ASes can only forge a valid Passport header that shows a packet comes from themselves. They cannot forge a Passport header as if the packet were from an AS outside the colluding set.

We also note that although two ASes share a symmetric key, they can not use this key to spoof each other's addresses at other ASes, because other ASes use separate keys shared with these two ASes respectively to validate their addresses.

The security of Passport is bound to routing security. We rely on routing to distribute the correct public Diffie-Hellman public values across ASes. By the Diffie-Hellman construction, we do not rely on routers to keep the public values secret from attackers, since it is computationally infeasible to find the long-term pair-wise

keys given only the public values. However, if an attacker can successfully hijack a prefix announcement and replace the Diffie-Hellman public value, it can both receive packets for the specific prefix, and send packets as if they were originated from that prefix.

7.1.3 Reflector Attack with Replayed Packets

A monitor attacker may attempt to launch reflector attacks by sniffing packets sent by a victim, and injecting duplicate copies of them from other network locations. Those replayed packets will be demoted because each Passport header is bound to an AS path. An upgraded, Passport-enabled host that understands the demotion mark in an IP header should echo back the demotion mark in its reply packets to avoid becoming a reflector.

Unfortunately, if a host is not upgraded to echo back a demotion mark, it may respond to replayed packets without demoting the reply packets, thereby becoming a reflector. Passport alleviates this problem by including the destination address, IP ID, IP length field of an IP header, and 8 bytes of payload in the MAC computation. The 8-byte payload covers the TCP sequence number and UDP checksum. A replayed packet must have the same source address, destination address, IP ID, IP length, and TCP sequence number or UDP checksum as the sniffed packet. These fields can help end systems detect and discard replayed packets. For instance, an upper-layer application or a transport protocol such as TCP may detect a duplicate packet, or a sequence number or checksum mismatch, and then discard the packet.

We believe that in practice, these mechanisms can effectively mitigate reflector attacks launched using replayed packets even if a host is not upgraded to echo back a demotion mark. However, if this type of attack becomes a serious concern, Passport can be extended to detect and discard packets with duplicate Passport headers in the network at a higher cost, using a combination of sequence numbers and bloom

filters as described in an early version of the Passport design [54].

In the case of a router attacker, packets can be duplicated without being demoted. As described above, an AS should avoid forwarding packets via a router attacker.

7.1.4 Security with Partial Deployment

In the incremental deployment phase, Passport prevents the addresses of upgraded ASes from being spoofed at other upgraded destination ASes by host attackers. If a destination AS is not upgraded, then as long as there is an upgraded AS on the path, packets that spoof the upgraded ASes' addresses will be demoted.

Similarly, a packet sniffed on one path but replayed on another path by a monitor attacker or a router attacker will be demoted as long as the replayed path differs from the sniffed path by one link whose end node is an upgraded AS.

7.2 NetFence

Next we summarize how NetFence withstands various attacks from different types of attackers.

7.2.1 Malicious End Systems

Forgery or tampering: Malicious end systems may attempt to forge valid congestion policing feedback. But NetFence protects congestion policing feedback with MAC signatures. As long as the underlying MAC is secure, malicious end systems cannot spoof valid feedback. A malicious sender may selectively present $L^{\uparrow}$ feedback to (and therefore hide $L^{\downarrow}$ feedback from) its access router, but NetFence's robust AIMD algorithm (Section 6.3.4) prevents it from gaining a higher rate limit.

Evading attack detection: Malicious end systems may attempt to prevent a congested router from starting a monitoring cycle. This attack is ineffective when a well-provisioned router uses high link utilization to detect attacks. When an under-

provisioned router uses the packet loss rate to detect attacks, NetFence limits the damage of this attack with a low loss detection threshold p_{th} (Section 6.3.1).

On-off attacks: Attackers may attempt to launch on-off attacks. In a macroscopic on-off attack, attackers may flood the network again after a congested router terminates a monitoring cycle. NetFence uses a prolonged monitor cycle (Section 6.3.1) to mitigate this attack. In a microscopic on-off attack, attackers may send traffic bursts with a short on-off cycle, attempting to congest the network with synchronized bursts, yet maintaining average sending rates lower than their rate limits. Our theoretical bound in Section 4.2.4 and simulation results in Section 10.2.2 both show that the shape of attack traffic cannot reduce a legitimate user's guaranteed bandwidth share, because a sender cannot send faster than its rate limit at any time (Section 6.3.3), and NetFence's robust rate limit adjustment algorithm (Section 6.3.4) prevents a sender from suddenly increasing its actual sending rate.

7.2.2 Malicious On-Path Routers

A malicious router downstream to a congested link may attempt to remove or modify the $L^{\downarrow}$ feedback stamped by a congested router in order to hide upstream congestion. But such attempts will make the feedback invalid, because the malicious router does not know the original $token_{nop}$ value needed to compute a valid MAC (Section 6.4).

A malicious on-path router may discard packets to completely disrupt end-to-end communications, duplicate packets, or increase packet sizes to congest downstream links. It may also change the request packet priority field in a NetFence header to congest the request channel on downstream links. Preventing such attacks requires Byzantine tolerant routing [73], which is not NetFence's design goal. Instead, we aim to make these attacks detectable. As presented in Section 7.1.3, the underlying source authentication sub-system Passport partially protects the integrity of a packet and enables duplicate detection: it includes a packet's length and the first 8 bytes

of a packet's transport payload (which includes the TCP/UDP checksum) in its MAC computation. We can further extend Passport's MAC computation to include NetFence's request packet priority field to protect it.

8

Implementation

We have implemented the Passport sub-system both in Linux and on a NetFPGA board [64], and the NetFence sub-system in Linux. We have also implemented a complete prototype system in Linux that integrates both Passport and NetFence. In this chapter, we present the details of these prototype implementations.

8.1 Passport

8.1.1 Header Format

Figure 8.1 shows a Passport header format used in our Passport-only prototype implementations. A destination MAC is 64-bit long. To save header space, each intermediate MAC is 32-bit long if there are more than one intermediate hop, and 64-bit long otherwise. As we will show soon, the nonce field is mainly used by specific MAC algorithms. It can also assist mitigating replay attacks if needed, as presented in an early version of the Passport design [54].

With this header format, the size of a Passport header depends on the number of ASes on the packet's forwarding path towards the destination. Assuming an average AS path length of four [1], the average header overhead for Passport is 24 bytes.

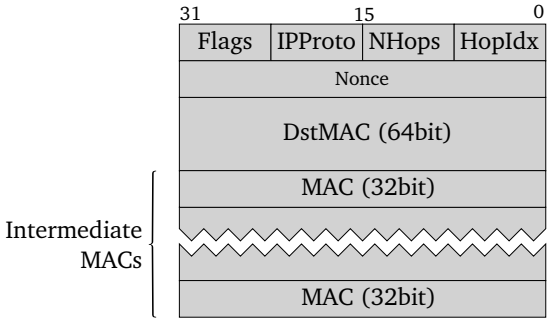


FIGURE 8.1: **Passport header format.**

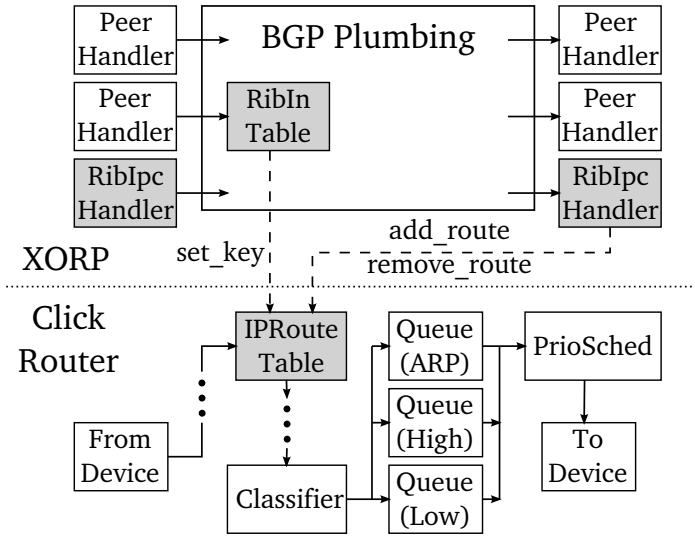


FIGURE 8.2: **Linux implementation of Passport using Click and XORP. The shaded boxes are the main modules we modify.**

8.1.2 Linux Implementation

We have implemented a prototype of Passport using Click [41] and XORP [35] in Linux. Figure 8.2 shows the structure of the prototype implementation. The shaded boxes are modules we modify. We modify XORP to piggyback the Diffie-Hellman key exchange protocol in BGP, and modify components in Click to support Passport header stamping and verification. We also modify XORP to communicate with kernel-space Click using the `/click` file system.

We add an optional and transitive AS path attribute DH_KEY to XORP's BGP

modules. This attribute encapsulates Diffie-Hellman public values as described in Figure 5.1. It is inserted into the BGP prefix announcements in the module `RibIpcHandler`, and later extracted from the prefix announcements in the module `RibInTable`. `RibInTable` is also modified such that whenever new Diffie-Hellman public values are received, the corresponding shared symmetric key is generated and sent to Click using the `set_key` interface in Figure 8.2. We also modify `RibIpcHandler` to update Click's routing table with Passport related information using the `add_route` and `remove_route` interfaces in Figure 8.2. The Passport related information includes AS paths and prefix-to-origin-AS mappings. In our current implementation, a new Diffie-Hellman public-private value pair is generated at XORP's startup time. We have not implemented periodic re-keying and private key distribution via iBGP.

We modify the `IPRouteTable` element in Click such that it receives shared symmetric keys from XORP and calls `generate_ppt()` or `verify_ppt()` in its `push()` method to stamp or verify Passport headers. We use Click's priority scheduler to handle normal and demoted traffic as shown in Figure 8.2. We use priority queuing instead of weighted fair queuing to emphasize the benefit of deploying Passport. The `ARP` queue ensures that link-local ARP packets have the highest forwarding priority.

Our Linux implementation of Passport uses UMAC because of its superior speed in a single-threaded environment. UMAC takes a nonce as input; therefore, we stamp a random number into the 32-bit nonce field of a Passport header and use it together with the 16-bit IP ID to generate a 48-bit nonce for UMAC computation. In the worst case when IP ID field never changes, the nonce space is only 2^{32}, and therefore a nonce may be reused before an AS re-keys. For this reason, we do not use the standardized UMAC as presented in [43], because it is vulnerable to even only a single nonce reuse. Instead, we use the original UMAC construction as described in [12] combined with UHASH [43] and AES. This construct is provably robust to occasional nonce reuse.

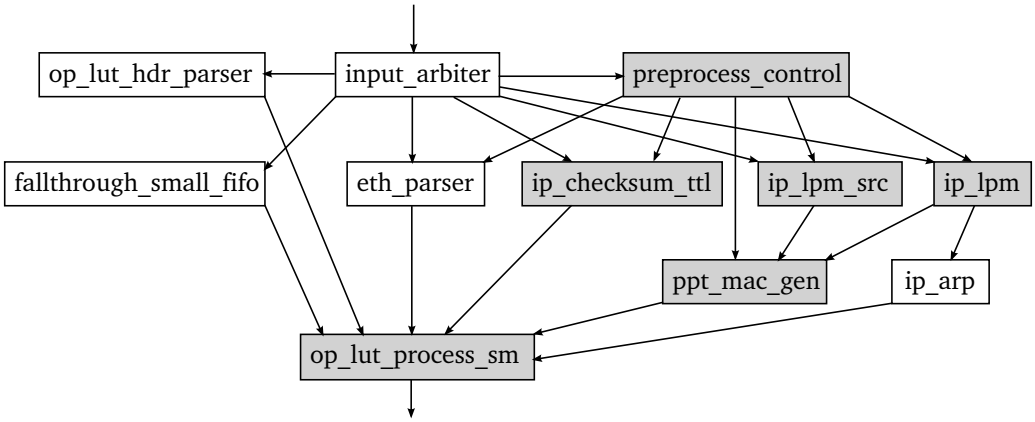


FIGURE 8.3: **The structure of the `output_port_lookup` module of our NetFPGA implementation of Passport. All the Passport related operations are performed in this module. The shaded boxes are the main sub-modules we add or modify.**

8.1.3 FPGA Implementation

As we will show in Section 9.1, although our Linux implementation of Passport shows a promising performance, its computational overhead at packet forwarding time is still too high for high-speed core routers. To further validate that the Passport design is indeed suitable for high-speed packet forwarding, we also implement the packet forwarding logic of Passport on a NetFPGA board.

Our NetFPGA implementation of Passport is based on the reference router design included in NetFPGA Package 2.1.2. The core module of the reference router design is named `output_port_lookup`: given a packet, it determines the output interface and the source and destination MAC addresses, decreases the TTL, and then sends the packet to the corresponding output queue. We modify this module to perform all the Passport related operations. The structure of the updated `output_port_lookup` module is shown in Figure 8.3. The shaded boxes represent the main sub-modules we add or modify. We add the `ip_lpm_src` sub-module to lookup the source AS and the symmetric key shared with the source AS. We also add the `ppt_mac_gen` module to either compute all the MACs to be inserted into the Passport header

when the router is an egress border router of the source AS, or verify a MAC in the existing Passport header when the router is an ingress border router of a non-source AS. We update the `preprocess_control` sub-module to extract the additional information needed by the Passport operations from a packet, *e.g.*, the source address and the MAC to verify. We also update the `ip_lpm` sub-module to get the AS path towards the destination address and the symmetric keys shared with the ASes on the path, and update the `ip_checksum_ttl` sub-module to pre-compute the new IP checksum as if the router were to insert a Passport header. Finally, we update the `op_lut_process_sm` sub-module to take all the pre-computed information and either insert a new Passport header if the router is the source AS's egress border router, or update the existing Passport header according the verification result generated by the `ppt_mac_gen` sub-module if the router is an ingress router of a non-source AS.

To keep our prototype implementation simple, we do not use UMAC as in the Linux implementation; instead, we use AES-CBC-128 as our MAC function. As the total length of the fields that need to be protected by the MACs is longer than 128 bits but shorter than 256 bits, computing a MAC requires two AES-128 encryption operations. The AES-128 module in our implementation is obtained from ICING [82].

8.2 NetFence

8.2.1 Header Format

Figure 8.4 shows the format of a NetFence header in our NetFence-only prototype implementation. It does not contain Passport related fields, because NetFence itself does not make use of those fields.

A full NetFence header from a sender to a receiver includes a forward header and a return header. The forward header includes the congestion policing feedback on the forwarding path from the sender to the receiver, and the return header includes the reverse path information from the receiver to the sender. Most fields are self-

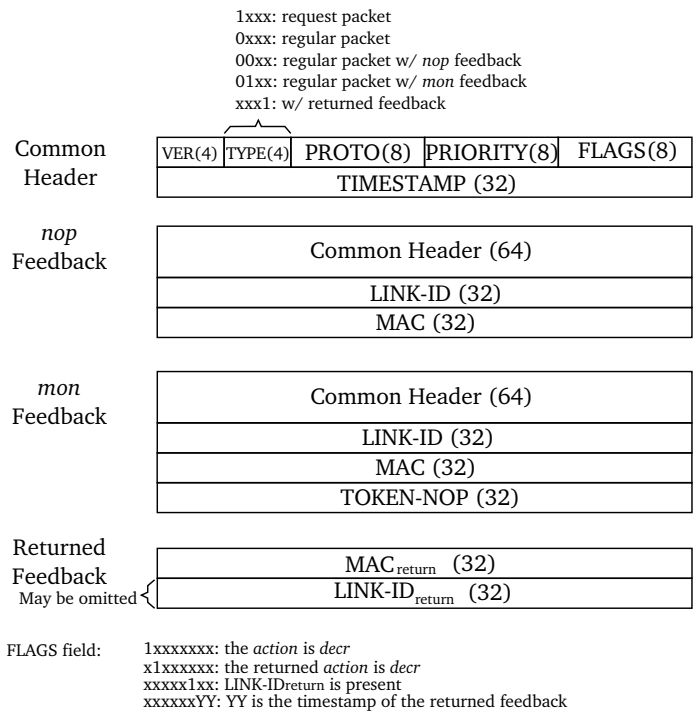


FIGURE 8.4: **NetFence header format.**

explained. A NetFence header is implemented as a shim layer between IP and an upper-layer protocol, and the PROTO field describes the upper-layer protocol (*e.g.*, TCP or UDP). The unit of a timestamp is one second.

The return header may be omitted if a sender has previously returned the latest feedback to the receiver. Even when the return header is present, it does not always include all the fields. If the returned feedback is *nop*, the LINK-ID$_{return}$ field will be omitted because it is zero, and one bit in the FLAGS field will indicate this omission.

To further save header space, a NetFence header only includes the last two bits of the returned timestamp. In the subsequent packets from the sender to the receiver, the sender's access router will reconstruct the full timestamp from its local time and the returned two bits, assuming that the timestamp is less than 4 seconds older than its current time.

With this implementation, a regular packet's NetFence header overhead is 20 bytes in the common case that the congestion policing feedback is *nop* in both the

forward and return paths. In the worst case that both paths contain the *mon* feedback, the header overhead is 28 bytes.

8.2.2 Linux Implementation

Similar to Passport, we have also implemented a prototype of NetFence on Linux using Click. This prototype implementation depends on the Passport's XORP implementation to establish the symmetric keys shared between ASes; it communicates with XORP via the `/click` file system, similar to Passport's Click implementation.

The structure of a NetFence router implemented using Click is similar to that of Passport as shown in Figure 8.2. We modify the element `IPRouteTable` to receive pairwise symmetric keys and AS path information from XORP, and let its `push()` function call a new function `rate_limit_pkt()` to process NetFence headers and rate limit both initial and regular packets. We also implement a new element `NetFenceQueue` to replace `Queue (High)` and `Queue (Low)` shown in Figure 8.2. `NetFenceQueue` implements the three channels shown in Figure 4.3, and uses per-source-AS queuing to localize the damage of compromised ASes.

A NetFence-enabled end host must be able to handle NetFence headers: it has to return the congestion policing feedback to legitimate senders, and it also has to attach the feedback it obtains from these senders to its own packets. To avoid modifying upper-layer applications, we also run Click routers on end hosts and implement a new module `CongestionAgent` to handle NetFence headers.

Our implementation uses AES-128 as the MAC function because of its superior speed when handling short messages. AES is amenable for hardware implementation: commercial hardware already supports high speed AES operations [37].

TYPE1 field:

1xxx: demoted
x1xx: is intra-domain auth
xxx1: use Diffie-Hellman value 1

TYPE2 field:

1xxxxxxx: request packet
0xxxxxxx: regular packet
00xxxxxx: regular pkt w/ *nop* feedback
01xxxxxx: regular pkt w/ *mon* feedback
xx1xxxxx: w/ returned feedback
xxx1xxxx: the *action* is *decr*
xxxx1xxx: the returned *action* is *decr*
xxxxx1xx: LINK-IDreturn is present
xxxxxxYY: the timestamp of the returned feedback

VER(4)	TYPE1(4)	HN(4)	HI(4)	PROTO(8)	TYPE2(8)
NONCE (32)					
DSTMAC (64)					
INTMAC (32)					
...					
INTMAC (32)					
PRIORITY(8)	TIMESTAMP (24)				
LINK-ID (32)					
MAC (32)					
TOKEN-NOP (32)					
MAC_{return} (32)					
$LINK\text{-}ID_{return}$ (32)					

FIGURE 8.5: **A integrated header that includes both Passport and NetFence fields.**

8.3 Integrating Passport and NetFence

Passport and Netfence are two loosely coupled sub-systems that have a simple and clear interface between them. A packet can have separate Passport and NetFence headers; they work seamlessly together. However, to further save header space, our system may also support a single integrated header that covers both Passport and NetFence fields, as shown in Figure 8.5. This header format allows us to save 4 byte header overhead for each packet compared to having separate Passport and NetFence headers.

In a partial deployment scenario where some hosts and ASes have deployed both Passport and NetFence, some have only deployed Passport and some have deployed neither, the legacy channel in the NetFence design needs to be further split into two sub-channels: one for packets with valid Passport headers but without NetFence headers, and one for legacy packets. The former sub-channel should have a higher forwarding priority than the latter sub-channel, because packets with authentic source addresses are easier to policy when they are considered harmful.

9

Performance Evaluation

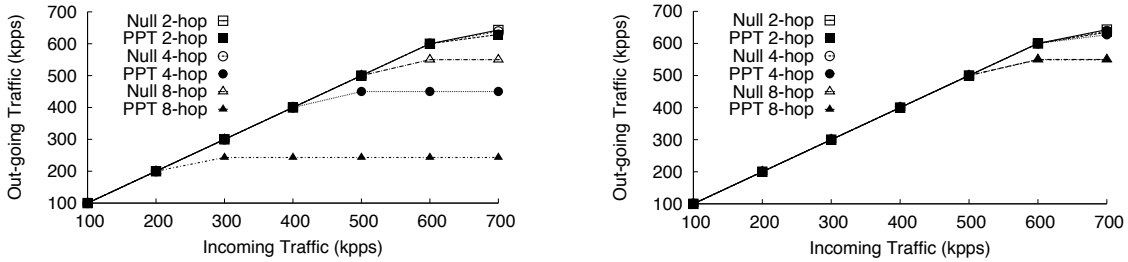
In this section, we use our prototype implementations to evaluate the overhead introduced by our design.

9.1 Linux Implementation of Passport

9.1.1 Header Processing Overhead

We use three PCs in our laboratory to measure the computational overhead of Passport header stamping and verification in our Linux prototype implementation. One PC is used as a router, connecting a packet generator PC and a sink PC. The router has an AMD Opteron 285 Dual Core 2.6GHz CPU, 2GB memory, and an Intel PRO/1000 GT Quad Port Server Adapter. The packet generator and sink are equipped with Pentium-D 3.4GHz CPU, 2GB memory and Intel PRO/1000 PT Server Adapter. We measure both the throughput and CPU cycles of Passport stamping and verification.

To measure the throughput of Passport header stamping, we let the packet generator send minimum sized packets (40 bytes TCP/IP headers) at various input rates. Our experiments assume legacy hosts, and the router PC inserts Passport headers



(a) Stamping (b) Verification

FIGURE 9.1: **The throughput of Passport header stamping and verification for various AS hops with minimum sized packets (each with a 40-byte TCP/IP header plus a Passport header). The Click null forwarding throughput for packets with the same sizes are also shown for comparison.**

	Operation	Time		
		2-hop	4-hop	8-hop
Per Packet	Passport Stamping	655 ns	1493 ns	3190 ns
	Passport Verification	578 ns	618 ns	631 ns
Re-key	DH value pair (1024-bit)	5.64 ms		
	Symmetric key (128-bit)	5.64 ms		

FIGURE 9.2: **Micro-benchmark result of various Passport operations. Time is converted from CPU cycles.**

	Security	Sig. Size	Signing	Verification
RSA-512	60-bit	64 bytes	512 us	40 us
RSA-1024	72-bit	128 bytes	2214 us	102 us
DSA-512	65-bit	40 bytes	368 us	443 us
ECDSA-160	78-bit	40 bytes	300 us	1400 us

FIGURE 9.3: **Equivalent MAC security level, signature size and computational costs of well-known public key schemes.**

into the packets. The sink measures the output rate. We vary the number of AS hops for each experiment, because a router needs to stamp a MAC for each AS on the path. Note that N AS hops implies $N - 1$ MAC computations. Ten million packets are sent for each experiment.

Similarly, to measure the throughput of Passport header verification, we let the

packet generator send minimum sized packets with preset Passport headers at various rates with various AS hops, use the router PC to verify Passport headers, and measure the output rates at the sink.

Figure 9.1 shows the Passport header stamping and verification throughput, together with Click null forwarding throughput for packets of the same sizes. The Passport header verification throughput matches well with Click's null forwarding throughput, as it only involves one MAC computation. The verification throughput varies from 636 kpps to 549 kpps for Passport headers of two to eight AS hops. The slight decrease is primarily due to the increase of packet length, not by the MAC computation.

The Passport header stamping throughput decreases as the AS path length increases. For Passport headers with two to eight AS hops, the throughput varies from 628 kpps to 243 kpps. If we assume that the average packet size is 400 bytes [1], the PC router can stamp Passport headers for average sized packets with two to eight AS hops at 2 Gbps to 0.9 Gbps. As the mean of the AS path length is between 3 and 4 [56], we expect that the stamping throughput for real Internet traffic exceeds 0.9 Gbps. We also note that an AS only needs to stamp Passport headers for traffic originated within its network, and not for transit traffic. This result is promising for a software implementation and suggests that Passport may be practical with the present hardware technology.

We also measure the CPU cycles for Passport header stamping and verification using the *get_cycles()* Linux kernel function. Figure 9.2 shows the result, with CPU cycles converted to time. The per-hop increment of a Passport header stamping time is about 420 ns.

Passport's header processing and communication overhead is inherent to the symmetric cryptography it uses. For comparison, we list the processing overhead as well as signature sizes of well-known public key signatures that provide a similar level of

security as one 64-bit MAC in Figure 9.3. The tests are done using the *openssl* speed test on the router PC, and the security levels are estimated according to [48].

Passport-enabled routers also exchange Diffie-Hellman keys on the routing plane. The cryptographic operations include generating Diffie-Hellman public-private value pairs and computing shared symmetric keys from Diffie-Hellman values. Both Diffie-Hellman value pairs and shared symmetric keys are generated using exponentiation. We tested the overhead to generate a Diffie-Hellman value pair and to compute a shared symmetric key on the router PC. As shown in Figure 9.2, each public key operation takes 5.64 ms.

The public key operations are expensive, but are unlikely to become a performance bottleneck, because an AS only performs public key operations when it re-keys. Re-keying should happen infrequently such as no more than once per week. When an AS itself re-keys, it needs to generate a shared symmetric key with all other ASes. As there are less than 40K ASes seen in a BGP routing table in 2010 [11], it takes less than four minutes to generate all the symmetric keys on the router PC. If another AS re-keys, an AS only needs to generate one symmetric key when it receives a new Diffie-Hellman public value from that AS. If we assume all ASes re-key randomly during a period of seven days, then on average, an AS may receive less than four new Diffie-Hellman public values per minute, and spend 23 ms to compute the shared secret keys.

9.1.2 Memory Overhead

Passport maintains per-AS key information. A router keeps a shared symmetric key per AS for Passport header stamping and verification. A MAC computation typically requires the initialization of a key context. It is desirable that a router initializes and stores the key context for fast processing. The size of a key context depends on the specific MAC; Our Linux implementation uses a UMAC key context that consumes

Table 9.1: **Benchmarked Throughput of Passport's NetFPGA Implementation**

operation	observed throughput (Gbps) w.r.t. AS path length		
	2 ASes	4 ASes	9 ASes
generate passport hdr	1.73	1.79	1.93
validate passport hdr	1.73	1.82	2.06

Table 9.2: **Estimated Throughput Limit of Passport's NetFPGA Implementation**

operation	estimated throughput limit (Gbps) w.r.t. AS path length		
	2 ASes	4 ASes	9 ASes
generate passport hdr	2.8	2.78	2.72
validate passport hdr	3.04	3.4	4.26

388 bytes. It requires less than 12MB memory to store the symmetric keys and their key contexts.

When an AS re-keys, another AS may need two different keys for Passport stamping and verification: one generated with the old Diffie-Hellman public value, and the other with the new value. This requires additional memory. As shown above, the average re-keying rate of all ASes is less than 4 per minute. If we assume it takes at most an hour for BGP to converge, then the number of simultaneously re-keying ASes is around 240, adding an additional 93KB memory overhead.

9.2 FPGA Implementation of Passport

As described in Section 9.1.1, throughput for routers that stamp Passport headers may become lower than 1Gbps when AS paths towards destinations are long. This motivates us to explore the option of implementing our design in hardware that has a fundamentally different packet processing architecture and verify that our design can have much higher performance.

To measure the packet forwarding performance of our FPGA implementation of Passport, we install a NetFPGA board on one PC. Each of the 4 1Gbps NIC ports on the NetFPGA board is connected to a separate PC that acts as both a

packet generator and a packet receiver. The routing table on the NetFPGA board is configured such that packets sent from one PC are returned to the same PC. Each PC connected to the NetFPGA board has a 1Gbps NIC and uses the Linux kernel module `pktgen.o` to generate packets as fast as it can.

Table 9.1 shows the observed throughput when each IP packet is 44-byte long excluding the Passport header. As can be seen, the throughput varies between 1.73Gbps and 2.16Gbps, a significant improvement over our Linux implementation. This improvement mainly comes from the interrupt-less and pipelining architecture of the FPGA implementation. However, the observed throughput is still far below the line speed (4Gbps). Further investigation reveals that this is because the packet forwarding capacity of the FPGA implementation has exceeded the packet generation capacity of the connected PCs. For instance, when the NetFPGA board is configured to insert Passport headers into packets and the AS path length is 9 (*i.e.*, 8 MACs are computed to generate the Passport header for a packet), we observe that the four PCs can only generate 2366K packets per second in total, and all the packets are successfully forwarded by the NetFPGA board.

As we cannot reach the throughput limit of our FPGA implementation with the benchmarking hardware, we give an estimation of this limit in Table 9.2. The clock of our NetFPGA board runs at 125MHz. Our micro-benchmarking result shows that the MAC generation module needs $22 + x$ clock cycles to generate all the MACs for a Passport header, where x is the number of needed MACs. When an incoming packet is small enough (*e.g.*, 40-byte), this module becomes the critical path in the preprocessing stage (Figure 8.3), resulting in the throughput limit as shown in Table 9.2. When the incoming packet size is large enough such that the packet transmission becomes the critical path in the preprocessing stage, our NetFPGA implementation can achieve line speed forwarding. For instance, when the NetFPGA board inserts a Passport header into a packet and the AS path to the destination

Table 9.3: **Micro-benchmarking Result for NetFence's Linux Implementation**

Packet Type	Router Type	Processing Overhead (ns/pkt)	
		NetFence	TVA+
request	bottleneck	w/o attack: 0 w/ attack: 492	389
	access	546	
regular	bottleneck	w/o attack: 0 w/ attack: 554	791
	access	w/o attack: 781 w/ attack: 1267	

has a length of 9, line speed forwarding can be achieved when the incoming packet size reaches 218 bytes.

The performance of our NetFPGA implementation is partially limited by the size of the FPGA chip and the AES-128 implementation. If we had enough gates on the FPGA chip and a fully pipelined AES-256 implementation, we could compute multiple MACs simultaneously and achieve back-to-back line speed forwarding even with the smallest IP packets. Being able to achieve line speed forwarding in a FPGA implementation suggests that it is promising to implement Passport on very high speed core routers without performance penalties.

9.3 NetFence

We benchmark our Linux implementation of NetFence on Deterlab [21] with a three-node testbed. A source access router A and a destination C are connected via a router B. The B—C link is the bottleneck with a capacity of 5Mbps. Each node has two Intel Xeon 3GHz CPUs and 2GB memory. To benchmark the processing overhead without attacks, we send 100Kbps UDP request packets and 1Mbps UDP regular packets from A to C respectively. To benchmark the overhead in face of DoS attacks, we send 1Mbps UDP request packets and 10Mbps UDP regular packets simultaneously.

The benchmarking result is shown in Table 9.3. When there is no attack, a

request packet does not need any extra processing on the bottleneck router B, but it introduces an average overhead of 546ns on the access router A because the router must stamp the *nop* feedback into the packet. A regular packet does not incur any extra processing overhead on the bottleneck router either, but it takes the access router 781ns on average to validate the returned feedback and generate a new one. When the bottleneck link enters the *mon* state during attack times, the bottleneck router takes 492ns to process an 88-byte request packet, or at most 554ns to process a 1500-byte regular packet. The access router takes on average 1267ns to process a regular packet at attack times.

The performance of a capability system TVA+ [52] on the same topology is also shown in Table 9.3 for comparison. We can see that the processing overhead introduced by NetFence is on par with that of TVA+. Note that we do not show the result when TVA+ caches capabilities, because such caching requires per-flow state on routers, while NetFence does not have this requirement.

These results show that NetFence's per-packet overhead is low. The CPU-intensive operations are primarily AES computation. Since there exists commercial hardware that can support AES operations at 40Gbps [37], we expect that NetFence's per-packet processing will not become a performance bottleneck.

10

Effectiveness Evaluation

In this chapter, we evaluate the effectiveness of our design using small-scale testbed experiments and extensive large-scale simulations.

10.1 Preventing Source Spoofing and Reflector Attacks with Passport

We run experiments on a Deterlab [21] testbed to demonstrate the effectiveness of Passport preventing source address spoofing and reflector attacks. We emulate a scenario in which malicious Host attackers (as defined in Section 7.1.1) in legacy ASes spoof the source address of a victim in an upgraded, Passport-enabled AS to launch reflector attacks.

The experiment topology is shown in Figure 10.1. Each circle represents one AS. Only the shaded ASes deploy Passport. We configure each AS to have only one router D_i. The bottleneck link is between D_0 and D_1. The victim host V is connected to D_1, the attacker host A is connected to D_9, and D_{11} to D_{19} each has one host connected to it. We wish to use a topology with more hosts, but we were limited by the number of PCs we could hold on Deterlab.

In our experiments, hosts U_1 to U_9 each run a reflector application that replies

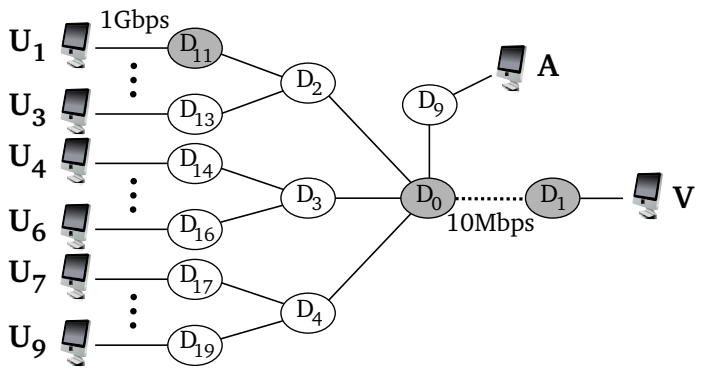


FIGURE 10.1: **Topology of the Deterlab experiments to evaluate the effectiveness of Passport.**

to incoming UDP packets with an amplification ratio of 40. The attacker spoofs the victim's address and sends UDP packets to all reflectors $U_1 \sim U_9$ uniformly. Each UDP packet sent by the attacker has 32 bytes payload. These parameters are set to emulate a DNS reflector attack [81].

After the attack is started, hosts U_1 to U_9 also each send 100 files to the victim using TCP. These TCP transfers are used to measure how the reflector attack affects the network performance seen by an end system. The size of each file is 20KB, and a file transfer aborts if it cannot finish in 10 seconds. We vary the attacker's sending rate from 1% to 20% of the bottleneck link bandwidth and measure the file transfer time.

The results are shown in Figure 10.2. Hosts U_2 to U_9 are in legacy ASes. Their file transfer traffic is legacy traffic and competes for bandwidth with the legacy reflector traffic at D_0. When the reflector traffic congests the bottleneck link, their file transfer traffic suffers from congestion. The file transfer traffic from U_1 carries Passport headers and only competes for bandwidth with verified Passport traffic at D_0. Therefore, U_1 is not affected by the reflector traffic and can finish all the file transfers quickly. Note that the reflector application on U_1 will not receive attack traffic and therefore will not send out reflector traffic to compete with U_1's file transfer

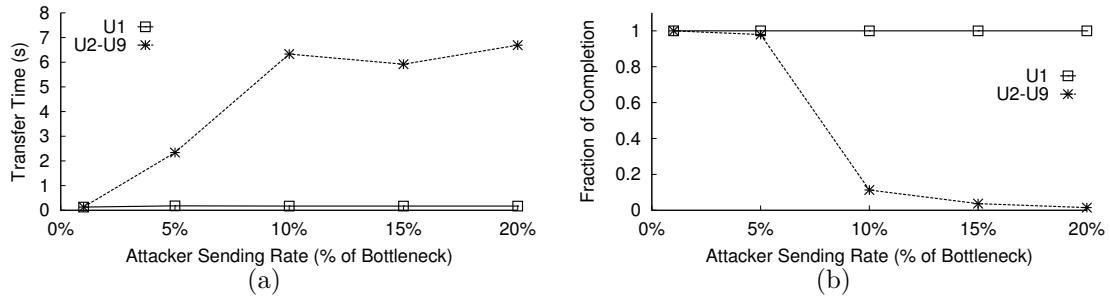


FIGURE 10.2: **The average file transfer times and fractions of completion for hosts U_1 to U_9 when the attacker A spoofs V's address to launch a reflector attack. U_1 is in an upgraded AS D_{11}, while U_2 to U_9 are in non-upgraded ASes D_{12} to D_{19}.**

traffic at the bottleneck link, because when attack packets to U_1 reach D_0, D_0 will discard them as they do not include Passport headers but both their source and destination addresses belong to upgraded, Passport-enabled ASes (See § 5.6.3).

10.2 Mitigating DoS Flooding Attacks with NetFence

Next we evaluate how well NetFence mitigates various DoS flooding attacks using ns-2 simulations. We compare NetFence with three other representative DoS flooding attack mitigation schemes:

TVA+ : TVA+ [103, 53] is a network architecture that uses network capabilities and per-host fair queuing to defend against DoS flooding attacks. TVA+ uses hierarchical fair queuing (first based on the source AS and then based on the source IP address) at congested links to mitigate request packet flooding attacks, and per-receiver fair queuing to mitigate authorized traffic flooding attacks in case (colluding or incompetent) receivers grant network capabilities to attack traffic.

StopIt : StopIt [53] is a filter and fair queuing based DoS defense system. A targeted victim can install network filters to stop unwanted traffic. Similar to TVA+, in case receivers fail to install filters, StopIt uses hierarchical fair queuing (first based on the

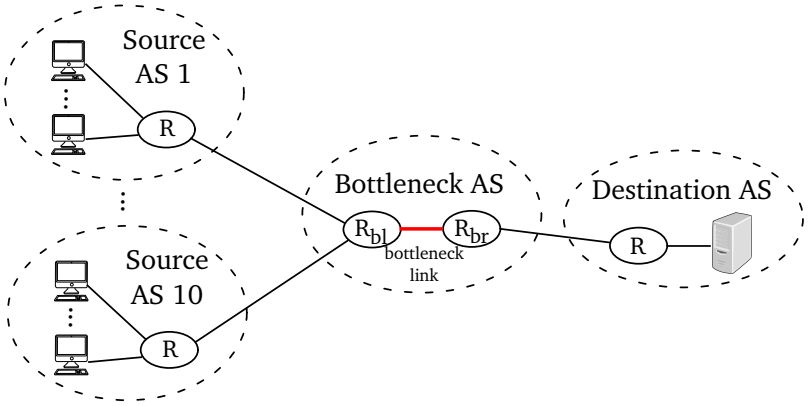


FIGURE 10.3: **The simulation topology for unwanted traffic flooding attacks.**

source AS and then based on the source IP address) at congested links to separate legitimate traffic from attack traffic.

Fair Queuing (FQ) : Per-sender fair queuing at every link provides a sender its fair share of the link's bandwidth. We use this scheme to represent any DoS defense mechanism that aims to throttle attack traffic to consume no more than its fair share of bandwidth.

We have implemented TVA+ and StopIt as described in [53, 103]. We use the Deficit Round Robin (DRR) algorithm [83] to implement fair queuing because it has $O(1)$ per packet operation overhead. In our simulations, attackers do not spoof source addresses, because source spoofing would be prevented by Passport: traffic with spoofed source addresses are either discarded or forwarded with lower priority such that it cannot cause congestion for traffic with authentic source addresses.

10.2.1 Unwanted Traffic Flooding Attacks

We first simulate the attack scenario where the attackers directly send packet flood to a victim, but the victim can identity the attack traffic and uses the provided DoS defense mechanism: capabilities in TVA+, secure congestion policing feedback in NetFence, and filters in StopIt, to block the unwanted traffic.

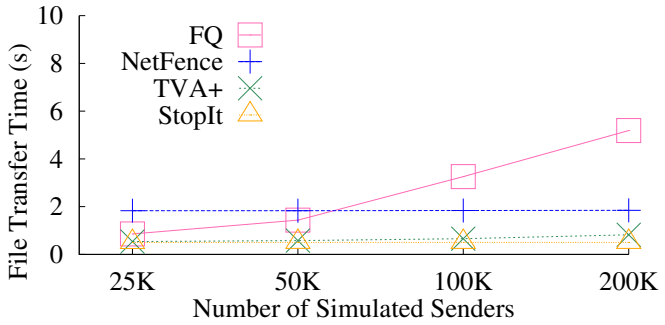


FIGURE 10.4: **The average transfer time of a 20KB file when the targeted victim can identify and wish to remove the attack traffic. The file transfer completion ratio is 100% in all simulated systems.**

We desire to simulate attacks in which thousands to millions of attackers flood a well provisioned link. However, we are currently unable to scale our simulations to support beyond several thousand nodes. To address this limitation, we adopt the evaluation approach in [103]. That is, we fix the number of nodes, but scale down the bottleneck link capacity proportionally to simulate the case where the bottleneck link capacity is fixed, but the number of attackers increases.

As shown in Figure 10.3, We use a dumb-bell topology in which ten source ASes connect to a destination AS via a transit AS. Each source AS has 100 source hosts connected to a single access router. The transit AS has two routers R_{bl} and R_{br}, and the destination AS has one victim destination host. The link between R_{bl} and R_{br} is the bottleneck link, and all other links have sufficient capacity to avoid congestion. We vary the bottleneck link capacity from 400Mbps to 50Mbps to simulate the scenario where 25K $\sim$ 200K senders (both legitimate and malicious) share a 10Gbps link. Each sender's fair share bandwidth varies from 400Kbps $\sim$ 50Kbps, which is NetFence's targeted operating region. The propagation delay of each link is 10ms.

In the simulations, each sender is either a legitimate user or an attacker. To stress-test our design, we let each source AS have only one legitimate user that repeatedly sends a 20KB file to the victim using TCP. We let each attacker send

1Mbps constant-rate UDP traffic to the victim. We measure the effectiveness of a DoS defense system using two metrics: 1) the average time it takes to complete a successful file transfer; and 2) the fraction of successful file transfers among the total number of file transfers initiated. We set the initial TCP SYN retransmission timeout to 1 second, and abort a file transfer if the TCP three-way handshake cannot finish after nine retransmissions, or if the entire file transfer cannot finish in 200 seconds. We terminate a simulation run when the simulated time reaches 4000 seconds.

For each DoS defense system we study, we simulate the most effective DoS flooding attacks malicious nodes can launch. In case of an unwanted traffic flooding attack, the most effective flooding strategy in NetFence and TVA+ is the request packet flooding attack. Under this attack, In NetFence, each sender (either malicious or legitimate) needs to choose a proper priority level for its request packets. We make an attacker always select the highest priority level at which the aggregate attack traffic can saturate the request channel. A legitimate sender starts with the lowest priority level and gradually increases the priority level if it cannot obtain valid congestion policing feedback.

Figure 10.4 shows the simulation results. The average file transfer completion ratio is omitted because all file transfers complete in these simulations. As can be seen, StopIt has the best performance, because the attack traffic is blocked near the attack sources by network filters. TVA+ and NetFence also have a short average file transfer time that only increases slightly as the number of simulated senders increases. This is because in a request packet flooding attack, as long as a legitimate sender has one request packet delivered to the victim, it can send the rest of the file using regular packets that are not affected by the attack traffic. The average file transfer time in NetFence is about one second longer than that in TVA+, because a legitimate sender will initially send a level-0 request packet that cannot pass the bottleneck link due to attackers' request packet floods. After one second retransmission backoff, a

sender is able to retransmit a request packet with sufficiently high priority (level-10) to pass the bottleneck link. Attackers cannot further delay legitimate request packets, because they are not numerous enough to congest the request channel at this priority level.

Figure 10.4 also shows that FQ alone is an ineffective DoS defense mechanism. With FQ, the average file transfer time increases linearly with the number of simulated senders, as each packet must compete with the attack traffic for the bottleneck bandwidth.

These results show that NetFence performs similarly to capability-based and filter-based systems when targeted victims can stop the attack traffic. A legitimate sender may wait longer in NetFence to successfully transmit a request packet than in TVA+ or StopIt. This is because NetFence uses coarse-grained exponential backoff to schedule a request packet's transmission and set its priority, while TVA+ uses fine-grained but less scalable per-sender fair queuing to schedule a request packet's transmission, and StopIt enables a victim to completely block unwanted traffic.

10.2.2 Colluding Attacks

Next we present our simulation results for regular traffic flooding attacks in which malicious sender-receiver pairs collude to flood the network. Such attacks may also occur if DoS victims fail to identify the attack traffic.

Single Bottleneck: As shown in Figure 10.5, we use a topology similar to Figure 10.3. The router R_{br} at the right-hand side of the bottleneck link connects to one destination AS with a victim host and nine additional ASes, each having a colluding host (colluder). Each source AS has 25% legitimate users and 75% attackers, simulating the case where the attackers are numerous but there are still a reasonable number of legitimate users in each source AS.

Each legitimate user sends TCP traffic to the victim host. We simulate two types

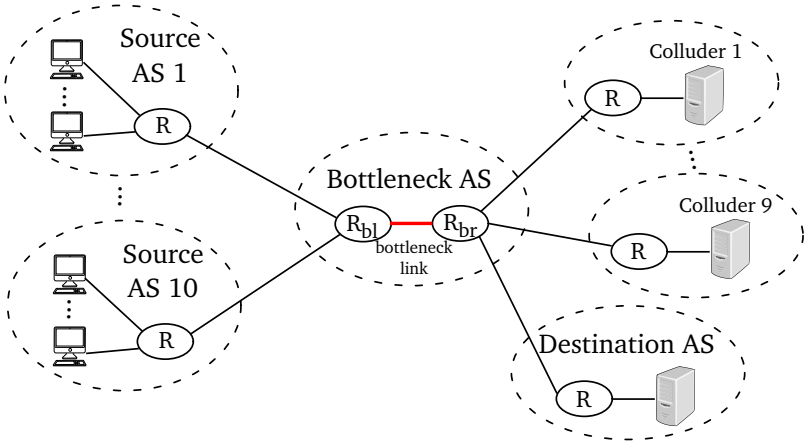


FIGURE 10.5: **The simulation topology for single-bottleneck colluding attacks.**

of user traffic: 1) long-running TCP, *i.e.*, a legitimate sender sues TCP to send a single large file during a simulation run; 2) web-like traffic, *i.e.*, a sender uses TCP to send small files whose size distribution mimics that of web traffic. We draw the file size distribution from a mixture of Pareto and exponential distributions as in [55], and make the interval between two file transfers uniformly distributed between 0.1 and 0.2 seconds. The maximum file size is limited to 150KB to make the experiments finish within a reasonable amount of time.

To simulate colluding attacks, we let each attacker send 1Mbps UDP traffic to a colluder. The attackers in TVA+ and NetFence send regular packets. Colluders in StopIt do not install filters to stop the attack traffic. Each experiment runs for 4000 seconds in simulated time.

When compromised nodes organize into pairs to send attack traffic, NetFence aims to guarantee each legitimate sender its fair share of the bottleneck bandwidth without keeping per-sender queues in the core of the network. We use two metrics to measure a DoS defense system's performance under this type of attack: 1) *Throughput Ratio*, the ratio between the average throughput of a legitimate user and that of an attacker; and 2) *Fairness Index* among legitimate users [18]. Let x_i denote a le-

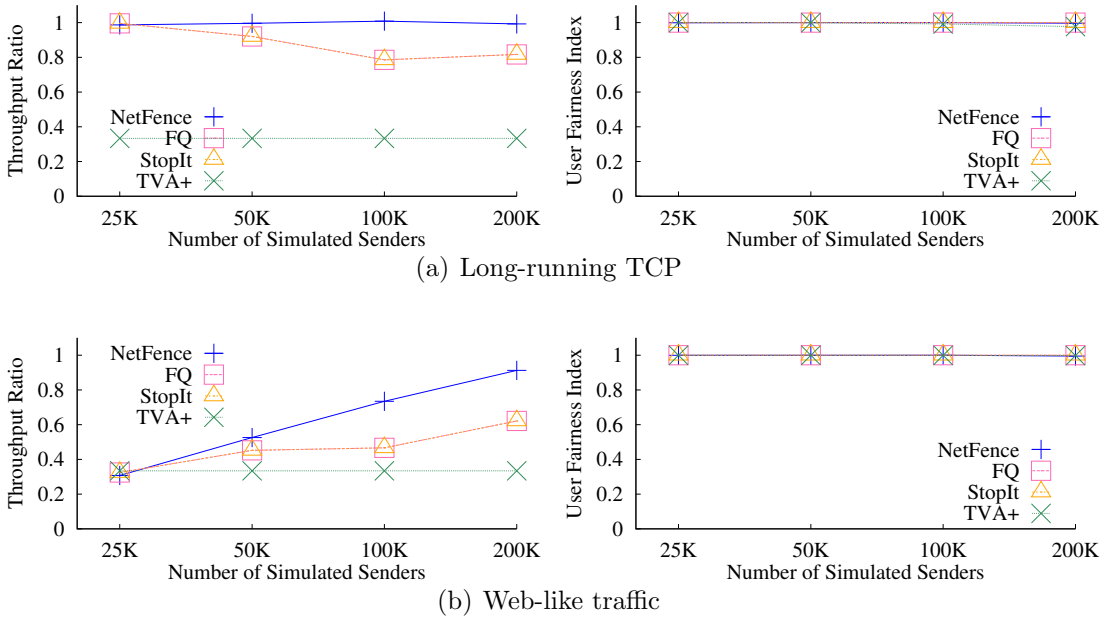


FIGURE 10.6: ***Throughput Ratio*** **between legitimate users and attackers and** ***Fairness Index*** **among legitimate users when receivers fail to suppress the attack traffic.**

gitimate sender i's throughput, and the fairness index is defined as $(\sum x_i)^2/(n \sum x_i^2)$. The ideal throughput ratio is 1, indicating that a legitimate user obtains on average the same bottleneck bandwidth as an attacker. The ideal fairness index is also 1, indicating that each legitimate sender has the same average throughput. We only measure the fairness index among legitimate users because *Throughput Ratio* has already quantified how well a legitimate user performs relatively to an attacker.

Figure 10.6 shows the simulation results. The fairness index for all systems is close to 1 in all the simulations, indicating that all the evaluated DoS defense systems can achieve fairness among legitimate users. For long-running TCP, NetFence's throughput ratio is also close to 1. This result shows that NetFence provides a legitimate sender its fair share of bandwidth despite the presence of DoS flooding traffic, consistent with the theoretic analysis in § 4.2.4. For the web-like traffic, NetFence's throughput ratio increases gradually from 0.3 to close to 1 as the number of simu-

lated senders increases. The throughput ratio is low when the number of senders is small, because a legitimate sender cannot fully utilize its fair share bandwidth: each sender has a large fair share of bandwidth, but a legitimate sender's web-like traffic has insufficient demand and there are gaps between consecutive file transfers.

FQ and StopIt perform exactly the same, because in these colluding attacks, they both resort to per-sender fair queuing to protect a legitimate user's traffic. However, unexpectedly, we note that they provide legitimate users less throughput than attackers even when the user traffic is long-running TCP. By analyzing packet traces, we discover that this unfairness is due to the interaction between TCP and the DRR algorithm. A TCP sender's queue does not always have packets due to TCP's burstiness, but a constant-rate UDP sender's queue is always full. When a TCP sender's queue is not empty, it shares the bottleneck bandwidth fairly with other attackers, but when its queue is empty, the attack traffic will use up its bandwidth share, leading to a lower throughput for a TCP sender.

TVA+ has the lowest throughput ratio among all systems in this simulation setting, indicating that a small number of colluders can significantly impact TVA+'s performance. This is because TVA+ uses per-destination fair queuing on the regular packet channel. With N_C colluders, a DoS victim obtains only $\frac{1}{N_C+1}$ fraction of the bottleneck capacity C at attack times, and each of the victim's G legitimate senders obtains $\frac{1}{G(1+N_C)}$ fraction of the capacity C. The attackers, however, obtain an aggregate $\frac{N_C}{(1+N_c)}$ fraction of C. If this bandwidth is shared by B attackers fairly, each will get a $\frac{N_C}{B(1+N_c)}$ fraction of the bottleneck capacity. A sender's bottleneck bandwidth share in other systems (NetFence, StopIt, and FQ) is $\frac{1}{G+B}$, and it does not depend on the number of colluders N_C. In our simulations, $N_c = 9$, $G = 25\% \times 1000$, and $B = 75\% \times 1000$. A legitimate TVA+ sender obtains $\frac{1}{2500}$ of the bottleneck bandwidth, while an attacker obtains $\frac{9}{7500}$ of the bottleneck bandwidth, three times

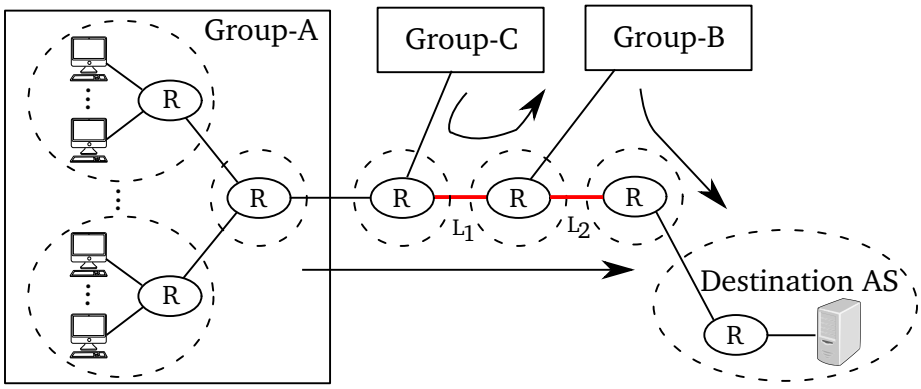


FIGURE 10.7: **The parking-lot simulation topology for multi-bottleneck colluding attacks.**

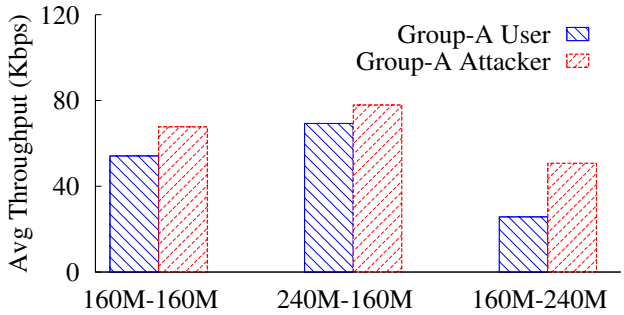


FIGURE 10.8: **Sender throughput (Kbps) under regular packet flooding attacks in a parking-lot topology with two bottleneck links. The fair share rate for each sender is 80Kbps.**

higher than a legitimate sender, as shown in Figure 10.6.

In these simulations, we also measure the bottleneck link utilization. The result shows that the utilization is above 90% for NetFence, and almost 100% for other systems. NetFence does not achieve full link utilization mainly because a router stamps the $L^{\downarrow}$ feedback for two extra control intervals after congestion has abated, as explained in § 6.3.4.

Multiple Bottlenecks: To evaluate NetFence's performance with multiple bottlenecks, we simulate colluding attacks on a parking-lot topology as shown in Figure 10.7. It has two bottleneck links in the *mon* state: L_1 and L_2. 3000 senders

are organized into three groups of the same size: Group-A senders send through both L_1 and L_2, Group-B senders send through the second link L_2, and Group-C senders send through the first link L_1. Each group contains 75% attackers and 25% legitimate users. Each attacker sends UDP traffic in regular packets at 1Mbps to a colluder, while each user sends long-running TCP traffic to a victim. Every simulation terminates at 4000 seconds in simulated time.

We simulate three different pairs of bottleneck capacities: 1) $C_{L_1} = C_{L_2} = 160Mbps$; 2) $C_{L_1} = 240Mbps, C_{L_2} = 160Mbps$; and 3) $C_{L_1} = 160Mbps, C_{L_2} = 240Mbps$. In these cases, a Group-A sender's max-min fair share bandwidth is 80Kbps, while a Group-B or Group-C sender's max-min fair share is either 80Kbps or 160Kbps. The simulation results are shown in Figure 10.8. The x-axis shows different simulation cases, and the y-axis is a sender's average throughput. Senders in Group-B and Group-C are omitted because they each can get at least its fair share bandwidth in all the simulations. However, on average a sender in Group-A obtains much smaller throughput than its fair share rate when $C_{L_1} < C_{L_2}$. This is because traffic from a Group-A sender traverses both bottleneck links L_1 and L_2. As discussed in § 6.3.5, if a flow traversing both L_1 and L_2 switches between the two corresponding rate limiters frequently, its rate limit may become smaller than its fair share bandwidth. This problem does not significantly affect the first two cases in which $C_{L_1} \geqslant C_{L_2}$, because in these simulations a Group-A sender's traffic carries L_1's feedback most of the time. As a result, the rate limiter (src, L_1) for a Group-A sender src is not idle most of the time, and it can have a rate limit close to the fair share bandwidth of the sender src.

Figure 10.8 also shows that a TCP user in Group-A has an even lower average throughput than an attacker in Group-A (the rightmost case). This is because when a TCP flow switches between two rate limiters that have very different rate limits, TCP cannot catch up with the abrupt rate limit change.

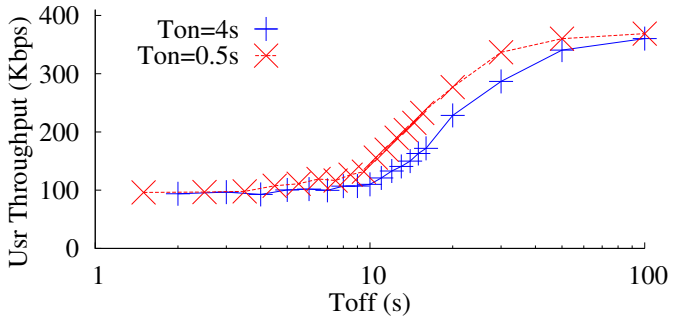


FIGURE 10.9: **Average user throughput in face of microscopic on-off attacks. The user traffic is long-running TCP. There are 100K simulated senders. Each sender's fair share bottleneck bandwidth is 100Kbps.**

NetFence's performance with multiple bottleneck links can be improved with more complicated designs, as discussed in § 6.3.5 and Appendix B. We believe the current design is a reasonable tradeoff to balance between complexity and performance.

strategic attacks: attackers may launch attacks that are much more sophisticated than brute-force flooding. Next we simulate microscopic on-off attacks and show that with NetFence, the attack traffic's shape does not reduce a legitimate user's throughput.

The simulation topology is the same as in shown in Figure 10.5. All legitimate users send long-running TCP traffic, while attackers send on-off UDP traffic. In the on-period T_{on}, an attacker sends at the rate of 1Mbps; in the off-period T_{off}, it does not send any traffic. All attackers synchronize their on-periods to create the largest traffic bursts. There are 100K simulated senders, each having a fair share bandwidth of at least 100Kbps.

In these simulations, we use two different values for T_{on}: 0.5s and 4s. For each T_{on}, we vary the off-period length T_{off} from 1.5s to 100s. Figure 10.9 shows the simulation results. As we can see, the average user throughput is at least a user's fair share rate as if attackers were always active (100Kbps), indicating that the attack

cannot reduce a legitimate user's fair share of bandwidth. As the attackers' off-period length increases toward 100 seconds, a legitimate user can achieve a throughput close to 400Kbps, indicating that long running TCP users can use most of the bottleneck bandwidth when the attackers' off-period is long.

11

Discussion

In this chapter, we discuss some of our design choices and their interaction with some of the existing protocols in the Internet. We also briefly discuss how Passport may help to mitigate application-level DoS attacks.

11.1 Passport

Packet fragmentation: Packets fragmented in the middle of the network will not have valid Passport headers. Passport demotes all fragments without valid Passport headers, including at the destination AS. A destination host receiving demoted fragments should not trust their source addresses at all. We are not concerned much with this issue, because fragmentation by the network is discouraged, and has been disabled by IPv6.

In case it is undesirable to let a destination host receive any packet with a totally untrustworthy source address, its access router may temporarily hold the first fragment of a packet and validate its Passport header after obtaining the packet length from the last fragment of the packet. If the Passport header is invalid, the access

router can drop the first fragment, preventing the destination host from receiving the complete packet and responding to it.

Prefix aggregation: If an AS's prefix is aggregated by its provider AS, then its AS path attributes, including its Diffie-Hellman value will be lost. In this case, an AS should rely on its provider to stamp and verify Passport headers for its traffic. A customer AS that desires to stamp and verify Passport headers on its own should request its providers not to aggregate its prefix. As an AS only needs one prefix to distribute the key exchange information, even if that prefix is not aggregated, it will not significantly increase the BGP table size, as there are much more prefixes than ASes in the Internet (>360K versus <40K in 2010 [11]).

Multi-origin prefixes: BGP prefix announcements may have multi-origin AS conflicts (MOAS), a practice not recommended by IETF [36]. MOAS interferes with the key lookup process, as a router needs to use the correct key from the origin AS to verify a Passport header. MOAS can be a signal of prefix hijacking; in this case, Passport relies on the routing system to resolve MOAS conflicts. MOAS can be caused by sibling ASes announcing each other's prefixes [60]; in this case, the sibling ASes should share Diffie-Hellman values so that a verifier can use the key shared with either AS to verify their addresses. MOAS can also be caused by multi-homed ASes connecting to its providers without BGP [104]. In this case, a simple solution is for the site to run BGP, in order to be compatible with Passport and IETF's recommendation. Our observation from a BGP table obtained from the Oregon RouteViews server on Aug 1st, 2007 shows that 1385 out of 244095 (0.57%) prefixes have more than one origin. Therefore, we expect that MOAS prefixes are uncommon, and are unlikely to be a deployment hurdle.

Anycast addresses have legitimate multi-origins. A source should not send traffic with anycast source address. An anycast address can always be used as a destination

address, because when computing a MAC for a destination AS, a source AS uses the key shared with the destination AS that will actually receive the packet, and the MAC verification at the destination AS will succeed.

Path discrepancy: Temporary routing changes may cause the data forwarding paths to be different from BGP paths. Passport demotes rather than discards packets at the intermediate ASes. Demoted traffic can still reach a destination if the network is not congested. In addition, Rexford et al. observe that BGP routes to popular destinations are stable [78], having less than 0.3 update events per day. Therefore, demotion caused by temporary path disagreement may not significantly degrade end-to-end performance.

Mao et al. [60] observe that paths inferred by traceroute may be different from BGP paths even when routing is stable. They postulate several causes. Other than prefix aggregation and MOAS, most of them are due to traceroute not accurately reflecting forwarding paths or AS boundaries. One rare cause is an iBGP misconfiguration of a transit AS. Passport can become a diagnosis tool to such routing anomalies. If a router discovers that all traffic it forwards cannot be verified, it is a strong indication of misconfiguration.

Inter-domain multicast: Passport only provides source authentication for unicast traffic, because the origin of a duplicated multicast packet does not match its source address. Routers should use separate authentication mechanisms such as [74] to authenticate multicast traffic.

11.2 NetFence

Fair Share Bound: When a disproportionately large number (B) of attackers attack a narrow link C (*e.g.*, when a million bots attack a 1Mbps link), the fair share lower bound $O(\frac{C}{G+B})$ achieved by NetFence or per-sender fair queuing (*e.g.*,

[53]) is small. However, this lower bound is still valuable, because without it, a small number of attackers can starve legitimate TCP flows on a well-provisioned link (*e.g.*, 10Gbps). Although this guarantee does not prevent large-scale DoS attacks from degrading a user's network service, it mitigates the damage of such attacks with a predictable fair share, without trusting receivers or requiring the network to identify and remove malicious traffic. Other means, like congestion quota discussed below, can be used to further throttle malicious traffic.

Equal Cost Multiple Path (ECMP): NetFence assumes that a flow's path is relatively stable and the bottleneck links on the path do not change rapidly. One practical concern arises as routers may split traffic among equal-cost multi-paths for load balancing. Fortunately, most ECMP implementations in practice (*e.g.*, [19]) would assign a flow's packets to the same path to avoid packet reordering. Thus, we expect NetFence to work well with ECMP.

Congestion Quota: If we assume legitimate users have limited traffic demand at attack time while attackers aim to persistently congest a bottleneck link, we can further weaken a DoS flooding attack by imposing a congestion quota, an idea borrowed from re-ECN [15]. That is, an access router only allows a host to send a limited amount of "congestion traffic" through a bottleneck link within a period of time. Congestion traffic can be defined as the traffic that passes a rate limiter when its rate limit decreases. With a congestion quota, if an attacker keeps flooding a link, its traffic through the link will be throttled after it consumes its congestion quota. Different from re-ECN, NetFence can impose a separate congestion quota for each sender-bottleneck pair so that a sender's traffic not traversing any link that is under attack will not be incidentally throttled when its quota for a link under attack is used up.

Rate Limit Control: For simplicity, the NetFence design uses a closed-loop AIMD

algorithm to adjust rate limits. Another option to set a sender's rate limit is to let a congested router compute a max-min share rate of each sender, and periodically send this rate to a sender's access router. We discard this approach because it involves per-sender state and has higher computation and message overhead than AIMD.

Convergence Speed: It may take a relatively long time (*e.g.*, 100s-200s) for NetFence to converge to fairness when a sender's rate limit is significantly different from its fair share of bottleneck bandwidth. This is because the control interval I_{lim} is on the order of a few seconds (two seconds in our current implementation), much longer than a typical RTT on the Internet. This convergence speed is acceptable in the NetFence design, because a rate limiter persists for a much longer period of time (*i.e.*, on the order of hours), which prevents attackers from constantly taking advantage of the unfairness during the convergence by sending strategic on-off traffic.

TCP and Rate Limiter Interaction: TCP's window size adjustment may be temporarily out of synchronization with the rate limiter adjustment. For instance, TCP may just receive its ACK before the rate limit is reduced, in which case TCP's sending rate will increase. This temporary out of synchronization is not a big issue, because the rate limit adjustment interval I_{lim} is much larger than a typical RTT in the Internet such that the out of synchronization may occur at most once among many RTTs.

11.3 Application-Level DoS Attacks

By establishing authentic source addresses, Passport is also a valuable tool to assist mitigating application-level DoS attacks (as defined in Section 2.1). With Passport, once an application-level DoS defense system identifies some attack traffic, it can easily install local network-layer filters to discard packets from the attack sources, or lower the priorities of the requests from the attack sources at the application level.

Since application-level DoS mitigation typically occurs close to or on the destination victim systems, the network traffic volume involved is much lower than on core routers, and therefore it is usually not a problem to keep states (*e.g.*, network-layer filters) for identified attack sources, as the states can be stored in slower, cheaper but larger memory (*e.g.*, low-cost DRAM). For instance, even if a server installs one network-layer filter per IP address for the whole IPv4 address space, there are only 4 billion filters. If these filters are implemented as a bit map with 4 billion bits, they require 512MB of memory, which costs less than 20USD in 2011 if the memory type is DDR-400 for a desktop PC.

12

Conclusion and Future Work

In this chapter, we summarize the work in this dissertation and propose possible directions for future work.

12.1 Mitigating DoS Flooding Attacks in Two Steps

DoS flooding attacks are difficult to mitigate partly because of the lack of accountability in the network layer of the current Internet. This dissertation proposes a two-step process in mitigating DoS flooding attacks: first to establish the network-layer accountability using Passport, and second to defend against the attacks using NetFence. Passport enables the cryptographic verification of a packet's true origin at the AS level, and it does so without much performance penalty: our preliminary evaluation suggests that it is practical for high-speed routers to support Passport at line speed. On top of Passport, NetFence establishes multiple levels of defense against DoS flooding attacks. In the typical case that attackers only control end hosts and the attack traffic can be detected by the destination hosts, NetFence enables the destination hosts to throttle the attack traffic close to the attack sources. If the attackers can make the attack traffic indistinguishable from legitimate traffic,

NetFence can still guarantee that each attack source can at most consume its own fair share of the bottleneck bandwidth. Even in the extreme case that attackers compromise source ASes, NetFence can still ensure that each compromised source AS can only consume its own fair share of the bottleneck bandwidth. Our preliminary evaluation suggests that NetFence achieves all the three levels of defense in a scalable and efficient way such that high-speed core routers can implement NetFence and still achieve line speed forwarding even at attack time. In addition to the effectiveness, efficiency and scalability goals, both Passport and NetFence are also designed to be secure and robust against various attack strategies, incrementally deployable, and self-contained without depending on non-routing infrastructures such as DNS.

12.2 Further Refining NetFence

Congestion Detection and Rate Limit Control: For simplicity, the current implementation of NetFence uses RED as the congestion detection algorithm. RED is not a strictly load-based congestion detection algorithm. A future research direction is to explore a load-based congestion detection algorithm that may react sooner to congestion than RED.

Control Interval Length: The current NetFence design uses a fixed control interval I_{lim} for all senders. It is difficult to optimize the value of I_{lim}, because a short I_{lim} cannot accommodate heterogeneous RTTs on the Internet, while a long I_{lim} slows down the convergence of rate limits. A potential improvement is to have a variety of control interval lengths $\{I_{lim}^k\}$. A sender may signal which length it prefers to use with a few bits in its NetFence header. Accordingly, we need to update the rate limit adjustment algorithm on an access router and the congestion hysteresis on a congested router to ensure fairness among senders with different control interval lengths.

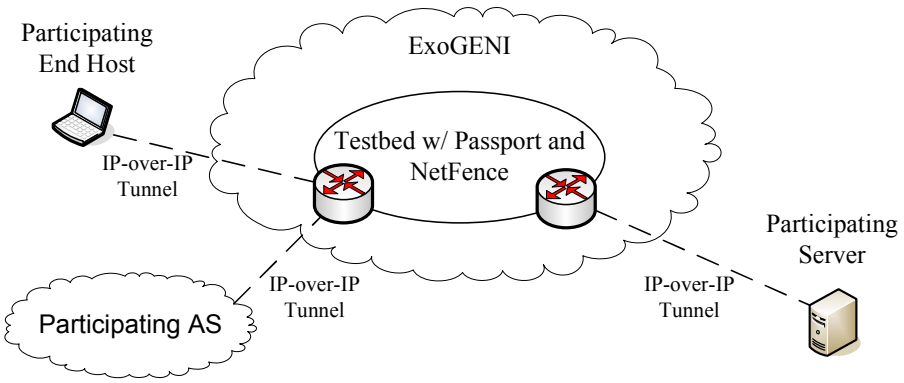


FIGURE 12.1: **One possible solution to build a virtual testbed with Passport and NetFence inside ExoGENI and connect external users to the testbed.**

12.3 One Step Closer to Reality: Deployment on ExoGENI

In this dissertation we evaluate our design of Passport and NetFence with analysis, benchmarking and simulations. While they all show promising results, the ultimate verification of whether our design achieves its goals is to deploy and use it in the real Internet. As the first step towards the real deployment, we can deploy our design on the ExoGENI testbed [8].

ExoGENI is a federation of cloud computing infrastructures connected by a variety of links and circuits. Since it adopts a flexible control framework with heterogeneity in mind, each site in the federation is autonomous, i.e., free to choose the software and hardware technologies to implement its cloud service. This enables participation from all types of cloud service providers; As a result, ExoGENI could potentially provide us with rich and plentiful computational resources (e.g., virtual machines, programmable routers, controllable virtual links with QoS guarantees) at diverse geographic locations. Since ExoGENI allows researchers to build fully customizable virtual network topologies on top of it, we can build our own large-scale virtual testbed that resembles the real Internet more than many other infrastructures such as Deterlab [21].

In addition to having diverse and plentiful computational resources, another key advantage of ExoGENI is that it allows user participation. A virtual network topology in ExoGENI is not restricted to be an isolated small world; it may interconnect with the real Internet or other virtual network topologies in ExoGENI, and the nodes in the network may have public IP addresses to directly interact with end users. This enables us to include real users and real network traffic into the deployment of our design and may potentially make our virtual testbed part of the real Internet.

There are two key challenges to deploying our design of Passport and Netfence on ExoGENI. The first challenge is to have high-performance implementations of our design's router logics in the ExoGENI environment. As shown in Section 9.1 and Section 9.3, our current Linux prototype implementation is likely to be able to handle 1Gbps-level traffic if deployed on bare-metal PCs. However, to fully make use of the available resources on ExoGENI, we also need a Linux implementation that can run equally well in virtual machines as well as good implementations for other computation platforms (e.g., RouteBricks [24] when it becomes available in ExoGENI).

The second challenge, which is even more important in the long run, is to allow user participation. One potential solution is to allow end hosts, servers and even ASes anywhere in the current Internet to connect to our virtual testbed using IP-over-IP tunnels, as shown in Picture 12.1. As long as the participants deploy our design, they would enjoy the full benefits provided by our design within our virtual testbed.

To lower the deployment hurdle for participating individual end hosts and servers, when we operate in a small-to-medium scale, these computers do not need to enable Passport and NetFence themselves; instead, their access points in our testbed could insert and remove Passport and NetFence headers and perform all the Passport and NetFence related operations on behalf of them. The end hosts and servers and

their access points may exchange information such as packet demotion using the IP header, similar to the mechanism described in Section 5.6.3. For this approach to work properly, the participating end hosts and servers need to guarantee that all the corresponding traffic go through their access points in our testbed; one potential solution is for them to use IP addresses allocated to them by our testbed, but then the servers have to publish their additional addresses.

IP-over-IP tunneling is by no means the only way to allow user participation. For instance, an alternative approach is to install HTTP proxies on leaf nodes in our virtual testbed and invite end users to use these proxies. Servers can be connected to our testbed similarly using reverse HTTP proxies. This approach does not require any changes to the external computers' software, but the functionalities provided by our proxies limit what the participants can do in our testbed.

Appendix A

Convergence Analysis for NetFence

NetFence uses an Additive Increase and Multiplicative Decrease (AIMD) algorithm to adjust a sender's rate limit. AIMD has been proved to converge onto efficiency and fairness [18]. We first briefly summarize the AIMD results, and then analyze how well NetFence converges to fairness and efficiency.

A.1 AIMD Preliminary

Consider a simplified fluid model where one link is shared by N synchronous flows, all of which having the same round-trip time (RTT), T. Each flow i uses the following rule to update its sending rate $x_i(t)$ at time t:

$$\begin{array}{rrcll} \text{AI:} & x_i(t+T) & = & x_i(t)+\alpha & \text{if no congestion} \\ \text{MD:} & x_i(t+\delta t) & = & x_i(t)\times\beta & \text{otherwise} \end{array}$$

where $\alpha > 0$, $1 > \beta > 0$, $\delta t > 0$ and $\delta t \to 0$.

Convergence to efficiency: When the link is under-utilized, Additive Increase (AI) is applied. The aggregate rate continues to increase as $\sum_i x_i(t+T) = \sum_i x_i(t) + N\alpha > \sum_i x_i(t)$. At some point the link will eventually be over-utilized, leading to

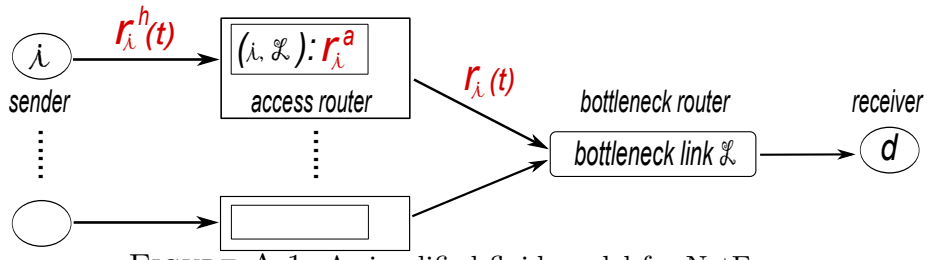


FIGURE A.1: A simplified fluid model for NetFence

Multiplicative Decrease (MD) being applied, causing the decrease of the aggregate rate: $\sum_i x_i(t+\delta t) = \beta \times \sum_i x_i(t) < \sum_i x_i(t)$. The decrease caused by MD finally results in link under-utilization. This pattern of oscillation around full utilization repeats, with the overshoot and undershoot decided by $N\alpha$ and β, respectively. Efficiency convergence is achieved if $\alpha \to 0$ and $\beta \to 1$.

Convergence to fairness : We measure fairness with Jain's fairness index [18]: $(\sum_i x_i(t))^2/N\sum_i x_i^2(t)$. It is easy to show that MD keeps the index unchanged, while AI increases it. Therefore, given sufficient number of AI rounds, the fairness keeps increasing, and will eventually reach its maximum, 1. At this point, all the flows share the link capacity equally.

A.2 Capacity Share Lower Bound

Given G good and B bad hosts sharing a bottleneck link of capacity C, can NetFence guarantee that each good host obtains its fair share $O(\frac{C}{G+B})$ of the capacity, regardless of the attack strategy bad hosts may apply? Here we prove this is indeed true.

Again we consider a simplified fluid model in Figure A.1. There are a fixed number of senders and one receiver. To make our analysis tractable, we do not consider delay: any rate change at the senders immediately takes effect at the bottleneck link. The bad hosts can use traffic of any shape to attack. As shown in the figure, for any host i, let $r_i^h(t)$ be its sending rate at time t, and r_i^a be its rate limit at its access router.

r_i^a is a constant value during a control interval $[t_0, t_0 + I_{lim})$, where t_0 is a particular point in time. At the output link of the access router, the egress rate of the host i is $r_i(t) = \min(r_i^h(t), r_i^a)$ due to the rate limit.

Definition 1. *We say a host has* ***sufficient demand*** *if it sends out data fast enough such that its corresponding rate limit is not punished by the robust rate limit adjustment algorithm described in § 6.3.4.*

Definition 2. *During any control interval, given a rate limit r_i^a, the* ***rate limit utilization*** *is defined as $\nu_i = \overline{r_i}/r_i^a$, where the host average egress rate $\overline{r_i} = \frac{1}{I_{lim}} \int_{t_0}^{t_0+I_{lim}} r_i(t)dt$. Obviously, we have $0 < \nu_i \leqslant 1$.*

Remark: If a sender i's traffic is sent in TCP, ν_i depends on TCP's congestion control efficiency. Given enough buffers at the congested router, this ν_i is close to 1 in typical scenarios, because NetFence uses a control interval $I_{lim} = 2s$ for rate limit adjustment, which is typically one order of magnitude larger than the end-to-end RTT (on average 100∼200ms in the Internet). We might have a low ν_i (*e.g.*, less than 50%) under very low/high bandwidth-delay product networks where TCP is inefficient. If a sender i's traffic is sent in UDP, ν_i depends on the host sending rate $r_i^h(t)$ during the control interval I_{lim}. For a source that sends as fast as allowed by the rate limiter, we have $\nu_i = 1$; for an on-off traffic, ν_i is decided by its duty cycle.

Lemma 3. *For any host s with sufficient demand, its rate limit will eventually be the highest among all the rate limits,* i.e., $r_s^a = \max_i(r_i^a)$.

Proof. We first show that if all the hosts have sufficient demand, they will eventually reach the same r_i^a. This is due to the AIMD rule. As we have shown in Section A.1, MD keeps fairness unchanged, but AI increases it. Under NetFence, whenever MD occurs, there is at least one AI round after it. This is because if there were only MD

rounds, the aggregated rate limit would eventually become very small, and then the bottleneck link would not be congested, which would lead to an AI round. Therefore, the rate limit fairness increases over time, and eventually all the hosts will have the same r_i^a. For a host without sufficient demand, NetFence punishes it by not increasing or even decreasing its rate limit (§ 6.3.4). Therefore, the host can only get a rate limit lower than one with sufficient demand. □

Assumption 4. *Congestion detection signals if and only if aggregate demand surpasses bottleneck link capacity,* i.e., *when* $\sum_i \overline{r_i} \geqslant C$.

Remark: In practice, this assumption is often true for load-based congestion detection schemes that signal congestion when the average traffic arrival rate exceeds a high-load threshold. Queue-based congestion detection (like RED) is more sensitive to traffic dynamics, possibly signaling congestion due to bursty traffic even if the link average utilization in one control interval is low. However, in the case of NetFence, according to our simulations, queue-based congestion detection satisfies this assumption well. This is because each sender's traffic is shaped by a rate limiter before it enters a bottleneck, which significantly limits the peak rate of the aggregate incoming traffic.

Theorem 5. *Given G good and B bad senders sharing a bottleneck link of capacity C, regardless of attack strategy, any good sender g with sufficient demand eventually obtains a fair share* $\frac{\nu_g \rho C}{G+B}$ *where* $\rho = (1-\delta)^3$.

Proof. During a congested control interval, we have $C \leqslant \sum_i \overline{r_i} \leqslant \sum_i r_i^a \leqslant \sum_i \max_i(r_i^a) = r_g^a(G+B)$ and thus $r_g^a \geqslant \frac{C}{G+B}$. For any uncongested control interval, the rate limit is reduced from the peak rate at the congested control interval, so $r_g^a \geqslant \frac{\rho C}{G+B}$ where $\rho = (1-\delta)^3$. This particular ρ value is due to the fact that our rate limit adjustment may apply one MD cut to make the link not congested

and then at most two more extra MD cuts to prevent malicious senders from hiding $L^{\downarrow}$ feedbacks. Therefore, we have, for all the control intervals in the steady state, $\overline{r_g} = \nu_g r_g^a \geqslant \frac{\nu_g \rho C}{G+B}$. $\square$

Remark: For a multi-homed host h_m that uses the bottleneck link l via multiple access routers, there is one rate limiter (h_m, l) in *each* access router. The host's bandwidth share can thus be higher than a single-homed host. This is not a big issue since the number of access routers of any multi-homed host is often small (*e.g.*, 2). It is also reasonable since the host pays for multiple access links.

Appendix B

Alternative Designs of NetFence to Handle Multiple Bottleneck Scenarios

Next we present two possible solutions to improve NetFence's performance in the multiple bottleneck case as described in § 6.3.5.

B.1 Multi-bottleneck Feedback in One Packet

A clean solution to handle multiple bottlenecks is to let a single packet carry the congestion policing feedback from multiple bottleneck links on a forwarding path. Compared to NetFence's core design described in § 6, this solution requires a new NetFence header format, a new set of algorithms to stamp and verify feedback, and a new regular packet policing algorithm; other parts of NetFence's core design do not need to change.

B.1.1 Multi-bottleneck Feedback in a NetFence Header

A NetFence header may carry the congestion policing feedback from zero or more bottleneck links on the packet's forwarding path. A link L may stamp one of two types of feedback: $L^{\uparrow}$ or $L^{\downarrow}$. Their meanings are the same as defined in § 6.1. The ith

bottleneck's feedback in a packet is encoded in two fields: $link_i$ and $action_i$, whose values are the same as defined in § 6.4.

All the feedback fields in a NetFence header are made unforgeable by a single *token* field, as we will show soon. A NetFence header also contains a single timestamp field ts to indicate the freshness of all the feedback.

If a packet does not traverse any bottleneck link, its NetFence header will only contain a ts field and a valid *token* field stamped by its access router. We refer to such a NetFence header as containing the *nop* feedback.

B.1.2 Stamping and Verifying Feedback

When a packet leaves its access router, the access router stamps the *nop* feedback. That is, it sets ts to its local time and computes the *token* field as follows:

$$token_{nop} = MAC_{K_a}(src, dst, ts) \tag{B.1}$$

K_a is the same as defined in § 6.4.

When the packet traverses a bottleneck link L in *mon* state, the bottleneck router inserts its feedback into the NetFence header. It inserts $L^{\downarrow}$ if the algorithm in § 6.3.4 determines to do so, or $L^{\uparrow}$ otherwise. The bottleneck router updates the *token* field as follows:

$$token = MAC_{K_{ai}}(src, dst, ts, L, action, token) \tag{B.2}$$

K_{ai} is the same as defined in § 6.4. The computation of the new *token* value covers the old *token* value to prevent a malicious downstream router from tampering the feedback stamped by other bottleneck links.

When an access router verifies a NetFence header, it first checks the ts field as described in § 6.4. Then it reconstructs the original $token_{nop}$ using Eq (B.1), and recomputes the *token* field using Eq (B.2). When the packet carries feedback

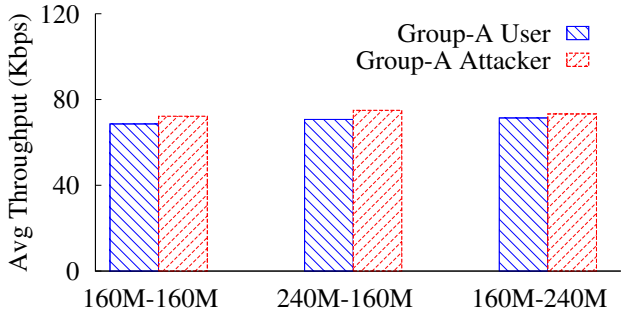


FIGURE B.1: **Sender throughput (Kbps) with the same simulation setting as in Figure 10.8, but with each packet carrying congestion policing feedback from multiple bottleneck links. The fair share rate for each sender is 80Kbps.**

from multiple bottlenecks, the router will have to apply Eq (B.2) multiple times to calculate the final *token* value.

B.1.3 Policing Regular Packets

The multi-bottleneck feedback in a regular packet clearly indicates the bottleneck links on the packet's forwarding path. An access router rate-limits a regular packet with all the rate limiters each associated with one on-path bottleneck link. If the packet cannot pass any of them, it is discarded. This algorithm prevents the rate limit for a flow from changing abruptly, and it also ensures that the flow's sending rate is smaller than the lowest rate limit for all the on-path bottleneck links.

B.1.4 Simulation Results

Figure B.1 shows the simulation results when a packet can carry multi-bottleneck feedback. The simulation setting is the same as in Figure 10.8. As can be seen, each sender in Group-A can get roughly its fair share of bottleneck bandwidth.

B.2 Inferring Rate Limiters

The solution in Appendix B.1 requires a new NetFence header format, which may be longer than that in the core NetFence design described in § 6. An alternative design

is that a packet still carries the congestion policing feedback from a single bottleneck, but an access router can infer the bottleneck links a flow traverses and police the flow's packets using all the corresponding rate limiters. This solution is compatible with NetFence's core design, because each access router can independently deploy it.

B.2.1 Inferring On-path Rate Limiters

An access router can keep a per-destination-prefix cache that records what bottleneck links are on the path towards a particular prefix. Whenever an access router receives $L^{\uparrow}$ or $L^{\downarrow}$ feedback from a sender, it adds L into the cache entry associated with the destination prefix. L may be removed from the cache entry if all the rate limiters for L have been removed or the $L^{\uparrow}$ and $L^{\downarrow}$ feedback has not appeared in packets toward the destination prefix for a long period of time.

The number of entries in the inference cache is at most the number of prefixes in a full BGP table. An access router also needs the prefix list of a full BGP table in order to locate the cache entry given the destination IP address in a packet.

B.2.2 Policing Regular Packets

Based on the inference cache, an access router can pass a packet with *mon* feedback through all the rate limiters associated with the bottleneck links on the packet's forwarding path. If the packet cannot pass any of them, it is discarded.

The inference cache may be inaccurate: a link L may stay in the cache entry of a particular prefix even after it is no longer in *mon* state or on the path toward the prefix. In this case, traffic toward the destination prefix may be unnecessarily throttled by the rate limiters associated with L. However,this is not a big problem because such cases only occur infrequently, and L will be removed from the cache entry after traffic toward the destination prefix no longer carries the $L^{\uparrow}$ and $L^{\downarrow}$ feedback for some time.

When a packet with the $L^{\uparrow}$ or $L^{\downarrow}$ feedback passes all the associated rate limiters, let (src, L_{low}) denote the one with the smallest rate limit. Unlike the algorithm described in § 6.3.3, the access router will reset the feedback to $L^{\uparrow}_{low}$.

B.2.3 Inferring Rate Limits

With the rate limiter inference cache, the single-bottleneck feedback in a packet may be used to infer the feedback from other bottleneck links on the packet's forwarding path. If a packet carries the $L^{\uparrow}$ feedback, other on-path bottleneck links must not be congested at this moment, because otherwise the $L^{\uparrow}$ feedback would have been replaced with an $L^{*\downarrow}$ feedback; on the other hand, if a packet carries an $L^{\downarrow}$ feedback, other on-path bottleneck links will not be able to stamp their feedbacks, and therefore the rate limits for them should be temporarily held unchanged.

With this inference algorithm, the rate limit adjustment algorithm in § 6.3.4 needs to be updated accordingly. In addition to t_s and $hasIncr$, a rate limiter (src, L) is also associated with three more state variables: $hasIncr*$, $isActive$, and $isActive*$. $hasIncr*$ records whether the rate limiter has seen the $L^{*\uparrow}$ feedback with a timestamp newer than t_s; $isActive$ records whether the rate limiter has seen any $L^{\uparrow}$ or $L^{\downarrow}$ feedback regardless of the timestamp value; and $isActive*$ records whether the rate limiter has seen any $L^{*\uparrow}$ or $L^{*\downarrow}$ feedback. At the end of each control interval I_{lim}, the rate limiter (src, L)'s rate limit r_{lim} is adjusted as follows:

1. If $hasIncr$ or $hasIncr*$ is true, the rate limiter's throughput r_{tput} is compared with $\frac{1}{2}r_{lim}$: r_{lim} is increased by Δ if $r_{tput} \geqslant r_{lim}$;

2. Otherwise, if $isActive$ is true, r_{lim} is decreased to $(1-\delta)r_{lim}$;

3. Otherwise, if $isActive*$ is true, r_{lim} is kept unchanged;

4. Otherwise, r_{lim} is decreased to $(1-\delta)r_{lim}$.

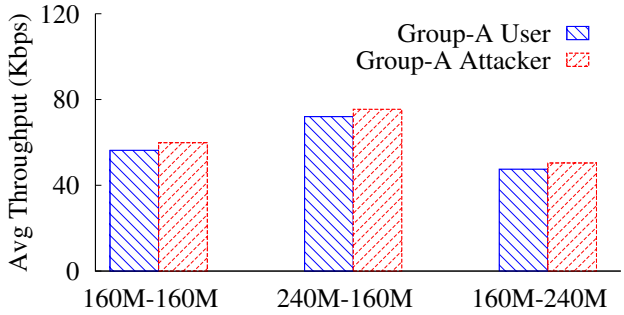


FIGURE B.2: **Sender throughput (Kbps) with the same simulation setting as in Figure 10.8 but also with rate limiter inference. The fair share rate for each sender is 80Kbps.**

B.2.4 Simulation Results

Figure B.2 shows the simulation results with the above rate limiter inference algorithm. The simulation setting is the same as in Figure 10.8 and Figure B.1. We can see that with this algorithm, TCP senders and attackers in Group-A have roughly the same throughput, because the rate limit of a flow no longer jumps between significantly different values. The throughput of a Group-A sender is improved compared to Figure 10.8; however, it may still be much smaller than its fair share. This is because for any Group-A sender src, the rate limits of both rate limiters (src, L_1) and (src, L_2) can only increase when neither bottleneck links is congested. This is a fundamental limitation of embedding single-bottleneck feedback in a packet: the packet simply cannot carry enough congestion information.

Appendix C

Pseudo Code

This appendix includes the pseudo code of the key procedures of Passport and NetFence.

```
1: procedure PROCESS_PACKET(pkt)
2:     if HasPassport(pkt) then
3:         if FromOtherDomain(pkt) then
4:             return verify_passport(pkt)
5:         else if FromCustomerHost(pkt) then
6:             return pkt, DROP
7:         else
8:             return pkt, PASS
9:         end if
10:    else
11:        if FromOtherDomain(pkt) then
12:            if SrcAddr(pkt) ∈ MyAddrSpace() then
13:                return pkt, DROP
14:            end if
15:            if SrcDomainIsDeployed(pkt) then
16:                if DstDomainIsDeployed(pkt) then
17:                    return pkt, DROP
18:                else
19:                    return pkt, DEMOTE
20:                end if
21:            end if
22:        else if FromCustomerHost(pkt) then
23:            if SrcAddr(pkt) ∉ CustomerAddr(pkt) then
24:                return pkt, DROP
25:            end if
26:        end if
27:        if ToOtherDomain(pkt) then
28:            if SrcAddr(pkt) ∉ MyAddrSpace() then
29:                return pkt, PASS
30:            else
31:                return generate_passport(pkt)
32:            end if
33:        else
34:            return pkt, PASS
35:        end if
36:    end if
37: end procedure
```

FIGURE C.1: **Pseudo-code describing how a Passport-enabled router at the administrative boundary (*e.g.*, between ASes, between an AS and its own customer) processes a received packet.**

```
procedure GENERATE_PASSPORT(pkt)
    dst ← DstAddr(pkt)
    path ← ASPathTo(dst)
    if DstDomainIsDeployed(pkt) then
        newpkt ← pkt
    else
        last ← LastDeployedDomain(path)
        newdst ← Decapsulator(last)
        newpkt ← Encapsulate(pkt, newdst)
    end if
    ppt ← NewPassport(newpkt)
    for each AS_i ∈ path do
        if AS_i supports Passport then
            key ← GetKeyWith(AS_i)
            if AS_i = DstDomain(pkt) then
                prev ← PrevAS(path, AS_i)
                mac ← GetMAC(newpkt, prev, key)
            else
                mac ← GetMAC(newpkt, key)
            end if
            AppendMAC(ppt, mac)
        end if
    end for
    InsertPassport(newpkt, ppt)
    return newpkt, PASS
end procedure
```

FIGURE C.2: **Pseudo-code for generating a Passport header.**

```
1: procedure VERIFY_PASSPORT(pkt)
2:     if PassportDemoted(pkt) and not ToMyDomain(pkt) then
3:         return pkt, DEMOTE
4:     end if
5:     if not DstDomainIsDeployed(pkt) then
6:         return pkt, DROP
7:     end if
8:     dst ← DstAddr(pkt)
9:     prev ← GetIncomingAS(pkt)
10:    if IsEncapsulated(pkt) then
11:        odst ← EncapsulatedDstAddr(pkt)
12:        if not DstAnnouncedToAS(odst, prev) then
13:            return pkt, DROP
14:        end if
15:        if DstDomain(pkt) ∉ ASPathTo(odst) then
16:            return Decapsulate(pkt), DEMOTE
17:        end if
18:    end if
19:    key ← GetKeyWith(SrcDomain(pkt))
20:    if ToMyDomain(pkt) then
21:        mac ← GetMAC(pkt, key)
22:    else
23:        mac ← GetMAC(pkt, prev, key)
24:    end if
25:    if mac ≠ MACToVerify(pkt) then
26:        if ToMyDomain(pkt) then
27:            return pkt, DROP
28:        else
29:            DemotePassport(pkt)
30:            return pkt, DEMOTE
31:        end if
32:    end if
33:    if dst = MyDecapsulator() then
34:        pkt ← Decapsulate(pkt)
35:    else if ToMyDomain(pkt) then
36:        StripOffPassport(pkt)
37:    else
38:        UpdatePassport(pkt)
39:    end if
40:    return pkt, PASS
41: end procedure
```

FIGURE C.3: **Pseudo-code for verifying a Passport header.**

```
1: procedure RATE_LIMITER.RATE_LIMIT_REQUEST_PACKET(pkt)
2:     hdr ← get_netfence_header(pkt)
3:     if hdr.priority == 0 then
4:         return PASS
5:     end if
6:     ts_now ← get_current_time()
7:     token_now ← m_token_init + (ts_now − m_ts_init) ∗ m_rate_init
8:     token_remove ← 2^(hdr.priority−1)
9:     if token_remove > token_now then
10:        return DROP
11:    end if
12:    if token_now > m_depth_init then
13:        token_now ← m_depth_init
14:    end if
15:    m_token_init ← token_now − token_remove
16:    if m_token_init < 0 then
17:        m_token_init ← 0
18:    end if
19:    m_ts_init ← ts_now
20:    return PASS
21: end procedure
```

FIGURE C.4: **NetFence request packet rate limiting pseudo-code.** m_* **are member variables of the rate limiter.**

```
1: procedure RATE_LIMITER.RATE_LIMIT_REGULAR_PACKET(pkt)
2:     rate_limiter.update_outgoing_rate(pkt.len)
3:     r ← rate_limiter.cache_packet(pkt)
4:     if r == CACHED then
5:         return CACHED
6:     else if r == DROP then
7:         return DROP
8:     end if
9:     return PASS
10: end procedure
11: procedure RATE_LIMITER.CACHE_PACKET(pkt)
12:     ts_now ← get_current_time()
13:     if m_cache.size_pkts()== 0 then
14:         if (ts_now − m_ts_depart_regular) ∗ m_rate_regular ⩾ pkt.len ∗ 8 then
15:             m_ts_depart_regular ← ts_now
16:             return PASS
17:         else if caching_delay_too_long(pkt) then
18:             return DROP
19:         end if
20:     end if
21:     m_cache.enque(pkt)
22:     if m_cache.size_pkts()== 1 then
23:         rate_limiter.schedule_next_unleash()
24:     end if
25:     return CACHED
26: end procedure
27: procedure RATE_LIMITER.UNLEASH_PACKET( )
28:     if m_cache.size_pkts()== 0 then
29:         return NULL
30:     end if
31:     pkt ← m_cache.deque()
32:     m_ts_depart_regular ← get_current_time()
33:     if m_cache.size_pkts()> 0 then
34:         rate_limiter.schedule_next_unleash()
35:     end if
36:     return pkt
37: end procedure
```

FIGURE C.5: **NetFence regular packet rate limiting pseudo-code.** m_* **are member variables of the rate limiter.**

```
1: procedure RATE_LIMITER.UPDATE_STATUS(pkt)
2:     hdr ← get_netfence_header(pkt)
3:     ts_pkt ← recover_timestamp_from_packet(hdr)
4:     if ts_pkt ⩾ m_t_s then
5:         if hdr.action == INCR then
6:             m_hasIncr ← TRUE
7:         end if
8:     end if
9: end procedure
10: procedure RATE_LIMITER.ADJUST_RATE_LIMIT( )
11:     action ← KEEP
12:     rate_old ← m_rate_regular
13:     if m_hasIncr then
14:         if rate_limiter.get_outgoing_rate()> m_rate_regular/2 then
15:             action ← INCREASE
16:         end if
17:     else
18:         action ← DECREASE
19:     end if
20:     if action == INCREASE then
21:         m_rate_regular ← m_rate_regular + Δ
22:     else if action == DECREASE then
23:         m_rate_regular ← m_rate_regular * (1 − δ)
24:     end if
25:     if action ≠ KEEP then
26:         rate_limiter.update_packet_cache(rate_old)
27:     end if
28:     m_hasIncr ← FALSE
29:     m_t_s ← get_current_time_in_seconds()
30:     rate_limiter.schedule_next_rate_adjustment()
31: end procedure
```

FIGURE C.6: **NetFence rate limit adjustment pseudo-code.** m_* **are member variables of the rate limiter.**

```
1: procedure ROUTER.FORWARD_PACKET(pkt)
2:     q ← router.find_output_queue(pkt)
3:     if q == NULL then
4:         discard_packet(pkt)
5:         return
6:     end if
7:     if is_legacy_packet(pkt) then
8:         q.enque(pkt)
9:         return
10:    end if
11:    if router.is_from_my_hosts(pkt) then
12:        r ← router.rate_limit_packet(pkt)
13:        if r == PASS then
14:            q.enque(pkt)
15:        else if r == DROP then
16:            discard_packet(pkt)
17:        end if
18:    end if
19: end procedure
20: procedure ROUTER.RATE_LIMIT_PACKET(pkt)
21:    hdr ← get_netfence_header(pkt)
22:    if hdr.mac == 0 or not router.mac_is_valid(pkt) then
23:        r ← router.get_init_pkt_rate_limiter(pkt)
24:        if r.rate_limit_init_packet(pkt)== DROP then
25:            return DROP
26:        end if
27:    else
28:        if hdr.linkid ≠ 0 then
29:            r ← router.get_regular_pkt_rate_limiter(pkt)
30:            r.update_status(pkt)
31:            x ← r.rate_limit_regular_packet(pkt)
32:            if x == DROP then
33:                return DROP
34:            else if x == CACHED then
35:                return CACHED
36:            end if
37:        end if
38:    end if
39:    router.update_packet(pkt)
40:    return PASS
41: end procedure
```

FIGURE C.7: **NetFence packet forwarding pseudo-code.**

```
1: procedure QUEUE.DEQUE_REGULAR_PACKET( )
2:     pkt ← queue.pick_next_regular_packet_to_deque()
3:     q ← queue.get_regular_queue(pkt)
4:     if q.ts_mon ⩾ 0 then
5:         queue.update_netfence_header(pkt)
6:     end if
7:     return pkt
8: end procedure
9: procedure QUEUE.CHECK_PACKET_LOSS( )
10:     ts_now ← get_current_time()
11:     qset ← queue.get_regular_queue_set()
12:     for q in qset do
13:         dr ← q.get_drop_rate()/q.get_deque_rate()
14:         q.drop_rate ← q.drop_rate ∗ 0.9 + dr ∗ 0.1
15:         if q.drop_rate > p_th then
16:             q.ts_mon ← ts_now
17:         else if q.ts_mon > 0 and ts_now − q.ts_mon > T_recover then
18:             q.ts_mon ← −1
19:         end if
20:     end for
21: end procedure
```

FIGURE C.8: **Pseudo-code showing how a bottleneck router may update a packet's congestion policing feedback.**

Bibliography

[1] CAIDA Report on Packet Size Distribution. `http://www.caida.org/analysis/AIX/plen_hist/`, 2000.

[2] D. Andersen. Mayday: Distributed Filtering for Internet Services. In *USENIX USITS*, 2003.

[3] D. G. Andersen, H. Balakrishnan, N. Feamster, T. Koponen, D. Moon, and S. Shenker. Accountable Internet Protocol (AIP). In *ACM SIGCOMM*, 2008.

[4] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet Denial of Service with Capabilities. In *ACM HotNets-II*, 2003.

[5] Arbor Networks. Worldwide Infrastructure Security Report, Volume VI. `http://www.arbornetworks.com/report`, 2010.

[6] K. Argyraki and D. R. Cheriton. Active Internet Traffic Filtering: Real-Time Response to Denial-of-Service Attacks. In *USENIX*, 2005.

[7] F. Baker and P. Savola. Ingress Filtering for Multihomed Networks. RFC 3704, 2004.

[8] I. Baldine, Y. Xin, A. Mandal, P. Ruth, A. Yumerefendi, and J. Chase. ExoGENI: A Multi-Domain Infrastructure-as-a-Service Testbed. In *TridentCom: International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*, June 2012.

[9] H. Ballani, Y. Chawathe, S. Ratnasamy, T. Roscoe, and S. Shenker. Off by default! In *ACM Hotnets-IV*, 2005.

[10] T. Bates, E. Chen, and R. Chandra. BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP). RFC 4456, 2006.

[11] BGP Routing Table Statistics. `http://bgp.potaroo.net/as6447/`, 2010.

[12] J. Black, S. Halevi, H. Krawczyk, T. Krovetz, and P. Rogaway. UMAC: Fast and Secure Message Authentication. In *CRYPTO*, 1999.

[13] A. Bremler-Barr and H. Levy. Spoofing Prevention Method. In *IEEE INFOCOM*, 2005.

[14] B. Briscoe, A. Jacquet, C. D. Cairano-Gilfedder, A. Salvatori, A. Soppera, and M. Koyabe. Policing Congestion Response in an Internetwork using Re-feedback. In *ACM SIGCOMM*, 2005.

[15] B. Briscoe, A. Jacquet, T. Moncaster, and A. Smith. Re-ECN: A Framework for Adding Congestion Accountability to TCP/IP. `http://tools.ietf.org/id/draft-briscoe-tsvwg-re-ecn-tcp-motivation-01.txt`, 2009.

[16] M. Casado, A. Akella, P. Cao, N. Provos, and S. Shenker. Cookies Along Trust-Boundaries (CAT): Accurate and Deployable Flood Protection. In *USENIX SRUTI*, 2006.

[17] M. Casado, P. Cao, A. Akella, and N. Provos. Flow-Cookies: Using Bandwidth Amplification to Defend Against DDoS Flooding Attacks. In *IWQoS*, 2006.

[18] D.-M. Chiu and R. Jain. Analysis of the Increase and Decrease Algorithms for Congestion Avoidance in Computer Networks. *Comput. Netw. ISDN Syst.*, 17(1), 1989.

[19] CSS Routing and Bridging Configuration Guide. `http://www.cisco.com/en/US/docs/app_ntwk_services/data_center_app_services/css11500series/v7.30/configuration/routing/guide/IP.html`, 2010.

[20] J. Crowcroft, T. Deegan, C. Kreibich, R. Mortier, and N. Weaver. Lazy Susan: Dumb Waiting as Proof of Work. Technical Report UCAM-CL-TR-703, University of Cambridge, Computer Laboratory, 2007.

[21] Deterlab. `http://www.deterlab.net/`, 2010.

[22] W. Diffie and M. E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 1976.

[23] C. Dixon, A. Krishnamurthy, and T. Anderson. Phalanx: Withstanding Multimillion-node Botnets. In *USENIX/ACM NSDI*, 2008.

[24] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism To Scale Software Routers. In *ACM Symposium on Operating Systems Principles (SOSP)*, 2009.

[25] Z. Duan, X. Yuan, and J. Chandrashekar. Constructing Inter-Domain Packet Filters to Control IP Spoofing Based on BGP Updates. In *IEEE INFOCOM*, 2006.

[26] D. Estrin, J. C. Mogul, G. Tsudik., and K.Anand. Visa Protocols for Controlling Inter-Organization Datagram Flow. *IEEE JSAC*, 1989.

[27] F-Secure. Calculating the Size of the Downadup Outbreak. `http://www.f-secure.com/weblog/archives/00001584.html`, 2009.

[28] P. Ferguson and D. Senie. Network Ingress Filtering: Defeating Denial of Service Attacks which Employ IP Source Address Spoofing. RFC 2827, May 2000.

[29] S. Floyd and V. Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM ToN*, 1(4), 1993.

[30] K. Foster. Application of BGP Communities. *The Internet Protocol Journal*, 6(2), 2003.

[31] L. Garber. Denial-of-Service Attacks Rip the Internet. *IEEE Computer Magazine*, 33, 2000.

[32] S. Gibson. Distributed reflection denial of service. `http://www.cs.washington.edu/homes/arvind/cs425/doc/drdos.pdf`, 2002.

[33] S. Gibson. The Strange Tale of the Attacks Against GRC.COM. `http://knight.segfaults.net/EE579/grcdos.pdf`, 2002.

[34] A. Greenhalgh, M. Handley, and F. Huici. Using routing and tunneling to combat DoS attacks. In *USENIX SRUTI*, 2005.

[35] M. Handley, E. Kohler, A. Ghosh, O. Hodson, and P. Radoslavov. Designing Extensible IP Router Software. In *USENIX/ACM NSDI*, 2005.

[36] J. Hawkinson and T. Bates. Guidelines for creation, selection, and registration of an Autonomous System (AS). RFC 1930, 1996.

[37] Helion Technology. AES Cores. `http://www.heliontech.com/aes.htm`, 2010.

[38] Intel AES Instructions Set. `http://software.intel.com/en-us/articles/intel-advanced-encryption-standard-aes-instructions-set/`, 2010.

[39] C. Jin, H. Wang, and K. G. Shin. Hop-Count Filtering: An Effective Defense Against Spoofed DDoS Traffic. In *ACM CCS*, 2003.

[40] A. Keromytis, V. Misra, and D. Rubenstein. SOS: Secure Overlay Services. In *ACM SIGCOMM*, 2002.

[41] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click Modular Router. *ACM TOCS*, 18(3), 2000.

[42] C. Kreibich, A. Warfield, J. Crowcroft, S. Hand, and I. Pratt. Using Packet Symmetry to Curtail Malicious Traffic. In *ACM Hotnets-IV*, 2005.

[43] T. Krovetz. UMAC: Message Authentication Code using Universal Hashing. RFC 4418, 2006.

[44] C. Labovitz. Round 2: DDoS Versus Wikileaks. `http://asert.arbornetworks.com/2010/11/round2-ddos-versus-wikileaks`, 2010.

[45] E. Larkin. Storm Worm's Virulence may Change Tactics. `http://www.networkworld.com/news/2007/080207-black-hat-storm-worms-virulence.html`, 2007.

[46] R. Lemos. Attack knocks out Microsoft Web sites. `http://news.cnet.com/2100-1001-251573.html`, 2001.

[47] R. Lemos. Bots Surge Ahead in March. `http://www.securityfocus.com/brief/466`, 2007.

[48] A. K. Lenstra and E. R. Verheul. Selecting Cryptographic Key Sizes. *Lecture Notes in Computer Science*, 1751:446–265, 2000.

[49] J. Leyden. Duo charged over DDoS for hire scam. `http://www.theregister.co.uk/2005/03/22/ddos_for_hire_plot_arrests`, 2005.

[50] J. Li, J. Mirkovic, M. Wang, P. Reiher, and L. Zhang. SAVE: Source Address Validity Enforcement. In *IEEE INFOCOM*, 2002.

[51] X. Liu, A. Li, X. Yang, and D. Wetherall. Passport: Secure and Adoptable Source Authentication. In *USENIX/ACM NSDI*, 2008.

[52] X. Liu, X. Yang, and Y. Lu. StopIt: Mitigating DoS Flooding Attacks from Multi-Million Botnets. Technical Report 08-05, UC Irvine, 2008.

[53] X. Liu, X. Yang, and Y. Lu. To Filter or to Authorize: Network-Layer DoS Defense Against Multimillion-node Botnets. In *ACM SIGCOMM*, 2008.

[54] X. Liu, X. Yang, D. Wetherall, and T. Anderson. Efficient and Secure Source Authentication with Packet Passports. In *USENIX SRUTI*, 2006.

[55] S. Luo and G. A. Marin. Realistic Internet Traffic Simulation through Mixture Modeling and A Case Study. In *Winter Simulation Conference*, 2005.

[56] D. Magoni and J. Pansiot. Analysis of the Autonomous System Network Topology. *ACM SIGCOMM CCR*, 31(3), July 2001.

[57] R. Mahajan, S. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling High Bandwidth Aggregates in the Network. *ACM SIGCOMM CCR*, 32(3), 2002.

[58] R. Mahajan, S. Floyd, and D. Wetherall. Controlling High-Bandwidth Flows at the Congested Router. In *IEEE ICNP*, 2001.

[59] A. Mahimkar, J. Dange, V. Shmatikov, H. Vin, and Y. Zhang. dFence: Transparent Network-based Denial of Service Mitigation. In *USENIX/ACM NSDI*, 2007.

[60] Z. M. Mao, J. Rexford, J. Wang, and R. Katz. Towards an Accurate AS-Level Traceroute Tool. In *ACM SIGCOMM*, 2003.

[61] M. Mathis, J. Semke, J. Mahdavi, and T. Ott. The Macroscopic Behavior of the TCP Congestion Avoidance Algorithm. *ACM SIGCOMM CCR*, 27(3), 1997.

[62] D. McGuire. DDoS Attacks Still Pose Threat to Internet. `http://www.mail-archive.com/isn@attrition.org/msg02208.html`, 2003.

[63] R. Moskowitz, P. Nikander, P. Jokela, and T. Henderson. Host Identity Protocol. RFC 5201, 2008.

[64] J. Naous, G. Gibb, S. Bolouki, and N. McKeown. NetFPGA: Reusable Router Architecture for Experimental Research. In *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, 2008.

[65] J. Nazario. Estonian DDoS Attacks – A Summary to Date. `http://asert.arbornetworks.com/2007/05/estonian-ddos-attacks-a-summary-to-date`, 2007.

[66] J. Nazario. Georgia DDoS Attacks – A Quick Summary of Observations. `http://asert.arbornetworks.com/2008/08/georgia-ddos-attacks-a-quick-summary-of-observations`, 2008.

[67] K. Nichols, S. Blake, F. Baker, and D. Black. Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers. RFC 2474, 1998.

[68] E. Nordmark. Stateless IP/ICMP Translation Algorithm (SIIT). RFC 2765, 2000.

[69] D. Pappalardo and E. Messmer. Extortion via DDoS on the rise. `http://www.networkworld.com/news/2005/051605-ddos-extortion.html`, 2005.

[70] K. Park and H. Lee. On the Effectiveness of Route-Based Packet Filtering for Distributed DoS Attack Prevention in Power-Law Internets. In *ACM SIGCOMM*, 2001.

[71] B. Parno, D. Wendlandt, E. Shi, A. Perrig, B. Maggs, and Y.-C. Hu. Portcullis: Protecting Connection Setup from Denial-of-Capability Attacks. In *ACM SIGCOMM*, 2007.

[72] D. Pauli. PayPal hit by DDoS attack after dropping Wikileaks. `http://www.zdnet.com/news/paypal-hit-by-ddos-attack-after-dropping-wikileaks/489237`, 2010.

[73] R. Perlman. Network Layer Protocols with Byzantine Robustness. MIT Ph.D. Thesis, 1988.

[74] A. Perrig, R. Canetti, D. Song, and D. Tygar. Efficient and Secure Source Authentication for Multicast. In *NDSS*, 2001.

[75] K. Ramakrishnan, S. Floyd, and D. Black. The Addition of Explicit Congestion Notification (ECN) to IP. RFC 3168, 2001.

[76] S. Ramasubramanian. Renesys Dissects the Bluesecurity DDoS. `http://www.merit.edu/mail.archives/nanog/2006-05/msg00238.html`, 2006.

[77] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, 2006.

[78] J. Rexford, J. Wang, Z. Xiao, and Y. Zhang. BGP Routing Stability of Popular Destinations. In *ACM IMW, 2002*, 2002.

[79] P. Roberts. Massive Denial Of Service Attack Severs Myanmar From Internet. `http://threatpost.com/en_us/blogs/massive-denial-service-attack-severs-myanmar-internet-110310`, 2010.

[80] S. Savage, D. Wetherall, A. Karlin, and T. Anderson. Practical Network Support for IP Traceback. In *ACM SIGCOMM*, 2000.

[81] F. Scalzo. Recent DNS Reflector Attacks. `http://www.nanog.org/meetings/nanog37/presentations/frank-scalzo.pdf`, 2006.

[82] A. Seehra, J. Naous, M. Walfish, D. Mazieres, A. Nicolosi, and S. Shenker. A Policy Framework for the Future Internet. In *ACM HotNets-VIII*, 2009.

[83] M. Shreedhar and G. Varghese. Efficient Fair Queueing Using Deficit Round Robin. In *ACM SIGCOMM*, 1995.

[84] D. R. Simon, S. Agarwal, and D. A. Maltz. AS-based Accountability as a Cost-effective DDoS Defense. In *USENIX HotBots*, 2007.

[85] A. Snoeren, C. Partridge, L. Sanchez, C. Jones, F. Tchakountio, S. Kent, and W. Strayer. Hash-Based IP Traceback. In *ACM SIGCOMM*, 2001.

[86] D. Song and A. Perrig. Advanced and Authenticated Marking Schemes for IP Traceback. In *IEEE INFOCOM*, 2001.

[87] K. Spiess. Worm 'Storm' Gathers Strength. http://www.neoseeker.com/news/story/7103/, 2007.

[88] The spoofer project. `http://spoofer.csail.mit.edu/`, 2011.

[89] A. Stavrou and A. Keromytis. Countering DoS Attacks with Stateless Multipath Overlays. In *ACM CCS*, 2005.

[90] I. Stoica, S. Shenker, and H. Zhang. Core-Stateless Fair Queueing: a Scalable Architecture to Approximate Fair Bandwidth Allocations in High-Speed Networks. *IEEE/ACM ToN*, 2003.

[91] DDoS Mitigation to the Rescue. `https://www.arbornetworks.com/dmdocuments/DDoS%20Mitigation%20to%20the%20Rescue.pdf`, 2010.

[92] TCPHA Project. `http://dragon.linux-vs.org/~dragonfly/htm/tcpha.htm`, 2011.

[93] A. Tsotsis. WordPress.com Suffers Largest DDoS Attack In Its History. `http://techcrunch.com/2011/03/03/wordpress-com-suffers-major-ddos-attack`, 2011.

[94] J. S. Turner. New Directions in Communications (Or Which Way to the Information Age?). *IEEE Communications Magazine*, 1986.

[95] J. Vijayan. Update: MasterCard, Visa others hit by DDoS attacks over WikiLeaks. `http://www.computerworld.com/s/article/9200521/Update_MasterCard_Visa_others_hit_by_DDoS_attacks_over_WikiLeaks`, 2010.

[96] R. Wesson. Botnets and the Global Infection Rate: Anticipating Security Failures. `http://www.stanford.edu/class/ee380/Abstracts/070606-slides.pdf`, 2007.

[97] Y. Xia, L. Subramanian, I. Stoica, and S. Kalyanaraman. One More Bit is Enough. *IEEE/ACM ToN*, 16(6), 2008.

[98] A. Yaar, A. Perrig, and D. Song. Pi: A Path Identification Mechanism to Defend Against DDoS Attacks. In *IEEE Security Symposium*, 2003.

[99] A. Yaar, A. Perrig, and D. Song. SIFF: A Stateless Internet Flow Filter to Mitigate DDoS Flooding Attacks. In *IEEE Security Symposium*, 2004.

[100] A. Yaar, A. Perrig, and D. Song. FIT: Fast Internet Traceback. In *Proc. of IEEE Infocom*, 2005.

[101] A. Yaar, A. Perrig, and D. Song. StackPi: New Packet Marking and Filtering Mechanisms for DDoS and IP Spoofing Defense. *IEEE JSAC*, 2006.

[102] X. Yang, D. Wetherall, and T. Anderson. A DoS-Limiting Network Architecture. In *ACM SIGCOMM*, 2005.

[103] X. Yang, D. Wetherall, and T. Anderson. TVA: A DoS-limiting Network Architecture. *IEEE/ACM ToN*, 16(6), 2008.

[104] X. Zhao, D. Pei, L. Wang, D. Massey, A. Mankin, S. F. Wu, and L. Zhang. An Analysis of BGP Multiple Origin AS (MOAS) Conflicts. In *ACM IMW*, 2001.

Biography

Xin Liu was born in Chongqing, China on November 16th, 1979. He received a B.S. and a M.S. in Electronic Engineering from Tsinghua University, Beijing and a PhD in Computer Science from Duke University.