ABSTRACT

Title of dissertation: An Accountability Architecture for the Internet

Adam Bender, Doctor of Philosophy, 2010

Dissertation directed by: Bobby Bhattacharjee
Department of Computer Science

In the current Internet, senders are not accountable for the packets they send. As a result, malicious users send unwanted traffic that wastes shared resources and degrades network performance. Stopping such attacks requires identifying the responsible principal and filtering any unwanted traffic it sends. However, senders can obscure their identity: a packet identifies its sender only by the source address, but the Internet Protocol does not enforce that this address be correct. Additionally, affected destinations have no way to prevent the sender from continuing to cause harm.

An accountable network binds sender identities to packets they send for the purpose of holding senders responsible for their traffic. In this dissertation, I present an accountable network-level architecture that strongly binds senders to packets and gives receivers control over who can send traffic to them. Holding senders accountable for their actions would prevent many of the attacks that disrupt the Internet today.

Previous work in attack prevention proposes methods of binding packets to senders, giving receivers control over who sends what to them, or both. However, they all require trusted elements on the forwarding path, to either assist in identifying the sender or to filter unwanted packets. These elements are often not under the

control of the receiver and may become corrupt. This dissertation shows that *the Internet architecture can be extended to allow receivers to block traffic from unwanted senders, even in the presence of malicious devices in the forwarding path.*

This dissertation validates this thesis with three contributions. The first contribution is DNA, a network architecture that strongly binds packets to their sender, allowing routers to reject unaccountable traffic and recipients to block traffic from unwanted senders. Unlike prior work, which trusts on-path devices to behave correctly, the only trusted component in DNA is an identity certification authority. All other entities may misbehave and are either blocked or evicted from the network.

The second contribution is NeighborhoodWatch, a secure, distributed, scalable object store that is capable of withstanding misbehavior by its constituent nodes. DNA uses NeighborhoodWatch to store receiver-specific requests block individual senders.

The third contribution is VanGuard, an accountable capability architecture. Capabilities are small, receiver-generated tokens that grant the sender permission to send traffic to receiver. Existing capability architectures are not accountable, assume a protected channel for obtaining capabilities, and allow on-path devices to steal capabilities. VanGuard builds a capability architecture on top of DNA, preventing capability theft and protecting the capability request channel by allowing receivers to block senders that flood the channel. Once a sender obtains capabilities, it no longer needs to sign traffic, thus allowing greater efficiency than DNA alone.

The DNA architecture demonstrates that it is possible to create an accountable network architecture in which none of the devices on the forwarding path must be trusted. DNA holds senders responsible for their traffic by allowing receivers to block senders; to store this blocking state, DNA relies on the NeighborhoodWatch DHT. VanGuard extends DNA and reduces its overhead by incorporating capabilities, which gives destinations further control over the traffic that sources send to them.

An Accountability Architecture for the Internet

by

Adam Bender

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2010

Advisory Committee:

Professor Bobby Bhattacharjee, Chair
Professor Michael Hicks
Professor Pete Keleher
Professor Neil Spring
Professor Lawrence C. Washington

Acknowledgments

I would first like to thank my advisor, Bobby, without whom this dissertation would not exist. He always encouraged me to do my best and compelled me to achieve more than I thought was possible. He has taught me much about research, and about life. I am grateful for all that I have learned from him. Neil deserves just as much praise for guiding my research and introducing me to new areas, but I am perhaps most thankful for how he has helped me think analytically about all matters of presentation and in so doing become a better writer and speaker. He taught me the value of precision and of the ability to distill arguments to their essence.

Much of my time at Maryland was spent in the lab. There is no other place where I would rather have worked, simply because of the people who shared it with me. Each of them deserves my thanks and praise: Randy, who weathers life's challenges better than anyone I know (despite the occasional little ball of hate); Aaron, who has his quinine moments; Lex, who is awesome; Katrina, who introduced the lab to games; Dave, whose curiosity and sense of humor were an inspiration; Cristian, who taught us much about the world but will be better remembered for what we taught him; and Rob, who will always be a gentleman.

I am also grateful to all of the people whom I've worked with over the years: Ruggero Morselli, Derek Monner (who also deserves credit for introducing me to great music), Nate Goergen, Jonathan Katz, Mike Hicks (who gave me more support than I perhaps deserved), Bjoern Leuttmann, Don Beaver, Pat Stephenson, Walter Willinger, Bala Krishnamurthy, Craig Wills, Mudhakar Srivatsa, and Carl Landis.

Finally, I would like to thank my family for their unwavering support and unconditional love. My parents and sister were always there to encourage me in whatever it was I wanted to do. Meeting Hyunyoung is the best thing that has ever happened to me; that alone would have made grad school worth it. Words cannot express how grateful I am for all that she has done for me.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the early 1970's, computer networks were limited to isolated instances at academic and government sites. The Internet, as it is known today, began as an effort to connect these networks with a single protocol [50]. At the time, threats to communication were external to the network: protocols were designed provide a reliable communication medium even in the case of gateway or network failure [26].

In the forty years since its creation, the Internet has grown exponentially. Commercial networks, as opposed to academic, dominate. There are nearly two billion users worldwide [42]. One of the reasons for this growth is the design decision to make the Internet open: the Internet architecture provides a protocol, TCP/IP, that connects endhosts, and any device or network that can speak that protocol can join the Internet. Communication does not require prior approval; anyone may communicate with anyone.

An open communication model suits the original Internet architecture's goal of providing efficient connectivity, assuming that hosts do not maliciously deviate from protocol. This assumption may have been true when the Internet was designed, but is not true today. As a result, the Internet is susceptible to internal attacks by malicious hosts. Furthermore, hosts can send packets without divulging their identity, allowing them to abuse the network with little consequence.

In this dissertation, I present protocols for an accountable network, DNA, which prevents such abuse. An accountable network layer binds sender identities to packets so that the receiver (or some other device) can take action based on the true identity of the sender. Strongly binding senders to packets supports many classes of possible actions. The original impetus for providing accountability, according to the designers of the Internet protocols, was to account for resource usage [26]. Accountability facilitates network auditing, by allowing administrators to determine what principal was responsible for causing a certain action (such as causing a device to fail). In this proposal, I examine the use of accountability to selectively *filter* repeated unwanted, abusive, or non-compliant traffic from malicious sources on a per-destination basis, while permitting traffic from others to proceed. The result is that receivers can control who can send packets to them, and thus who can consume their resources.

1.1 Lack of accountability facilitates abuse

The current Internet is not accountable. The Internet protocols were designed to be resilient to external disruptions—faulty hardware and unreliable links—rather than internal misbehavior. As such, they lack means to both identify senders and hold them responsible. There is no strong identification mechanism built into the Internet protocols. Senders can be identified by their IP address, yet there are no requirements that this address be correct. Similarly, there are no rules that designate what actions a sender can be held accountable for, or who should hold senders responsible. The situation would improve if ISPs were to police their customers, but often they do not have incentive to do so.

As a result, attacks on the Internet infrastructure—the hosts, links, and routers that compose the network—are not only possible but common. Such attacks are called denial of service (DoS) attacks. One example of a DoS attack is a bandwidth flooding attack, in which the attacker (or, more commonly, attackers) sends packets as quickly as it can towards the victim, without regard to how or if the victim responds. The goal is to consume all of the available bandwidth on the victim's access link, so that other, legitimate traffic cannot be delivered. However, even a small amount of attack traffic is sufficient to disrupt communication: TCP's exponential backoff mechanism will reduce throughput drastically when the packet loss rate is above 5% [79].

Adding to the difficulty in preventing DoS attacks is the tendency of attackers to spoof their source address. A victim that can identify an attacker by its IP address could filter traffic from that address with a firewall. Attackers adapted by spoofing their source address to avoid such filters. The result is that attackers are unidentifiable to the destination and therefore cannot be held accountable for their actions.

1.2 Accountability as a solution to DoS attacks

My approach to preventing DoS attacks is motivated by the belief that the victim of the attack, and receivers in general, should have control over the packets they receive. If a victim were able to stop unwanted packets before they arrived, it could recover from the attack. Blocking unwanted packets while permitting desired traffic requires distinguishing between the two, but it is possible that the only difference between wanted and unwanted packets is their source. Currently, the only means to identify senders are by temporal properties, such as IP address or network location. Even if a mechanism to block senders did exist, attacker could evade blocks by changing these properties.

Incorporating accountability into the network solves this problem. An accountable architecture strongly identifies senders with persistent attributes and binds a sender's identity to its packets. Blocking attackers is more effective, as the attackers cannot bypass blocks. Building a mechanism to block senders based on accountable identities satisfies our goal of giving receivers control over the packets they receive, thus mitigating DoS attacks.

Trusted components

Existing proposals to extend the Internet to allow receivers control over the traffic the receive share a common characteristic: they trust some component on the forwarding path to not misbehave. This is undesirable, as these elements are often not under the control of the receiver and may become corrupted. An ideal accountability architecture would not place trust assumptions on any elements of the forwarding path to achieve either identification or filtering. This dissertation proposes the following thesis: *the Internet architecture can be extended to allow receivers to block traffic from unwanted senders, even in the presence of malicious devices on the forwarding path.* I demonstrate this hypothesis by presenting the DNA and VanGuard accountable architectures, showing how they maintains correctness if in-network devices are corrupt, and evaluating their performance with an implementation and simulations.

1.3 Goals of an accountability architecture

Incorporating accountability into the Internet is an immense task. All aspects of the design need to be chosen properly so that the resulting architecture provides the desired properties while incurring minimal overhead and change. Here I discuss the goals we have for an accountability architecture and the reasons they are important.

Trust lies outside the forwarding path

Components of the Internet are operated and maintained by diverse organizations, often with no mutual trust of each other. An accountable Internet architecture should reflect this by trusting as few of the components as possible. DNA minimizes this trust by depending upon only a trusted third party to issue identities and, when necessary, resolve disputes.

If any part of the network deviates from the protocol, DNA defines measures to filter, disconnect, or expunge the offending component. This is in stark contrast to even the most recent DoS prevention techniques and accountability protocols [81, 53, 52, 3], which all rely on some component of the network to correctly enforce the protocol.

Receivers can block senders

Receivers should have control over who sends to them. Accepting a packet from the network consumes resources, and receivers should be able to control allocation of their resources. To give receivers this control, an accountable architecture must allow receivers to block senders. Blocking a sender should prevent any packets from that sender from reaching the destination. By allowing receivers to express this control, DNA gives them control over who consumes their network resources and allows them to protect their networks from attack.

Senders may remain anonymous

Traditionally, accountability and anonymity have been mutually exclusive. Binding senders to strong, persistent identifiers can compromise privacy, by making it easier to track users across multiple connections. An accountable Internet architecture should not provide privacy-compromising adversaries with more abilities than they have today.

DNA uses temporary blinding keys to prevent traffic analysis; this comes at the cost of requiring extra communication to map the temporary key to a permanent key before placing a block. Existing techniques such as onion routing [36] can be used to further mask communication patterns.

Sender identity is unforgeable

In order for blocks to be effective, they must unambiguously identify the entity to be blocked. A sender must not be able to avoid a block because of an imprecise match. Similarly, identities must be long-lived: blocks are ineffective if a principal can mint a new, unblocked identify. DNA uses the strongest identification methods available, namely cryptographic signatures, to provide unforgeable, long-lived identities to senders.

Every destination is protected

Currently, only high-profile sites are able to employ any form of DoS protection, while many others are vulnerable. Some DoS-mitigation proposals also offer protection only to individual sites [23, 48, 4]. Our goal is to provide protection for every host in the network. This motivates building accountability into the network. Changing the fundamental nature of the Internet to provide accountability requires changes at the underlying protocol levels.

Traffic is blocked near the source

There are three places in the network where unwanted traffic can be blocked: near the destination, in the core, or near the sender. By the time the traffic has reached the destination, it is too late: bottleneck links may already be congested.

Filtering traffic in the core is infeasible for several reasons. High-speed routers can only perform cursory checks, router state is limited, and undesired traffic may take many paths through the network, each requiring its own filter.

The only choice left is to block traffic near the source. Doing so not only avoids the above problems, but by dropping packets early, does not waste network resources on transmitting packets that will only be discarded later.

Incremental deployment

The growth of the Internet has led to an ossification of its core protocols. It is no longer possible to coordinate global shifts from one protocol to the next; change must occur gradually. The difficulties and delays encountered in the current attempt to upgrade from IPv4 to IPv6 are an example of this phenomenon. Therefore, new designs must be able to coexist with current protocols. DNA is designed to allow for incremental deployment. Most changes happen at endhosts and in edge networks, allowing for sites to upgrade at their own pace. Networks in the core of the Internet do not have to upgrade simultaneously; accountable packets can be tunneled through non-accountable networks.

As the guarantees DNA offers (specifically, minimal trust assumptions and strong sender identification) are very strong, it is not surprising that DNA incurs costs (in terms of packet size, signature checking and verification, and additional components in the network) greater than those in today's Internet. However, DNA makes design choices (such as using elliptic curve cryptography and the use of waivers) to reduce these costs.

1.4 Thesis outline

In Chapter 2 I give a detailed survey of work related to DNA and VanGuard. I describe previous attempts at preventing and mitigating DoS attacks and to what extent they provide accountability of senders. I focuses on three approaches: capabilities, which routers can use to prioritize packets that the destination has expressed desire in; filters, which, like DNA, allow receivers to block packets from unwanted senders; and alternate designs of an accountable architecture for the Internet. It also discusses work that proposes uses of accountability in networking for purposes other than preventing DoS attacks.

In Chapter 3 I present the design and evaluation of the DNA architecture. It describes the protocol in detail and shows how DNA continues to be effective even when devices on the forwarding path, including senders, routers, and the components of DNA, are corrupted. DNA uses strong cryptography, rather than network-based properties such as point of attachment, to identify senders. The result is that once a sender is blocked, it cannot move or change properties of its network to bypass a filter. DNA includes provisions for allowing senders to remain anonymous, yet still be blockable. This chapter also proposes extensions to decrease the packet-transmission overhead (waivers) and prevent replay attacks at the cost of router state (Bloom filters in routers).

DNA provides the ability for destinations to block senders. These filters must be stored for as long as the destination desires and be readily available in order to determine if a sender is attempting to contact a destination that has blocked it. DNA stores filters in a globally-available distributed hash table (DHT). A DHT is a structured overlay network with a put/get interface that scales efficiently with the size of the network. As such, it is a natural design choice for storing such a large amount of state. However, DHTs are vulnerable a variety of attacks. Chapter 4 presents the design of the NeighborhoodWatch DHT, which prevents known attacks by relying on a trusted authority.

An accountable Internet architecture provides an accountable channel that can be used as a building block for other systems. In Chapter 5 I present one such application. VanGuard is a network architecture that incorporates both accountability and capabilities, which are an efficient way of allowing destinations to tell routers whether traffic is desired. VanGuard is built on DNA and uses it to hold senders accountable for their traffic and allow receivers to block senders. VanGuard ties capabilities to accountability information and treats them as secret values, meaning senders (once they obtain capabilities) do not need to sign packets in order to be held accountable. This allows VanGuard to provide the same benefits as DNA with less overhead.

I conclude my dissertation in Chapter 6, in which I summarize my contributions and suggest areas of future work.

Chapter 2

Related Work

This chapter presents a survey of work related to my thesis, with a focus on network-based DoS protection mechanisms give some amount of control to the receiver. It begins with early proposals based on overlay networks (Section 2.1) and router-placed filters (Section 2.2). Passport, a building block for many later systems, follows (Section 2.3). Capabilities, the approach which has been most-heavily explored in the research literature, is presented in Section 2.4. Filtering systems, the family in which DNA falls, are discussed in Section 2.5. Section 2.6 describes AIP, which shares many of the same goals as DNA, and derivative work. Section 2.7 briefly presents other related systems. I follow with a description of existing DoS-prevention techniques, both Internet practices (Section 2.8) and commercial services (Section 2.9). Finally, Section 2.10 presents other uses of accountability in networking.

Table 2.1 presents a comparison of the pieces of work discussed in this chapter with DNA and VanGuard.

2.1 Overlay-based services

Overlay-based DoS-prevention services leverage a dedicated infrastructure that uses a combination of tunneling and filtering to protect against DoS attacks. They do not aim to solve the general DoS problem, but rather protect individual hosts or services, such as emergency services. The two proposed systems discussed below, SOS and Mayday, have two requirements that make them impractical for wide-area

	Senders identified by	Destination protection mechanism	Mechanism enforced by	Trusted components	Notes
DNA	Signatures	Filtering near source	LRep, source gateway	TAP	Blocks enforced regardless of location
AIP [3]	Self-certifying addresses	Filtering at source	Trusted hardware	Sender hardware, routers, source networks	Requires re-addressing hosts
IPa+ [80]	Passport	None	-	DNSSEC, secure BGP	Passport does not identify senders
StopIt [52]	Passport	Filtering near source	Source gateway	StopIt servers	Passport does not identify senders
AITF [8]	Path according to record route	Filtering near source	Source gateway	Gateways	
AS-based [72]	IP address	Filtering near source	Source gateway	Filter request servers	Relies on ingress filtering
TVA [82]	Path identifier; capabilities	Denying use of privileged channel	Routers	Routers	Fair-queuing to protect request channel
NetFence [53]	Local address; Passport	Rate-limiting	Source gateway	Source gateway	
CAT [22]	Ability to receive traffic at source address	Routing traffic through middlebox	Cookie boxes	Cookie boxes	Cookie boxes unprotected
Passport [55]	Source AS	None	-	BGP, to establish shared keys between ASes	Identifies only the source AS, not the sender
Pushback [58]	-	Rate-limiting	Routers	Routers	
SOS [48], Mayday [4]	Shared key	Filtering at overlay	SOAPs	SOAPs, secret servlets, beacons	

Table 2.1: Summary of related work

DoS protection. First, they require a large, dedicated infrastructure that can authenticate and route a large number of packets on its own. Second, the participants are limited to senders and receivers that exchange secret information before using the overlay.

2.1.1 SOS

The Secure Overlay Service, or SOS, is the first overlay-based DoS-prevention service [48]. SOS protects a single *target* from attack. Surrounded the target is a set of filters, such a firewalls, that create a "filtered region" around the target consisting only of authorized traffic. In order to pass through a filter, traffic must first be routed through an overlay; SOS uses the Chord [75] overlay network.

There are three types of special nodes in the overlay:

1. SOAPs, which will verify senders with an authentication protocol such as IPsec or TLS.

2. Secret servlets, which route traffic directly to the target through the filter. Filters only allow traffic with an IP source address that is one of the secret servlets. By keeping the secret servlets addresses secret, SOS limits the ability of an attacker to spoof a source address that allows his traffic to traverse the filter.

3. Beacons know the identity of secret servlets and route (authenticated) packets towards them.

If any overlay node is attacked or otherwise fails, it simply leaves the overlay. The Chord protocols ensure the overlay recovers. If a secret servlet is attacked or discovered, the target simply selects another and only need inform the beacons of the change.

2.1.2 Mayday

Mayday [4] generalizes SOS by allowing for alternative overlay routing and client authentication methods. The author presents a variety of probing and scanning attacks that can be used to determine the secret values (such as secret servlet IP addresses) used to protect the target. To protect against these attacks, Mayday allows filters to authenticate secret servlets (*egress nodes* in Mayday parlance) by one or more lightweight authenticators, such as a secret value for the destination port or address. While it does solve some deployment issues and formalize the design-space of overlay-based protection, Mayday does not (aim to) offer a service that could be used to protect every end host.

2.2 Pushback

Pushback [58, 43] is a technique in which routers recognize that one of their outbound links are congested and take measures to reduce this congestion. Rather than dropping packets randomly, Pushback attempts to discover an *aggregate*, or well-defined subset of traffic, that is causing congestion. Pushback consists of two mechanisms: a local mechanism, employed by the congested router to identify and control the aggregates, and a pushback mechanism in which that router asks upstream routers to help control the aggregate. Pushback can be though of as a means of aggregate-based congestion control, an intermediate between fine-grained per-flow control and coarse per-packet control.

When a router begins to drop packets due to congestion, it passes relevant information from the dropped traffic to an aggregate congestion control (ACC) agent, which uses clustering to find a small number of aggregates that are responsible for the most congestion. Aggregates can be based on any available fields, such as destination address, source prefix, or port number. Once the router identifies the larger aggregates, it determines the minimum number that must be rate-limited, and to what extent, in order for the over-all drop rate (not including rate limiting of select aggregates) to fall below a threshold.

In order to further reduce congestion at the router, and to limit collateral damage caused by benign traffic that matches the rate-limited aggregate, Pushback propagates aggregates through the network. A router R that drops excessive traffic due to rate limiting will request that its upstream neighbors limit the amount of aggregate-matching traffic sent to R. These requests contain the aggregate and the desired arrival rate for that aggregate. The original router only sends these requests to routers that are responsible for a large amount of the traffic matching the aggregate. Routers that receive pushback messages occasionally send a status message back to the requesting router, so that the latter may decide whether to let the pushback requests expire. Pushback has the advantage that, when high-bandwidth traffic originates from a localized source, only nearby routers need to use rate-limiting.

While Pushback can control flooding attacks in certain situations, it has several drawbacks that prevent it from being an effective, general solution to the DoS problem. First, while aggregation means that routers need to store fewer rules in order to block traffic, this comes at the cost of accuracy. The greater the degree of aggregation, the more benign packets are lost due to collateral damage. Second, as the authors acknowledge, it may not be possible to always find an aggregate that describes the high-bandwidth traffic. In this case, Pushback is no more effective than randomly dropping packets, which is what occurs on congested links anyways. Third, routers decide on their own when to start rate-limiting. Without an explicit (authenticated) notification mechanism, Pushback does not allow destination to block unwanted but non-overwhelming traffic.

2.3 Passport

In Passport [56, 55], ASes use shared secret keys to enable on-path ASes to determine a packet's source AS. Each AS uses routing announcements to share a secret key with each other AS. When a packet is sent, the source AS uses these keys to create a series of secure tokens, one for each AS through which a packet will be forwarded, and places them in outgoing packets. The token created for AS A identifies the source AS to A. Passport is used as a spoofing-prevention mechanism in several other DoS-prevention architectures, including NetFence [53] (discussed in Section 2.4.6), StopIt [52] (discussed in Section 2.5.3), and IPa+ [80] (discussed in Section 2.6.2).

Key exchange in Passport is accomplished via Diffie-Hellman key exchange [30], piggybacked on top of BGP announcements. Passport assumes global parameters p, a prime, and g, a generator mod p. Each AS_i generates a random r_i and computes $b_i = g^{r_i} \bmod p$. AS_i then includes b_i in its BGP announcements. The shared secret key between AS_i and AS_j is $K_{i,j} = b_j^{r_i} \bmod p = b_i^{r_j} \bmod p$. The security of Passport key distribution is therefore tied to the security of the routing system.

When a packet P leaves its source AS AS_1, a border router in that AS determines the sequence, or path, of ASes that P will pass through, using BGP routing tables. For each AS_j on this path, the border router finds (or computes) the secret key $K_{1,j}$ and uses that key to create a token. The format of a token is:

$$t_{1,j} = MAC_{K_{1,j}}(src, dst, len, IP\ ID, payload[0..7], AS_k)$$

where src is the source address, dst is the destination address, len is the packet length, $IP\ ID$ is P's IP identifier, $payload[0..7]$ is the first 8 bytes of P's payload, and AS_k is the AS preceding AS_j in the AS path. The border router places these tokens into the Passport header, a shim header between IP and the transport protocol.

When P arrives at AS_j along the path, a border router in AS_j verifies the corresponding MAC. It determines the source AS, and hence which shared key to use, by examining the source address of the packet and mapping that to an AS via its routing tables. Having obtained the key, the border router computes the MAC over the same fields. If the MACs are equal, the border router forwards the packet. If the comparison fails, however, the source AS is possibly spoofed and the border router performs one of two actions, depending on if AS_j is the destination AS or not. If AS_j is the destination AS, it drops P. Otherwise, the router "demotes" P, that is, it gives a lower priority to forwarding P and sets a bit in P that tells other ASes to do the same. The reason for not dropping P in the middle of the network is that a (temporary) routing variation may have occurred, sending P along a valid path that AS_i did not correctly predict.

Security of key exchange

While relying on existing mechanisms to distribute keys may be efficient, the potential damage caused by routing system failures increases. Diffie-Hellman is vulnerable to a man-in-the-middle attack [45]. An attacker that can replace the public value of one participant in the key exchange protocol can later impersonate that party. This implies that an AS can impersonate any of the ASes for which it provides connectivity. Additionally, an attacker that hijacks a victim's prefix can impersonate the victim, by issuing a new DH value for that AS. This can increase the damage caused by a prefix hijacking attack, which frequently occur today [14]. The only known methods of securing Diffie-Hellman key exchange require either a PKI, which Passport deliberately does not assume.

Trust assumptions

Passport trusts the routing system (in this case, BGP) to securely distribute Diffie-Hellman key exchange values. Passport trusts ASes to police their networks to prevent internal address spoofing. Passport places minimal trust in intermediate routers as they have only a small role: they can forward traffic as normal or demote traffic before forwarding it.

Sender identification

Contrary to the authors' claims that Passport allows routers to validate the source *address* of packets, there are no provisions to prevent address spoofing inside an AS, nor mechanisms to prove that an address is not spoofed. Therefore, Passport allows the validation of only the source *autonomous system*. Because senders are not identified, Passport does not provide a mechanism for receivers to control who can send to them.

2.4 Capabilities

Capability-based architectures split traffic into multiple classes; in general, the two predominant classes are traffic from senders whom the receiver *has* consented to receive traffic from and traffic from senders whom the receiver *has not* consented to receive traffic from. Consent is given in the form of a *capability*: a difficult-to-forge token that is granted to a sender when the receiver responds to an initial request to send traffic. Capabilities are usually created and checked by an in-network device.

2.4.1 Initial proposal

Anderson, Roscoe, and Wetherall [5] were the first to propose using capabilities as a means of DoS prevention. They add two types of entities to the Internet architecture: Request-To-Send (RTS) servers and verification points (VPs). RTS servers help senders obtain capabilities and are co-located with BGP speakers. VPs are deployed near RTSes at network choke points. BGP announcements are extended to include a list of RTS servers along the path to a destination network. A sender obtains tokens by sending a request to the first RTS server on the list, where it is relayed along the list of RTS servers until it reaches the destination. The RTS server overlay is used to prevent flooding: each RTS server rate limits the requests it forward to any prefix.

Once a request arrives at the destination, it must decide whether to grant a capability. Capabilities allow a sender to send n packets in t seconds; these values are system-wide parameters. The destination creates a capability (in fact, a sequence of capabilities) using a reverse hash chain (see Section 2.4.5 for a description of reverse hash chains). The last hash chain value is the first capability, and the destination sends it to the sender through the RTS server overlay. Each RTS server observes the capability and transmits it to its nearby VP, which stores it. The VP ensures that each packet has a capability, that the capability has not been used for more than n packets, or is older than t seconds; this requires that each VP store each capability, its age, and the number of packets send with it. Destinations can issue new capabilities by including the latest unreleased hash value in a response packet, where it can be observed and verified by RTS servers/VPs.

2.4.2 SIFF

SIFF [79] shows how to use capabilities without requiring per-flow state on routers and provides the basis for subsequent work in capabilities [81, 64, 23]. SIFF partitions traffic into two classes: privileged and unprivileged. Endhosts can establish a channel of privileged traffic, which takes precedence over non-privileged packets at routers. Privileged packets carry capabilities that are verified by routers en-route. Routers drop packets that fail verification. Routers also prefer privileged packets over unprivileged, so that floods of unprivileged packets do not affect privileged channels. SIFF gives receivers a simple mechanism to stop packets from an unwanted sender: ceasing to forwarding packets to the sender, which causes the privileged channel to eventually close.

To establish a privileged channel, a sender must obtain a capability from the recipient through a handshake. The sender sends an *explorer* packet to the destination, and each (SIFF-enabled) router on the path inserts a marking. Each marking is a keyed hash of the router's incoming interface, the last-hop router's outgoing interface, and the source and destination IP addresses. The key is known only to the router.

When the sender's explorer packet arrives at the destination, the receiver sends its own explorer packet that includes the routers' markings. When the original sender receives this response packet, these markings serve as the capabilities that allow the sender to establish a privileged channel. The TCP handshake can be carried on top of the channel-establishment handshake.

Routers check privileged packets with the same computation used when creating markings: each router that forwards a (purportedly) privileged packet computes the marking it would insert into the packet if it was an unprivileged, explorer packet. The router forwards the packet if this marking matches the marking already present in the packet; otherwise, the router drops the packet.

To prevent an attacker from obtaining a capability and using it to flood a receiver, routers change their private keys (and thus computed capabilities) frequently. This does not affect normal communication, as these new capabilities are reflected to the sender by the receiver. However, a receiver that wishes to no longer receive traffic from a sender can stop relaying capabilities to the sender. The sender's capabilities will eventually expire and it will no longer be able to send privileged traffic to the destination.

Trust assumptions

SIFF trusts that (SIFF-enabled) routers check the validity of capabilities in packets and drop packets with invalid capabilities. Tacitly, SIFF also assumes that there is at least one SIFF-enabled router near every source, so that valid capabilities cannot be shared by many hosts behind their collective first-hop SIFF router.

In addition, SIFF makes the assumption that an explorer packet will eventually arrive at its destination and be returned, even when the unprivileged channel is being flooded with traffic.

Sender identification mechanism

SIFF capabilities identify the sender as someone able to receive traffic at the source address included in its packets. While this limits the number of spoofable addresses that are available to an attacker, it does not prevent source spoofing nor uniquely identify a sender. Namely, it does not offer protection against an attacker that has a large network at their disposal, as they can obtain valid capabilities for many senders. SIFF and other capability-based architectures define identification in terms of network location [72]; VanGuard, presented in Chapter 5, is the first exception to my knowledge.

2.4.3 TVA

TVA [81, 82] builds upon SIFF to allow destinations to express basic policies over how senders can use capabilities and to protect the capability request channel. TVA also makes minor improvements on SIFF, such as using larger, harder-to-forge capabilities and allowing senders to replace a capability with a nonce if that capability's router caches the capability.

As in SIFF, senders request a capability by sending an unprivileged capability request to the destination. TVA routers insert a pre-capability into capability-request packets. The format of a pre-capability is:

$$\text{timestamp} \mid\mid H(\text{source IP, destination IP, timestamp, secret})$$

where *timestamp* is the router's local time and *secret* is a value either known only to the router or shared across the router's trust domain.

Expressing policy in capabilities

To turn a pre-capability into a capability, a destination first selects values for T, the lifetime of the capability, and N, the amount of bytes the source can send in that time. The destination constructs the capability as:

$$\text{timestamp} \mid\mid H(\text{pre-capability, } N, T)$$

To verify a capability, a router re-constructs the capability (requiring two hashes) and compares against the value in the packet. A router must also check that its current time does not exceed $\textit{timestamp} + T$.

Ensuring that a sender does not use a capability to send more than N bytes requires storing state at the router. However, TVA makes assumptions that bound this state. Routers only keep state for senders that send at a rate faster than N/T, and this state is kept only for $\frac{LT}{N}$ seconds, where L is the size of the packet. Thus, when a sender's rate drops below N/T, meaning it cannot send more than N bytes over the lifetime of the capability, its state is eventually reclaimed. TVA imposes a minimum value on N/T, $(N/T)_{min}$. Thus each flow consuming state at a router represents bandwidth greater than $(N/T)_{min}$, of which there can only be $\frac{C}{(N/T)_{min}}$ for a link of capacity C. Thus TVA allows destinations to include rate limiting policies in capabilities, at the cost of (bounded) router state.

Fair-queuing capability requests

To prevent attackers' capability requests from overwhelming legitimate requests, TVA approximates per-sender fair-queuing at each *trust domain*, e.g., AS. However, because sender addresses may be spoofed (on the capability-request channel), TVA routers use a path identifier similar to Pi [78] to tag packets according to their incoming interface, which provides an approximate source identifier. Each network places packets that share an identifier into a queue and uses fair-queuing across all queues.

One consequence of this is that senders with the same path identifier share fate. While this limits the rate at which an attacker's capability requests reach the destination, it also allows a malicious sender to starve other senders whose packets map to the same queue. The authors of Portcullis [64] point out that, due to "mixing" of legitimate traffic with attack traffic, the probability of successful packet delivery drops exponentially in the number of hops along the path.

Because TVA does not allow receivers to block capability requests from unwanted senders, it does not prevent attackers from overwhelming portions of the capability request channel. Similarly, a large group of attackers (such as a botnet) can cooperate to fill most or all queues at a given router.

2.4.4 Denial of Capability Attacks

While SIFF and TVA can protect established, privileged channels, their weakness is that they cannot afford the same level of protection to channel establishment with unprivileged packets. Argyraki and Cheriton [9] claim that the capability distribution mechanism is the main deficiency of capability-based architectures. They describe a "denial-of-capability" attack, in which attackers flood the capability request channel, preventing legitimate capability requests from succeeding. TVA's fair-queuing is not an effective solution because it restricts all interfaces to equal connection rates, regardless of whether that rate is appropriate for (different) receivers. The underlying cause of this problem is that SIFF and TVA rely on the ability to partition traffic into two classes: "wanted", privileged traffic and "unwanted", unprivileged traffic, whereas connection-setup requests cannot be correctly classified as either wanted or unwanted.

More recent capability architectures take Argyraki and Cheriton's arguments into account. Portcullis (Section 2.4.5) responds to these observations by requiring clients to solve a puzzle in order to request a capability. Flow-cookies (Section 2.4.7) aims to protect only high-value targets and makes the assumption that all traffic sent to and from this target passes through a middlebox.

2.4.5 Portcullis

Portcullis [64] uses computational proofs of work, i.e., puzzles, to enforce fair sharing of the capability request channel. The authors of Portcullis take this approach because they observe that sender identifiers are either spoofable (IP addresses) or too course-grained (path identifiers) to provide adequate fairness. By tying a capability to an amount of work, Portcullis aims to improve sender fairness.

When a sender wishes to send a capability request, it generates a puzzle based on a *seed* and solves it. Each solution has an associated *puzzle level.* The sender includes the puzzle and solution in the capability request. Routers verify that the puzzle was created with the latest seed and that the solution is correct, and give priority to higher-level puzzles.

The trusted *seed generator* periodically releases a new seed to ensure that puzzle solutions expire. Seeds need to be unpredictable yet easily verifiable. The Portcullis seed generator uses a reverse hash chain by picking a random value h_0 and computing $h_{i+1} = H(h_i||i)$. The seed generator signs the final h_n value and releases it. Every t minutes, the seed generator releases a new seed, which is the predecessor of the previous seed in the hash chain; if the current seed is h_i, the next seed to be released will be h_{i-1}. Thus seeds are easily verifiable: one can easily check that $h_i == H(h_{i-1}||i)$. Routers learn of new seeds by observing the seed value in capability requests; if a router observes and verifies a new value, it assumes that is the current seed.

A Portcullis puzzle has the form:

$$p = H(x||r||h_i||\text{destination IP}||\ell)$$

where r is a random nonce, h_i is the current seed, and x is the solution such that the last ℓ bits of p are 0. ℓ is the level of the puzzle; solutions with a greater value of ℓ receive priority over those with lesser values. The sender includes r, h_i, ℓ, and x in the packet so that routers can verify the solution.

The authors of Portcullis prove that the best strategy for an attacker who wishes to flood the capability-request channel is to solve puzzles at the highest level at which it can keep the channel saturated. Solving puzzles of a greater level means the attacker is not able to consume the entire bandwidth of the capability-request channel. Solving puzzles of a lesser level means it is easier for requests from legitimate senders to be given a higher priority at routers. Given this optimal attacker behavior, the authors show that a legitimate sender need only solve a puzzle at a level slightly greater than the attacker's chosen level to successfully transmit a capability request with high probability. However, there is no guarantee that an under-powered (compared to the attacker) sender will be able to solve a puzzle at this level during the lifetime of a seed.

In order to prevent puzzle re-use and puzzle sharing, routers must store previously-seen puzzles. Portcullis stores puzzles in Bloom filters [21]. This requires routers to maintain a (bounded) amount of state. When the router encounters a new puzzle with a valid solution, it adds the puzzle to the Bloom filter and inserts the request into a priority queue based on its puzzle level.

Trust assumptions

Portcullis trusts the underlying capability architecture to correctly prevent (or limit) unwanted communication. Portcullis requires a trusted seed distribution service, for which they use DNS. The signing of the DNSSEC root [40] suggests that distributing seeds can be secured. However, Portcullis implicitly assumes that the seeds themselves, implemented with a reverse hash chain, are secure. Determining an "early" value in the hash chain would reveal all following values as well. An attacker capable of finding such a value could solve puzzles using unreleased seeds to trick routers into believing a new seed had been released. This would prevent legitimate senders from being able to use valid puzzle solutions. Additionally, the attacker could pre-compute and solve puzzles with a very high level, allowing him to flood the capability-request channel with requests that always take priority over legitimate requests. Even if H is preimage-resistant, the authors suggest that a hash chain should be valid for one year, which gives attackers a long time to mount a (massively parallel) attack.

Sender identification mechanism

Senders are differentiated based on what puzzles they solve, rather than by any identifier or network-based property.

2.4.6 NetFence

In NetFence [53], routers encode congestion information into packets using a variant of capabilities, allowing congested routers to signal to first-hop routers that a sender should be rate limited. The aim of NetFence is to guarantee each sender at least an equal share of each bottleneck link. To do so, it moves the task of congestion control from the the endhost, which NetFence does not trust, to the first-hop router, which NetFence does (although it details provisions for handling malicious routers). NetFence uses Passport [55] to establish shared keys between ASes and to prevent source spoofing.

Like SIFF [79] and TVA [81], NetFence splits traffic into three categories: request packets, regular packets, and legacy packets. Request and regular packets carry "secure congestion policing feedback", which first-hop routers use to rate limit senders if necessary. There are three types of congestion feedback: *nop*, indicating no rate limiting is required, $L^{\downarrow}$, indicating that link L is congested and the first-hop router should rate limit traffic traversing L, and $L^{\uparrow}$, indicating that (so far) the path is free of congested links. Each type of feedback is expressed as an unforgeable marking in the packet. The capability-request channel is limited to 5% of the bandwidth on any link. NetFence adopts a mechanism similar to Portcullis [64] to protect the capability-request channel, although it is enforced only at the first-hop router. In NetFence, senders assign a priority level to their packets. Packets at level k are given priority over packets at level $k-1$, but a rate limiter at the first-hop router accepts level k packets at half the rate of level $k-1$.

To initiate communication with a destination, a sender sends a request packet to the destination. The sender's first-hop router inserts the *nop* feedback into this packet. The destination returns the feedback to the sender in its response. The sender must include the returned feedback in its regular packets. In this sense, NetFence is a generalization of capabilities: if a destination does not want the sender to be able to send regular packets, it does not return the congestion feedback.

A congested router, R, may update feedback in order to throttle a sender S (similar to TCP explicit congestion notification [68]). If R detects a high loss rate on one of its outgoing links L, which may be a symptom of an ongoing attack, it starts a *monitoring cycle*. During a monitoring cycle, R stamps $L^{\downarrow}$ feedback into S's packets (unless a router preceding R has already stamped such feedback). R may modify both request and regular packets. Congestion feedback includes a timestamp to ensure freshness. R's feedback will eventually reach S's first-hop router, R_S, which creates a rate limiter for (S, L) if one does not already exist. So long as S's feedback is $L^{\downarrow}$, R_S will decrease the rate at which S can send through L. To detect when congestion is alleviated, R_S resets the feedback on outgoing packets, changing $L^{\downarrow}$ to $L^{\uparrow}$. R updates this feedback only when it is experiencing congestion; if an $L^{\uparrow}$ feedback is returned, then R_S increases the rate at which S can send through L. If S's packets do not cross a congested link, the returned feedback will be *nop*, R_S will not rate limit S, and R_S will continue to stamp *nop* feedback in S's packets.

A first-hop routers' (S, L) rate limiter throttles S with an additive increase, multiplicative decrease (AIMD) algorithm. Each rate limiter includes two variables: one to track whether S has received $L^{\uparrow}$ feedback and the other to record start time of the current *control interval*. S's rate is adjusted only at the end of each control interval. The default length of control intervals is two seconds. The only way for a sender's rate limit to increase is if it presents the first-hop router with fresh $L^{\uparrow}$ feedback. Otherwise, if the sender presents $L^{\downarrow}$ feedback, presents feedback from before the start of the control interval, or sends no packets, its rate limit will decrease. This prevents S from hiding $L^{\downarrow}$ feedback or sending at a slow rate to increase its rate limit. The first-hop router will terminate a rate limiter if it has not received $L^{\downarrow}$ feedback, or discarded a packet, for "a few hours". This rate limiting guarantees that, in the long run, each sender will have at least a $\frac{1}{N}$ share of the bandwidth of the congested link when there are N senders using the link.

To be effective, congestion feedback must be unforgeable; neither endhosts nor routers should be able to modify existing feedback other than to replace *nop* or $L^{\uparrow}$ with $L^{\downarrow}$. A congestion feedback has six fields: a mode (either *nop* or *mon*), a link identifier (containing an IP address if mode is *mon* and invalid otherwise), an action (↑ or ↓; valid if mode is *mon*), a timestamp indicating the time at the first-hop router, and a token, which is a MAC of various fields. The format of the MAC for a *nop* feedback, stamped by a first-hop router, is:

$$token_{nop} = MAC_K(src, dst, ts, null, nop)$$

where K is a key known only to the router, ts is a timestamp, and $null$ is a link identifier. A $token_{L^{\uparrow}}$ contains the ↑ action but it otherwise similar to a $token_{nop}$. A $token_{L^{\downarrow}}$, however, is created by a remote router. Its format is:

$$token_{nop} = MAC_{K'}(src, dst, ts, L, mon, \downarrow, token_{nop})$$

where K' is the key shared between the source AS and the congested router's AS, as established by Passport [55].

When a first-hop router receives a regular packet from a local sender, the router verifies the packet by re-computing the MAC and comparing it to the token in the packet. If the MAC is incorrect, or the timestamp in the feedback is older than the "feedback expiration time", which defaults to four seconds, the router demotes the packet to a request packet and applies the appropriate rate limiting.

The authors propose several possible remedies for handling a compromised first-hop router, including per-AS queuing at each router, per-AS rate limiting at each router, and heavy-hitter detection. Each approach relies on routers being able to map a source address to an AS; NetFence uses Passport to prevent source spoofing.

NetFence does permit at least one attack, though it is difficult to execute. If a sender can collude with an on-path attacker, such as an upstream router, the router can observe the sender's request packets with *nop* feedback and tunnel this feedback to the sender. The sender can then successfully ignore $L^{\downarrow}$ feedback, which it can replace with fresh, valid *nop* feedback. This attack also undermines NetFence's ability to allow destinations control over who sends to them: even if the destination does not return any feedback, the sender will still obtain it. This attack does not work in TVA [81] unless the colluding attacker is located in the destination's network; otherwise the sender will not be able to obtain the capabilities necessary to traverse all routers along the path. The crucial difference is that in TVA, a router anywhere on the path verifies the markings it places in a packet, whereas in NetFence, only the first-hop router does so.

Sender identification

The majority of NetFence's mechanisms do not require identification of a sender by any entity other than the sender's first-hop router; therefore identification is a matter of securing the local network. Remote identification of senders occurs only when a first-hop router is suspected of misbehavior. In this case, senders are identified at the granularity of their AS; Passport enables this identification. Neither routers nor destinations are able to identify the first-hop router that inserted initial feedback into a packet; this is surprising given that this router is trusted to ensure protocol compliance. While the authors suggest that a destination may refuse to return feedback to a sender, presumably based on identifying the sender by their IP address, Passport only identifies the AS from which a packet originated—it is not guaranteed to prevent source spoofing.

Trust assumptions

NetFence trusts first-hop routers to correctly police their attached senders. If they do not, NetFence must resort to secondary mechanisms that require thousands of queues per router in the core and may harm legitimate senders.

2.4.7 Flow-cookies and CAT

Flow-cookies [23] is a capability-based system that provides protection by routing all traffic between senders and a target server through a third-party middlebox called a *cookie box*. The cookie box has a high-speed connection to the Internet. Senders wishing to initiate a TCP flow with the server first contact the cookie box. This is accomplished by having the cookie box announce the routes to the protected server and then communicating with the server through a private tunnel. The cookie box uses SYN cookies [19] to ensure that the sender is not spoofing its address. Once the connection is established, the cookie box hands the connection off to the server. As in TVA [81], the sever consents to receiving traffic from the sender by sending a response, which is routed through the cookie box.

Return packets from the server are routed through the cookie box, which adds a *flow-cookie*. A flow-cookie is a MAC of a counter value and the connection 4-tuple, keyed with a secret known only to the cookie box. The counter is used so that flow-cookies expire. The cookies box places the flow-cookie in the TCP timestamp option field; most TCP implementations copy timestamped values from received packets into return packets. The cookie box then inspects non-SYN packets to ensure they contain a valid flow-cookie. The server can tell the cookie box to block certain IP addresses, which are stored in a blacklist at the cookie box. Flow-cookies cannot be used to protect links on the path to the cookie box, which is why is must be well-provisioned.

Because flow-cookies does not require modification of any existing protocols or client applications, it is simple to deploy. It does not use heavy-weight cryptography, and the authors' implementation is able to process packets at a rate of 2.38 Gb/s under idealized conditions.

CAT [22] extends flow-cookies to allow for a destination to employ multiple cookie boxes in separate networks. Multiple cookie boxes, placed on high-bandwidth links, are better able to handle high-volume flooding. By placing cookie boxes in or near Tier-1 networks, CAT reduces both the potential for bottlenecks and the possibility that the destination can be reached through an unprotected link.

CAT places cookie boxes in networks on the *trust boundary* around the destination's network. The trust boundary is the (transitive closure of) networks that have a commercial relationship with the destination's network. CAT assumes that these commercial relationships can be bootstrapped into trust relationships, which can be used to determine which cookie box is responsible for establishing connections and filtering traffic. CAT includes modifications to BGP and routing tables to allow networks to make these decisions.

Because CAT is a bolt-on solution, it can only provide protection for sites that can route their traffic through a cookie box (notably, cookie boxes themselves can not protected by CAT).

Trust assumptions

Flow-cookies trusts cookie boxes to not fail, maliciously or otherwise. If a cookie box suffers a simple fail-stop failure, the destination is unreachable. CAT solves this problem by incorporating multiple cookie boxes; however, a compromised cookie box may still fail to block unwanted traffic. In addition, CAT trusts remote networks to correctly determine whether or not they are responsible for filtering traffic.

Sender identification mechanism

A cookie box identifies senders at two different times. When a sender first establishes a connection, the cookie box uses SYN cookies, and when a sender uses a previously-established connection, the cookie box uses flow-cookies. Both types of cookie ensure that the sender is able to receive traffic at the source address included in its packets. Because cookies are hard to forge without knowing the MAC key, an attacker cannot spoof an arbitrary address and flood the protected server using flow cookies that are valid for that address.

2.5 Filtering

Filter-based systems a complementary approach to that of capability-based systems. In capability-based systems, routers by default provide a low level of service and a receiver must explicitly allow traffic for it to receive a higher level of service, while in filter-based systems, routers provide a high default level of service and a receiver must explicitly block senders from whom it does not want to receive traffic.

Filter-based designs have their own set of trade-offs. By design, filter-based systems require devices other than the receiver to store filter state. Unlike capability systems, there is no capability-request channel to protect; however, there must be a mechanism to identify senders to block and ensure that blocks are enforced.

2.5.1 AITF

In capability-based architectures like TVA [81], decisions on whether to grant capabilities or not are made at the destination. In Pushback [58], rate-limiting aggregates are created and stored at congested routers near the destination. Active Internet Traffic Filtering, or AITF [8], pushes filtering decisions even further away from the destination by placing them at the gateway of the blocked sender. Placing filters near the sender increases the filtering capacity of the network as a whole, distributing the task of storing filters among the greatest number of devices. It also drops unwanted traffic before it has had a chance to mix with other traffic or consume excessive forwarding resources bandwidth.

In order to allows destinations to place filters at the source's gateway, AITF must first provide a way to identify the gateway in question and then dictate how a destination can establish a filter at a remote device. To identify the source of packets, AITF uses a variant of the IP Record Route option. Routers that participate in AITF stamp their IP address and a keyed MAC (created with a key known only to itself) of the packet's destination into the packet, using a shim protocol between

IP and the transport protocol. The destination thus receives an ordered (but likely incomplete) list of routers through which the packet traversed. This *path* has the form $\{S, S_{gw}, ..., D_{gw}, D\}$ where S is the source, D is the destination, and X_{gw} is X's gateway.

When a destination D receives unwanted packets stamped with a path P, it asks its gateway D_{gw} to filter packets with this path. D_{gw} then installs a short-lived filter that blocks traffic with path P from arriving at D and sends a request to S_{gw} to block traffic from S to D. This request takes the form of a three-way handshake, to prevent an off-path attacker from installing a bogus filter at S_{gw}. However, the handshake packets traverse the same links as the attack traffic, so they may be lost. AITF does not offer protection against this possibility [52].

If the handshake is successful, S_{gw} installs a short-lived filter to block S to D traffic and requests that S no longer send traffic to D for a fixed length of time. If this fails to staunch the flow of traffic, e.g., if S_{gw} misbehaves, then the local gateway can *escalate* the filtering request to the next router on the path (that is, the router two hops away from S). This request is the same as the original request to S_{gw}, only now D_{gw} asks that all traffic from S_{gw} to D be blocked. Escalation can have negative side effects for both S_{gw} and D_{gw}; as such it can serve as a powerful motivation for S_{gw} to behave, but may not always be a credible threat. Escalation continues until either the traffic stops or D_{gw} installs a filter locally that blocks all traffic from S_{gw} to D. As there on the order of tens of thousands of edge networks, and assuming there is a unique border router per customer-provider connection, D_{gw} can block arbitrarily many unwanted senders with only tens of thousands of filters.

If AITF is only partially deployed, it may be the case that there are no AITF-enabled devices unique to the path from an attacker A to a destination D, i.e., devices that are not also on the path from a distinct sender S to D. This means that A can spoof the path in its packets, essentially framing S and S_{gw}. The keyed MAC is used to prevent this attack: when S_{gw} receives a request to block traffic that it did not forward, it responds with the correct MAC for packets destined to D. However, a malicious S_{gw} could always respond in this way (as the key used to create the MAC is known only to itself) and never be blamed for forwarding malicious traffic.

Trust assumptions

AITF trusts nearly every element of the architecture. Routers must place the correct markings in packets for blocking to be effective. While escalation protects against certain malicious gateway behaviors, it is only possible after a sufficient number of networks deploy AITF, and routers in the middle of the network may not have the capacity to handle a large number of escalation requests.

Sender identification

A sender is identified by markings placed into its packets by AITF-enabled routers. These markings do not identify a unique sender so much as a sequence of routers that a packet traversed. Because identification is effectively at the granularity of a subnet, senders can spoof the addresses of other hosts located behind the first-hop AITF router.

2.5.2 AS-based Accountability

Simon et al. were the first to describe the role of accountability as a means of DDoS defense [72]. They define accountability as the combination of accurate and reliable source identification along with a mechanism for allowing receivers to block traffic from any source. Their proposed system extends AITF with stronger sender identification and hop-by-hop filtering requests, and the authors argue that implementing accountability in this way is the most cost-effective method (compared to over-provisioning and traffic scrubbing) of defending against DoS attacks.

In order for blocking to effective, identification must be in terms of a persistent attribute, rather than some network-specific property. Otherwise, senders can simply change their network properties to evade a block. In this system, a sender is identified as a *customer* of an ISP, i.e., an entity that receives an allocation of one or more IP addresses. A customer could be either a single user, a company, or another ISP. Participating ISPs store records of which IP addresses they allocate to each customer and at what time; this entails, for instance, retaining DHCP and NAT logs. To ensure that source IP addresses can be traced to the sender, these ISPs also deploy strict ingress filtering [32].

When a destination in a participating ISP wants to block traffic from a sender, it sends a filter request to its local *filter request server*, or FRS. These devices process and forwards filter requests, and each accountable ISP must operate at least one. The FRS determines if the requesting customer is allowed to issue a filter request, which is a matter of local policy. If the request is allowed, the FRS forwards the request to the FRS in the neighboring ISP through which the offending traffic arrived (as determined by BGP tables). This process repeats until the request arrives at the FRS in the ISP from which the traffic originated. There, the FRS examines its records to find the customer that was using the IP address specified in the request and places the blocks necessary to prevent that customer from sending packets to the destination.

Not all ISPs will implement accountability. This has several implications, the foremost of which is that a packet originating from an unaccountable ISP may have a forged source address. To address this issue, the accountability scheme uses an "evil bit"—a one-bit field in each packet. A packet's evil bit is initially set to 0, and if the packet ever crosses a border from an unaccountable ISP to an accountable one, the accountable ISP sets the packet's evil bit to 1. The evil bit is otherwise preserved. If a packet arrives at a destination (in an accountable ISP) with the evil bit set to 0, then the destination knows (1) that the source address in the packet is not spoofed (since it originated from an accountable ISP) and (2) that there is a *path* of accountable ISPs from the source to the destination, which is required for filter requests forwarding. If offending traffic with the evil bit set to 1 arrives at a destination, the only option is to rate-limit such traffic.

The evil bit has additional uses. An overwhelmed router should preferentially drop traffic with the evil bit set. If a malicious sender garners an excessive number of filter requests, its ISP can reclaim filter state by removing these filters and setting the evil bit in that sender's packets. An ISP that claims to be accountable when it is not can be detected via out-of-band mechanisms, and its neighboring ISPs can set the evil bit on its traffic. Finally, to prevent reflection attacks, in which a sender issues a request from a spoofed source address to cause the response to be sent to a third-party victim, the evil bit is preserved in all responses (as spoofing is only possible from unaccountable ISPs, these evil bits will be set to 1).

Trust assumptions

This scheme trusts ISPs to accurately map traffic to customers. There is no mechanism to detect a misbehaving ISP that allows its customers to evade blocks by changing their IP address. The scheme also trusts ISPs to correctly implement ingress filtering. While not specifically a trust assumption, the filter request mechanism requires a path of neighboring, accountable ISPs between sender and receiver.

Sender identification

In this scheme, sender identification is possibly only for traffic from accountable domains. There, ingress filtering ensures that senders can use only the addresses allocated to them, and ISP logs map these addresses to customer records. By identifying senders by something other than IP address, this system is able to block senders regardless of their network properties, so long as they remain in the same ISP.

2.5.3 StopIt

StopIt is a filtering system designed to address the deficiencies of AITF [8]. Its design incorporates a device dedicated to handling filter requests, the StopIt server, in every AS (similar to the FRS of Simon et al. [72]). StopIt server addresses are published via BGP, so that any StopIt server can send a filter request to the StopIt server in a given AS. The only devices allowed to contact a remote StopIt server are other StopIt servers; filters configured with StopIt server addresses protect StopIt servers from potentially malicious inter-domain traffic.

StopIt relies on Passport [55] to prevent source address spoofing. However, Section 2.3 describes how Passport is insufficient for this task. Establishing a filter that blocks traffic from S to D for a length of time t is a five-step process:

1. A destination D sends a StopIt request to its gateway D_{gw}, requesting to block the flow (S, D) for time t.

2. D_{gw} confirms that S is sending traffic to D by looking at a log of recent packets, installs a temporary filter blocking S, and sends a StopIt request directly to S.

3. If S continues to send traffic to D, D_{gw} sends a StopIt request to its local StopIt server, D_{SIS}. D_{SIS} forwards the StopIt request to the StopIt server in S's AS, S_{SIS}. StopIt servers communicate on a closed channel: routers block traffic destined to a local StopIt server that was not sent by a remote StopIt server.

4. D_{SIS} forwards the request to S_{gw}.

5. S_{gw} confirms that S is sending traffic to D, installs a filter blocking S to D traffic, and sends an additional StopIt request to S.

StopIt improves upon AITF by incorporating defenses against several attacks that AITF is vulnerable to. AITF's primary weakness is that filter requests share link capacity with attack traffic. StopIt prioritizes inter-domain filter requests by ensuring attack traffic does not reach StopIt servers. However, the method described in the paper to enforce this property requires that border routers maintain a whitelist of the addresses of 30,000+ remote StopIt servers and compare packets' source addresses against this list.

StopIt requires that S send traffic to D before D can block S. This prevents a type of filter-exhaustion attack at S at the cost of keeping per-flow state at every gateway: D_{gw} maintains a *flow cache* that logs each flow passing through D_{gw} over the past several seconds. When D sends a StopIt request to D_{gw}, D_{gw} checks that it has seen traffic from S to D before forwarding the request. Likewise, S_{gw} will ensure that is has seen (S, D) traffic before blocking that flow.

Trust assumptions

StopIt relies on Passport [55] to prevent source spoofing and relies on the same trust assumption (see Section 2.3). In addition, StopIt trusts StopIt servers to honor filter requests from remote ASes.

Sender identification

Senders are identified using Passport.

2.6 AIP and IPa+

2.6.1 AIP

The Accountable Internet Protocol [3] (AIP) takes an innovative approach to providing accountability at the network layer: hierarchical addresses derived from public keys. While this prohibits compatibility with IPv4 and IPv6, it allows verification of source addresses with a level of certainty that no other scheme (other than DNA) offers. AIP uses secure hardware in each endhost to allow destinations to block unwanted traffic.

In the AIP architecture, an *accountability domain* (AD) plays the role of today's network or AS. Each host has a globally-unique endpoint identifier (EID). The full address of a host located in AD AD has the form AD:EID. AD names and EIDs are *self-certifying*: they are hashes of self-generated public keys. As such, they are flat identifiers that cannot be aggregated, and the current practice of routing based on prefixes would no longer be possible. In AIP, inter-network routing protocols operate at the granularity of ADs, and routers forward a packet using only the destination AD until it reaches this AD. Within the destination AD, routers forward the packet using only the EID.

The goal of using self-certifying addresses is to prevent source spoofing. The notion of address spoofing in AIP is different than in today's Internet. With IP, interfaces are assigned IP addresses (either by an administrator or via DHCP), and a spoofed packet is any packet leaving that interface with a different source address. In AIP, a sender may create as many valid source addresses as they wish. The concern, therefore, is not sending with a different address but with impersonation: sending a packet using a source EID for which the sender does not know the corresponding private key. To prevent impersonation, source address are verified in multiple places in the network: at first-hop routers and when crossing AD boundaries. Within the source AD, a first-hop router R_1 verifies that a connected sender S is not using a spoofed address. When S first sends a packet through R_1 after not having sent a packet for a given amount of time, R_1 drops this packet and sends a *verification packet* to S. S signs this packet with the private key corresponding to its EID and forwards the signature to R_1. If the signature is correct, R_1 will forward packets for S.

When a packet crosses the boundary from AD A to AD B, there are three possible cases. If B trusts A to have verified the source address, then B forwards the packet. Otherwise, B uses a variant of uRPF (discussed in Section 2.8.1) to determine if the packet arrived on the same interface that offers the best path back to the packet's source AD. If so, B forwards the packet. If both tests fail, B follows the same verification procedure as a first-hop router and sends a verification packet to the packet's source address $AD{:}EID$. If the sender is able to correctly sign the packet, the router in B creates an entry in its *accept cache* stating that it should forward packets from $AD{:}EID$. This last check has the potential to create a large amount of state at routers in B, but will only be used in the case of path asymmetry.

A destination that receives unwanted traffic can block the sender by sending a *shut-off packet* (SOP) to the traffic source. AIP assumes that hosts (which are operated by well-intentioned users) are equipped with a trusted network interface card ("smart-NIC"). The smart-NIC receives the SOP and installs a filter blocking further packets to the destination, for an amount of time specified in the SOP. Smart-NICs require that SOPs include the hash of a packet recently sent by the NIC to the destination; this prevents replay attacks and the spoofing-capable attackers from blocking traffic between innocent hosts.

While novel, AIP's blocking mechanism has drawbacks. First, the blocking mechanism will not work for all endhosts. Destinations can block only hosts that have a smart-NIC (and allow the smart-NIC to receive SOP packets). Second, there is a mismatch between the entity that appears in packets (EID) and the entity that can be blocked (a NIC). Blocking a NIC does not prevent a malicious user from sending traffic from the same EID on a different host. Likewise, any NIC can (legitimately) send from a given EID; blocking a NIC prevents all EIDs from sending from that NIC. A more appropriate mechanism would be to block an EID, rather than a NIC, from sending to a destination.

Trust assumptions

AIP relies on trusted hardware at the sender to enforce its blocking mechanism. AIP also trusts a source AD to verify that a sender is not sending with a spoofed address; while an upstream AD can also verify this, the process requires the AD to store state on behalf of a remote sender and is therefore used only as a last resort. Source ADs are trusted to enforce a limit on the number of addresses a sender can use; no other AD can enforce this. The authors of IPa+ [80] note that a compromised AD can spoof the address of any host whose traffic is forwarded through the AD, because it can pass the uRPF test.

Sender identification

Senders are identified by a hash of a self-generated public key. Source ADs are trusted to enforce a limit on the number of keys (addresses) a sender can use.

2.6.2 IPa+

IPa+ [80] is an attempt to provide the same accountability guarantees as AIP [3] while staying compatible with IP. IPa+ focuses on securing the Internet's routing infrastructure and resorts to existing mechanisms to provide source accountability and stop unwanted traffic.

IPa+ adopts one of the addressing conventions of AIP—AS names are the hashes of public keys—while eschewing the other—host addresses are IP addresses, not flat hashes. As AS names are currently flat, this does not affect the scalability of existing interdomain routing protocols, and it preserves the ability to aggregate host addresses. IPa+ binds an IP address prefix to an AS's public key and uses the hash of the key as the AS's BGP name. These bindings are stored as signed reverse DNSSEC records. Specifically, the root Internet registry, IANA, signs records that certify prefix allocations to individual RIRs, RIRs certify prefix allocations to ASes, and ASes may sign allocations to customers. Each of these records is stored in DNSSEC.

Given these secure bindings, IPa+ uses sBGP [46] or another secure routing protocol to announce prefixes. Validation of announcements is done by querying the appropriate DNSSEC records, assuming IANA's root public key is globally known.

In order to retain the ability to aggregate IP addresses, IPa+ does not adopt AIP's concept of self-certifying host addresses. In fact, IPa+ claims that assigned, rather than self-generated, addresses are more accountable, as they limit "whitewashing" attacks in which attackers mint a new address or identifier after they have been blocked. To provide source accountability, IPa+ uses Passport [55] (Section 2.3). In the interest of facilitating deployment, IPa+ does not adopt AIP's trusted smart-NIC. Instead, the authors IPa+ suggest using StopIt [52], AITF [8], or Portcullis [64] to allow destinations to stop unwanted traffic.

Trust assumptions

IPa+ relies on the correct deployment of many systems: DNSSEC, a secure BGP, Passport, and a capability- or filter-based architecture. While securing the Internet's routing infrastructure may mitigate the risk of man-in-the-middle attacks on Passport (Section 2.3), IPa+ trusts ASes to prevent spoofing in their own domains.

Sender identification

IPa+ uses Passport [55] to identify the sender's AS.

2.7 Other schemes

This section contains proposed systems that are related to DNA, but have little in common with previously-discussed work.

2.7.1 NUTSS

NUTSS is an alternative Internet architecture that incorporates policy into connection establishment. When initiating a flow, a sender first forwards a signaling message through an off-path overlay of "P-boxes", which authenticate the sender and apply policies such as access control. Signaling messages are routed by name: an extension to DNS provides the path of P-box names from the sender to the destination's hostname. P-boxes insert tokens into this message, and the receiver responds with both addressing information (its address and port) and the inserted tokens.

The sender can now send messages directly to the destination's address. These address-routed messages pass through on-path "M-boxes": each P-box has one or more corresponding M-boxes that lie on the data path. M-boxes enforce P-box policy, by allowing traffic only if it contains a token granted by that P-box.

Applying policy during connection establishment allows receivers and networks to prevent unwanted connections and to explicitly negotiate the use of middleboxes. This allows NUTSS to provide partial DoS protection, but NUTSS relies on external mechanisms to protect P-boxes and M-boxes from attacks. NUTSS does not depend on specific authentication protocols; it suggests using standard mechanisms such as public-key signatures and challenge-response protocols to identify senders.

2.7.2 Default-off

Default-off [13] allows hosts to declare to the network infrastructure what traffic it wants routed to it. It changes the Internet's open model, in which any host may send to any other, into a network that is "off by default": hosts that want unsolicited traffic, such as servers and peers in P2P networks, must proactively say so. Other hosts that do not want unsolicited traffic need not do anything. Public servers publish these reachability constraints to their first-hop router, who propagate them through the network, similar to, but separately from, routing advertisements. Routers enforce reachability constraints at forwarding time, dropping packets destined to unreachable destinations (unless they are responses to traffic initiated by that destination). The network acts as a large, distributed firewall, enforcing hosts' policies at various points in the network.

Default-off relies on routers to store reachability information, which may prove prohibitive, as each destination can publish a list of IP addresses which are not allowed to contact it. To save space, routers may aggregate reachability information, at the cost of accuracy.

2.7.3 Traceback

Related work discussed so far combats source spoofing by either authenticating traffic based on shared knowledge rather than address or network information (overlays [48, 4] and capability-based architectures [5, 81, 79, 84]), recording the packet's path in the packet (AITF [8], Passport [55]), or using cryptographically-unforgeable addresses (AIP [3]). An alternative approach, called *traceback*, is to modify IP to allow a receiver to determine the origin of a (spoofed) packet. With traceback, on-path routers assist the destination in determining the path that a packet takes. Most proposals require the destination to receive many packets from a source, and perform computation on those packets, before it is able to reconstruct the path. Traceback schemes do not allow a single packet to unambiguously identify a sender, nor do they incorporate blocking mechanisms.

Several different types of traceback have been proposed. Bellovin et al. detail a scheme in which routers, upon receipt of a packet, will forward a special ICMP message to the destination with very low probability. These messages contain the router's address and allow the destination to re-assemble the path that a packet flood takes [16]. Savage et al. propose *probabilistic edge sampling* in which routers will randomly encode link information in the IP ID field of a packet [71]. Pi [78] modifies Savage's scheme to use deterministic markings, so that destinations may filter based on path information. SPIE [74] avoids packet modification, and instead requires routers to store hashes of packets so that the destination may query them to determine through which routers a packet has passed.

2.7.4 Reliable routing

The architecture presented in this thesis uses signatures on packets to show provenance. One of the first appearances of digital signatures in network protocols is Perlman's Byzantine fault tolerant routing protocol [65], which uses digital signatures to unforgeably identify senders so that routers can perform fair, per-source queuing.

2.8 Internet Practices

Two Internet practices are particularly relevant to this thesis: ingress filtering, which aims to prevent source spoofing as it is used in DoS attacks, and IPsec, which, among other things, enables mutual authentication at the network layer.

2.8.1 Ingress filtering

Network ingress filtering, as described in BCP 38 (also RFC 2827) [32], is a simple measure that ISPs can take to ensure that they do no accept spoofed packets. If an edge network announces a set of prefixes, then a router in that ISP that implements ingress filtering will not accept packets that have a source address from a prefix outside of those announced by the network. This ensures that a sender cannot spoof arbitrary addresses, but does not prevent it from using other addresses in the same prefix.

Unicast reverse path forwarding (uRPF) extends ingress filtering by requiring routers to forward a packet only if it arrives on the interface which is the router's best route to the packet's source address. BCP 38 claims that uRPF is not effective due to route asymmetries. BCP 84 (RFC 3704) [12] describes possible relaxations to uRPF that allow it to be effective in the presence of asymmetry and extends ingress filtering to multihomed networks.

In order to be effective, network ingress filtering needs to be universally deployed. A single ISP that does not implement ingress filtering allows hosts in that network to spoof arbitrary addresses, rendering address-based filtering at the destination ineffective. As spoofed packets traverse networks, they mix with legitimate packets, making ingress filtering impossible to implement in the core of the Internet. Unfortunately, an ISP has little intrinsic incentive to deploy ingress filtering, as it does not prevent spoofed traffic from other non-filtering ISPs from entering the ISP.

2.8.2 IPsec

The Security Architecture for IP [47], or IPsec, defines various security services for IP traffic, including encryption and authentication. It consists of two protocols, Authenticated Header, which authenticates traffic, and Encapsulating Security Payload, which encrypts and optionally authenticates traffic.

As one of the functions of authentication is to identify senders, IPsec could be used to hold senders accountable. However, IPsec uses MACs, created with symmetric keys, to authenticate packets. In order to establish a shared key, a sender and receiver must exchange a key out of band, or use the Internet Key Exchange protocol. To provide wide-area accountability, IPsec would require that every (sender, receiver) pair establish a shared secret key, which is infeasible. Additionally, because the keys are symmetric, the sender is not accountable to anyone other than the receiver.

2.9 Commercial services

Two types of commercial services are available to help servers survive DoS attacks: traffic scrubbing and content distribution networks (CDNs).

Traffic scrubbing

An ISP that wishes to protect itself from DoS attacks can employ a traffic scrubbing service, such as those offered by Arbor Networks [6] or Prolexic [66]. These services typically use anomaly detection to determine when an attack is occurring and then re-route traffic to a high-capacity network. This network “scrubs” the traffic by removing packets that match a pattern of malicious traffic. The remaining packets are then routed to their original destination.

Content distribution networks

CDNs such as Akamai [2], Limelight [51], and Coral [34] replicate a server’s (mostly static) content at multiple points throughout the Internet. Clients are directed towards their nearest copy, usually with DNS-based routing, rather than to the central server. CDNs protect against bottlenecks and flash crowds at the server, but not malicious DoS attacks.

2.10 Other uses of accountability in networks

2.10.1 Accountability in distributed systems

Accountability in the Internet fits into the broader problem of accountability in distributed systems, a topic which has been studied previously. Yumerefendi and Chase [83] suggest accountability as a design goal of network services and propose a framework towards that goal. They developed CATS [84], an accountable network storage service, within this framework. PeerReview [37] provides accountability for general distributed protocols through per-node, secure, consistent logs of (signed) messages that node has sent and received. A detector module can detect Byzantine faults by comparing message logs to a reference implementation of the protocol. Each of these systems uses digital signatures to identify principals and prove authenticity.

2.10.2 Service accountability

An Internet service provider that violates a service level agreement (SLA) may cause an endhost to experience poor service, such as unacceptable levels of packet delay or loss. The host may be unable to determine who is responsible, as a malicious ISP can identify the sender's diagnostic measurement probes and give them a higher level of service, while still providing poor service to ordinary traffic. Several pieces of work propose methods of holding ISPs accountable for providing the level of service stipulated in their business agreements, allowing hosts to detect and react to SLA violations.

AudIt [10] argues that ISPs should collect statistics on traffic aggregates and report to a separate ISP I the statistics of the traffic that originated from I. AudIt is resilient to ISPs that produce false statistics: they are exposed to their peers.

Goldberg et al. [35] present secure sketching and secure sampling protocols that can localize failures. The sender and receiver apply a shared psuedorandom function to data packets to create a small set of probe packets, which the receiver explicitly acknowledges. An adversarial ISP is unable to determine which packets are probes ahead of time, and therefore must provide the same level of service to all packets.

Chapter 3

An Accountability Architecture for the Internet

Accountability in a network, defined as the ability to map packets in a network back to the principals responsible for their transmission, coupled with the capacity to block those principals, is vital to preventing misbehavior. Blocking malicious traffic in routers requires filters to distinguish malicious from valid traffic far from the victim. This architecture conflates the roles of ISPs as providers of connectivity (the pipe) and of accountability (stopping abuse). We believe that ISPs need not bear the responsibility for deciding the policy (whom to block) and implementing the mechanism (filters) of network layer accountability.

Towards this end, this chapter presents DNA, an accountable Internet architecture. DNA assumes the existence of trusted identity authorities, the ability to compute HMACs at intermediate routers and verify signatures at customer-edge routers, and the availability of servers to act as partially-trusted storage devices and representatives of an accountability provider.

With these features, DNA contains protocol mechanisms that allow high performance verification of accountable traffic, traceback of unaccountable traffic, and the revocation of permission to connect.

3.1 Introduction

Misbehavior in the Internet can be attributed to two major factors: first, that attacks can be launched with relative anonymity through source spoofing or networks

This chapter is based on joint work with Rob Sherwood, Neil Spring, and Bobby Bhattacharjee.

of compromised machines (botnets), and second, that it may not in a provider's interest to deny connectivity to a paying customer simply because some attack traffic leaves the network. Packet traceback [71, 74, 78] and reverse-path filtering to prevent spoofing [55, 72] attack the first problem, limiting the anonymity of attackers and allowing an abused destination to reliably accuse a sender of misbehavior. In each approach, however, the network providing connectivity to the source machine is expected to disconnect it (perhaps completely) to protect a distant destination.

As profit-driven companies, Internet service providers are not impartial: each has a financial interest in turning a blind eye to misbehavior by their own customers.[1] Rogue ISPs such as Atrivo [11] actively prosper from hosting services that are widely known to be malicious. for any protocol designed to find, block, and prevent network abuse is to *not rely on any ISP*. Restated, *providing connectivity must not imply accepting responsibility.*

DNA takes this principle one step further; that a sender's ability to inject traffic into the network should be decoupled from his point of attachment. The action by a destination to *block* a misbehaving source should prevent that source from obtaining a new address or spoofing a different address to send traffic to the same destination. This suggests that identities, or more specifically blockable principals, should be issued as public keys from a globally-known, trusted authority. This trusted accountability provider (TAP) prevents arbitrary users from sending traffic and, more importantly, bars blocked subscribers from sending under a pseudonym. DNA supports the coexistence of many TAPs, each with its own set of policies and underlying implementations, which agree on a protocol for validating packets in transit. That is, DNA is designed for the tussle [27], not for a specific outcome.

This chapter presents the network components of DNA, a complete network accountability system in which:

1. An accountability provider is the sole trusted entity; all other participants

[1] Protecting their customers from outside attack may be a separate service.

(routers, accountability provider representatives, etc.) can be implicated if they misbehave. The trusted entity's services need not be implemented centrally, allowing it to be made scalable and resilient to attack.

2. Endhosts can block principals from sending traffic to them. Blocked principals cannot send traffic to the blocking destination regardless of their point of attachment.

3. Long-term blocking filter state is not stored at any router; a combination of Passport [55]-inspired tokens and distributed filter storage compels the source network to check for blocks.

4. A privacy certificate authority (CA) [76] authenticates temporary keys, rather than long-lived identifiers, for use in contacting destinations, preserving (some) privacy. Machines near senders guard the sender's long-lived identity, though we do not explicitly anonymize clients or servers as in SOS [48].

5. What constitutes abuse is determined by a destination alone, it need not be proven to an authority: if traffic is undesired, future traffic can be blocked. The sender can continue to send to other destinations.

6. Cryptographic operations can be largely avoided along the fast path through *waivers*, explicit statements that a server will not be implicated by a client, and *tokens* that enable traceback.

To implement this network accountability system, we designed, implemented, and evaluated protocol mechanisms for high-performance verification of accountable traffic without all-pairs shared keys, traceback of unaccountable traffic (improperly signed, but carried anyway) to a misbehaving inter-domain link, and revoking permission to connect without a separate, protected control channel.

The rest of this chapter is structured as follows: we summarize related work in Section 3.2. In Section 3.3 we present the design of DNA. We discuss attacks on DNA in Section 3.4 and DNA partial deployment approaches in Section 3.6. We evaluate DNA overhead in Section 3.7 and conclude in Section 3.8.

3.2 Related work

DNA allows receivers to block specific senders. Several recent proposals achieve similar goals using different mechanisms.

In TVA [81], senders initiate a connection by sending a *capability request* on a dedicated channel; this request is marked by each router along the way. If the traffic is desired by the receiver, the marked request is returned to the sender. The sender then includes this returned *capability* in subsequent packets. During an attack, packets without capabilities are preferentially dropped. Portcullis [64], an extension to TVA, uses sender-solved puzzles to protect the capability request channel. Unlike DNA, TVA and Portcullis both pin the network path between the sender and the receiver (and require a symmetric path for optimal performance). DNA relies on the TAP to issue global identities that enable receivers to block senders regardless of their network point of attachment; such a facility does not exist in TVA or Portcullis. Finally, DNA requires only the TAP nodes be correct and trusted; all other entities may be corrupt. TVA and Portcullis require a trusted and correct transit infrastructure.

AIP [3] binds self-certifying addresses with network interfaces and assume that all interfaces have trusted hardware that obeys a "shut-off" packet sent by a receiver. Using these primitives, AIP builds an accountability infrastructure that receivers can use to stop abusive senders. Like DNA, AIP provides receivers fine-grained control over which senders are blocked; unlike DNA, the blocks in AIP are bound to network interfaces and not to packet-sending principals. DNA relies on a trusted authority to certify network-layer-independent identities. In comparison, AIP relies on a pervasive trusted hardware deployment at end hosts. DNA is designed as a shim layer between network (IP) and transport (TCP) and is agnostic of the network protocol, whereas AIP requires a new addressing scheme at the network layer.

Passport [55] is a system for cryptographically identifying a packet's source AS. This information can be used to track abusive senders. Passport relies on a shared key between every pair of participating ASes (whether they are neighboring ASes or not). These keys are used to create HMACs (like the tokens of DNA) that can eventually be used to identify the source AS of any packet. Passport requires the predetermination of AS paths for every end-to-end route. In previous work [18], I show how Passport can be extended to identify and block individual senders.

The requirement of shared keys between every AS pair (Passport) vs. pervasive trusted hardware (AIP) vs. trusted identity authorities in DNA represent explorations in three starkly different points in the design space for network-layer accountability. DNA minimizes the number of trusted entities in the system (only the TAP nodes are trusted) and is the only system that can withstand malicious transit routers and ASes.

Simon et al. [72] present a system where receivers can identify the source of traffic and block traffic from any source. Their scheme consists of per-customer ingress filtering at ISPs, and a trusted Filter Request Server (FRS) located in each ISP. To block a source, a receiver contacts the local FRS, which in turn contacts the FRS in the source's ISP. The source's FRS is responsible for filtering traffic from that source to the receiver. AITF [8] uses a mechanism similar to IP Record Route to mark the provenance of packets. When a receiver wants to block traffic that follows a certain path, the receiver asks its gateway to contact the sender's gateway, similar to how FRSes contact each other. In comparison to these systems, DNA's notion of blocks transcend the sending principal's network point of attachment. Perhaps more significantly, these schemes hold the sender's AS responsible for traffic from the AS and for blocking abusive senders. In fact, AITF includes a provision for disconnecting an entire AS if blocks are not properly administered. In

DNA, destinations block individual sending principals; once blocked, these senders can no longer obtain necessary tokens to send packets to the blocked destination. The source AS is unaware of specific blocks, and instead simply verifies that only accountable packets depart its network.

Finally, IP traceback [71, 74, 78] is a mechanism that victims invoke to (partially) trace back the path an attack packet traversed. Traceback mechanisms require minimal changes to hosts and can potentially be used as a building block for an end-to-end accountability solution. The "accusation" protocol in DNA uses tokens to traceback unaccountable packets. The DNA traceback mechanism always isolates the malicious entity that sourced the attack packets.

3.3 DNA: Design

The primary goal of DNA is to ensure that only accountable packets may transit networks to destinations that have not blocked the sender. Accountable packets are those signed by a principal authorized by a trusted authority (*waivered* packets require no signatures, but carry proof that the destination expressly consented to receiving such packets). I term "unaccountable" those packets with an invalid signature and "legacy" those packets having no accountability material at all.

In DNA, *principals send packets to hosts.* Each packet contains sufficient information for a destination host to block the sending principal (for any reason). DNA provides the following property:

> Packets are forwarded to a destination only if the destination has not blocked the sending principal, regardless of the sender's network point of attachment.

Lesser goals that shape the design of DNA are as follows. Verifiable functions should be distributed away from the trusted authority, to protect it from attack or overload with mundane requests; this leads to complexity but should provide resilience.

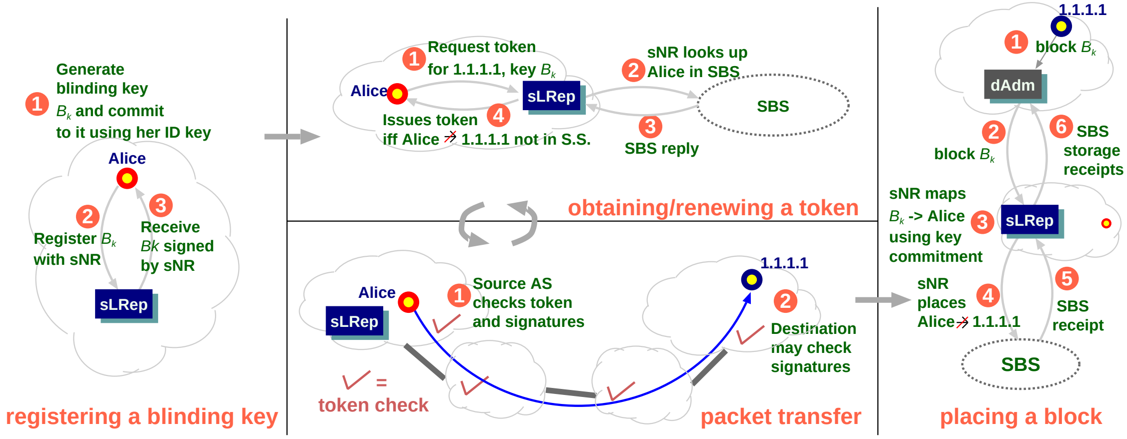

1 Generate blinding key B_k and commit to it using her ID key
Alice
2 Register B_k with sNR
3 Receive Bk signed by sNR
sLRep
registering a blinding key
1 Request token for 1.1.1.1, key B_k
Alice
sLRep
2 sNR looks up Alice in SBS
SBS
3 SBS reply
4 Issues token iff Alice ↛ 1.1.1.1 not in S.S.
obtaining/renewing a token
Alice
sLRep
1 Source AS checks token and signatures
1.1.1.1
2 Destination may check signatures
✓ = token check
packet transfer
1.1.1.1
1 block B_k
dAdm
2 block B_k
6 SBS storage receipts
sNR maps B_k -> Alice using key commitment
3
sLRep
sNR places Alice ↛ 1.1.1.1
4
5 SBS receipt
SBS
placing a block

Figure 3.1: Life cycle of accountable packets. SBS is the secure block storage mechanism.

Computation cost in transit networks is limited by avoiding asymmetric cryptography in transit; this leads to including network-by-network tokens in packets. DNA avoids revealing user identifiers: even though IP addresses could be used to correlate accesses by the same user, network-layer accountability should not require revealing identity to destinations. Return traffic should be able to be sent cheaply; this allows servers and high-volume connections to avoid much of the DNA overhead. Finally, DNA must support many fine-grained blocks in the system, leading to a need to distribute the blocks across many servers.

DNA divides time into *epochs* on the order of minutes. All participants are synchronized to within an epoch; all certifications of public keys have an expiry time expressed as the expiration epoch. The DNA implementation uses seconds since midnight UTC, January 1, 1970, as returned by `gettimeofday()`, divided by 128, to provide an easily computed, uniform epoch in 32 bits.

A sender may be able to continue sending packets to a destination even after the destination has blocked the sender. However, the “period of vulnerability” is bounded by the epoch. A blocked sender may also be able to send packets if it can find aid from a corrupt component; however, DNA will expose the corrupt entity to keep principals blocked.

3.3.1 Components Overview

The DNA architecture consists of five components which I describe in turn.

The TAP

The TAP is the trusted accountability provider. Its role in the architecture is to certify and renew the keys of well-behaved, but untrusted, participants (each of which is described below). It must be able to accept proofs that a component misbehaved (signed statements that catch the component in a lie), deny certificate renewal to those participants, and not reissue new certificates to the same. All entities know the TAP's public key and can verify TAP signatures.

The TAP is the only trusted entity in the system. DNA assumes that the TAP never divulges its private key and always discharges its protocol obligations correctly. The TAP may be implemented in a distributed manner with arbitrarily many replicas. These replicas need to synchronize only the current TAP public/private key pair and a list of principals whose keys are not to be renewed in the current epoch.

Packet-sending Principals (Users)

A principal (or a user) $\mathcal{P}$ in DNA is identified by a public key, $\mathcal{P}_{pk}$, which the TAP certifies (by signing) through an out-of-band mechanism. Section 3.6 describes principals that forward legacy traffic (from existing machines, bearing no signatures) into the accountable network, likely with limitations. These *legacy gateways* appear to the architecture as ordinary users that can be blocked.

Autonomous Systems (ASes)

If the entire network were under the control of the same administrator, protecting against misbehavior would be simpler. Although no part of DNA requires that the trust domains be split along autonomous system boundaries, I use the term AS because the idea connotes responsibility for routers and ownership of IP address ranges for hosts. The terms of pairwise connections between ASes are encoded in service level agreements, similarly, these pairwise connections may represent different levels of trust or different ways to address misbehavior.

Administrators of ASes have *speaks-for* keys, signed by the TAP, to express that the administrator is permitted to place blocks on behalf of specific destination addresses or to grant waivers that allow return traffic to be unsigned. These keys may be delegated, restricting sub-keys to more-specific address prefixes. The TAP binds a speaks-for key to a prefix by including the prefix its the signature of the key.

Neighboring ASes periodically exchange a symmetric key; border routers can create and verify message authentication codes (as realized in *tokens*, described below in Section 3.3.5) using this key.

An AS may permit hosts to exchange packets within the same AS without verification; one might allow traffic to LReps, to diagnostic services, or to bootstrapping (DHCP or DNS) servers, for example, without signatures. However, such legacy traffic would likely not be accepted by neighboring ASes. An AS *may* choose to transit, limit, or (likely with increased deployment) outright reject legacy traffic, but *must not* transit unaccountable traffic—that is, packets with invalid signatures.

Routers in an AS may be malicious or be compromised; handling attacks by routers is discussed in Section 3.4.3.

LReps

DNA local representatives (LReps) are servers, distributed in the network, that act as a privacy CA by certifying short-lived *blinding* keys for legitimate principals and generate initial *tokens* after checking for a block or validating a waiver. Blinding keys “hide” principal keys to provide a measure of unlinkability across connections. Tokens are an optimization that allow ASes to forgo asymmetric cryptographic operations when accepting packets from (locally) trusted neighboring networks.

Each LRep has a key signed by the TAP. Each AS[2] contains at least one LRep; like neighboring ASes, LReps periodically exchange a symmetric key with its host

[2] As Section 3.6 shows, it is not strictly necessary for *each* AS to have an LRep.

AS.

An LRep may be malicious or compromised.

Secure storage

The list of blocks—statements of the form “10.0/16 rejects packets from Principal A”—is maintained in a distributed, secure block storage (SBS) mechanism. DNA uses the NeighborhoodWatch secure DHT [17] (Chapter 4), which nicely dovetails with the assumptions of DNA. The NeighborhoodWatch DHT (NWDHT) is resilient to malicious nodes that drop or misroute queries. NWDHT provides security via a centralized authority, which DNA already assumes (the TAP). Conversely, DNA can be used to protect the NWDHT central authority, weakening NWDHT's assumption that the central authority be protected.

SBS nodes may be co-located with LReps or be independently provisioned. Each SBS node has a key signed by the TAP authorizing it to act in that role.

3.3.2 An Accountable Packet

Accountable packets have a valid signature chain from the TAP to the packet: the TAP's signature of the LRep's public key, the LRep's signature of the blinding key, and the blinding key's signature of the packet contents. This chain make the packet accountable to the sender by providing a means to identify and block the sender. The sender is responsible for the packet, the LRep is responsible for verifying the sender is not blocked, and the TAP is responsible for ensuring correct behavior of the LRep. If the sender or LRep is not behaving according to protocol, the destination can appeal to the next-highest authority present in the signature chain.

Verifying the entire signature chain of a packet in transit would be computationally infeasible. Thus, certain routers in every AS place tokens in the packet to signify that they have checked the signatures, or trust someone that has. An initial token, provided by the LRep, indicates that the destination has not blocked the sender. The source AS is responsible for checking both the token and the signature chain. Subsequent tokens, added by each AS, indicate that the signatures verify—or the packet was received from another AS that claimed the signatures verify.

To further reduce the computational overhead of sending traffic, *waivered* traffic is exempted from signatures. If a sender S trusts return traffic from an intended destination D, S may give D permission to send return traffic without inserting *any* signatures into the packet. D's LRep verifies that S has permitted unsigned return traffic before issuing an initial token to D.

3.3.3 Process Overview

I next briefly outline how DNA operates in normal operation, with misbehavior possible only by the principal that sources traffic. A detailed description of each step follows.

Assume that Alice is a principal, with public key $\mathcal{P}_{pk}$ certified by the TAP, who wishes to send packets to a host D with address D_{ip}; in turn, D will wish to block Alice from sending further messages. Figure 3.1 illustrates this process and participants at a high level.

Registering a blinding key

1. Alice commits to a blinding key that she will later use to obtain tokens from a nearby "source" LRep (sLRep) (Section 3.3.4).

Obtaining an initial token

1. Alice contacts sLRep with a registered blinding key and an intended destination (D_{ip}) (Section 3.3.5).

2. sLRep examines the SBS to find any block that the administrator of D_{ip}'s AS may have placed on Alice.

3. If there is no block, sLRep provides a token that enables Alice to send to D_{ip} during the current epoch (Section 3.3.5).

Constructing a waiver (optional)

1. Alice signs a message for the destination's LRep that will instruct it to provide tokens to the destination without a lookup into the SBS (Section 3.3.7).

Packet Transfer

1a. Alice signs her outgoing packets with her blinding key and includes the initial token and the signature chain from the TAP to the LRep to her blinding key in the packet. A router in her AS verifies all signatures and the token.

Or,

1b. For waivered packets, Alice applies a hash function to the waiver token and the message.

2. An egress router in Alice's AS places a token in the packet, indicating that that AS has validated the packet. When the packet crosses a peering link to a different AS, the ingress router checks the token and discard packets with invalid tokens. When the packet leaves a network, the egress router inserts a token for the next AS (Section 3.3.6). This occurs on each peering link.

Placing a block

1. D decides to block Alice. He persuades his AS administrator to place the block and provides the packet signed by the blinding key. The administrator signs a block request message, which includes Alice's blinding key, with his speaks-for key. It then sends the block request to sLRep. (Section 3.3.8).

2. sLRep places a block against $\mathcal{P}_{pk}$ (not the blinding key) into the SBS and returns a receipt to the administrator (Section 3.3.8). sLRep places a temporary filter in its AS to block further packets from Alice until her initial token expires.

Tokens are valid for only a single epoch. Therefore Alice must receive new tokens for every epoch in which she wishes to send. Once sLRep receives a request to block Alice from sending to D_{ip}, sLRep no longer gives new tokens to Alice (for sending messages to D_{ip}). Alice cannot send messages to D_{ip} by changing her network point-of-attachment since her (new) local LRep will find the block in the SBS and not issue her an initial token.

In the rest of this section, I describe the messages exchanged for each of these sub-protocols. I denote a principal A's public and private key pair by A_{pk} and A_{sk} respectively, a message m signed using key k by $\mathsf{Sign}_k(m)$ and a message m encrypted using key k by $\mathsf{Enc}_k(m)$.

3.3.4 Registering a blinding key

Assume that Alice has generated a key pair ($\mathcal{P}_{pk}$, $\mathcal{P}_{sk}$), that the TAP has certified $\mathcal{P}_{pk}$ by signing it, and that Alice wants to send messages to D_{ip}. Assume that sLRep is within Alice's AS (AS$_1$) and that Alice's host is pre-configured with sLRep's address. Alice generates a blinding key pair ($\mathcal{B}_{pk}$, $\mathcal{B}_{sk}$) and commits to it by signing $\mathcal{B}_{pk}$ with $\mathcal{P}_{sk}$. The format of Alice's commitment (Commit_{Alice}) is $\mathsf{Commit}_{Alice} = \mathsf{Sign}_{\mathcal{P}_{sk}}(\mathcal{B}_{pk}, \nu_{\mathcal{B}})$ where $\nu_{\mathcal{B}}$ is the epoch until when Alice proposes to use this blinding key. The key registration process is as follows:

$$
\begin{array}{ll}
\mathit{Alice} \mapsto \mathsf{sLRep} : \mathsf{Commit}_{Alice}, \mathsf{Sign}_{\mathsf{TAP}}(\mathcal{P}_{pk}) & \textit{; registration request} \\
\quad \textit{; } \textsf{\textit{sLRep}} \textit{ checks that the } \textsf{\textit{TAP}} \textit{ signed } \mathcal{P}_{pk} & \\
\quad \textit{; } \textsf{\textit{sLRep}} \textit{ stores } \mathcal{B}_{pk} \leftrightarrow \mathcal{P}_{pk} & \\
\mathsf{sLRep} \mapsto \mathit{Alice} : \mathsf{Sign}_{\mathsf{sLRep}}(\mathcal{B}_{pk}, \nu_{\mathcal{B}}) & \textit{; signed blinding key}
\end{array}
$$

Alice may register as many blinding keys as sLRep will allow.

3.3.5 Obtaining an initial token

Initial tokens are bound to a blinding key, $\mathcal{B}_{pk}$, a destination, D_{ip}, the current epoch, c, and an issuing LRep. Initial tokens make the statement "a valid $\mathcal{P}_{pk}$ registered $\mathcal{B}_{pk}$ and D_{ip}'s administrator has not blocked $\mathcal{P}_{pk}$." An initial token is required to send packets to a different domain and is checked by routers in Alice's domain.

When Alice wishes to communicate with D_{ip} she obtains an initial token as follows:

$Alice \mapsto \mathsf{sLRep}$: $\mathsf{Sign}_{\mathsf{sLRep}}(\mathcal{B}_{pk}, \nu_{\mathcal{B}}), D_{ip}$; *token request*

; *sLRep verifies its signature on $\mathcal{B}_{pk}$*

; *sLRep verifies that $c \leq \nu_{\mathcal{B}}$*

; *sLRep retrieves $\mathcal{P}_{pk}$ and checks the SBS*

; *to ensure Alice has not been blocked by D_{ip}*

$\mathsf{sLRep} \mapsto Alice$: $\mathcal{T}_0$; *initial token*

The format of the initial token ($\mathcal{T}_0$) is:

$$H(K_{\mathsf{sLRep}\leftrightarrow \mathrm{AS}_1}, \mathcal{B}_{pk}, D_{ip}, c, \mathsf{sLRep}_{pk}, \mathsf{sLRep}_{ip})$$

where $K_{A\leftrightarrow B}$ is the shared symmetric key between A and B (A and B are neighboring ASes or an LRep and the AS it is in). $H(\cdot)$ is a secure MAC, such as HMAC, where the first parameter is the secret key.

A blocked sender will not be able to obtain an initial token. If D_{ip} has blocked Alice, then the sLRep will find such a block in the SBS, and not issue a token to Alice.

3.3.6 Packet Transfer

With an initial token, Alice can send a packet to D_{ip}. She signs her message with her blinding key and includes her initial token in the packet. Figure 3.2 shows the

IP Header	20
Version + Flags	2
Transport protocol	1
TAP index	2
sLRep_{ip}	4
sLRep_{pk}	14
$\mathcal{B}_{pk}$	14
$\mathsf{Sign}_{\mathsf{TAP}_{sk}}\left(\mathsf{sLRep}_{pk}, \mathsf{sLRep}_{ip}, \nu_{\mathsf{sLRep}}\right)$	40
$\mathsf{Sign}_{\mathsf{sLRep}}\left(\mathcal{B}_{pk}, \nu_{\mathcal{B}}\right)$	28
$\mathsf{Sign}_{\mathcal{B}_{sk}}\left(m\right)$	28
ν_{sLRep}	4
$\nu_{\mathcal{B}}$	4
c	4
$\mathcal{T}_0 = H(K_{\mathsf{sLRep} \leftrightarrow \mathrm{AS}_1}, \mathcal{B}_{pk}, D_{ip}, c, \mathsf{sLRep}_{pk}, \mathsf{sLRep}_{ip})$	4
$\mathcal{T}_1 = H(K_{\mathrm{AS}_1 \leftrightarrow \mathrm{AS}_2}, \mathcal{B}_{pk}, D_{ip}, c, H(m))$	4
$\mathcal{T}_2 = H(K_{\mathrm{AS}_2 \leftrightarrow \mathrm{AS}_3}, \mathcal{B}_{pk}, D_{ip}, c, H(m))$	4
$\ldots$	
$\mathcal{T}_{n-1} = H(K_{\mathrm{AS}_{n-1} \leftrightarrow \mathrm{AS}_n}, \mathcal{B}_{pk}, D_{ip}, c, H(m))$	4
Transport (message m)	*var*

Figure 3.2: Accountable packet format. The rightmost column represents the number of bytes required for each field, when using cryptographic primitives described in Section 3.7.

fields Alice embeds.

Forwarding an accountable packet consists of first verifying the material within the packet, and second, appending the next token that the subsequent AS will verify. These two steps may be separated, so that packets are verified when they enter a network and given tokens just before they exit.

The first gateway to see the packet must verify the embedded signature chain from the TAP to the message: $\mathsf{Sign}_{\mathsf{TAP}_{sk}}\left(\mathsf{sLRep}_{pk}, \mathsf{sLRep}_{ip}, \nu_{\mathsf{sLRep}}\right)$, $\mathsf{Sign}_{\mathsf{sLRep}}\left(\mathcal{B}_{pk}, \nu_{\mathcal{B}}\right)$, and $\mathsf{Sign}_{\mathcal{B}_{sk}}\left(m\right)$. It then verifies the initial token $\mathcal{T}_0$ by checking that (1) sLRep_{pk} and sLRep_{ip} belong to the same sLRep (which is located in the same AS as the gateway), (2) $\mathcal{T}_0$ was created for $\mathcal{B}_{pk}$, and (3) the MAC computed with the $K_{\mathsf{sLRep} \leftrightarrow \mathrm{AS}_1}$ that it shares with sLRep is correct. If any check fails the packet is dropped. These steps are unique to the first gateway because it is responsible for validating the entirety of the packet; later gateways check only that the packet was validated by a (transitively)

IP Header	20
Version + Flags	2
Transport protocol	1
$\mathrm{HMAC}_{nonce}(m)$	4
$\mathcal{T}_0 = H(H(K_{\mathsf{sLRep} \leftrightarrow \mathrm{AS}_1}, D_{ip}, c), m)$	4
$\mathcal{T}_1 = H(K_{\mathrm{AS}_1 \leftrightarrow \mathrm{AS}_2}, \mathcal{B}_{pk}, D_{ip}, c, H(m))$	4
$\mathcal{T}_2 = H(K_{\mathrm{AS}_2 \leftrightarrow \mathrm{AS}_3}, \mathcal{B}_{pk}, D_{ip}, c, H(m))$	4
$\cdots$	
$\mathcal{T}_{n-1} = H(K_{\mathrm{AS}_{n-1} \leftrightarrow \mathrm{AS}_n}, \mathcal{B}_{pk}, D_{ip}, c, H(m))$	4
Transport (message m)	*var*

Figure 3.3: Waivered return packet format.

trusted neighbor.

Before the packet leaves the network, a router stamps the next token on the packet. This token conveys that the packet is valid: that the signatures and previous token have been checked and that the next domain should transit it, even if the token's creator has not directly verified the signatures in the packet. The format of the next (and any subsequent) token is:

$$\mathcal{T}_1 = H(K_{\mathrm{AS}_1 \leftrightarrow \mathrm{AS}_2}, \mathcal{B}_{pk}, D_{ip}, c, H(m))$$

The packet arrives at the destination with a chain of tokens that can be used to diagnose in-network misbehavior (Section 3.4).

Tokens are significantly faster to verify than signatures, since they use only symmetric key operations. The use of tokens does not require shared keys between every AS pair or the predetermination of AS paths, as in Passport [55].

3.3.7 Waivers

High-volume servers and power-constrained devices may not be able to sign each packet they transmit. DNA allows hosts to state with a waiver that unsigned return traffic will not be blocked. Hosts must possess or have access to a valid speaks-for keypair (SF_{pk}, SF_{sk}) delegated to their prefix; we expect the key to be delegated specifically for one address for a short duration, but a waiver may be generated by a service, such as the LRep.

We assume that Alice can transmit a connection nonce n to Bob in secret; perhaps by encrypting it using Bob's well-known or stored public key, in-band in an established secure transport protocol atop accountable packets, or by other means. Alice then sends a waiver ω of the form:

$$\mathsf{SF}_{pk}, \mathsf{Sign}_{\mathsf{TAP}_{pk}}\left(\mathsf{SF}_{sk}\right), Alice_{ip}, e, \mathsf{Sign}_{\mathsf{SF}_{sk}}\left(H(n), Alice_{ip}, e\right)$$

where e is the epoch until which any node possessing n may send waivered packets to $Alice_{ip}$. The waiver is only valid if SF_{pk} is signed; if SF_{sk} is delegated, the certificate chain from the TAP is included in the waiver. Alice sends ω to Bob.

Bob verifies the signature of the hash of the nonce and forwards ω to its LRep. The LRep verifies that $Alice_{ip}$ created a waiver allowing Bob to send to Alice and grants a *secret* waiver token for sending to $Alice_{ip}$ without querying the SBS. The format of a waiver token is $H(K_{\mathsf{sLRep} \leftrightarrow \mathrm{AS}_1}, Alice_{ip}, c)$. However, because this token is not bound to a principal, only to the holder of a nonce, it could be abused if eavesdropped. To keep this waiver token secret and prevent replay, LRep encrypts it before transferring it to Bob. The waiver token is not included directly in the packet, but hashed with the message.

Bob does not include any signatures in packets for Alice; instead, Bob includes a HMAC of the packet computed using Alice's nonce as the key. Bob's AS allows waivered packets from Bob to depart without alarm. The format of the waivered packet is shown in Figure 3.3.

Figure 3.4 presents a state transition diagram showing how a sender transitions to and from being able to send waivered traffic. The sender traverses the path through obtaining an initial token to being able to send accountable packets. It may receive a waiver to send outgoing packets, $\overrightarrow{\omega}$; this causes it to obtain a waiver

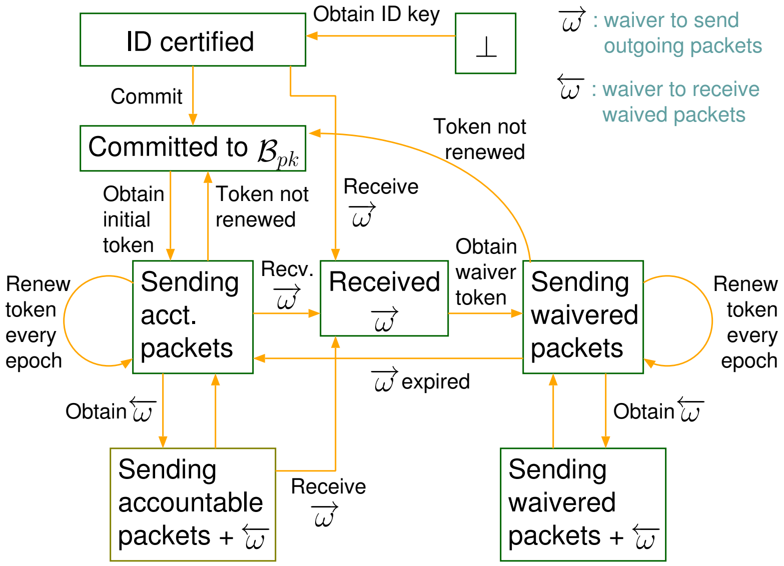

ID certified
Obtain ID key
⊥
$\overrightarrow{\omega}$: waiver to send outgoing packets
$\overleftarrow{\omega}$: waiver to receive waived packets
Commit
Committed to $\mathcal{B}_{pk}$
Token not renewed
Obtain initial token
Token not renewed
Receive $\overrightarrow{\omega}$
Renew token every epoch
Sending acct. packets
Recv. $\overrightarrow{\omega}$
Received $\overrightarrow{\omega}$
Obtain waiver token
Sending waivered packets
Renew token every epoch
$\overrightarrow{\omega}$ expired
Obtain $\overleftarrow{\omega}$
Obtain $\overleftarrow{\omega}$
Sending accountable packets + $\overleftarrow{\omega}$
Receive $\overrightarrow{\omega}$
Sending waivered packets + $\overleftarrow{\omega}$

Figure 3.4: DNA sender states and transitions.

token from the LRep and then begin sending waivered packets. From either sending state, the sender may add the $\overleftarrow{\omega}$ to permit unsigned incoming traffic. If a token or waiver is not renewed, the sender reverts to a less-authorized state.

3.3.8 Placing a Block

Suppose the owner of D_{ip} wishes to block Alice. He forwards one of Alice's packets to his administrator, $D.admin$. Given an accountable packet from Alice, $D.admin$ can extract sLRep's address and Alice's blinding key. Assume that $D.admin$ possesses the speaks-for key corresponding to the address block containing D_{ip} (`1.1/16`). $D.admin$ places the block:

$$\begin{aligned}\mathsf{BlockReq}_{\mathtt{1.1/16}} = {} & \text{“}\mathtt{1.1/16}\text{”},\ D.admin_{pk},\ \mathcal{B}_{pk},\ D_{ip},\ type,\\ & \mathsf{Sign}_{\mathsf{TAP}_{sk}}\left(\text{“}\mathtt{1.1/16}\text{”},\ D.admin_{pk}\right),\ \mathsf{Sign}_{D.admin_{sk}}\left(\mathcal{B}_{pk}, D_{ip}, type\right)\end{aligned}$$

where *type* $\in$ {“new”, “republish”}, as follows:

$D.admin \mapsto \mathsf{sLRep}$: $\mathsf{BlockReq}_{\mathsf{1.1/16}}$; *block request*

; *sLRep retrieves Alice’s* $\mathcal{P}_{pk}$ *using* $\mathcal{B}_{pk}$ *and places a block in the SBS*

; *The SBS nodes return* k *signed storage receipts* (r_i).

$\mathsf{sLRep} \mapsto D.admin$: $r_1, ..., r_k, \mathsf{Enc}_{\mathsf{LRep}_{pk}}(\mathsf{Commit}_{Alice})$; *block receipt*

The LRep in the destination network, dLRep, may grant tokens without consulting the SBS for the purpose of sending block messages to other LReps. This permits sending block requests while overloaded. dLRep must inspect the combination of the block request and the packet headers to ensure that the request is not spurious: for example, that $\mathcal{B}_{pk}$ followed a chain from the TAP and that the destination for the token is sLRep_{ip}.

Depending on the implementation, the SBS may not permanently store blocks. A *D.admin* that insists upon permanent blocks may republish them before they expire. To facilitate republishing, the sLRep returns an encrypted copy of Alice’s commitment to her blinding key $\mathcal{B}_{pk}$. A request to republish a block request includes this encrypted commitment, so that sLRep can decrypt it to find Alice’s commitment. After Alice’s original commitment expires, sLRep may expunge it; it is the responsibility of *D.admin* to keep this state so the block can be republished.

To block further waivered traffic, the victim must both block the principal used to solicit a waiver, if applicable, and refuse to renew the waiver.

3.3.9 Multiple TAPs

The description so far has focused on a single TAP. DNA in fact allows multiple TAPs to be simultaneously deployed and for destinations to accept packets signed with identities issued by TAPs of their choosing. We assume that each TAP has a globally-unique index to be carried in all accountable packets and used to identify TAP_{pk}.

Senders must know which TAPs a destination honors, and LReps issue tokens to a sender only if the sender has a key signed by a TAP that the destination accepts. DNA assumes DNS can be augmented to return, along with a list of IP addresses, the indexes of one or more TAPs, in response to a lookup request. The current effort to secure DNS with DNSSEC [40] prevents tampering with DNS requests and responses. To avoid bootstrapping problems, DNA assumes DNS serves will accept queries from identity keys certified by a well-known TAP, perhaps run by ICANN.

3.3.10 Discussion

This approach protocol has an additional benefit, beyond the goals at the opening of this section. Accountable packets are locally verifiable: any entity along the path can determine if a packet is accountable, by checking all signatures and that the fields in the packet are consistent. Invalid packets are dropped near the source and far from the destination. Misbehavior can be detected locally, either by neighbors exchanging traffic, by a first hop router, or by the LRep in the source network. Further, blocking decisions—deciding whether a source should be denied the right to send to controlled destinations—can be made without proving harm. Because the incremental cost of a block is small, they need no external justification.

3.4 Attacks

The previous sections have described the default operation of DNA, without failures or node corruption. This section considers misbehavior by different components and shows how DNA detects misbehavior and eventually evicts malicious nodes from the system.

Attacks are classified into three categories based on how attackers are identified:

- **DNA**: The protocol messages in DNA include sufficient information to unam-

Attack Description	Resolution
Sources may	
Flood destination*	DNA: Victim places a block; tokens not granted
Change network point of attachment	DNA: Block found in SBS from any location
Replay old tokens	DNA: Tokens expire after one epoch
Use tokens to send to arbitrary destination	DNA: Tokens bound to destination addresses
Exhaust SBS storage by incurring many blocks*	DNA: LRep rate-limits tokens to blocked senders
Falsely accuse waivered traffic of being unaccountable	Local: Destination can prove waiver is valid and accusation is false
LReps may	
Fabricate blinding keys; issue tokens to unregistered users*	DNA: LRep cannot produce a commitment from a principal to that key; the SBS will not return a receipt
Flood destination*	TAP: Administrator can expose LRep to the TAP
Not place blocks; ignore block request*	TAP: Victim LRep receives no receipt, redirects request to the TAP
Not lookup blocks before issuing tokens; issue tokens to a blocked sender*	TAP: SBS does not return receipts, reports duplicate block to the TAP
Routers may	
Not check signatures*	Local: Tokens will lead back to this network
Flood using bogus tokens	DNA: Neighbors will discard packets with invalid tokens
Flood using bogus source	Local: Cannot create accountable traffic; tokens will lead back to this network
Replay packets*	Local: Neighbors install Bloom filters
Fabricate upstream tokens*	Local: Fabricator and neighbor network accuse each other
Not check upstream tokens	Local: If an invalid token is accepted, network will not deflect blame

Table 3.1: DNA attacks and resolution. Items with a * are explained in Section 3.4.

biguously implicate corrupt nodes.

- **Local**: The DNA protocol can resolve the misbehavior to two neighboring entities, either of which may be corrupt. However, the "good" entity knows the corrupt neighbor and modify its trust policy accordingly.

- **TAP**: Some attacks require the assistance of a TAP node for resolution. Victims provide the messages that indicate a corrupt entity to the TAP, which can verify the deceit, decide to not recertify the node for future epochs, and communicate the decision to replica TAPs.

Table 3.1 matches the most prominent attacks DNA components may attempt with the mechanism that would isolate the attacker. The rest of this section describes selected attack resolution mechanisms in detail.

3.4.1 Source-Generated Attacks

Source floods

DNA enables destinations to explicitly block source principals. If a source floods a remote host with accountable traffic (i.e., packets with valid blinding and LRep keys), the victim network can block the source. A block request message (from the *D.admin*) fits into a single Internet MTU IP datagram and does not require an acknowledgment for the block to be activated. Thus, the block protocol will be able to block the source, even if the source can saturate the victim's access link.

If the source sends unaccountable traffic (i.e. traffic with an invalid signature), then the source's domain—which is responsible for verifying each signature—will discard the packet.

If the source sends legacy traffic (without an accountability header), then the traffic may be forwarded or discarded as described in Section 3.6.

SBS storage exhaustion

A malicious sender may try to overload the SBS by attacking many destinations, causing the SBS to hold a large amount of block information, generate many block receipts, and cause its LRep to verify those DHT receipts.

For every block than an LRep publishes, it has to verify the returned receipts. An LRep will rate limit or stop issuing token to a sender that is blocked by multiple senders, causing excessive work for the LRep to verify and return receipts.

Changing Network Attachment Point

After being blocked or banned, a principal may try to continue an attack by changing their LRep or ISP (which would map the principal to a different LRep). However, the block in the SBS would stop the source from receiving an initial token.

3.4.2 Corrupt LReps

Flooding

Like any sender, a corrupt LRep may try to flood a destination. If the LRep node sends packets using a blinding key registered to LRep_{pk}, the blinding key would be blocked as if it were a normal sender. If the misbehaving LRep does not honor the block, the administrator who blocked the LRep does not receive receipts and forwards the request to the TAP, who eventually evicts the LRep (as it would for any LRep that did not respond to any block request). If the LRep does return receipts, but continues to send traffic, the administrator presents the TAP with the returned receipts and an example of the traffic that violates the receipts. This is evidence of misbehavior and the TAP evicts the LRep.

An LRep is unable to send packets using a blinding key registered by another user (without the corresponding secret key $\mathcal{B}_{sk}$, it is unable to sign packets). Thus a corrupt LRep can never implicate an honest user of misbehavior.

Ignoring block requests

If an LRep ignores block requests (or is unable to return the necessary number of receipts), the victim administrator appeals to the TAP. The TAP challenges the source LRep to place the blocks and return receipts, otherwise it will evict the LRep.

Skipping lookups or issuing tokens to a blocked sender

A source and an LRep may collude. The LRep could issue a token allowing a sender (using a different blinding key) to send packets to a destination that has previously blocked the sender. LReps that skip SBS lookups may do the same. In this case, the destination will presumably block the sender a second time.

The next course of action depends on whether the corrupt LRep publishes this second block request or not. If it does publish the block requests, the SBS will will observe that the destination has blocked $\mathcal{P}_{pk}$ twice—an indication that the LRep misbehaved. The SBS will not return a receipt, and may alert the TAP. If the LRep does not publish the request, the destination appeals to the TAP as above. In addition, it also has the option of publishing a block request for the LRep itself, which the LRep must honor as a request to not issue tokens to *any* sender to contact the destination. This block request has the same format as a request to block an sender's $\mathcal{P}_{pk}$, but includes the LRep's public key (which is included in any accountable packet that has an initial token from the offending LRep) instead of a sender's key. Any LRep can publish this block, as it does not require reversing a sender's commitment to a blinding key. Once the block is published, the LRep will implicate himself of misbehavior if it continues to issue tokens to a destination that has blocked the entire LRep. The destination can use this to appeal to the TAP, without requiring the TAP to attempt to publish the block itself. Incorporating this measure requires that an LRep make two lookups—one for the sender and one for itself—before issuing an initial token.

It is possible that a victim administrator asks a TAP node to challenge a source TAP because multiple principals are attacking a domain (but the source LReps are not corrupt). That is, each block that is placed is being honored, but each attack (with a new blinding key) corresponds to a new principal. In this case, all of the suspected attack packets will map to different (unblocked) blinding keys, and the source LReps will not be falsely implicated.

3.4.3 Corrupt Routers

Corrupt routers may fabricate packets with (bogus) accountability headers, resulting in packets with an invalid signature chain—preventing the destination from blocking the sending principal—to arrive at a destination. Similarly, routers may not check packet signatures, allowing unaccountable traffic to enter the network. Additionally, a corrupt router may simply replay valid packets to flood the victim.

Generating or forwarding invalid packets (Accusation Protocol)

If a victim receives an unaccountable packet, it must rely on the tokens to isolate the faulty node. The victim's administrator inspects the last token in the packet and determines which of its upstream ASes the packet came from (by determining which shared key was used to check the token). The victim administrator then "accuses" the upstream AS of sending an invalid packet. The upstream AS (say AS_1) verifies the last token, and tries to map the second-to-last token to one of *its* upstream neighbors. If this process is successful (and the second-to-last token maps to AS_2), AS_1 "accuses" AS_2. This process recurses until the packet traverses the last "good" AS (say AS_g) into the AS that generated it (say AS_b).

Note that all that is required to verify a token is the blinding key, destination IP, epoch the packet was created, the hash of the packet contents, and the token itself. Thus accusation messages (containing the full token chain), like block requests, fit into a single packet.

When the original packet was generated, the corrupt router in AS_b may have fabricated all upstream tokens, in which case the AS_b administrator will have no other upstream AS to blame. In the worst case, the AS_b router may have been able to add one "valid" upstream token, since it may know the secret shared with the upstream AS.

In the first case, either the AS_b administrator will locate and expunge the corrupt router (and stop the attack), or the attacks will continue. If the attacks continue, each AS will continue to receive accusations from downstream routers and AS$_g$ must apply some local policy. This policy could include technical solutions (e.g. check signatures on incoming packets) or monetary remedies (e.g. the AS peering SLA could state that the offending AS will pay a small sum for each unaccountable packet that it transits).

Suppose the AS chooses to validate signatures if the accusations continue. This will enable AS$_g$ to immediately isolate AS$_b$ (since AS$_g$ will immediately know that AS$_b$ is transiting unaccountable packets). AS$_g$ will not forward these packets, stopping the flow of accusations. If AS$_b$ continues to source unaccountable packets, AS$_g$ may take further action.

If the SLA between AS$_g$ and AS$_b$ includes the monetary provision, then AS$_b$ will have to pay for each unaccountable packet. AS$_g$ may have also had to pay its downstream AS (after all, AS$_g$ did transit an unaccountable packet); however, in this case, AS$_g$ recoups its cost because of the payment from AS$_b$.

If the router in AS$_b$ added a "valid" token for a previous hop AS (since the neighboring ASes share a secret key), AS$_b$ will be able to "blame" a good upstream AS (say AS$_{up}$). However, AS$_{up}$ will not be able to map the packet to any of *its* upstream neighbors. If the AS$_{up}$ administrator can immediately assert that its routers have not been corrupted, it can isolate AS$_b$.

However, it is often difficult for a network administrator to certify that none of the hosts or routers in its AS has been compromised. In this case, AS$_{up}$'s administrator can locally audit the router between itself and AS$_b$, ensure it is not compromised, and verify the signatures of all packets traversing the peering link. If the accusations continue, then AS$_b$ is unambiguously implicated, and AS$_{up}$ may choose to renegotiate the peering.

Verifying signatures on every packet may be prohibitively expensive on high speed links. AS$_{up}$ can implicate AS$_b$ without verifying packet signatures. Once the AS$_{up}$ administrator verifies that its peering router with AS$_b$ (say R) is not compromised, it generates a temporary symmetric key (k), and instructs R to add *two* tokens on each outgoing packet. The first outgoing token is computed using k (known only to R) and the second token is the regular token computed using the key shared between AS_{up} and AS_b. If subsequent accusations from AS_b includes a token signed with k, then some host or router within AS_{up} is compromised, and the AS_{up} administrator has to audit its internal routers. However, if AS_{up} is not compromised, then the accusation packets from AS_b will not contain a token signed with k (since AS_b does not know k). However, all packets transited from AS_{up} included a token signed with k; this enables AS_{up} to unambiguously implicate AS_b.

Replaying valid packets

A router can replay a fixed packet only until the next epoch, when it will be dropped by subsequent domains. However, this does not solve router replay in general. If routers persist in replaying accountable packets, neighboring ASes can install Bloom filters [21] on their routers. Passport [55] also uses Bloom filters to catch replayed packets. Bloom filters are described in greater detail in Section 3.5.

When a router forwards a packet, it tests its Bloom filter to see if the packet exists in the Bloom filter. Such a test is probabilistic and does not guarantee that packet is an actual replay. Therefore, the router also stores a set of hashes of possible duplicate packets. If a router suspects a packet is a duplicate because it appears in the Bloom filter, it then hashes the packet and compares against the set of hashes, and only rejects the packet if its hash appears in the set. If it does not, the router adds the hash to the list and forwards the packet. If the packet does not match the Bloom filter, the router forwards it and inserts a copy into the filter.

While this allows exactly two duplicate packets to be forwarded (the first, original packet, and the second that matched the Bloom filter and whose hash was stored), it prevents false positives when detecting duplicates and allows the router to tune the size of its Bloom filter based on how many hashes it can store. The filters and set of hashes can be refreshed every epoch, limiting the amount of storage needed.

3.4.4 Attacks using waivered packets

Assume Alice sent a waiver allowing Bob to send unsigned packets to Alice. When Alice receives an unsigned packet, she can use the HMAC to assert that she had explicitly permitted the sender to send unsigned packets. If the HMAC in the packet is corrupt (or not present), Alice may use the usual "accusation" protocol to stop/isolate the sender.

If Alice were malicious, she may try to evict Bob (who is good) by initiating the accusation protocol on packets that include a proper HMAC. The tokens in the packet would eventually terminate at Bob, who can produce Alice's waiver explicitly allowing Bob to send unsigned packets. Bob does not have to reveal his private key to decrypt the nonce in order to prove that Alice sent a waiver; he only has to produce the nonce and show that Alice signed a hash of the nonce. Bob now accuses Alice (tracing the packet tokens *forward*) of misbehavior. Bob's accusation includes Alice's waiver, her accusation message, and the nonce *in cleartext*:

$$\mathsf{SF}_{pk}, \mathsf{Sign}_{\mathsf{TAP}_{pk}}\left(\mathsf{SF}_{sk}\right), \text{``accuse''}, nonce$$

Each AS on the forward path can now independently check that (1) Alice did allow Bob to send unsigned messages, (2) Alice signed the nonce, and (3) Bob had included a proper HMAC on the return packet. This evidence is sufficient for Alice's AS to isolate Alice's host (or for Alice's upstream AS to isolate Alice's AS).

Bob's AS does not check signatures on waivered packets. However, this does not allow any new attacks. If Bob sends malicious traffic to a destination that has given a waiver to Bob, the victim domain will not renew Bob's waiver, and the LRep in Bob's domain will stop issuing tokens to Bob when the waiver expires.

3.5 Bloom filters: An alternative to tokens

When an accountable packet crosses an AS boundary, a router in the upstream AS stamps a token into the packet. This token expresses that AS's belief that the packet is valid: the signatures verify and the sender is not blocked. In the case of misbehavior, routers can use these tokens to the source of a packet. When an unaccountable packet arrives at a destination, the destination can initiate the accusation protocol, in which ASes use the tokens to find the principal that sent the packet.

However, tokens consume space in the packet and require neighboring ASes to share secret keys. This section explores an alternate method of traceback, similar to Snoeren et al.'s Source Path Isolation Engine (SPIE) [74], that eliminates the need for ASes to stamp tokens into packets. Unlike other traceback systems [78, 71], which require the destination to collect multiple packets in order to reconstruct the path, SPIE enables traceback of a single packet. SPIE also does not alter packet contents. Instead, a SPIE router stores a log of packets in a Bloom filter, which DNA already assumes exist in routers to detect replayed packets (Section 3.4.3).

In SPIE, each router maintains a log of the packets that it has recently forwarded. The log is implemented with a Bloom filter. When a packet arrives at the router, the router creates k different m-bit digests of the packet by hashing part of the packet with k independent hash functions. The input to each hash is the first 24 invariant bytes of the packet (the IP TTL, checksum, and QoS can change during transmission, so the router zeroes these fields before computing the digest). These k values are then inserted into the Bloom filter, a bit-vector of 2^m bits, by using the m-bit results as indices and setting the bits at those indices to 1.

The SPIE log can be used for both traceback and detecting duplicate packets. However, detecting duplicates based on a hash of only the first 24 bytes of a packet produces false positives. To prevent this, routers hash the entire packet, which incurs a marginally higher computing cost.

The task of tracing a packet back to its source is handled by a SPIE Traceback Manager (STM), a device in each AS responsible for querying routers in that network and communicating with other STMs. When presented with an unaccountable packet p, a destination D sends a request to its local STM to find the source of p. The STM queries the routers in its AS to determine the path p took, called the *attack path*. The STM starts with D's first-hop router, then queries its neighbors to find the preceding router in the path. One or more routers report that they forwarded the packet; the STM then queries the upstream neighbors of these routers. The STM continues this process of simulating reverse-path flooding until it reconstructs the path of p, terminating at some inter-AS link. Having identified the previous AS hop, the STM issues a request to the STM in that AS to do the same, until the source is found.

Ideally, this process would terminate at the device that sent the packet. However, there are two ways in which (an honest) router's answer could be incorrect. The first is a false negative: if a router forwarded the packet in the previous epoch, it may have refreshed its filter. To prevent this, routers archive their Bloom filters for several epochs and examine each for p. A false positive can occur due to the probabilistic nature of Bloom filters. This will cause a fork in the reconstructed path. In most cases, forks are easy to detect because the false path will end quickly (the probability a router produces a false positive is independent of the probability of the same at other routers), while the true path will continue on to further hops. However, this may not always occur. In order to minimize false positives and prune the reconstructed path, the destination should forward each unaccountable packet to the SPIE. Each additional packet reduces the probability of Bloom filter false positives, and thus of forks in the reconstructed path, because the Bloom filter hash functions are independent.

Using SPIE to find the source of an unaccountable packet is an alternative to accumulating tokens in packets. The LReps in an AS would serve as the traceback manager, which local hosts would contact to initiate traceback of a packet. However, the security model of SPIE is different than that of DNA. SPIE discusses how to limit the effect of malicious routers that alter packets or duplicate packets to undetectably amplify a flow, but not routers that attempt to subvert the traceback process. In order to be a viable alternative to token, a SPIE-like traceback mechanism must be resilient to such misbehavior.

A malicious router R_{bad} may subvert the traceback process by causing false negatives or false positives. Fortunately, neither behavior allows the malicious router to prevent reconstruction of the attack path. DNA handles a deliberate false positive in the same way as an accidental one: the probability that a router upstream of R_{bad} also produces a false positive is low. This probability decreases for each distinct packet provided by the destination during the traceback process. Thus, the false branch of the reconstructed path terminates at R_{bad} and implicates R_{bad} in misbehaving. A false negative will end the reconstructed path prematurely—but again, immediately before R_{bad}. If the STM suspects a router is lying, it can query that router's upstream neighbors to see if any of them have the packet in their logs. Additionally, the network operator can monitor the links between these routers for misbehavior.

In addition to eliminating the need for tokens in packets, this method of traceback makes it easier to find a malicious router. The links in reconstructed SPIE paths are between routers, whereas the DNA accusation protocol produces links between ASes. With SPIE, the source of an unaccountable packet is narrowed down to a single device rather than a network. Network operators do not need to search their networks for misbehaving devices. This benefit comes at the cost of requiring all routers to store packet digest logs (to catch replays, it is sufficient that only one router on the path between the attacker and victim detect replays).

Pretty Good Packet Authentication (PGPA) [38] uses a similar concept to create a service that can determine whether or not a given host has sent a packet.

In PGPA, edge ISPs deploy traffic monitors on their customer access links. Traffic monitors store the SHA-1 hash of each packet, along with a timestamp, that is sent by the ISPs customers and can be queried to verify whether a given packet was recently sent by one of these customers. PGPA does not provide general traceback, as no packets are logged outside of edge ISPs.

Performance

The size of a router's Bloom filter determines its false positive rate. The larger the filter, the lower this rate and greater the accuracy of the reconstructed packet paths, but the higher the cost in terms of memory. The false-positive rate p of a Bloom filter that uses k hash functions to store n packets in m bits is given by the equation

$$p \approx \left(1 - e^{kn/m}\right)^k$$

The authors of SPIE suggest using $k = 3$ hash functions and a *memory efficiency factor* (n/m) of 0.2, which gives a false-positive rate of 0.092 when storing n items. To achieve the above false-positive rate, a router that forwards P packets per second must use a Bloom filter with at least $5P$ bits $(5 = 1/0.2)$. Assuming an average packet size of 1000 bits, a 40 Gbit/s link requires 200 Mbit/s of memory. If the average packet size is 3200 bits, a more realistic value [60], the requirement is reduced to less than 64 Mbit/s.

Given the above false-positive rate of p, how accurate is the process of constructing the attack path? This depends on the length L of the path and the degree d of each router. An analytic bound on the number of extra nodes in the attack path due to false positives, assuming d is constant, is $\frac{Lpd}{1-pd}$. This is not encouraging: for $d > 5$, the reconstructed attack path will contain more than 50% false positive edges. The authors of SPIE show that the number of false positives is much lower in a real topology. Simulations of SPIE on a tier-one ISP topology show that the average number of false positives is less than one, even for paths of 30 hops.

3.6 Partial Deployment

DNA does not require global deployment. Partial deployment solutions depend on whether the original sender is DNA-aware and whether there are legacy transit domains between accountable domains. We assume that all senders and destinations in an accountable domain are DNA-aware; however, DNA-aware principals may send packets from legacy domains.

Two new entities interact with legacy ASes. An *accountability gateway* enables DNA-aware senders in legacy domains to send traffic to accountable domains. A *legacy gateway* forwards sanitized legacy traffic onto accountable domains.

Sending accountable traffic from legacy domains

Accountable senders in legacy domains may wish to send traffic to domains that only accept accountable traffic. The sender's traffic is routed through an *accountability gateway*, LRep_{gw}.

Accountability gateways have a TAP-signed LRep key that they use to issue tokens to senders that have committed to a blinding key and have not been blocked, much like a normal LRep. Unlike a normal LRep, accountability gateways issue tokens to senders in legacy ASes; these senders must tunnel accountable packets to the accountability gateway's AS to use the token.

The blocking procedure is the same: should a receiver decide to block Alice, it follows the standard blocking protocol by sending a block request to LRep_{gw}, who publishes the block in the SBS.

Transiting through legacy domains

Accountable ASes can tunnel over legacy ASes to extend the reach of deployed, accountable "islands", much like the MBone and 6Bone. The tunnel endpoints exchange symmetric keys to create tokens. These tokens are bound to the tunneled "link" between accountable ASes.

It may be the case that an egress router R in an accountable domain cannot forward an accountable packet to an accountable domain, for instance if it cannot find a tunnel that reduces the distance to the packet's destination or if R determines that the destination is in an adjacent legacy domain. In this case, R removes the accountability header from outgoing packets, changes the IP protocol field to the protocol field in the accountability header, recomputes the IP header checksum, and forwards the resultant legacy packet.

Legacy senders

Legacy gateways allow legacy senders to send packets to accountable ASes. A legacy gateway ($\mathcal{L}$) possesses a $\mathcal{P}_{pk}$ from the TAP. $\mathcal{L}$ accepts legacy traffic, creates and commits to a blinding key corresponding to the source IP address, and forwards the packets using the usual DNA protocol. In effect, $\mathcal{L}$ takes responsibility for the legacy traffic by signing it. Destinations that block $\mathcal{L}$ no longer receive any legacy traffic through this gateway. Like any user $\mathcal{P}_{pk}$, if sufficient destinations block $\mathcal{L}$, its key may no longer be renewed by the TAP.

3.6.1 Distributing Keys to New Users

When a new user Alice joins an AS that implements DNA, or Alice's AS first deploys DNA, she may not have a valid principal key needed to commit to a blinding key. Rather than distribute keys from the TAP out of band, new users can contact the TAP using their local LRep as an accountability gateway. In this case, Alice must authenticate to LRep via an out of band mechanism. Alice generates a ($\mathcal{B}_{pk}$, $\mathcal{B}_{sk}$) for herself and registers $\mathcal{B}_{pk}$ with the LRep, who will let Alice contact only the TAP. Alice uses $\mathcal{B}_{pk}$ to contact the TAP and then authenticates to the TAP with whatever credentials are required. Alice generates her ($\mathcal{P}_{pk}$, $\mathcal{P}_{sk}$) and gives $\mathcal{P}_{pk}$ to the TAP to sign. This is done to prevent impersonation attacks: if the TAP generated the key, it could send traffic as Alice. Finally, the TAP responds with a signature on the public key, allowing Alice to send accountable packets.

If Alice uses the temporary key to abuse the TAP, the TAP will forward a

Key Size	Sign	Verify
112 bits	$\mu = 0.91$, $\sigma = 0.03$	$\mu = 1.11$, $\sigma = 0.01$
160 bits	$\mu = 1.77$, $\sigma = 0.01$	$\mu = 2.07$, $\sigma = 0.02$

Table 3.2: Times in milliseconds to perform each operation on a 3.4 GHz Intel Pentium 4 CPU. μ is average, σ is standard deviation.

normal block request to the LRep, who revokes Alice's permission to authenticate.

3.7 Evaluation

This section evaluates the performance of DNA, based on an implementation of the TAP and a DNA router, and wide-area simulations using ns-2.

3.7.1 Implementation and analysis

Here, I present microbenchmarks for each type of overhead from our implementation. The TAP is implemented in Ruby, using SWIG wrappers to OpenSSL [63]. I have augmented the Click router [49] to perform DNA token manipulations and signature verification.

The implementation uses the OpenSSL implementation of ECDSA [54] for signatures and SHA-1 HMAC for tokens. ECDSA affords short public keys and signatures without expensive bi-linear pairing operations. DNA uses 160-bit keys ("long" keys) for TAP_{pk}, $\mathcal{P}_{pk}$, and speaks-for keys, and 112-bit keys for LRep_{pk} and $\mathcal{B}_{pk}$ keys ("short" keys). As LRep_{pk} and $\mathcal{B}_{pk}$ keys are refreshed relatively often; their keylength can be shorter.

DNA overheads can be partitioned into five broad categories:

- **Accountable traffic:** Accountable traffic incurs processing overheads for signatures and verifications, latency overheads for block lookups, storage overheads for storing commitments, and packet overheads for storing an accountability header.

				112-bit	Verifications		Local	Remote	Header	Msg.
	Operation	Actor	Frequency	Signatures	112-bit	160-bit	RTT	RTT	size	size
1	Send acc. packet	Sender	Per packet	1	·	·	·	·	185	*var.*
2	Verify acc. packet	Router		·	2	1	Local operation			
3	Commit to blinding key	Sender	As needed	1	·	·	1	·	20	100
4	Verify commitment	sLRep		1	1	1	Local operation			
5	Request (initial) token	Sender	Once per epoch	·	·	·	1	·	20	50
6	Return (initial) token	sLRep		·	[0..1]	·	1	·	20	4
7	Create a waiver	Admin	Per waivered connection	1	·	·	Local operation			
8	Return a waiver	Admin		·	·	·	1	·	20	76
9	Send a waiver	Sender		·	·	·	·	1	185	76
10	Check + register waiver	Server		·	·	2	1	·	20	76
11	Send waivered packets	Server	Per packet	·	·	·	·	·	48	*var.*
12	Create block request	Admin	Per block	1	2	1	1	·	185	124
13	Create accusation pkt	Admin	As needed	·	·	·	1	·	185	48
14	Forward accusation pkt	AS		·	·	·	1	·	185	48

Table 3.3: Overhead of DNA sub-protocols. Header size reflects packets with five tokens. but y in the worst case. All signatures and verifications for the same operation can be performed in parallel.

- **Traffic with waivers:** Once waivers are issued, DNA operations require no asymmetric cryptography. The primary overhead is in constructing and checking HMACs.

- **LRep and gateway:** LReps and legacy gateways must store blinding key mappings, publish blocks to the SBS, and verify SBS receipts.

- **Administrator:** Administrators sign block requests, create waivers, and store and verify SBS receipts.

- **TAP node:** TAP nodes sign keys for principals and LReps. They resolve disputes and evict misbehaving LReps.

The primary computational overhead in DNA comes from signature creation, signature verification and HMAC operations. The time required for the signature computations on a typical workstation is shown in Table 3.2. The HMAC operations require less than 0.02 ms for 1500 byte packets.

Table 3.3 shows the number of different cryptographic operations required for DNA component protocols, the number of local and remote round trips each protocol incurs, and the sizes of the protocol messages. Of note from Table 3.3 is the following:

Line 2 One router in the source AS must verify two short signatures and one long signature for accountable packets. These verifications can be parallelized and two of them may be cached.

Line 3 Principals can pre-commit to blinding keys and use them as necessary. There is no need to commit to a blinding key for each new connection.

Line 7 Creating a waiver requires two signature operations, only one of which must be performed at connection setup time. The nonces can be generated and signed offline.

Line 11 Once a waiver is verified and a token obtained, no more asymmetric operations are required.

Line 12 If a packet can be verified (three verification operations) and the destination wishes to block the sender, it needs to create one signature with its speaks-for key. Block requests fit in a single datagram.

Line 13-14 If a destination receives a packet with an invalid signature or an invalid nonce, it invokes the accusation protocol. The accusation packets also fit into a single datagram, and require no signatures beyond the accountability header. Each successive upstream AS must check their token (one HMAC) and then compare the previous token against all keys shared with their other AS neighbors (one HMAC operation per neighbor).

The compilation of overheads in Table 3.3 suggests that DNA can be deployed in the wide-area using current hardware. I now use the analysis in Table 3.3 the microbenchmarks from our implementation to extrapolate the performance of a hypothetical DNA deployment.

Generating a blinding key

For new connections, the sending principal must generate a blinding key. The overhead of generating a blinding key is low. Blinding keys can be generated offline on an as-needed basis. The sender implementation, when running on a 1.5 GHz CPU, can generate a 112-bit ECDSA blinding key in 1.2 ms.

LRep initial connection processing

An LRep must verify commitments to blinding keys and generate initial tokens. Not every connection requires commitment verification: a sender may elect to reuse a blinding key to multiple destinations. Registering a blinding key requires verifying one long and one short key, which takes approximately 3 ms (Table 3.2). Issuing an initial token requires only one verification (of the SBS response) and one HMAC operation.

The LRep also stores the mapping between blinding keys and users. A blinding key commitment consists of an $\mathcal{P}_{pk}$ (20 bytes), a $\mathcal{B}_{pk}$ (14 bytes), the expiration epoch $\nu_{\mathcal{B}}$ (4 bytes), and $\mathsf{Sign}_{\mathcal{P}_{sk}}(\mathcal{B}_{pk}, \nu_{\mathcal{B}})$ (40 bytes—ECDSA signatures are twice as long as the public key). This requires 78 bytes of storage per blinding key. An LRep can store 13.7 million keys in 1 GB of storage.

Sending an accountable packet

Though three signatures are included in outgoing packets, senders need to create only one per packet: the signature of m with $\mathcal{B}_{sk}$. Our sender, running on a single-threaded 1.5 GHz AMD XP CPU, is able to create 259 accountable packets, each containing 1000 bytes of payload, per second.

As shown in Figure 3.2, the accountability header consists of 145 bytes plus 4 for every token. Assuming the average AS path length is four [57], for most packets the tokens consume 20 bytes. Thus, the accountability header consumes 165 bytes.

Router processing

The first-hop router, implemented in Click and running on a 1.5 GHz CPU, is able to verify an entire signature chain at a rate of 79 per second.

Each border router either checks a token or creates one in every packet it forwards. The storage requirements on border routers are minimal: HMAC keys are 16 bytes long. The above-mentioned host can create (or check) tokens for 1024-byte packets at a rate of 80,000 per second. This corresponds to approximately 640 Mbps of throughput. Using dedicated hardware, such as the Intel IXP2855 network processing chip [41], routers can achieve through-puts up to 10 Gbps.

Block request creation

Block requests include the blocked blinding key, the IP of the blocker, and a signature of both. If the block request is treated as a packet, with the message consisting of the blocked key and the IP, then the administrator can sign the message with its

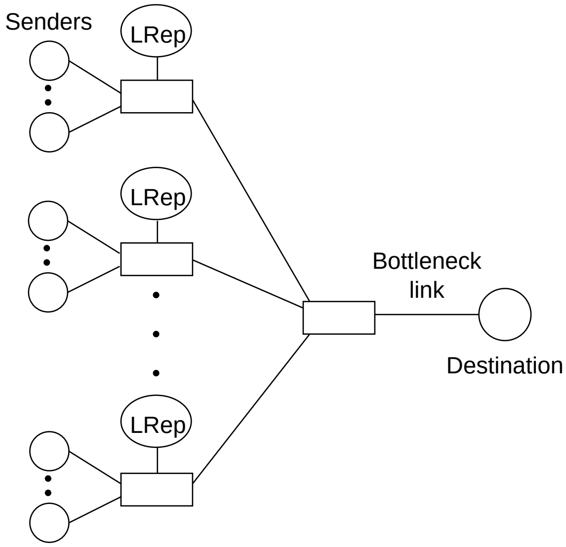


Figure 3.5: Simulation topology

speaks-for key and use that as the message signature in the accountable packet header. This limits the additional overhead at the administrator to verifying the signatures in the packet that it was requested to block.

The network administration at the University of Maryland Computer Science Department manually blacklists IP addresses by installing a firewall rule for persistent attackers. A trace collected between August 2007 and December 2007 shows 2,736 hosts were blacklisted (approximately 15 per day). The overhead of this level of blacklisting is negligible.

3.7.2 Simulation

In this section, we evaluate the performance of a wide-scale deployment of DNA. This deployment is simulated using ns-2. We first evaluate DNA's basic functionality to confirm that it performs as expected. We then show that performance is not significantly affected when certain components become compromised. For comparison, we consider TVA under the same scenario, and compare performance.

Topology

We use a simple dumbbell topology, shown in Figure 3.5, in our simulations. The sender networks are on the left-hand side of the dumbbell. Each network contains one LRep, a number of senders, and a first-hop router that connects them. Each network then feeds into a single node, whose outgoing link is the bottleneck. The destination is connected to this link.

The bottleneck link capacity varies, depending on the experiment, from 1 to 10 Mbit/s. The capacity of all other links is 100 Mbit/s. The latency on all links is 10 ms, except as noted in Section 3.7.2.3.

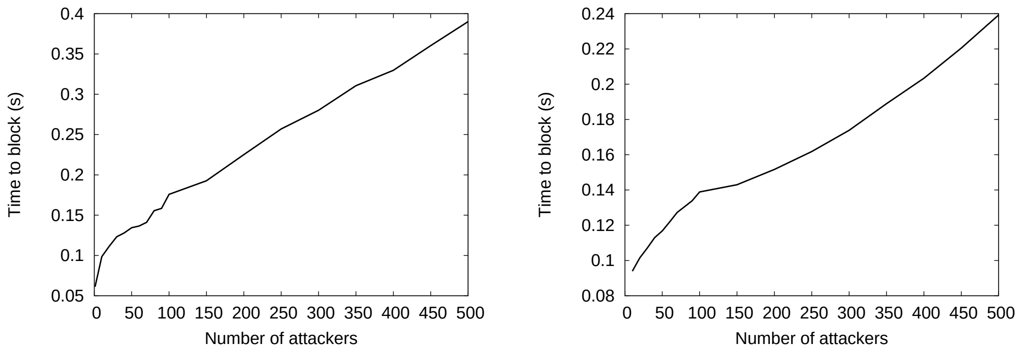
There are two types of senders: attackers that flood the destination and honest senders that attempt to transmit files to the destination over TCP. Attackers send traffic at 1 Mbyte/s.

3.7.2.1 Blocking senders in honest networks

When a destination sends a block request to an LRep, the LRep publishes the block in the DHT. It also places a temporary filter at the gateway, which drops packets that contain the initial token that the LRep issued to the blocked sender. This filter expires when the token expires. This simulation evaluates quickly a destination's block requests take effect if the sender's network is honest, i.e., its gateway honors these filters.

For these experiments, the bandwidth of the bottleneck link is set to 10 Mbit/s. Each sender is an attacker and begins by first obtaining a token from the LRep. It then sends at a constant bitrate until it is blocked. As soon as the destination receives a packet from an attacker, it creates a block request and sends it to the attacker's LRep.

We ran two experiments with sender networks of different sizes. In the first experiment, each LRep serves a single sender, that is, each network has one sender. In the second, each sender network has 10 senders. In each experiment and the number of attackers was varied from 1 to 500.



(a) One attacker per network

(b) Ten attackers per network

Figure 3.6: Average time required to block each attacker in a dumbbell topology network consisting solely of attackers. Each LRep installs a local filter that blocks the source once it receives a block request.

Figures 3.6(a) (1 sender per network) and 3.6(b) (10 senders per network) show the average time from when an attacker first sends a packet until when it is blocked by the destination.

3.7.2.2 Multi-stage attacks

A second experiment models a more powerful adversary that can coordinate multiple attacks against a destination using disjoint groups of nodes. This scenario models a botnet, where a single adversary in control of many bots can command different sets of bots to attack at different time.

TCP transfers

To measure the effect of such compromise, we emulate an experiment from the evaluation of TVA [82]. Honest senders attempt to transfer files to the destination using TCP. The files are 20 KBytes each, as in the evaluation of TVA. Senders serially transmit a fixed number of files, but this number is set so that file transfers are started throughout the series of attacks. In this experiment, there are 10 honest senders and each transfers 20 files. We then observe how file transfer completion time is affected at each state of the attack.

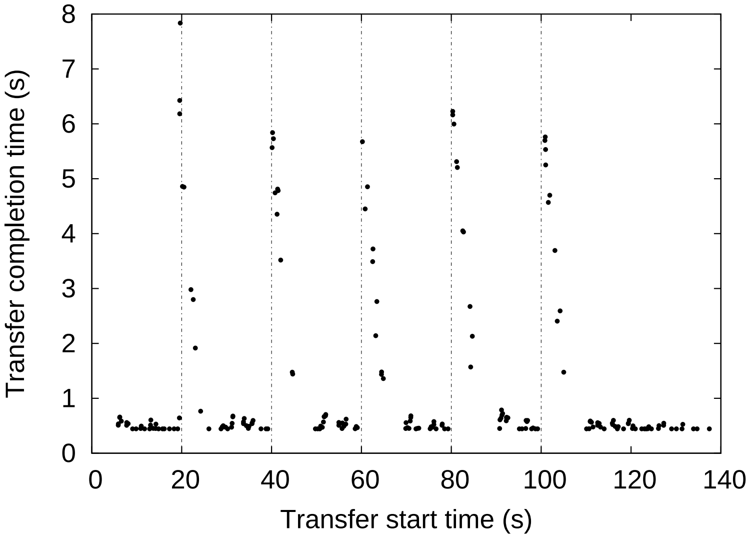


Figure 3.7: Time to complete a 20 Kbyte file transfer in the presence of multi-stage attack. Ten honest senders send ten files each. An attack stage starts every 20 seconds, indicated by the vertical bars. In each attack, 150 users flood at 1 Mbyte/s.

In all experiments that simulate file transfer via TCP, excessive packet loss may cause the transfer to abort. Specifically, the sender will abort the transfer if it does not receive a response to 8 SYN packets or if its retransmission timeout is greater than 64 seconds. In this particular experiment, no transfers were aborted.

A transfer that times out due to excessive packet loss will be aborted; no timeouts occurred in this experiment.

In this experiment, there are 750 adversaries, each sending at 1 Mbyte/s. They are split into five waves of 150 attackers each. The first wave starts sending at time 20 seconds, and subsequent waves begin every 20 seconds after that.

The results of the experiment are shown in Figure 3.7. The x-coordinate of each point is the time when the 20 Kbyte transfer started and the y-coordinate is the time required to complete the transfer. The results show that while transfer times increase when each wave begins, DNA recovers quickly and overall performance is restored. The effects of each wave of attacks is not cumulative: the effect of subsequent attacks is not worse than the earlier attacks.

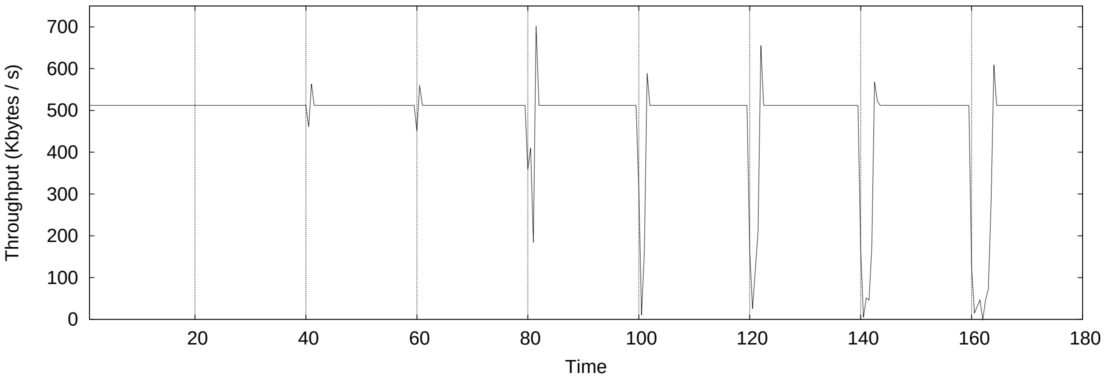


Figure 3.8: Bandwidth received at the destination from a single source sending at a constant bit rate equal to half of the bottleneck link. Every 20 seconds, attacks of increasing intensity begin.

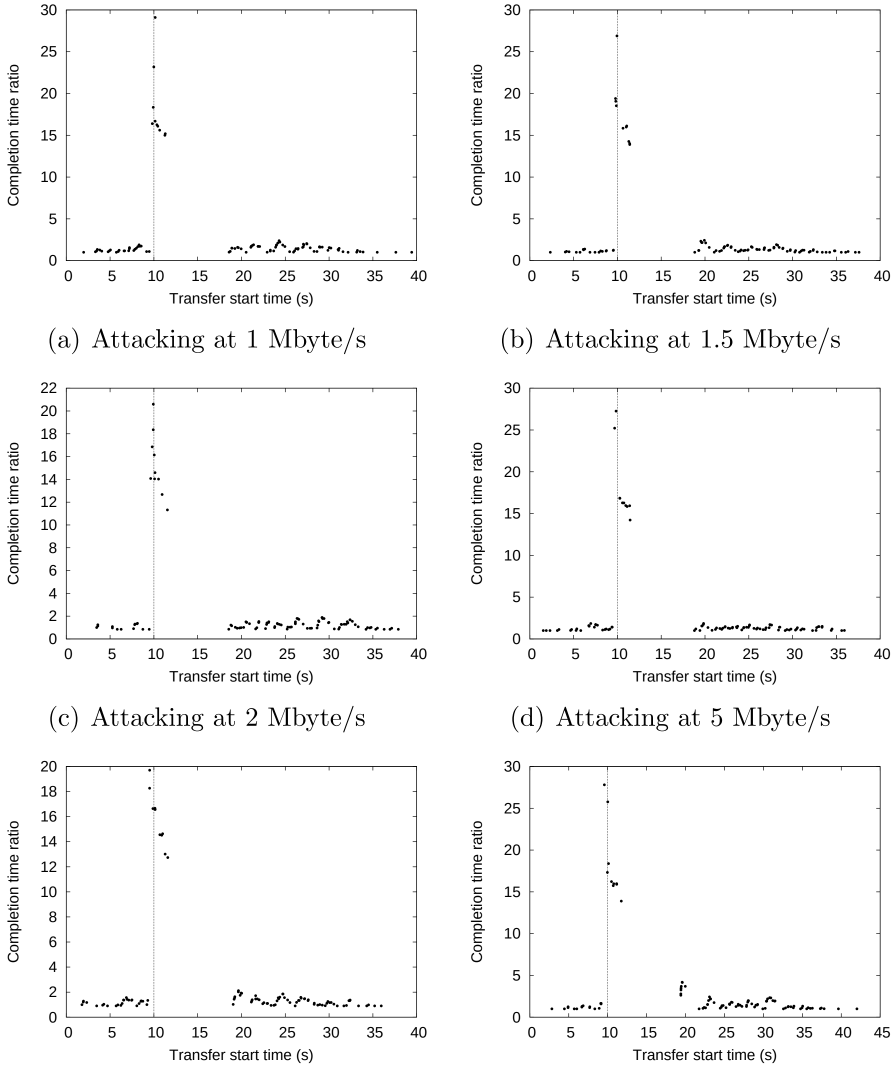
Effect on bitrate

To measure the available bandwidth on a link during an attack, we performed a similar experimet as the previous one. Instead of 10 TCP senders, we used a single source sending at a constant rate, equal to the half of the capacity of the bottleneck link. Also, the attacks increased in severity. At 20 seconds, one attacker begins flooding the destination at a bandwidth equal to the bottleneck link. 20 seconds later, two attackers begin; then four, and so on, up to attacks of 128 senders.

The results, shown in Figure 3.8, highlight the importance of blocking attackers quickly. While smaller attacks have limited effect, the magnitude of larger attack overwhelms the bottleneck link, forcing throughput close to zero, until all senders are blocked.

3.7.2.3 Handling compromised routers

We performed a third experiment to determine how well the accusation protocol allows DNA to recover from a compromised router in the middle of the network. When compromised, the router begins sending unsigned, unaccountable packets to the destination. It places correct tokens in its packets, and no downstream routers perform signature checks, allowing the attack packets to reach the destination.



(a) Attacking at 1 Mbyte/s

(b) Attacking at 1.5 Mbyte/s

(c) Attacking at 2 Mbyte/s

(d) Attacking at 5 Mbyte/s

(e) Attacking at 10 Mbyte/s

(f) Attacking at 20 Mbyte/s

Figure 3.9: This figure shows the effect that a compromised router that allows unaccountable packets into the network has on the time required for senders to transfer a 20 Kbyte file. The y-axis is the ratio of time required to complete a transfer compared to the time of the first transfer. The attack starts at 10 seconds, indicated by the vertical line. Each LRep installs a local filter that blocks the source once it receives a block request. The bottleneck link capacity is 1 Mbit/s.

This experiment uses a setting similar to the previous one: 10 senders send 10 copies of a 20 Kbyte file to the destination and we measure the effect of the compromised router in terms of how it affects file transfer completion time.

The topology in this experiment is changed slightly from the previous one.

The senders are attached as before (in this case, all senders are honest). However, the compromised router is connected through a series of three additional routers. Each of these represents a network that participates in the accusation protocol. As the compromised router can drop arbitrary packets, we did not place any honest senders behind that router. The bottleneck bandwidth in this experiment is set to 1 Mbit/s.

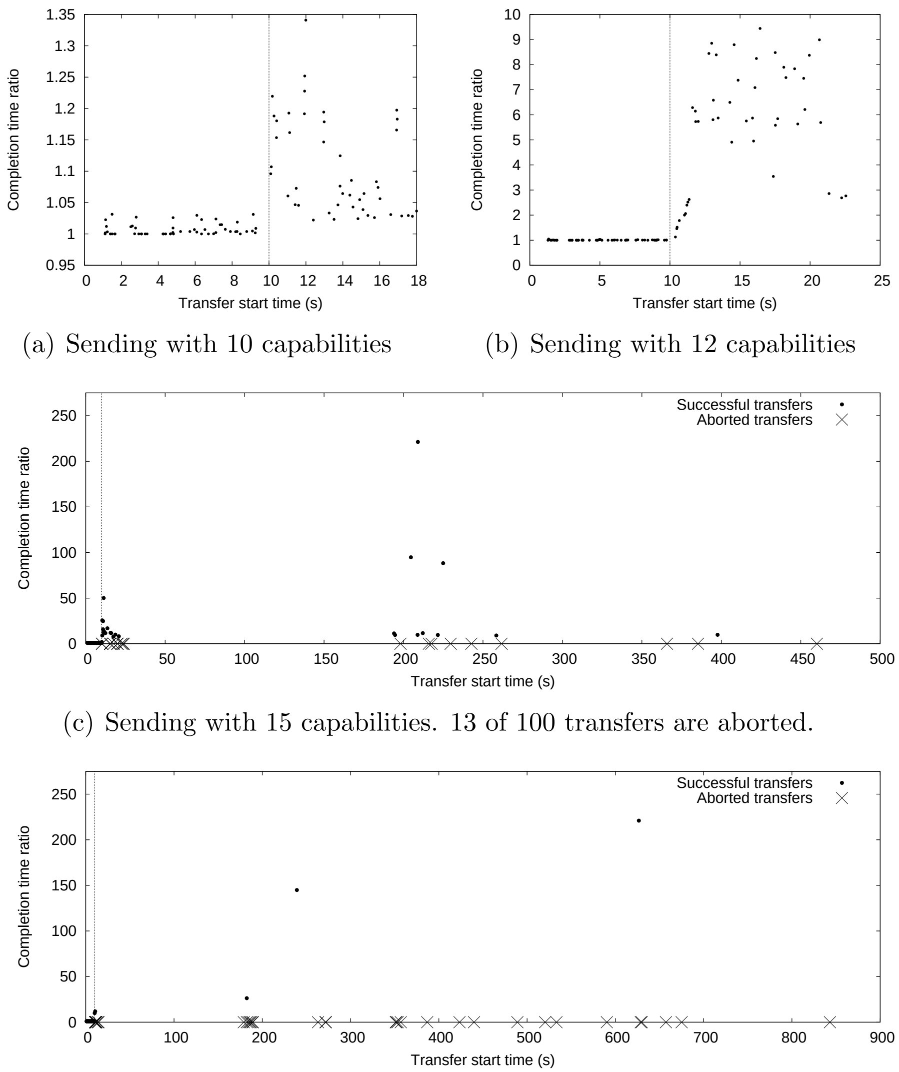
Because packets from the compromised router do not contain a valid LRep signature, a block request will likely be ineffective. The destination instead immediately starts the accusation protocol. In an actual deployment of DNA, the execution of the accusation protocol can vary depending on local policy. If the compromised router is still active, honest networks can use NetFlow [25] or similar tools to detect it. However, this may take some time. To emulate this, the simulation waits two seconds before forwarding the accusation message to an upstream network. We believe this to be a middle ground between an actively-policed network that ensures its routers are correct and a network that applies a more lax policy that may take longer to detect malicious behavior.

Figure 3.9 shows the results of this simulation. The x-axis shows time, and the y-axis shows the duration of transfers that started at that time. The router becomes compromised at 10 seconds and begins flooding, resulting in a spike in transfer time. However, as the accusation protocol executes, transfer time quickly returns to its prior levels, regardless of the rate at which the attacker sends.

Comparison to TVA

We now examine TVA in the same scenario, in order to compare how TVA and DNA perform under this type of attack. We used the TVA implementation made available by the authors [1].

As before, this experiment consisted of honest senders transmitting a 20 Kbyte file and a compromised router acting as an attacker. Each sender transfers files sequentially. A transfer that times out due to excessive packet loss will be aborted.



(a) Sending with 10 capabilities

(b) Sending with 12 capabilities

(c) Sending with 15 capabilities. 13 of 100 transfers are aborted.

(d) Sending with 20 capabilities. 36 of 100 transfers are aborted.

Figure 3.10: This figure shows the effect that a compromised router that steals and uses TVA capabilities has on the time required for senders to transfer a 20 Kbyte file. The y-axis is the ratio of time required to complete a transfer compared to the time of the first transfer. The attack starts at 10 seconds, indicated by the vertical line. The bottleneck link capacity is 1 Mbit/s.

A separate set of senders lies behind the compromised router. These senders obtain valid capabilities, which the router observes. The router then uses the capabilities to flood the destination with forged traffic. (Because of limitations in ns-2, the senders were actually co-located on the same node, which acted as the compromised router.)

Each capability allows the router to send at 100 Kbytes per second. We varied the number of capabilities the router obtains between 10 and 20. After 10 seconds, the router begins its attack.

Figure 3.10 shows the results of the experiment. Unlike in the previous experiment, the compromised router has a significant effect on file transfer completion time. Once the router obtains 13 capabilities, allowing it to send at 1.3 Mbyte/s, it is able to congest the network to the point that file transfers time out before completion.

We can compare how well DNA and TVA withstand a compromised router along two separate axes: (1) performance immediately after the attack vs. steady-state performance, and (2) how performance changes with the intensity of the attack.

For lower-intensity attacks, i.e. attacks that just manage to saturate the bottleneck link, TVA outperforms DNA immediately after the attack: transfer times in Figure 3.10(a) are less than 35% above pre-attack levels, whereas transfers in DNA (Figure 3.9(a)) take several times longer. However, once the compromised router is discovered, transfer times in DNA return to pre-attack levels. In TVA, they remain higher than before the attack.

As the power of the attack increases, the performance of TVA quickly deteriorates. Once the router is able to obtain 15 capabilities (Figure 3.10(c)), it is able to congest the bottleneck link to a point where some transfers are aborted and those that do finish take significantly longer. In comparison, Figure 3.9(b) shows how DNA performs under an attack of equivalent bandwidth: transfer times are affected nearly the same as in attacks at lesser bandwidths and no transfers are aborted. Figures 3.10(d) and 3.9(c) show a similar comparison between the systems when the attacker can flood at 20 Mbyte/s; 36% of file transfers are aborted with TVA, compared to none for DNA.

These results show that DNA is able to recover from a router compromise, while TVA is not. They also suggest that DNA performs similarly regardless of the power of the attacker, while increasingly powerful attacks have a drastic effect on TVA's performance.

3.8 Conclusion

DNA is a network-layer accountability architecture that explicitly decouples connectivity from responsibility. DNA's security is grounded in trusted identity certification authorities (TAPs). DNA allows destinations to specify per-source principal blocks, which apply regardless of the principal's network attachment point. DNA does not mandate local policy, e.g., each AS is free to choose how it classifies misbehavior, and can withstand the corruption of all non-TAP components.

Extrapolated numbers from our implementation of DNA show that computationally, it is feasible to implement DNA with current hardware. We have strived to make the trade-offs in DNA explicit and we expect the point in design space that DNA explores to be both viable and important.

Chapter 4

The NeighborhoodWatch DHT

In this chapter, I present the NeighborhoodWatch DHT, which DNA uses as its scalable block storage service.

4.1 Introduction

A distributed hash table (DHT) is a decentralized, distributed system that exports a put/get interface. Both of these functions are built on a lookup operation that scales logarithmicly with the size of the system. A DHT thus is a natural choice for DNA's SBS. However, the sensitive nature of the SBS's contents (namely, block requests) invite malicious users to attack the SBS.

Traditional DHTs can withstand fail-stop failures, but malicious nodes may provide incorrect routing information, refuse to return published items, or simply ignore certain queries. Efforts to protect DHTs from these attacks hurt their scalability. Specifically, the DHT of Castro et al. [24] relies on *redundant routing*, which floods messages along multiple paths, while S-Chord [33] requires occasional flooding of $O(\log^2 N)$ messages.

This chapter presents NeighborhoodWatch, a DHT that is resilient to malicious nodes and does not require flooding. NeighborhoodWatch maintains the $O(\log N)$ bounds on routing table size and expected lookup time. NeighborhoodWatch depends on an on-line trusted authority that periodically contacts and issues signed

This chapter is based on the paper [17], which is joint work with Rob Sherwood, Derek Monner, Nate Goergen, Neil Spring, and Bobby Bhattacharjee.

certificates to each node. Nodes need a certificate to participate; honest nodes in NeighborhoodWatch can detect malicious behavior and expel the responsible nodes from the DHT. The DNA TAP fills the role of the on-line trusted authority, administering a large set of untrusted nodes that provide secure content delivery. This design greatly increases the availability and capacity of the service without requiring that every node be trusted.

NeighborhoodWatch makes one additional assumption: in the flat identifier-space of the DHT, there is no sequence of $k+1$ consecutive nodes that are malicious, where k is a system parameter chosen by the blacklist administrator. That is, at least one in $k + 1$ consecutive nodes is alive and honest. A larger value of k means that this assumption is more likely to hold, but also that load on the trusted nodes increases (Section 4.4.1). NeighborhoodWatch maintains the logarithmic complexity of original DHTs and is secure as long as the assumptions are maintained. Even if the second assumption is violated, NeighborhoodWatch can provide many of its security properties.

Using a relatively small trusted resource to secure a scalable infrastructure is an important theme in distributed systems that makes several designs practical. For instance, in cryptography, a public key infrastructure (PKI) can be established by a single trusted keypair. Maheshwari et al. [59] present a system, TDB, which uses a small amounted of trusted storage to build a trusted database on untrusted hosts. NeighborhoodWatch employs a similar concept, in which few trusted hosts enforce the correctness of a DHT consisting of many untrusted hosts. The ratio of trusted hosts to untrusted hosts is only limited by the bandwidth and processing power of the trusted hosts. The end result is a potential increase, by several orders of magnitude, in the number of hosts responsible for content (in this case, blacklist) distribution.

The rest of this section is organized as follows. Section 4.2 presents the security model for NeighborhoodWatch. The design of NeighborhoodWatch is given in Section 4.3. An analysis of NeighborhoodWatch appears in Section 4.4, and an evaluation in Section 4.5. Section 4.6 presents related work, and Section 4.7 concludes.

4.2 Security Model and Assumptions

Previous designs of secure DHTs [24, 33] are able to guarantee security when a $\frac{1}{4}$ fraction of nodes are corrupt. NeighborhoodWatch adopt a similar model, in that it allows for some fraction f of the nodes to be malicious, but does not place hard bounds on f. Instead, NeighborhoodWatch assumes that for every sequence of $k+1$ consecutive nodes in the flat ID space of the DHT, at least one is alive and honest, where k is a system parameter. This is the *fundamental assumption*. The insight is that if nodes cannot choose where they are placed in the DHT (an assumption justified momentarily), malicious nodes would have to corrupt a large fraction of the N nodes in the DHT in order to obtain a long consecutive sequence of corrupted nodes. By storing sequences of nodes in routing tables, honest nodes are guaranteed to know of at least one other honest node that is "near" a given point in the DHT. For a given value of f, a corresponding k can be chosen so that the fundamental assumption holds with high probability. Section 4.4.1 analyzes how likely it is that the fundamental assumption holds for given values of f, k, and N.

As the fraction of malicious nodes in the DHT increases, the likelihood that the fundamental assumption holds decreases. If, however, malicious nodes could be discovered and removed from the DHT, then the number of malicious nodes would be kept at a manageable level. In order to remove nodes from the DHT, NeighborhoodWatch assumes the existence of an on-line trusted authority, potentially distributed, that periodically issues signed certificates to nodes. These certificates are called *neighborhood certificates*, or nCerts, for reasons which are explained in Section 4.3. nCerts have a relatively short expiration time compared to the average lifetime of a node. Nodes need a current, valid nCert in order to participate in the system. In order to remove a malicious node, the authority simply refuses to sign a fresh nCert for that node.

Maintaining security is then a matter of detecting malicious nodes. NeighborhoodWatch introduces several mechanisms by which misbehavior can be detected and proven. When a node is known (or strongly suspected) to be malicious, it is expelled from the system. This ensures the correctness and efficiency of operations on the DHT, by either enforcing that malicious nodes behave according to protocol, or by increasing the likelihood that the fundamental assumption holds.

NeighborhoodWatch assumes that the adversary cannot place a corrupt node anywhere it wishes in the DHT. This condition is enforced by requiring that nodes obtain a signed public key before they are admitted into the DHT. This certificate is distinct from a neighborhood certificate. Nodes use these certificates to sign responses to certain DHT messages. In addition, a node's ID is taken to be the hash of its public key. This prevents nodes from choosing their location in the DHT. By making certificates expensive to acquire, as do Castro et al. [24], Neighborhood-Watch combats the Sybil attack [31].

Public key certificates serve another purpose as well: when a node's key certificate expires, it must obtain a fresh one. As a consequence, its ID will change, and the node will be relocated to a new portion of the DHT. Although this is a potentially expensive operation, it also limits the lifetime of a sequence of $k + 1$ corrupt nodes, as eventually they will be redistributed.

Additionally, NeighborhoodWatch requires all nodes to have loose clock synchronization. If a node's clock differs from the system clock, it may impair its own ability to participate effectively in DHT operations, but it will not harm the operation of other nodes.

4.3 The NeighborhoodWatch DHT

This section presents the design of the NeighborhoodWatch DHT. The DHT supports three operations, all of which are reliable when the fundamental assumption holds:

1. *lookup*(id), which takes an ID and returns a reference (nCert) to the node responsible for storing items with the given ID
2. *publish*$(id, item)$, which stores data item $item$ in the DHT under ID id at the node responsible for id as well as its k successors
3. *retrieve*(id), which returns either a data item published under the given ID or a statement, signed by the node responsible for the ID, that no item was published with that key.

NeighborhoodWatch is based on Chord [75], which routes queries through a network of N nodes with only $O(\log N)$ messages while requiring each node to store only $O(\log N)$ links to other nodes. These links are called a node's *finger table.* The ID space of Chord is the integers between 0 and $2^m - 1$ from some integer m. Chord orders IDs onto a ring modulo 2^m. A node with ID x stores fingers to nodes with ID $x + 2^i \mod 2^m$ for integers $0 \leq i < m$. The *successor* of n is the node whose ID is immediately greater than n's ID modulo 2^m. Likewise, the *predecessor* of n is the node whose ID is immediately less than n's.

While NeighborhoodWatch will still operate correctly in most cases when the fundamental assumption is violated, security is not guaranteed. Malicious nodes can undetectably hide published items, prevent new nodes from joining, and cause routing failures. This motivates the reliance on a trusted authority: when malicious nodes misbehave, their behavior can be *proven*; upon witnessing proof of misbehavior, the trusted authority can remove malicious nodes from the DHT. NeighborhoodWatch is designed so that any attempt by a node to lie about whether or not an item was published will implicate that node as faulty, and it will be expunged from the system. Therefore, a corrupt node's only course of action is to be maliciously non-responsive.

4.3.1 Overview

Unlike S-Chord, which partitions nodes into disjoint neighborhoods, NeighborhoodWatch assigns a *neighborhood* to each node. This neighborhood consists of the node itself, its k successors, and k predecessors. A single node will therefore appear in $2k + 1$ neighborhoods. NeighborhoodWatch requires $2k + 1$ nodes to bootstrap.

NeighborhoodWatch employs an on-line trusted authority (the TAP) to sign certificates attesting to the constituents of neighborhoods. The TAP has a globally-known public key, TAP_{pk}, and corresponding private key TAP_{sk}. The TAP may be replicated, and the state shared between TAP replicas is limited to a private key, a list of malicious nodes, and a list of complaints of non-responsive nodes.

The TAP creates, signs (using a secure digital signature algorithm) and distributes neighborhood certificates, or nCerts, to each node. Nodes renew their nCerts on a regular basis by contacting the TAP. Similarly, joining nodes receive an initial nCert from the TAP. nCerts list the current membership of a neighborhood, accounting for any recent changes in membership that may have occurred. Using signed nCerts, NeighborhoodWatch is able to verify the set of nodes that are responsible for storing an item with ID x.

Nodes maintain and update their finger tables as in Chord. For each of n's successors, predecessors, and finger table entries, node n stores a full nCert (instead of only the node ID and IP address as in Chord). When queried as part of a lookup operation, nodes return nCerts rather than information about a single node.

4.3.2 Neighborhood Certificates

A node n has ID $n.id$, IP address $n.ip$, port $n.port$, public key $n.pk$ (along with a signed certificate from the CA), and private key $n.sk$. Let the *info* of node n be defined as $\widehat{n} = \{n.id, n.ip, n.port, n.pk\}$. The predecessor of n is $p(n)$, the i'th predecessor of $p(n)$ is $p^i(n)$, and likewise n's immediate successors are $s(n)$, $s^2(n)$, etc. The *range* of n is the integer interval $(p(n).id, n.id]$. Node n is said to be the $owner(x)$ for any ID x in the range of n.

Malicious nodes may try to subvert lookups by lying about their range. By including $p(n)$ in nCert_n, NeighborhoodWatch allows any node to determine the range of n given (a fresh copy of) nCert_n. As new nCerts are issued periodically, it is possible for a node to hold several nCerts at once. When queried, a malicious node might present an old nCert in an attempt to hide a newly-joined node. Therefore NeighborhoodWatch includes the entire neighborhood of n in nCert_n to serve as witnesses to the freshness of nCert_n. Anyone can determine the accuracy of nCert_n by querying each member of the neighborhood and comparing the returned nCerts. If at least one honest neighbor exists, its nCert will reveal any hidden nodes and implicate malicious ones.

Epochs

nCerts cannot be explicitly revoked—once a certificate is distributed, it cannot be "called back", since using certificate revocation lists requires a publish and lookup infrastructure very similar to the one we are trying to build. Therefore, to prevent malicious nodes from persisting in the DHT, nCerts must expire periodically. To facilitate this, NeighborhoodWatch divides time into sequentially-numbered epochs. Certificate expiration is implemented by including a timestamp, indicating the last epoch in which a key is valid, in the certificate.

Time is split into two kinds of epochs, *join* and *renew*, which alternate. New nodes may join only in a join epoch; existing nodes may renew their nCerts only in a renew epoch. Nodes must renew their certificates once during each renew epoch to remain in the DHT for subsequent epochs. The length of each is a system parameter, though typical values would be on the order of tens of minutes. The epoch length is a trade-off between the frequency of overhead incurred by the recertification process and the length of time that a proven malicious node can remain in the DHT, since a node can only be expelled by the TAP refusing its renew request. Epochs and epoch lengths in NeighborhoodWatch need not correspond with those of DNA.

The current epoch is denoted by an integer which monotonically increases over time. nCerts issued in a join epoch e_j are valid through epoch $e_j + 1$, i.e., the next renew epoch. nCerts issued in renew epoch e_r are valid through epoch $e_r + 2$, i.e., the next renew epoch. A node who fails to renew its nCert before it expires has effectively left the system, as other nodes simply ignore expired nCerts.

The reason for separating the periods in which a node can join from the periods where nodes renew their nCerts is to prevent the following scenario: an honest node n requests that a TAP replica renew nCert$_n$. While this is in progress, a new node j joins, perhaps through a different TAP replica, and an nCert$_n$ containing j is issued to n. Afterward, the initial renew operation completes, and n receives a new nCert without j. This would produce inconsistencies among the nodes in n's neighborhood, and could potentially lead to n being implicated as malicious. By separating join and renew epochs, only nodes who are affected by a joining node receive new certificates in a join epoch, and neighborhoods are stable (barring removal of unresponsive nodes) throughout a renew epoch.

nCert Format

Let $\mathsf{Sign}_K(msg) = (msg, \sigma_K(msg))$ denote the application of a secure digital signature algorithm to msg, where $\sigma_K(msg)$ is the secure digital signature of msg with key K. The format of nCert$_n$ is:

$$\begin{aligned} \widehat{n} &= \{n.id, n.ip, n.port, n.pk\} \\ nodes &= \widehat{p^k(n)}, ..., \widehat{p(n)}, \widehat{n}, \widehat{s(n)}, ..., \widehat{s^k(n)} \\ nCert_n &= \mathsf{Sign}_{\mathsf{TAP}.sk}\,(nodes, e) \end{aligned}$$

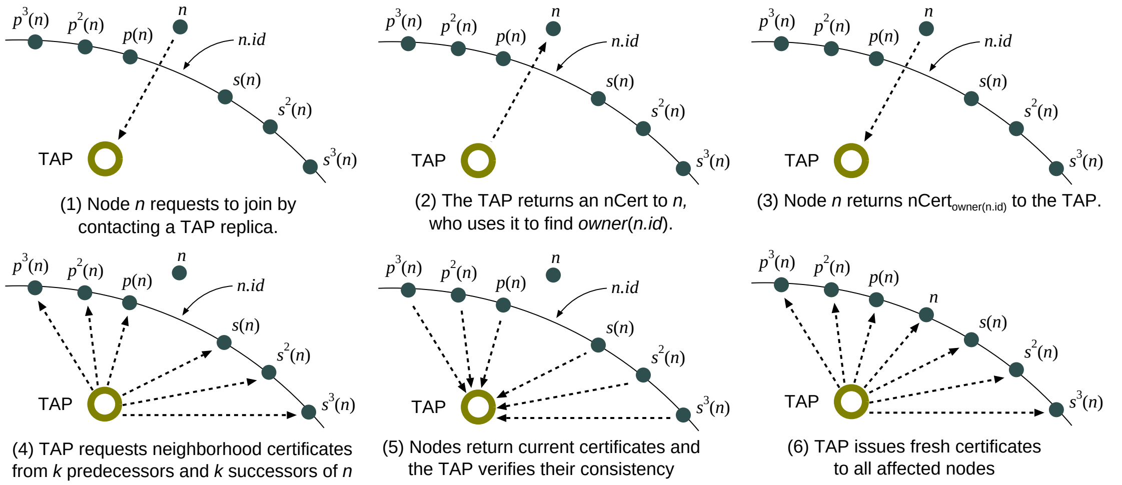
where e is the last epoch in which nCert$_n$ is valid.

4.3.3 Routing

NeighborhoodWatch uses iterative routing, meaning that a querier q searching for $owner(id)$ will contact each hop on the path to $owner(id)$, rather than passing the query off for another node to route. This allows q to recover from routing failures. By using iterative routing, q can ensure that each step of the routing protocol makes progress towards $owner(id)$.

To execute $lookup(id)$, a querier q that is a DHT node examines its finger table to find the nCert of the closest known predecessor of id; call this node p. If q is not a DHT node, it requests the nCert of any DHT node it knows about; in this case, that node is called p. In either case, q requests that p provide the nCert of its closest known predecessor of id. Let the nCert that p returns be nCert$_{next}$. q examines nCert$_{next}$ and determines if it is valid. Several criteria must be met for nCert$_{next}$ to be valid: clearly, it must be signed by the TAP and it must not have expired. Also, $next.id$ must be at least halfway between p and id, which will be the case if p's finger table is correct. If $next$ does not allow q to progress at least half of the distance to id, which may be the case if $next$'s finger table has not stabilized, then q has the option of either querying a different node in any nCert it has or continuing with a sub-optimal hop. If nCert$_{next}$ is valid, q replaces p with $next$ and repeats the process, stopping when it receives a valid nCert$_{owner(x)}$.

If p responds with an invalid nCert, or simply doesn't respond, q queries one



(1) Node *n* requests to join by contacting a TAP replica.

(2) The TAP returns an nCert to *n*, who uses it to find *owner*(*n.id*).

(3) Node *n* returns nCert$_{\text{owner(n.id)}}$ to the TAP.

(4) TAP requests neighborhood certificates from *k* predecessors and *k* successors of *n*

(5) Nodes return current certificates and the TAP verifies their consistency

(6) TAP issues fresh certificates to all affected nodes

Figure 4.1: The join process in the NeighborhoodWatch DHT. Here $k = 3$.

of the other nodes listed in nCert$_p$. If any of these nodes is correct, q is able to make progress towards id while querying at most $2k + 1$ nodes. Each successful query halves the remaining distance between p and id, resulting in at most $(2k + 1)\log N$ messages per query.

A malicious node m that was previously $owner(id)$, but has since relinquished that range to a newly joined node j, may present an old but valid nCert$_m$ which shows m as $owner(id)$ instead of j. Note that this can only occur during the epoch that j joins or the following recertify epoch; after that time, the nCert$_m$ showing m as $owner(id)$ will be expired. In this case, m cannot suppress the keys for which j is now responsible, as will be shown in Section 4.3.7.

4.3.4 Join and Renew

To renew its nCert, a node n presents nCert$_n$ to an TAP replica. The TAP obtains the nCert of each node in nCert$_n$ to check the validity of nCert$_n$ (i.e., to make sure n is not presenting an old-but-unexpired nCert that does not contain a newly-joined node). If nCert$_n$ matches the view of n's neighborhood that the other nCerts describe, the TAP issues a fresh nCert to n, containing the same neighborhood of nodes, but with an expiration time of the next renew epoch.

The process of joining the DHT is similar to that of renewing, but more nodes need to be queried since more nCerts need to be issued. When a node n wishes to join a NeighborhoodWatch instance, it finds nCert$_{owner(n.id)}$ and presents it to a TAP replica. First the TAP ensures that n has a valid private key certificate from the CA. Then the TAP uses the nCert to retrieve the nCerts of the k successors and k predecessors of $owner(n.id)$. The TAP then requests other nCerts as necessary to obtain a full view of the neighborhoods of each node in nCert$_{owner(n.id)}$, so as to verify the neighborhoods of each node that will receive a new nCert when n joins. The TAP then creates new nCerts for n and each of the $2k + 1$ nodes in nCert$_{owner(n.id)}$ ($owner(n.id)$ becomes $s(n)$), setting the expiration epoch for each to the next (renew) epoch e, and sends each new nCert to the associated node. The join process is shown in Figure 4.1.

Once n has joined the DHT, it fills in its finger tables by querying its neighbors for the appropriate nCerts. It also stores the nCerts of all nodes in its neighborhood.

When a node n receives a new nCert, the TAP verifies n's neighborhood. The TAP contacts each node in nCert$_n$. If a contacted node provides a conflicting certificate, it is malicious; if it provides nothing, it is unresponsive. Such nodes are not included in the new nCert$_n$ and are replaced by the appropriate neighbor. To prevent inconsistencies in issued certificates, the TAP replicas must coordinate to maintain a shared list of malicious and unresponsive nodes and refuse to insert these nodes into any granted nCert.

4.3.5 Publishing

NeighborhoodWatch provides a *publish*$(id, item)$ operation, which stores *item* in the DHT under *id*. Let $n = owner(id)$. When a node p wishes to publish *item* to the DHT with key *id*, it first finds nCert$_n$. Recall that the nodes in nCert$_n$ include $n, s(n), s^2(n), ..., s^k(n)$. Let these nodes be the *publish nodes* of nCert$_n$. p will contact each of the publish nodes and request that the node store *item*. This models the replication procedure in Chord, where the k nodes succeeding id store copies of $item$.

Let d be a publish node that p contacts. If d decides to store the item, it returns a signed *receipt* that it has stored the item. The format of a receipt is:

$$R_{id,d} = \mathsf{Sign}_{d.sk}\,(id, Hash(item), d.id, e)$$

where e is the current epoch. This receipt is used to implicate d if it maliciously refuses to return the item when requested to do so.

If d does not respond, then p considers d to be *unresponsive* and informs the TAP. The consequences of this action are detailed further in Section 4.3.8.

Stored items are self-certifying [28], meaning that given the pair $(id, item)$, there exists a way to verify that the object published under ID id is in fact $item$. One technique for self-certifying items is to set $id = hash(item)$, for some collision-resistant hash function. Another way to make items self-certifying is for the publisher to sign them, assuming anyone retrieving the item can locate the publisher's public key. By using self-certifying items, malicious nodes are prevented from returning "fake" items in response to a retrieve request.

4.3.6 Receipt Storage and Retrieval

When a node p publishes a data item to the DHT, it receives (up to) $k+1$ receipts from the nodes that store the item. In order to use receipts to expose nodes that refuse to return published items, the receipts must be made available to anyone that might observe a dishonest response to a retrieve request. One option is for p to store the receipts for items it publishes and give copies to whomever requests such a receipt. However, this introduces two problems: first, if p goes offline or crashes, the receipts of all items p published become unavailable. Second, when looking for an item published in an ID, a node must know who published that item in order to find the receipt. This is not feasible in the general case where the querier may not know (or care) who published the item it seeks.

NeighborhoodWatch incorporates a mechanism by which receipts are stored in the DHT itself. Of course, receipts must be stored at IDs independent of the item being stored, so that with high probability a corrupted neighborhood does not store all the receipts for items published to that neighborhood. NeighborhoodWatch must also avoid the problem of storage explosion: the publish of a single item could result in the storage of $k+1$ receipts; if receipts of *those* publish operations result in receipts stored in the DHT, another $(k+1)^2$ items would need to be stored, and so on. NeighborhoodWatch imposes a limit on the *receipt factor*, RF, that dictates how many levels of receipts are published to the DHT. In this chapter, $RF = 1$, that is, receipts are stored in the DHT, but receipts-of-receipts are not stored.

When a publisher p receives a receipt $R_{id,d}$ from d for an item published under ID id, p publishes the receipt under the ID $R.id = Hash(id||d.id)$ using the normal *publish* protocol. $R_{id,d}$ is then stored on the $k+1$ nodes following $R.id$ in the DHT. p receives receipts from these nodes to verify that the publish is successful, but does not publish these receipts. A node requesting an item with ID $item.id$ from d can check if d has created a receipt R because that node knows both $item.id$ and $d.id$, and can thus determine $R.id$.

Assume d is malicious and returns negative responses to retrieve requests for item i. For d to remain undiscovered, not only must the $k+1$ nodes responsible for storing i be corrupt, but all $k+1$ copies of receipts for i must be unavailable. For this last condition to be true, corrupt neighborhoods of $k+1$ nodes must exist around *each* of the receipts. This occurs with probability close to $\left(p^{k+1}\right)^{k+1}$, which is extremely small even for large values of p and small values of k.

4.3.7 Retrieving

Like the *publish*$(id, item)$ operation, a node r executing *retrieve*(id) also begins by using the *lookup*() operation to find nCert$_{owner(id)}$. r chooses a publish node d from nCert$_{owner(id)}$ and requests *item* from d by sending *id*. If d responds with *item* (which r can verify because items are self-certifying), *retrieve*(id) returns *item* and terminates. If d responds saying that no such item exists, it may be the case that *item* was never published, or it may be that d is trying to subvert r's request. To prevent subversion, all negative responses are signed by the responding node. r stores d's response and requests *item* from another publish node, until either *item* is found or all publish nodes have returned a negative response.

If a node d that r contacts responded with a signed statement that it does not have *item*, yet one of d's neighbors did possess *item*, then r searches for $R_{id,d}$. If it is able to find such a receipt, it presents the receipt and the signed negative response to the TAP, who expels d from the DHT. Even if all nodes issue a negative response, if r suspects that *item* was in fact published under *id*, then it may still search for receipts. Collusion-resistant receipt storage is discussed in Section 4.3.6, and the process of expelling nodes is discussed in Section 4.3.8.

A malicious node thus risks exposing itself if it issues a receipt when it intends to not return items, returns a negative response when an item was published to it, or returns an item that was not published. A malicious node can avoid exposure only by not responding to any requests.

4.3.8 Removing malicious and unresponsive nodes

Previously in this section, we have shown that any attempt by a malicious node to "lie" to an honest node, whether it be by refusing to return an item it stores or returning an unpublished item, will cause the node to be expelled from the DHT (as a result of the TAP not issuing a fresh nCert to the node). Thus it is in the best interest of malicious nodes to be maliciously unresponsive, that is, to communicate with the TAP to ensure that it receives fresh nCerts, yet not respond to any messages from peers. Here, we show how the system can evict such unresponsive nodes. This property comes at the cost of storing state at the TAP. Specifically, the TAP records the list of nodes that have "complained" about a given DHT node, as well as a list of expelled nodes to prevent them from joining.

Whenever a node d fails to respond to a message from node n, n sends a statement to the TAP to that effect. If the number of nodes that complain about d crosses some threshold, the TAP then determines whether d is being maliciously unresponsive. The TAP does this by finding random nodes in the DHT and asking them to submit requests to d. These nodes report to the TAP whether or not d responded. The TAP must do this so that d does not detect that it is being probed. Therefore the requests should be distributed over time and come from a randomly selected set of nodes.

The TAP requests m DHT nodes to make a request to d and observes the results. Let θ_{alive} and θ_{dead}, $\theta_{dead} < \theta_{alive}$, be two parameters that the TAP uses in determining whether d should be expelled:

- If more than $\theta_{alive} \times m$ nodes report d as alive, the TAP takes no further action.
- If fewer than $\theta_{dead} \times m$ nodes report d as alive, the TAP expels d.
- If number of "alive" responses is between $\theta_{dead} \times m$ and $\theta_{alive} \times m$, the TAP re-runs the test at a later time.

θ_{alive} can be set fairly high, as even if d is not malicious but still fails to respond to many requests, it should be removed from the DHT for better performance. θ_{alive} should be selected so that the probability of selecting $\theta_{alive} \times m$ malicious nodes, when an f-fraction of the DHT is malicious, is low. This process is similar in spirit to the *send challenge* of PeerReview [37].

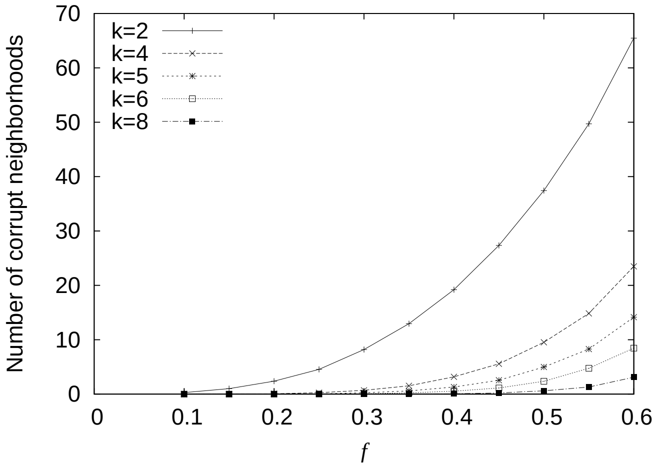


Figure 4.2: The number of corrupt neighborhoods as a function of the probability that a node is bad. Here, $N = 300$.

4.4 Analysis

This section analyzes two aspects of NeighborhoodWatch: how likely the fundamental assumption (that each sequence of $k + 1$ nodes contains at least one node which is honest and alive) will hold as a function of the probability that a random node is malicious, and the cost of storing an item and its receipts in the DHT. These components, along with the certification process (evaluated in Section 4.5), are the ways in which NeighborhoodWatch differs most from traditional DHTs.

4.4.1 Validity of fundamental assumption

If the fundamental assumption is violated, then NeighborhoodWatch cannot guarantee that a $retrieve(id)$ operation will succeed, even if an item has been published to the DHT under id. Note that $lookup(id)$ operations will be successful as long as there is no sequence of $2k + 1$ nodes that are all malicious.

For a system with N nodes, there are N sequences of $k + 1$ nodes. Let a node be bad with probability f. Each sequence individually violates the assumption with probability f^{k+1}. However, the sequences are not independent of each other; the expected number of bad sequence is not $N \times f^{k+1}$ (though this is a reasonable approximation for small f).

To examine the likelihood that the fundamental assumption holds, we simulated an instance of the DHT and counted the number of sequences of length $k+1$ that violate the assumption as a function of N (the number of nodes), k (the number of successors stored in a certificate), and f (the probability that a randomly-chosen node is malicious). Figure 4.2 shows the results of this simulation. Note that the greater the value of N, the higher chance that there is a corrupt sequence. However, even when $f = 0.5$, the number of expected bad sequences is small (below 1 when $k = 8$). This gives an improvement over Castro et al.'s system [24] and S-Chord [33], which are only secure when $f < 0.25$.

4.4.2 Cost of storing items in DHT

In order to store a B-byte item in NeighborhoodWatch, $k+1$ copies of the item are stored. Thus, $B(k+1)$ bytes are required for storing copies of the item. In addition, $k+1$ receipts are stored in the DHT. Each receipt consists of two IDs, one hash, one timestamp (4 bytes), and one signature. Assuming that IDs and hashes are 20 bytes (the length of a SHA-1 hash) and that signatures are 40 bytes, receipts consume 104 bytes.

Each publish results in $k+1$ receipts that are stored in the DHT. Each receipt is stored by $k+1$ nodes. Thus storing an item in the DHT incurs an additional cost of $104 * (k+1)^2$ bytes to store receipts, for a total cost of $(k+1)(B+104(k+1))$ bytes required to reliably store a B-byte item.

4.5 Implementation and Evaluation

We developed and deployed an implementation of NeighborhoodWatch on approximately 70 PlanetLab [15] nodes. The implementation is coded in 2400 lines of Ruby. For digital signatures, we used the elliptic curve digital signature algorithm (ECDSA) provided by OpenSSL. The ECDSA code is written in C++, with a wrap-

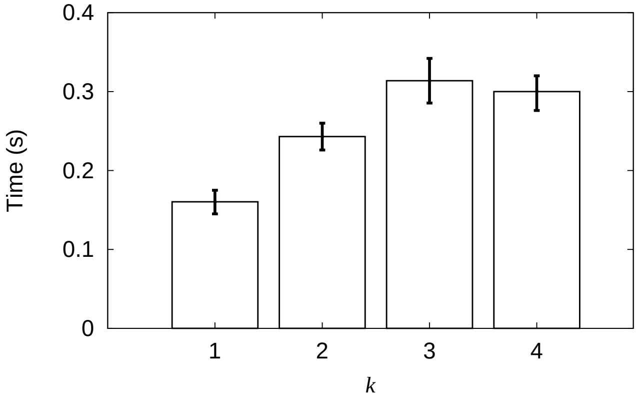


Figure 4.3: Recertification times vs k. The certificate size is $2k+1$. Vertical bars represent 95% confidence intervals.

per written in SWIG. The ECDSA keys in our implementation are 160 bits long, resulting in signatures that are 320 bits long.

PlanetLab nodes were selected to create a large diversity of geographic node locations, latencies, and response times. After deploying NeighborhoodWatch, we then collected statistics of our secure DHT in operation, to better understand the response times required by the routine mechanisms involved in building and maintaining our secure DHT.

In Figure 4.3, we show the average time in seconds for a recertification operation to complete. The average recertification time depends on the length of a timeout. When the TAP requests node certificates from nodes, it will wait for up to *timeout* seconds before requesting the node's nCert again; if the node does not respond to a second request, the TAP considers the node dead and replaces it with another. If *timeout* is too low, churn is unnecessarily introduced, whereas if *timeout* is too high, unresponsive nodes will have a greater effect on the average recertification time. When running experiments on PlanetLab, we set *timeout* to 5 seconds.

4.6 Related Work

Distributed hash tables (DHTs) were introduced as a method of organizing peer-to-peer nodes to provide decentralized storage. The intent of the original DHT protocols (such as Chord [75], CAN [69], and Pastry [70]) was to minimize both lookup time and the amount of routing state stored at each node. For instance, Chord requires nodes to store links to $O(\log N)$ other nodes and queries take $O(\log N)$ messages for a DHT with N total nodes.

Sit and Morris [73] describe several ways in which adversarial nodes may attempt to subvert a DHT. Malicious nodes might attempt to route requests to other incorrect nodes, provide incorrect routing updates, prevent new nodes from joining the system, and refuse to store or return items.

Castro et al. propose a system in which secure routing can be maintained even when up to $\frac{1}{4}$ of the nodes are malicious. Their system counters the Sybil attack [31], in which a single malicious node joins the DHT in multiple locations, by requiring each node to have a certificate that binds its ID to the node's public key. These certificates are provided by an off-line certificate authority, who limits the number of certificates issued to a single entity. When a node detects that one of its queries for $owner(x)$ has resulted in a node that is unlikely to be $owner(x)$, it floods its request along multiple paths, potentially requiring a number of messages that is polynomial in the number of nodes.

The concept of grouping consecutive nodes into neighborhoods has been a feature of several secure DHT designs. Fiat, Saia, and Young propose S-Chord, which is also resilient to a $\frac{1}{4}$ fraction of malicious nodes. S-Chord partitions consecutive nodes into *swarms*, which act as the basic functional unit of the DHT. Lookups in S-Chord take $O(\log^2 N)$ messages (compared to $O(\log N)$ in NeighborhoodWatch) and each node stores $O(\log^2 N)$ links (compared to $O(\log N)$). Myrmic [77] is a secure DHT that makes similar system assumptions as NeighborhoodWatch, and produce a similar solution. However, Myrmic does not consider securing the publish operation and has no mechanism for removing malicious nodes from the DHT when they are discovered. Morselli [62] proposes a method of ensuring that each neighborhood has a $\frac{2}{3}$-majority of honest nodes and then uses Byzantine agreement in each neighborhood to ensure correct behavior of each neighborhood. Bhattacharjee et al. [20] present a lookup primitive which can be verified through the use of threshold cryptography.

4.7 Conclusion

In this chapter, I presented the NeighborhoodWatch DHT, which is secure against a large fraction of malicious nodes. The system depends on a centralized (though replicated) trusted authority which contacts each node on a regular basis, as well as the assumption that sequences of consecutive corrupt nodes are not arbitrarily long. NeighborhoodWatch consists of a small set of trusted hosts that manage a large set of untrusted hosts, thereby allowing security guarantees to scale by orders of magnitude. By using an innovative receipt-storing scheme and digital signatures, NeighborhoodWatch is able to detect and prove malicious behavior; a corrupt node's only course of action is to be maliciously non-responsive. The centralized authority can remove malicious and non-responsive nodes from the DHT, leaving only correct peers.

Chapter 5

VanGuard: Adding Accountability to Capabilities

This chapter presents VanGuard, a system that combines DNA with capability architectures. VanGuard provides the best of both worlds: senders are held accountable and receivers can block senders, as in DNA, yet most packets do not require the use of asymmetric-key cryptography, reducing the overhead of verifying a capability-carrying packet to that of existing capability architectures.

5.1 Capabilities

Capabilities are a well-studied approach to mitigating the threat of DoS attacks [81, 82, 64, 5, 79]. Capability architectures allow receivers to inform routers that a certain flow is desired, with the expectation that routers will prioritize desired flows over potentially undesired flows. Capabilities themselves are light-weight, taking up little space in a packet and are efficiently verifiable by routers.

Capability systems introduce two new categories of traffic: capability-endowed privileged packets, and unprivileged request packets. Packet with valid capabilities take priority at routers, so that traffic desired by the destination takes priority over attack traffic. In order to obtain capabilities, a sender must first send an unprivileged capability request to the destination. Routers insert capabilities (or, in some cases, *pre-capabilities*) into request packets. Routers compute capability values from information in the packet and local to the router, such as the source and destination addresses, incoming and outgoing interfaces, the current time, or a secret known only to the router. If it so desires, the destination returns the capabilities to the sender. In some cases, such as in TVA [82], the destination must first transform the pre-capability into a capability by computing the hash of the pre-capability, a number N of bytes that the capability allows the sender to send, and a lifetime T of the capability.

The source sends a privileged packet by including the returned capabilities in the packet. Each (capability-aware) router along the path computes the capability that it would have inserted into the packet and compares this to the (appropriate) capability present in the packet. If the values match, the router forwards the packet; otherwise, it demotes the packet to un-privileged before forwarding, so that it will be preferentially dropped if it is queued a congested router.

Capabilities work well to protect privileged channels, but additional mechanisms must be used to help legitimate senders obtain capabilities. Argyraki and Cheriton argue that the mechanism used to protect the request channel can also be used to protect the privileged channel [9]. This is not entirely accurate, as the authors of Portcullis point out when they “strongly disagree” with the claim [64]. Their insight is that only a single capability request needs to succeed in order to bootstrap an arbitrarily large flow. However, the point remains: the request channel is vulnerable and needs to be protected. Previous capability architectures limit the capability request channel to 5% of the bandwidth on any link and resort to per-source [81] or per-path [82] fair-queuing, which requires a spoofing prevention mechanism in the first case and a packet marking system (such as Pi [78]) combined with hierarchical fair-queuing in the second. Portcullis [64] protects the capability request channel by requiring senders to solve a computational puzzle in order to send a request, but requires routers to prioritize requests by puzzle difficulty and an infrastructure for distributing puzzle seeds. None of these approaches allow destinations to block a sender from making capability requests; doing so would prevent a malicious sender from affecting legitimate traffic flooding a link with unwanted requests

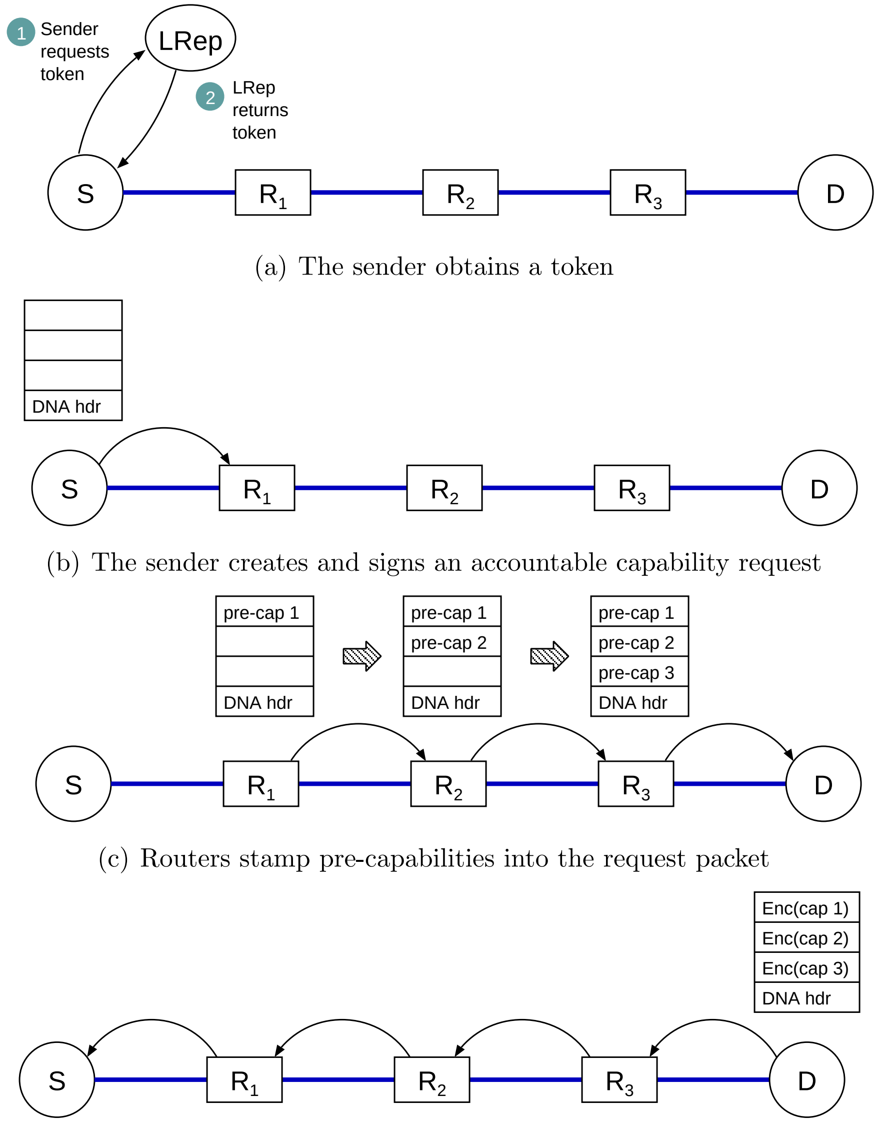
Existing capability systems face another limitation. Because they do not use asymmetric cryptography, they are efficient compared to. However, this also limits the application of capabilities. Capabilities are not bound to packets or to senders, meaning they can be observed by on-path devices. Such a device can then use the capability to impersonate the sender who was actually granted the capability.

This chapter presents VanGuard, which extends capability architectures to incorporate accountability. The goal of VanGuard is to provide the strong accountability of DNA, including the ability of destinations to block senders, with the efficiency of capabilities. Unlike previous work, which aims to limit the rate at which a sender can request capabilities, VanGuard uses DNA to allow destinations to block specific senders from issuing requests entirely. Destinations protect the capability request channel by blocking malicious senders from sending requests, which we show is more effective than fair-queuing on a bandwidth-limited channel. Another way in which VanGuard differs from previous work is that VanGuard treats capabilities as secret values. Senders do not include capabilities in privileged packets. Instead, capabilities serve as the secret input to a keyed MAC of the packet contents. The appropriate routers can derive the capability on demand to verify that the sender has received a valid capability. This prevents on-path observers from stealing capabilities.

5.2 VanGuard Overview

VanGuard is an accountable capability architecture. It is based on TVA [82], though its mechanisms can be applied to other capability architectures as well. This section gives an overview of how VanGuard operates and how it differs from TVA. VanGuard assumes a full deployment of DNA and TVA and shows how the two would inter-operate to provide sender accountability such that, once the sender obtains capabilities, the per-packet overhead is equal to that of TVA.

Like TVA, VanGuard introduces two new classes of traffic: unprivileged capability requests and regular, privileged packets. Routers give preference to privileged packets and will always drop unprivileged traffic before privileged traffic. In order to send privileged packets, a source needs capabilities, which it obtains by sending a capability request to the destination. Figure 5.2



(a) The sender obtains a token

(b) The sender creates and signs an accountable capability request

(c) Routers stamp pre-capabilities into the request packet

(d) The destination creates, encrypts and return the capabilities in a request packet

Figure 5.1: The VanGuard capability request process

To send a capability request in VanGuard, a sender must create a fully accountable packet by obtaining a token from a local LRep and signing the packet with its private key, in accordance with the DNA send protocol.

Along the way, each VanGuard router R_i stamps a *pre-capability*, $\textit{pre-cap}_i$, into the request packet. A pre-capability is a hash of the packet's source address, destination address, the local time, and a secret known only to the router.

Once a capability request arrives at the destination, it applies local policy to determine how to respond. If the destination wants to allow the source to send privileged traffic, it transforms each pre-capability into a capability by hashing $\textit{pre-cap}_i$ with the number of bytes (N) and time (T) for which the capability is to be valid. It then returns the list of capabilities, along with N and T, to the source. Because the destination does not have capabilities to send to the source, it must return the capabilities in an accountable request packet of its own.

On the other hand, if the destination is overwhelmed, it has two options. It can ignore the request (or return a TCP RST without capabilities via an accountable request packet) to the source, so that it can retry its request later. If the source is persistently sending capability requests, the destination may block the source. As the capability request is an accountable DNA packet, the destination creates a DNA block request and sends it to the source's LRep. Allowing destinations to block sources from sending unwanted capability requests is more effective at preventing "denial of capability" attacks [9] than rate-limiting, as it can prevent malicious requests from ever interfering with legitimate requests.

Unlike TVA, VanGuard keeps receiver-generated capabilities secret. This is to prevent on-path routers from observing capabilities, which would allow them to send privileged traffic themselves. Before sending capabilities back to the sender, the receiver encrypts them using a key provided by the sender.

Once a sender obtains capabilities, it can send privileged packets. To send a privileged packet, the source creates a *packet-capability* for each capability it received. A packet-capability is a hash of the capability and the invariant fields of the packet, including the payload. packet-capabilities are tied to individual packets, not just source-destination pairs as in TVA. Given a packet and a valid packet-capability, it is infeasible to determine the capability used to create the packet-capability. Privileged packets are unsigned: VanGuard relies on the underlying capability architecture to protect the network.

When a router receives a privileged packet, it verifies the corresponding packet-capability. Verifying a packet-capability requires computing three hashes. The router computes the pre-capability using the information from the packet and its secret. It then computes the (secret) capability in the same way as the destination: by hashing the pre-capability with N and T (which are included in the capability header in the packet). The router finally computes the packet-capability by hashing the capability with the invariant fields of the packet. If this matches the value in the packet, the router forwards the packet, otherwise it demotes or drops the packet.

Despite not having to sign privileged packets, the sender is still accountable for such packets. If it abuses the network, the destination can block the sender. Capabilities are tied to a source-destination IP address pair and cannot be stolen by third parties to send privileged packets. Thus capabilities unambiguously identify the sender. By caching the sender's LRep from its capability request, the receiver can send a block request to the LRep at any time.

To protect requests from overwhelming privileged traffic, TVA limits requests to 5% of the capacity each link. In VanGuard, this is not necessary, as receivers can block senders that flood the network with capability requests.

The combination of DNA and capabilities provides an accountability architecture with the same benefits of DNA (primarily, a destination can block a sender regardless of network point of attachment) and the small overhead of capability-based architectures (only request packets require signatures and multiple keys). VanGuard flows are bootstrapped using accountable packets, then continued using more efficient capabilities.

5.3 Challenges of making capabilities accountable

VanGuard augments capability architectures with DNA, allowing destinations to block traffic from unwanted senders. This effectively protects the capability request channel, solving one of the main problems with capability architectures. However, in order to ensure that the privileged channel is accountable as well, VanGuard must solve two additional problems. First, capabilities are efficient partly because they do not require asymmetric cryptography, but the strong accountability of DNA does. How can capabilities provide accountability without signatures? Second, malicious routers (or hosts near the sender or receiver) can observe capabilities in packets. Because capabilities are not strongly bound to senders, nothing prevents these malicious devices from sending packets with stolen capabilities.

5.3.1 Holding senders accountable with capabilities

DNA is able to hold senders accountable because every packet contains information sufficient to unambiguously identify the sender and to block the sender's packets at the source. VanGuard capabilities must provide this same information. This requires changes to the structure of capabilities and how they are transmitted. First, to prevent anyone from impersonating the sender, VanGuard makes it impossible to steal a sender's capabilities, as described in Section 5.3.2. Second, when a receiver returns capabilities in response to an accountable request packet, it caches the information needed to block a sender for the duration of the flow. If the receiver wishes to block a sender that is using capabilities to send privileged traffic, it examines its cache to find the sender's blinding key and the LRep that authorized the sender. It then creates a block request for that LRep. The block takes effect as soon as the LRep receives the request and installs a temporary filter at the sender's first-hop router. The LRep simultaneously publishes the block in the SBS, so that the sender will not be able to obtain further tokens to request capabilities from the destination.

Alternatively, capabilities could be bound to the blinding key and LRep as well, but this would require additional space in the packet in order to provide sufficient information for routers to verify the capability.

5.3.2 Preventing capability theft

In existing capability proposals, capabilities are only weakly bound to a sending principal. Any entity that observed a capability could use it to send traffic of its own, provided it was located on or near the path from sender to destination. Section 3.7.2.3 demonstrates this effect by simulating an attack in which a compromised router steals capabilities to impersonate legitimate senders.

Because it is easy to transform capabilities into packet-capabilities, and pre-capabilities into capabilities, VanGuard must ensure that neither capabilities nor pre-capabilities can be stolen by an on-path attacker. To prevent capability theft, senders do not directly include returned capabilities in packets. A capability is treated as a secret value, known only to the sender, the destination, and the router that can verify it (a router R_B located on the path between another router R_A and the destination can in fact observe pre-capabilities in request packets, but this will not allow it to create a capability to send traffic to the destination, as described in Section 5.5). The sender uses each capability to create a packet-capability, a hash of the secret capability and the packet contents, to include in packets. A router can verify the packet-capability created from the pre-capability it stamped into the request packet by reconstructing the intermediate capability, but is not able to determine the value of any other capabilities from inspecting the packet. Anyone that observes a packet-capability cannot use it to send packets of its own.

To prevent eavesdroppers from observing capabilities as they are returned to the sender, the sender encrypts a symmetric session key in its capability request, using the destination's public key obtained via DNS (Section 5.4.5). The destination encrypts the capabilities with this key to preserve their secrecy. By preventing capability theft, VanGuard allows capabilities, instead of public-key signatures, to prove ownership of packets.

5.4 VanGuard Architecture

This section describes the VanGuard architecture in detail.

5.4.1 Assumptions

Underlying capability architecture

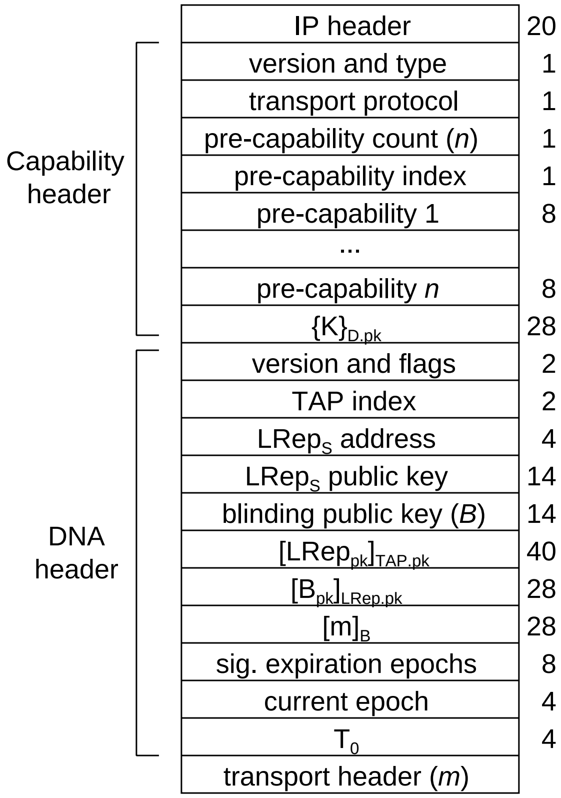
VanGuard is an extension to generic capability architectures. For ease of exposition, this chapter assumes the underlying architecture is TVA and presents its contributions as modifications to the TVA protocol.

VanGuard can adopt any mechanisms used by the underlying capability architecture. For instance, TVA incorporates measures to allow receivers to express fine-grained policies on capabilities. Specifically, when a receiver returns a capability to the sender, it restricts the capability to be valid for sending only N bytes in the next T seconds. Routers enforce these conditions, at the cost of storing bounded, temporary state. TVA also includes provisions for allowing senders to renew capabilities with privileged packets (avoided the need to send another request packet in order to continue sending packets), handling mid-flow route changes, and caching capabilities to save space in packets. As such, VanGuard does not address these concerns explicitly and relies on existing mechanisms for this functionality.

Like TVA, VanGuard assumes a globally known hash function H that can be computed at line speed. One-way compression functions based on AES are a good candidate, as specialized cores can perform AES encryption at up to 40 Gbit/s [39].

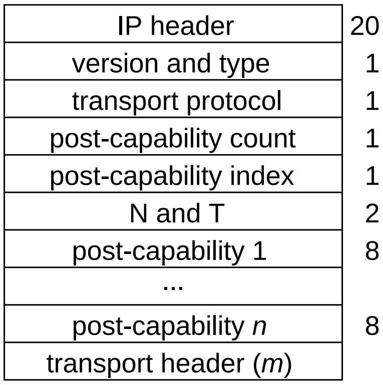
Deployment

Any subset of routers may implement VanGuard. Security increases with the size of the deployment: every time a capability is checked, it lowers the probability that the



	IP header	20
Capability header	version and type	1
	transport protocol	1
	pre-capability count (*n*)	1
	pre-capability index	1
	pre-capability 1	8
	...	
	pre-capability *n*	8
	$\{K\}_{D.pk}$	28
DNA header	version and flags	2
	TAP index	2
	$LRep_S$ address	4
	$LRep_S$ public key	14
	blinding public key (*B*)	14
	$[LRep_{pk}]_{TAP.pk}$	40
	$[B_{pk}]_{LRep.pk}$	28
	$[m]_B$	28
	sig. expiration epochs	8
	current epoch	4
	T_0	4
	transport header (*m*)	

Figure 5.2: Capability request packet format and byte count



IP header	20
version and type	1
transport protocol	1
post-capability count	1
post-capability index	1
N and T	2
post-capability 1	8
...	
post-capability *n*	8
transport header (*m*)	

Figure 5.3: Privileged packet format and byte count

sender is using forged capabilities. The probability of a path consisting of only faulty routers that do not perform checks also decreases. However, the more VanGuard routers there are on the path from sender to destination, the greater the overhead of privileged packets. Only one correct router is needed to demote a packet with an incorrect capability. Therefore, we assume capabilities are checked only when a packet crosses a trust boundary, i.e., enters into a new network. This limits the number of capabilities in the packet to the number of networks or ASes the packet traverses, which, for most packets, is less than six [57].

5.4.2 Packet types

There are two types of VanGuard packets: unprivileged, but fully-accountable, capability requests and privileged packets that carry capabilities. Legacy traffic may

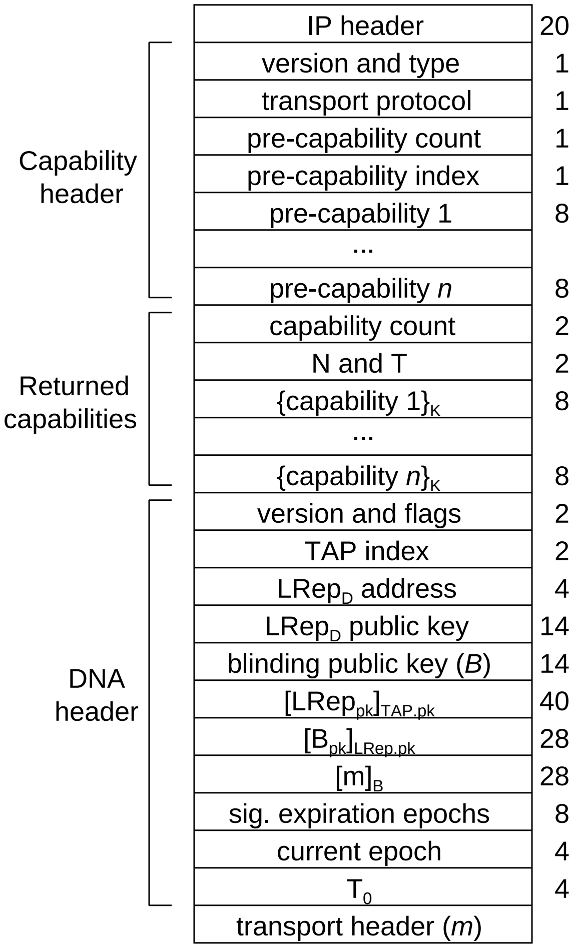


Figure 5.4: Capability response packet with returned capabilities and capability request for the reverse path

also exist, but it receives the lowest priority at routers. All VanGuard contain a capability header and a DNA header. These headers are shim headers between IP and the transport protocol. Because routers can modify packets by inserting or updating capabilities, the DNA sender's signature of the packet contents must not cover the capability header. Thus, the capability header precedes the DNA header.

Capability requests are fully accountable and carry a full DNA header. The packet format is shown in Figure 5.2. Capability response packets are also accountable (as they are requests for capabilities in the other direction) and thus have a full DNA header and two capability headers (one to request pre-capabilities and one to return the encrypted capabilities). Their format is shown in Figure 5.4. Figure 5.3 shows the format of a privileged packet.

5.4.3 Capability types

There are three types of packet markings in VanGuard. Pre-capabilities, which are stamped into request packets by routers, and capabilities, which the destination creates by hashing pre-capabilities and returning them to the sender, function almost identically in VanGuard as they do in TVA. VanGuard uses a second transformation, this time by the sender, to turn capabilities into packet-capabilities. In TVA, capabilities have two roles: they give the sender the ability to send privileged packets and they give routers the ability to verify that a packet is privileged. VanGuard uses different types of marking for each role: capabilities to bestow permission to the sender and packet-capabilities to allow routers to verify permission. This distinction is necessary to prevent unauthorized senders from being able to create valid packet-capabilities.

To maintain sender accountability, VanGuard must ensure that a sender can only create a packet-capability if it has received a capability from the destination. Because it is easy to transform a capability into a packet-capability, satisfying this requirement has two components: devices other than the sender and receiver must not be able to observe a capability in transit and capabilities must be hard to forge. VanGuard ensures the first component by using end-to-end encryption. The second component also breaks down into two sub-components: no device (other than routers in the destination's network) can observe all of the pre-capabilities in transit and pre-capabilities must themselves be hard to forge. The nature of packet forwarding trivially satisfies the first requirement, and VanGuard uses existing methods of creating pre-capabilities to ensure the second.

5.4.4 Capability formats

Pre-capabilities must be unforgeable. That is, a router R will accept a packet-capability only if it was created with a pre-capability stamped by a router in R's network. To accomplish this, VanGuard uses pre-capabilities similar to those in TVA. A VanGuard pre-capability is a hash of the sender's public key, the destination IP address, a local timestamp, and a secret known only to routers in the same network. The pre-capability for network i is thus

$$\textit{pre-cap}_i = t_i || H(S_{pk}, D_{ip}, t, secret_i)$$

where t_i is the time in network i. All packet marking fields are the same size as in TVA: an eight-bit timestamp and the last 56 bits of the output of H, creating a 64-bit value.

When a destination receives a request packet with a list of pre-capabilities, it has several options (discussed in Section 5.4.7). If it chooses to return capabilities to the sender, it transforms the pre-capabilities into capabilities. VanGuard is not particular as to how this transformation happens. Assuming VanGuard is used with TVA, the destination will create a list of capabilities that allows the sender to send N bytes over T seconds by hashing each pre-capability with N and T:

$$cap_i = t_i || H(N, T, \textit{pre-cap}_i)$$

In TVA, senders include capabilities in privileged packets, and the capabilities do not change during their lifetime. This enables capability theft. In contrast, VanGuard packet-capabilities depend on the returned capability, but are unique to every packet. To prevent others from stealing and using packet-capabilities, VanGuard binds the value of a packet-capability to the contents of a packet, so that they are valid only for that packet. Anyone that observes a packet-capability cannot use it to create packet-capabilities of their own. Of course, nothing can prevent the observer from sending an exact duplicate of the packet. For this, VanGuard uses Bloom filters to detect such duplicates 3.5.

A sender creates a packet-capability by hashing a capability with the contents of the non-invariant fields of the packet, i.e., the entire IP packet except for the TTL, QoS, and checksum:

$$\textit{post-cap}_i = t_i || H(\textit{cap}_i, \textit{pkt})$$

5.4.5 Requesting capabilities

Request packets are unprivileged, but accountable. They have two components: a capability header and a DNA header (Figure 5.2).

Sending a request packet requires knowing three pieces of information about the destination: its IP address, its list of preferred TAPs, and a public key. The public key is necessary to establish a shared symmetric key between the source and destination. The sender obtains this information through DNS. (Senders make DNS requests using accountable packets. A DNS server does not need to publish a public key, as it does not return capabilities to any sender.) Because DNA already modifies DNS records to include the list of TAPs a destination will accept, VanGuard assumes that records also store a public key. A host h generates a public keypair $(h^{VG}_{pk}, h^{VG}_{sk})$, separate from any DNA key, and publishes h^{VG}_{pk} in its DNS record. To prevent h^{VG}_{pk} from being tampered with en-route, h's administrator uses its speaks-for key to certify the key; this certificate is also stored in h's DNS record.

To send a request packet, a sender S first follows the DNA protocol to create an accountable packet. First, S commits to a blinding key. It then requests a token to talk to the destination, D, from its LRep. The LRep determines if D has blocked S by examining its local cache and the SBS. This is when blocks by a destination are enforced: if D has blocked S, the LRep will not return a token, preventing S's packets from leaving its network. Otherwise, the LRep returns a token, $\mathcal{T}_0$, to S. S inserts its payload into the packet (if creating a TCP connection, the TCP SYN is piggybacked into the capability request and the payload is the TCP header) and signs it with its blinding key. S can now populate the fields of an accountable packet according to Figure 5.2.

S now fills out the capability header. It leaves space for routers to stamp pre-capabilities (the default is six; if this is insufficient, a downstream router will increase the space in the packet). S also generates a random symmetric key K. The destination uses this key to encrypt the capabilities it returns to the sender, so that they cannot be observed in transit. S encrypts K with D_{pk}^{VG} and includes it in the capability request. S then sends the packet, and its first-hop router verifies the signature and $\mathcal{T}_0$before forwarding it.

5.4.6 Creating pre-capabilities

When a request packet crosses an AS boundary into AS_i, the receiving router checks the previous AS's token (as described in Section 3.3.6). If the token is valid, the router stamps the pre-capability for AS_i into the packet. This pre-capability, $\textit{pre-cap}_i$ is equal to $t_i || H(S_{pk}, D_{ip}, t, secret_i)$, where t_i is the time at AS_i and $secret_i$ is a secret known only to routers in AS_i. VanGuard assumes loose clock synchronization among routers in the same AS, which is already assumed by DNA. The secret value periodically changes so that capabilities expire.

If the token is invalid, the router drops the packet. Unaccountable capability requests must not be delivered to the destination. In this respect, VanGuard differs from TVA, which will forward requests with best-effort service.

5.4.7 Returning capabilities

When the destination D receives a request packet, it decides whether or not to grant capabilities to the sender. If not, it can ignore the request, or return a capability request (as it has no capabilities to send privileged packets) with empty capabilities and a TCP RST as the payload. Alternatively, if the sender is flooding the destination with capability requests, the destination creates a block request, has it signed with the administrator's speaks-for key, and sends it to the LRep named in the request packet. This prevents the sender from sending any further packets to D.

If the destination decides to grant capabilities, it first creates capabilities from the pre-capabilities, decrypts the secret key in the request packet, encrypts the capabilities with this key, and sends the capabilities in a request packet of its own (so that it can obtain capabilities to send to the source).

To turn these into capabilities, D first chooses how many bytes (N) to allow the sender to send and an interval in which to do so (T). For example, a capability may grant the sender the ability to send 100 Kbytes in 10 seconds. For each pre-capability $\textit{pre-cap}_i$ in the request packet, D sets $\textit{cap}_i$ to be $t_i || H(\textit{pre-cap}_i, N, T)$.

Next, D uses D^{VG}_{sk} to decrypt the symmetric key K. D encrypts each capability with K to prevent eavesdroppers from observing it in transit. D stores K, so that it can decrypt the capabilities that S returns to it. D creates a capability request packet and inserts each $\mathsf{Enc}_K(\textit{cap}_i)$ into the packet. D makes the packet accountable and sends it to S. This packet acquires pre-capabilities en route to S, which S returns.

D also caches the value of S's LRep, indexed by S_{pk}, so that D can still block S when it sends unsigned, privileged packets.

5.4.8 Sending privileged packets

Once S obtains (encrypted) capabilities from D, it can send privileged packets. It first decrypts the capabilities using the shared secret key K it previously generated. To send a packet P, S creates a packet-capability for each AS AS_i along the path to D, using $\textit{cap}_i$. The packet-capability is a hash of the capability and the invariant parts of the packet contents, including the payload and DNA header, but not the capability header. Because privileged packets are not signed and do not require tokens, the only value in the DNA header is S_{pk} (see Figure 5.3). S computes $\textit{post-cap}_i$ as $t_i || H(\textit{cap}_i, P')$ where P' is P with the IP TTL, QoS, and checksum fields replaced with zeroes.

5.4.9 Verifying capabilities

When a router R in AS AS_i receives a privileged packet, it validates the packet-capability for that AS, *post-cap*$_i$. It finds *post-cap*$_i$ by treating the packet-capability pointer as an index into the list of packet-capabilities. R validates the packet-capability by first computing the pre-capability it would have inserted into the packet by hashing D's IP address, S's public key from the packet, the timestamp t from the packet-capability, and its own secret. It then hashes this pre-capability with N and T to produce the capability created by S. Finally, it hashes this value with the invariant parts of the packet. If this value matches the packet-capability, the the router determines the packet to be valid.

The router also performs whatever additional checks are required by the underlying capability architecture. In the case of TVA, this includes verifying that the packet-capability has not expired (t_i is relatively fresh) and that the sender has not sent more then N bytes in the past T seconds. This requires storing a bounded amount of state for each sender. For a packet of B bytes, R creates a byte counter and a timer with a TTL of BT/N seconds. The TTL decrements every second and increases (along with the byte count) with every packet from S. If S sends less than N bytes in T seconds, then the TTL will never increase beyond $NT/N = T$ seconds and will eventually reach zero. In this case, the state is reclaimed. If the byte counter reaches N, R drops subsequent privileged packets from S until the TTL expires.

If the packet passes all checks, then the router increments the packet-capability index and forwards the packet. If a check fails, VanGuard again defaults to the underlying capability architecture as to how to handle the packet. Typically, the packet would be demoted to unprivileged legacy traffic before being forwarded to the destination.

	TVA	VanGuard
Request identifier	Path identifiers	Sender signatures
Bandwidth-limited request channel	Yes	No
Request channel scheduling policy	Hierarchical fair-queuing	None
Privileged channel scheduling policy	Per-destination fair-queuing	None
Trust assumption	All routers	The TAP
Observing capabilities allows adversary to:	Send arbitrary privileged traffic	Replay traffic

Table 5.1: Summary of differences between TVA and VanGuard.

5.4.10 Blocking a privileged sender

To block a sender S, D needs S's blinding key and the IP address of LRep_S. Both of these fields are present in request packets, so blocking S based on a request is straightforward: D creates a block request and has the network administrator sign it with its speaks-for key and forward it to LRep_S. LRep_S follows the DNA blocking protocol and returns a receipt to D's administrator. LRep_S places a filter at S's first-hop router, blocking traffic to D.

Once S has capabilities, however, it no longer includes the DNA header in its packets. When D issues capabilities to S it caches S's blinding key and LRep_S with S's address as they key. D can the blinding key and address in order to create a block. S can reclaim this state when S's capabilities expire.

5.4.11 Summary of differences between TVA and VanGuard

We now review the differences between TVA and VanGuard. The most fundamental difference is that in TVA there is no notion of sender identity other than network location. Only the path identifiers that routers insert into capability requests allow the destination to distinguish between requests from different senders. On the other hand, senders in VanGuard are uniquely identified by a public key. Malicious senders cannot cause long-term collateral damage to other senders: block requests affect only the guilty party, whereas in TVA, the attacker could continually force a nearby honest sender to share the same queues in downstream routers.

To prevent unprivileged capability requests from overwhelming privileged traffic, TVA limits requests to 5% of the bandwidth on any link. In VanGuard, requests and privileged packets share the same bandwidth. VanGuard relies on destinations to block the sources of packet floods.

As a result of using blocking to protect links, VanGuard does not need to use a scheduling policy. In TVA, routers hierarchically fair-queue request packets based on path identifiers. This prevents malicious senders from denying capabilities to honest senders by consuming the entirety of the request channel bandwidth. To prevent a malicious destination from granting capabilities to malicious senders, allowing them to congest a link that the malicious destination shares with other hosts, TVA uses per-destination fair-queuing on the privileged channel. While VanGuard could implement per-destination or per-flow queuing, we consider this attack to be outside of our threat model. There is no way to distinguish between the malicious behavior described above and fair use of the shared link, and we do not want to deny legitimate users their share of the network.

TVA and VanGuard differ in their architectural assumptions. VanGuard relies on DNA, which requires the deployment of additional infrastructure: a TAP, an SBS, LReps in networks, and modifying routers to support signature verifications and token operations. However, this architecture is not trusted. TVA, however, trusts all routers to not steal capabilities from privileged traffic: a malicious router can flood a destination using any capability it observes. On the other hand, by encrypted returned capabilities and binding packet-capabilities to packets, VanGuard prevents capability theft.

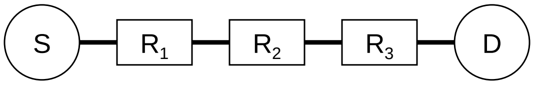


Figure 5.5: Router R_2 can observe pre-capabilities stamped by R_1 but not by R_3.

This comparison is summarized in Table 5.1.

5.5 Attacks

Capability theft

In order for unsigned packets to be accountable, no one but an authorized sender be able to create a valid packet-capability. One way capability theft can occur is if an on-path router observes a pre-capability. It can then use it to create capabilities for its own packets. While this is possible in VanGuard, it does not allow the router to send privileged packets to any destination. To see why, consider Figure 5.5. R_2 can observe the pre-capability that R_1 stamps into a request packet from S to D. If R_2 is malicious, it could turn that pre-capability into a packet-capability that would allow any sender to send a privileged packet through R_1 to D. However, R_2 cannot see the pre-capabilities that R_3 inserts into the request packet. Without this knowledge, any forged packet will be dropped (or, at best, demoted to legacy traffic) by R_3. So long as there is at least one honest router downstream of an on-path attacker, this attack will fail. VanGuard assumes that at least one router in D's network correctly stamps and verifies capabilities, so that spoofed packets will be dropped before they reach D. If routers in D's network do not operate correctly, then there is little that VanGuard can do to protect D.

The greater concern is when a router observes the full list of capabilities when they are returned to the sender. In this scenario, an on-path adversary does obtain the capabilities necessary to send spoofed traffic to D. TVA is vulnerable to this attack, as shown in Section 3.7.2.3. To prevent this attack, D encrypts capabilities before they are returned.

5.6 Evaluation

To evaluate the performance of VanGuard, we developed a simulation of a large-scale deployment of VanGuard using ns-2.

The biggest difference between VanGuard and TVA is in how they treat capability requests. In TVA, routers fair-queue requests using a series of path identifiers stamped in the packet. When a capability request crosses an AS trust boundary, the ingress router tags the request with a value that is unique among all ingress routers; this tag identifies the upstream AS. Routers hierarchically fair-queue requests based on path identifiers, which approximates per-source queuing. Even if a sender spoofs their source address and prepends fake path IDs, the path identifiers inserted by routers will be consistent across the sender's packets.

In contrast, VanGuard requires that capability requests be signed by a certified key and allows destinations to block requests from unwanted senders. Unsigned packets without capabilities are dropped by the network. This section compares the two methods of preventing unwanted capability requests from overwhelming legitimate requests.

Simulations of TVA use code provided by the authors [1]. Simulations of VanGuard use the DNA code from Section 3.7.2 to implement the blocking functionality and the above TVA code to obtain capabilities.

Topology

One of the goals of our simulations is to determine how quickly TVA senders can obtain a capability in the presence of attackers. TVA routers fair-queue capability requests based on path identifiers. Using a dumbbell topology, as in Chapter 3, would not accurately model how TVA's fair-queuing would perform on a topology closer to that of the Internet (the capability request channel was not under attack in the evaluation of TVA in Chapter 3, so the results are not topology-dependent). Instead, we use a tree-like topology, shown in Figure 5.6. The destination is connected

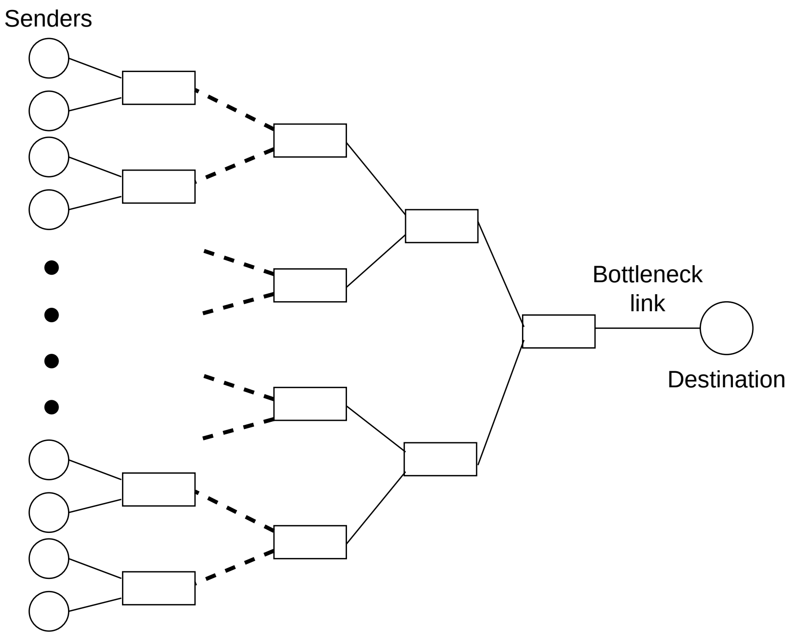


Figure 5.6: Simulation topology

to the root of the tree by the bottleneck link. Each upstream router represents a network and connects to two upstream devices: either routers or senders. Each sender is eight hops away from the destination.

In the simulations of VanGuard, the LRep is connected to each sender's first-hop router and serves both senders attached to that router.

Parameters

In each experiment there are 20 honest senders transmitting files. Honest senders attempt to transfer ten files of 20 Kbytes each to the destination using TCP.

Attackers flood the network with capability requests at a rate of 1000 per second. We vary the number of attackers between 0 and 100. Senders and attackers are placed in random locations. A first-hop router may serve two honest, two malicious, or a one honest and one malicious senders. Nodes attached to the same first-hop router and share TVA path IDs.

The bandwidth of all links except the bottleneck is 100 Mbit/s. The bottleneck link is limited to 1 Mbit/s. The latency on each link is 10 ms.

5.6.1 Experiments

We evaluated two aspects of each system: the time it takes for file transfers to complete and the time it takes for a sender to obtain a capability.

5.6.1.1 File transfer time

VanGuard

We first evaluate how well VanGuard combats bandwidth flooding attacks.

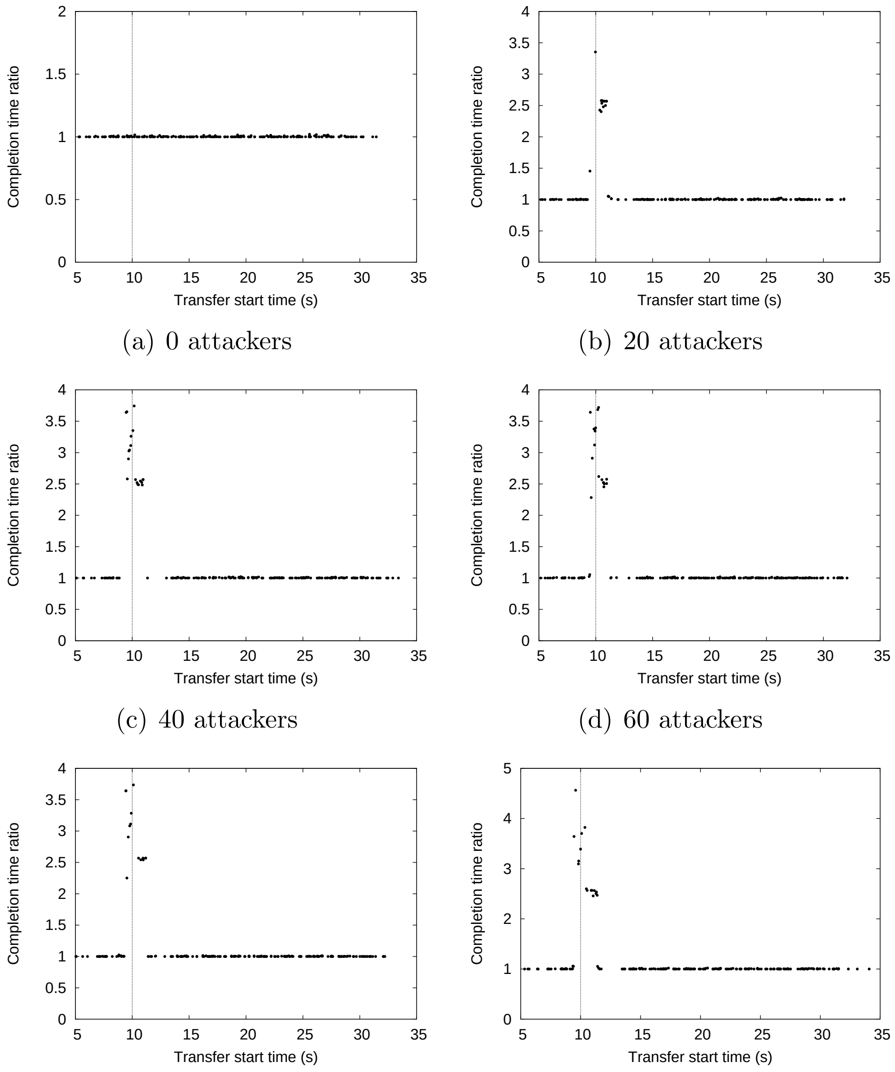
Because VanGuard does not use a dedicated channel for capability requests, requests from malicious senders compete with ongoing file transfers. This delays those transfers, as shown in Figure 5.7, until the senders are blocked. Once this occurs, file transfer time returns to pre-attack levels.

TVA

We then evaluate the performance of TVA on this topology. As both honest senders and attackers may share the same path ID (as they would in Internet-like topologies), we expect that requests from both will share the same queues, which will hamper senders that request capabilities after the attack begins.

Figure 5.8 confirms this hypothesis. Once honest senders finally obtain a capability, their transfer times decrease, although they do not return to the baseline, suggesting that senders sharing a first-hop router still have difficulty obtaining tokens.

Examining Figures 5.7 and Figures 5.8 closely to observe *which* transfers are affected reveals an insight into the nature of the difference between TVA and VanGuard. With VanGuard, the affected transfers began before the attack (or before the attackers were blocked): the attack traffic affects in-progress transfers. However, with TVA, the affected (aborted) transfers began after the attack, when senders requested capabilities; existing transfers were not affected.



(a) 0 attackers

(b) 20 attackers

(c) 40 attackers

(d) 60 attackers

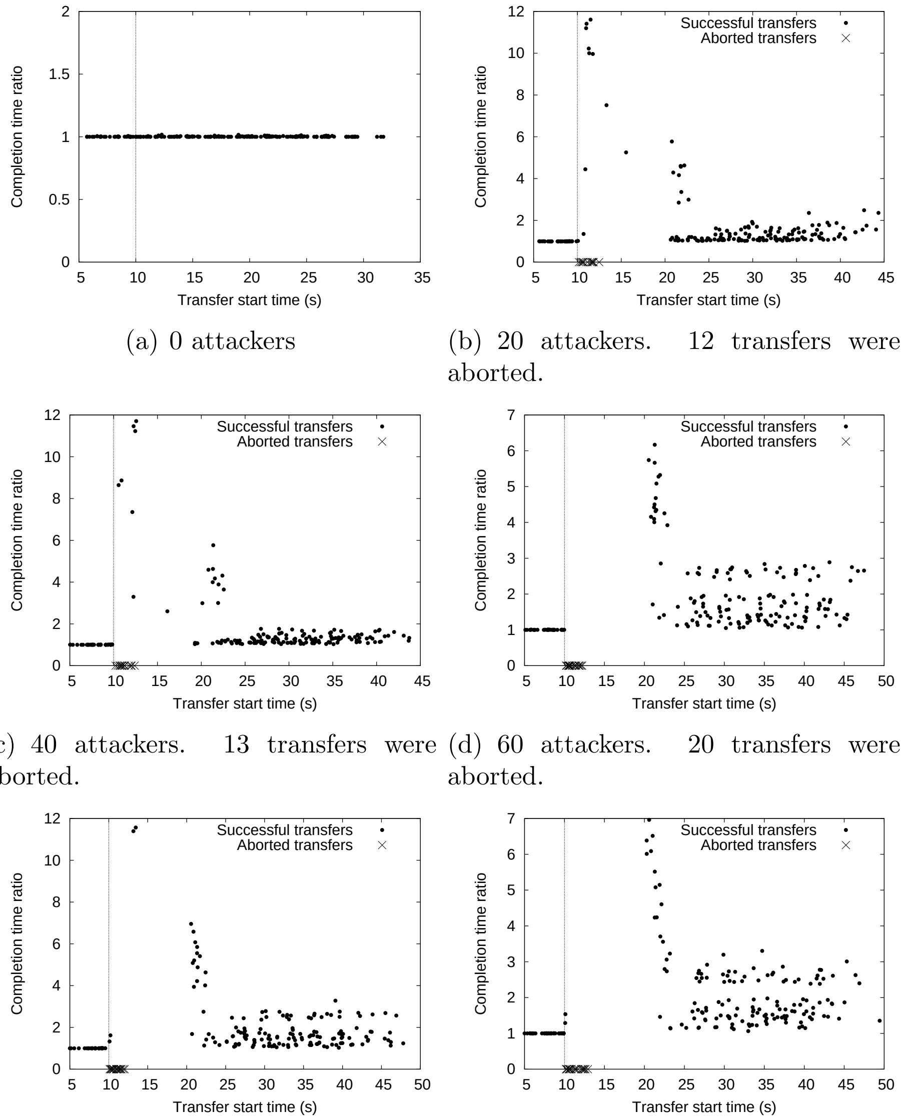
(e) 80 attackers

(f) 100 attackers

Figure 5.7: This figure shows how VanGuard performs under increasingly powerful request flooding attacks. The y-axis is the ratio of the time required to transfer a 20 Kbyte to the victim compared to the time of the first transfer. The attack starts at 10 seconds, indicated by the vertical line.

5.6.1.2 Time to obtain capabilities

Our next experiment focuses just on the ability of honest senders to obtain capabilities. This experiment focuses closer on the main difference between VanGuard and TVA, namely how each protects capability requests. Using the data from the previous experiment, we plot how long each capability request takes in each archi-



(a) 0 attackers

(b) 20 attackers. 12 transfers were aborted.

(c) 40 attackers. 13 transfers were aborted.

(d) 60 attackers. 20 transfers were aborted.

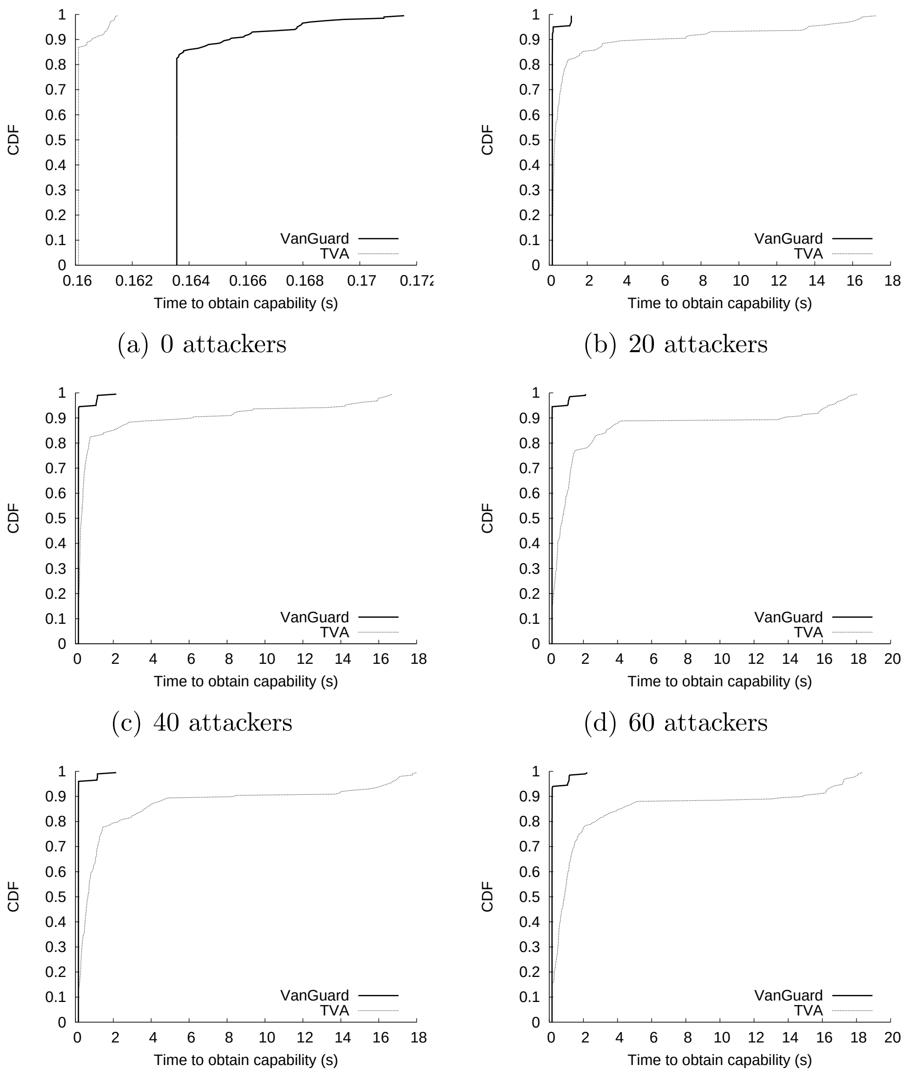
(e) 80 attackers. 18 transfers were aborted.

(f) 100 attackers. 20 transfers were aborted.

Figure 5.8: This figure shows how TVA performs under increasingly powerful request flooding attacks. The y-axis is the ratio of the time required to transfer a 20 Kbyte to the victim compared to the time of the first transfer. The attack starts at 10 seconds, indicated by the vertical line.

tecture.

The results, presented in Figure 5.9, show that senders are able to obtain a capability faster in VanGuard than in TVA, and that the difference between the two increases with the number of attackers. The disparity in Figure 5.9(a) is due to the



(a) 0 attackers

(b) 20 attackers

(c) 40 attackers

(d) 60 attackers

(e) 80 attackers

(f) 100 attackers

Figure 5.9: CDFs of the delay between sending a capability request and obtaining a capability VanGuard and TVA.

delay imposed on VanGuard packets by signature verification.

5.7 Conclusion

This chapter presents VanGuard which combines DNA with capabilities to create an accountable capability architecture.

Previous capability architectures include capabilities in the clear in packets.

Routers and other devices that observe capabilities can use the same capabilities to send privileged traffic. VanGuard solves this problem by introducing packet-capabilities, which are bound to individual packets. An adversary that eavesdrops a packet-capability can use the packet-capability to send only duplicates of the observed packet.

VanGuard outperforms both TVA and DNA. By allowing destinations to block malicious senders from sending capability requests, VanGuard protects legitimate requests better than TVA, which fair-queuing requests on a bandwidth-limited channel. The cost of sending packets in VanGuard is less than that of DNA, because VanGuard requires signatures only on request packets.

VanGuard can be thought of as a combination of both capabilities and filtering architectures like AITF [8] and StopIt [52]. Based on their comparisons of both types of DoS prevention approaches, the authors of StopIt conclude that filters outperform capabilities in terms of effectiveness, until the magnitude of the attack is so great that the destination cannot place filters at the source. By combining filters, a fail-open design, with capabilities, a fail-closed design, VanGuard offers the best of both approaches.

Chapter 6

Conclusion

In this chapter, I summarize the contributions of this thesis and propose areas of future work in network accountability.

6.1 Contributions

In this dissertation I support the following thesis: *the Internet architecture can be extended to allow receivers to block traffic from unwanted senders, even in the presence of malicious devices in the forwarding path.*

My approach to demonstrating this thesis is to incorporate accountability into the network. An accountable network architecture binds sender identities to packets and supports a set of actions based on those identities. The action I consider in my thesis is blocking traffic at the source, upon request of the receiver.

The first contribution in this thesis is the DNA accountable Internet architecture, in which every on-path component is held accountable for its actions. Malicious devices are eventually found and evicted from the network, limiting their impact. Determining if a packet should be blocked, based on the sender, requires identifying the principal that sent it. Properly identifying the sender requires two conditions: the packet is bound to a principal's identity, and this identity belongs to the sending principal.

DNA uses public-key cryptography for both tasks, as this is the only way we know of to provide the strong guarantees that DNA requires. Senders are identified by an asymmetric keypair. Senders sign outgoing packets with the private key.

Requiring signatures on packets not only allows others to identify the source of the packet, but also prevents innocent principals from being framed for malicious actions.

In order to block traffic from unwanted senders, the attribute which identifies senders (in this case, their keypair) must be long-lasting. Blocking a sender is ineffective if the sender can use a new, unblocked identity. To prevent principals from minting identities, DNA employs a centralized authority that certifies these identities. This is the only trusted component of DNA.

One of the goals of DNA is to ensure that unwanted traffic is blocked at the source, before it affects the network. To satisfy this, DNA stores block requests in a globally available repository, so that devices in the sender's network can determine if the sender has been blocked by its intended destination. DNA uses the NeighborhoodWatch distributed hash table, the second contribution of this thesis, to store block requests; NeighborhoodWatch is a scalable, distributed data store. It makes use of the trusted authority of DNA and is resilient to malicious activity by its constituent nodes, so long as the malicious nodes are unable to occupy a sufficiently long sequence of consecutive node IDs.

NeighborhoodWatch is based on Chord [75], which defines an efficient lookup operation that serves as the basis for storing and retrieving items. In a network with N nodes, a lookup takes a maximum of $O(\log(N))$ steps to complete. A lookup in NeighborhoodWatch is, at most, a small constant factor slower than in Chord.

The third contribution of this thesis is VanGuard, an accountable capability architecture built on top of DNA. VanGuard addresses the weaknesses of prior capability architectures by allowing receivers to block senders from sending unwanted capability requests, obviating the need to use fair-queuing over unreliable identifiers to protect the capability request channel. VanGuard performs better than DNA alone because capability-bearing packets need not be signed, as the capability is an indication that the destination desires traffic from the sender and has cached the information necessary to block the sender if it so desires. This is an improvement over DNA's waivers, which neither allow a waivered sender to be blocked nor provide the other benefits of capabilities such as fine-grained rate limiting [82].

These three pieces show an Internet architecture that identifies the senders of packets, allows destinations to block traffic from unwanted senders near the source even in the presence of malicious devices, and how to improve efficiency by using capabilities instead of signatures in established connections. Together, they validate my thesis.

6.2 Limitations and future work

One way to extend the work in this thesis is to to identify and remedy limitations of DNA, NeighborhoodWatch, and VanGuard. I identify several limitations below and suggest possible fixes. I also discuss two additional areas of future work: designing accountability architectures under alternate trust models and uses of accountability in networks beyond blocking unwanted senders.

6.2.1 Limitations

DNA

The major limitation of DNA is the cost of deployment. It requires significant changes to the Internet's architecture: the establishment of one or more TAPs, certifying the keys of all senders, dedicating nodes to participate in the NeighborhoodWatch SBS, deploying LReps in each AS, and modifying routers to support signature verifications and token operations. The upside to these changes is that they are mostly *additions* to the current Internet, as opposed to AIP [3], which requires changes to routing protocols, host addresses, and host hardware. Mitigating or removing any of DNA's deployment costs would produce a promising path towards deployment.

A second limitation of DNA a cost of a different type: the use of per-packet asymmetric cryptography. While this allows DNA to hold senders accountable for their packets, it is currently computationally infeasible to verify signatures at line speeds. DNA amortizes this cost by verifying signatures near the ends of the network, rather than in the middle. Of the three signatures in DNA packets, two (the signature of the LRep's key by the TAP and the signature of the sender's blinding key by the LRep) appear in multiple packets, allowing the first-hop router to cache valid signatures. VanGuard further reduces the cost of signatures. However, this does not eliminate the cost of requiring signatures on packets. It may be unrealistic to expect low-power devices such as mobile phones to sign each outgoing packet, and for first-hop routers to verify signatures on hundreds of thousands of packets per second. How can these costs be reduced so that signing packets does not affect transmission times? Can signatures be replaced with a different, more efficient mechanism, without sacrificing the ability to bind a sender to its packets?

A related limitation is the size of the DNA packet header. Including public keys and signatures in the packet header consumes a large amount of space, even compared to the 40-byte IPv6 header. As previously mentioned, two of the three signatures can be cached and replaced with smaller values. Can the need for these signatures be eliminated entirely?

NeighborhoodWatch

In order to function properly, the NeighborhoodWatch DHT must ensure that an adversary cannot control a sequence of k consecutive nodes in the identifier space, where k is the replication factor. NeighborhoodWatch places nodes randomly and evicts malicious nodes, which increase the likelihood that this condition holds but does not guarantee it. Is there a way to guarantee that this condition holds, without bounding the number of malicious nodes in the network?

Another limitation is the requirement that all nodes periodically contact the TAP. This contravenes the distributed nature of a DHT: if the TAP fails, so may the

DHT. Can the role of the TAP be fully decentralized? That is, can nodes generate neighborhood certificates in a reliable, distributed manner? Applying techniques from Morselli's dissertation [62] may yield progress towards a fully-decentralized, secure DHT.

VanGuard

Because it builds on DNA, VanGuard shares many of the same limitations, particularly the costs of deployment. VanGuard requires further changes to routers and end hosts, but does not require changes to additional devices than a deployment of DNA would.

In DNA, any device could verify is a packet was accountable or not. This is not true with capabilities. Because VanGuard uses only capabilities to protect the privileged channel, ASes may transit invalid traffic by, for instance, failing to drop or demote packets with incorrect packet-capabilities.

Of course, DNA and VanGuard do not permit the simplest DoS attack by a router: dropping packets destined to the victim. Both rely on correct routing and DNS operation in order to provide connectivity. In Section 6.2.3 I discuss how DNA can be used to secure routing protocols, and the deployment of DNSSEC [40] suggests that secure DNS is a reasonable assumption.

6.2.2 Alternate trust assumptions

Trusting in-network devices

One direction for future work is to explore how a relaxation of DNA's trust assumptions can make an accountability architecture easier to deploy. By relaxing the requirement that no devices involved in packet transmission be trusted, for instance, we could assume a limited deployment of trusted devices, on the order of several per network. Intuitively, this is a more reasonable assumption than requiring trusted hardware in every end host (as in AIP [3]) or trusting on-path routers (as in TVA [81]).

This assumption suggests an alternative, simpler design of an Internet architecture that allows destinations to block senders. As in VanGuard, there are two classes of traffic: accountable connection requests and packets in established connections. The trusted devices, or *T-boxes*, form an overlay which routes connection requests from the sender to the T-box near the destination. T-boxes ensure that only accountable requests enter the overlay and store the block requests created by hosts in their network.

Links in the trusted overlay are protected with cryptography. Two neighboring, accountable networks exchange a symmetric key that they use to authenticate traffic between the T-boxes in their networks; traffic addressed to a T-box that does not carry a MAC created with the correct key is dropped. Because the overlay transmits only accountable traffic, there is no need to trace traffic back to its source—i.e., there is no need for DNA-style, hop-by-hop tokens in packets.

To establish a connection with a destination D, a sender S finds the trusted device, T_D, in D's network via DNS, creates an accountable request to connect to D, and sends the request to its local trusted device, T_S. The request is then forwarded through the T-box overlay, network by network, until it reaches T_D. T_D examines its local state to see if D has blocked S. If not, T_D returns a token to S that allows (unsigned) traffic from S to enter D's network.

Storing block requests on a device near the destination obviates the need for globally available, secure storage for block requests. It also reduces connection establishment latency, as there is no need to query the block storage. This efficiency improvement comes at the cost of allowing requests from blocked senders to traverse the overlay, only to be denied by the destination's T-box.

Packets in established connections need not be accountable, but they must carry a token generated by a T-box. Only the T-box that is responsible for storing block requests created by a destination D will grant a token to establish a connection to D. Therefore, a token does not need to indicate which specific T-box created it, only that it was created by a T-box. It is an open question whether this relaxation affords a more efficient solution that using digital signatures.

Removing global trust

DNA has a single trusted component: the TAP, which certifies principals' identity keys. This trust can be distributed across different, competing TAP services. Is it possible to distribute this trust further? As a point of comparison, senders in AIP [3] generate their own keys, and networks are responsible for ensuring users do not generate "too many" keys. There is no need for a central authority to certify keys, though each network must be trusted to limit the number of keys a user creates.

In the T-box design described above, the TAP is not necessary: T-boxes can authenticate local users and certify their keys. What other assumptions lead to an accountable architecture that does not rely on a centralized trusted authority? Could such an architecture be more resilient to attacks than DNA? Finding the answers to these questions may lead to an accountability architecture that does not rely on a set of authorities that dictate what is and is not a valid principal.

6.2.3 Using accountability for security

An accountable network is a building block that has many security-related uses. This thesis shows two such uses: blocking unwanted senders and securing capability systems. Here, I discusses several more uses that would serve as areas of future research.

An obvious use of accountability is for network auditing. Because accountable packets can be mapped to their sender, it is easy to determine who is responsible for network events. This has applications ranging from improving the accuracy of intrusion detection systems to preventing unauthorized users, such as worms, from exploiting vulnerabilities.

Similarly, accountability can be used to dismantle botnets. Currently, bots are hard to identify, because they often attack with spoofed source addresses. However, the compromised hosts themselves are not malicious; their owners are simply unaware that they are participating in a botnet. Identifying the true source of attacks and holding the sending host accountable (temporarily disconnecting it, for instance) encourages users to properly maintain their machines.

Additionally, accountability can help shut down the botnet itself. Botnets are often managed via a command and control (C&C) server [29]. By intercepting traffic from the C&C server, perhaps with a honeypot [67], network operators can identify and quarantine the source of commands. Similar events have occurred in the past, such as the shutdown of the Atrivo network [11], though they are rare; accountability will facilitate this process.

The final example I discuss is how accountability can be used to create a secure inter-domain routing protocol. Route hijacking, when an AS announces a prefix that it does not own [14], is a common occurrence in the Internet. There have been several high-profile instances, most notably Pakistan's hijack of YouTube [61]. To prevent these attacks, routers must accept only *valid* prefix announcements. A router must know two facts to determine if an announcement is valid: the identity of the sender, and if the sender is authorized to make the announcement. In an accountable Internet, it is easy to answer the first question. To answer the second, the routing protocol can use a slight modification of DNA's speaks-for keys. In DNA, the TAP certifies speaks-for keys for administrators of each prefix. To extend the role of speaks-for keys to secure routing, the TAP can also include the AS name in the certificate. This allows the administrator (or a BGP-speaking router) to prove that they are the owner of the announced prefix.

6.3 Parting thoughts

I conclude my dissertation with one impression of the work contained within. What strikes me most about the work I have presented here is how many areas is draws from. DNA relies on the design of strong but efficient cryptographic protocols.

NeighborhoodWatch is a contribution to secure distributed system design. VanGuard is based on work in filtering and capability architectures. In order to determine how quickly routers could perform certain computations, we examined specifications of dedicated hardware. We used measurement data, collected both at our university and from a global network security community, to guide some of our design choices. We motivated the conference publication of NeighborhoodWatch [17] by showing how it could be used to fight spam; this required examining current practices in sending and combating spam.

When I started this work, I did not expect that creating systems that prevented denial of service attacks would require such broad knowledge and expose me to so many areas. I take it as a positive sign that results from one area of networking research can easily lend themselves to another. It is also interesting to me to see how, as research in certain areas matures (DHTs, for instance), it is adopted into work in other areas (mitigating DoS attacks).

On the other hand, one could argue that drawing from such a wide variety of areas is the symptom of a negative aspect of my work: its complexity. Simplicity is often cited as a design goal, that simple systems are easier to implement and reason about. While this may be true, I think that at some point, "simple" systems will face limitations on what they can accomplish, at least in regards to protecting resources on the Internet. Evidence suggests that this time is rapidly approaching: botnets are growing ever larger, due to host insecurity [7], subtle attacks on DNS accelerated the adoption of DNSSEC [44], and current routing systems are vulnerable to prefix hijacking. As the Internet grows, so must the complexity of the systems that run it. In order to be resilient to current and future threats, the Internet community needs to adopt secure systems like DNA and VanGuard, despite their complexity and costs.

Bibliography

[1] A DoS limiting network architecture. http://www.cs.duke.edu/nds/ddos/.

[2] Akamai Technologies. http://www.akamai.com.

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

[4] D. G. Andersen. Mayday: distributed filtering for Internet services. In *USITS*, 2003.

[5] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet denial-of-service with capabilities. In *HotNets*, 2003.

[6] Arbor Networks. http://www.arbornetworks.com.

[7] Arbor Networks. Worldwide infrastructure security report, 2009.

[8] K. Argyraki and D. R. Cheriton. Active Internet traffic filtering: Real-time response to denial-of-service attacks. In *USENIX*, 2005.

[9] K. Argyraki and D. R. Cheriton. Network capabilities: The good, the bad and the ugly. In *HotNets*, 2005.

[10] K. Argyraki, P. Maniatis, O. I. andSubramanian Ashish, and S. Shenker. Loss and delay accountability for the Internet. In *ICNP*, 2007.

[11] J. Armin. Atrivo - cyber crime USA, 2008. http://hostexploit.com/downloads/Atrivo%20white%20paper%20090308ad.pdf.

[12] F. Baker and P. Savola. Requirements for IP version 4 routers. RFC 3704, Mar. 2004.

[13] H. Ballani, Y. Chawathe, S. Ratnasamy, T. Roscoe, and S. Shenker. Off by default! In *HotNets*, 2005.

[14] H. Ballani, P. Francis, and X. Zhang. A study of prefix hijacking and interception in the Internet. In *SIGCOMM*, 2007.

[15] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating systems support for planetary-scale network services. In *NSDI*, 2004.

[16] S. M. Bellovin, M. Leech, and T. Taylor. ICMP traceback messages. Internet Draft: draft-ietf-itrace-04, Feb. 2003.

[17] A. Bender, R. Sherwood, D. Monner, N. Goergen, N. Spring, and B. Bhattacharjee. Fighting spam with the NeighborhoodWatch DHT. In *INFOCOM*, 2009.

[18] A. Bender, N. Spring, D. Levin, and B. Bhattacharjee. Accountability as a service. In *SRUTI*, 2007.

[19] D. J. Bernstein. SYN cookies. `http://cr.yp.to/syncookies.html`, 1996.

[20] B. Bhattacharjee, R. Rodrigues, and P. Kouznetsov. Secure lookup without (constrained) flooding. In *WRAITS*, 2007.

[21] B. Bloom. Space/time trade-offs in hash coding with allowable errors. In *Communications of the ACM*, 1970.

[22] M. Casado, A. Akella, P. Cao, N. Provos, and S. Shenker. Cookies along trust-boundaries: Accurate and deployable flood protection. In *SRUTI*, 2006.

[23] M. Casado, P. Cao, A. Akella, and N. Provos. Flow-cookies: Using bandwidth amplification to defend against DDoS flooding attacks. Technical report, Stanford HPNG, 2006.

[24] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure routing for structured peer-to-peer overlay networks. In *OSDI*, 2002.

[25] B. Claise. Cisco Systems NetFlow services export version 9. RFC 3954, Apr. 2004.

[26] D. D. Clark. The design philosophy of the DARPA Internet protocols. In *SIGCOMM*, 1988.

[27] D. D. Clark, J. Wroclawski, K. R. Sollins, and R. Braden. Tussle in cyberspace: Defining tomorrow's Internet. *IEEE/ACM Transactions on Networking*, 13(3):462–475, June 2005.

[28] I. Clarke, S. Miller, T. Hong, O. Sandberg, and B. Wiley. Protecting free expression online with Freenet. *IEEE Internet Computing*, 2002.

[29] E. Cooke, F. Jahanian, and D. McPherson. The zombie roundup: Understanding, detecting, and disrupting botnets. In *SRUTI*, 2005.

[30] W. Diffie and M. E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[31] J. Douceur. The Sybil attack. In *IPTPS*, 2002.

[32] P. Ferguson and D. Senie. Network ingress filtering. RFC 2827, May 2000.

[33] A. Fiat, J. Saia, and M. Young. Making chord robust to Byzantine attacks. In *ESA*, 2005.

[34] M. J. Freedman, E. Freudenthal, and D. Mazières. Democratizing content publication with Coral. In *NSDI*, 2004.

[35] S. Goldberg, D. Xiao, E. Tromer, B. Barak, and J. Rexford. Path-quality monitoring in the presence of adversaries. In *SIGMETRICS*, 2008.

[36] D. Goldschlag, M. Reed, and P. Syverson. Onion routing. *Commun. ACM*, 42(2), Feb. 1999.

[37] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical accountability for distributed systems. In *SOSP*, 2007.

[38] A. Haeberlen, R. Rodrigues, K. P. Gummadi, and P. Druschel. Pretty good packet authentication. In *HotDep*, 2008.

[39] Helion Technology. Giga AES cores.

[40] ICANN and VeriSign. Root DNSSEC. `http://www.root-dnssec.org/`. Accessed on 8/2010.

[41] Intel IXP 2855 network processor. `http://developer.intel.com/design/network/products/npfamily/ixp2855.htm`.

[42] Internet World Stats. Internet usage statistics. `http://www.internetworldstats.com/stats.htm`.

[43] J. Ioannidis and S. M. Bellovin. Implementing pushback: Router-based defense against DDoS attacks. In *NDSS*, 2002.

[44] D. Kaminsky. Black ops 2008 – its the end of the cache as we know it. `http://www.doxpara.com/DMK_BO2K8.ppt`.

[45] C. Kaufman, R. Perlman, and M. Speciner. *Network Security: Private Communication in a Public World*. Prentice Hall, 2 edition, 2002.

[46] S. Kent, C. Lynn, and K. Seo. Secure border gateway protocol (Secure-BGP). *JSAC*, 18(4), April 2000.

[47] S. Kent and K. Seo. Security architecture for the Internet Protocol. RFC 4301, Dec. 2005.

[48] A. D. Keromytis, V. Misra, and D. Rubenstein. SOS: secure overlay services. In *SIGCOMM*, 2002.

[49] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. F. Kaashoek. The Click modular router. *ACM Transactions on Computer Systems*, 18(3):263–297, August 2000.

[50] B. M. Leiner, V. G. Cerf, D. D. Clark, R. E. Kahn, L. Kleinrock, D. C. Lynch, J. Postel, L. G. Roberts, and S. Wolff. A brief history of the Internet. `http://www.isoc.org/internet/history/brief.shtml`.

[51] Limelight Networks. `http://limelightnetworks.com`.

[52] X. Liu, X. Yang, and Y. Lu. To filter or to authorize: Network-layer DoS defense against multimillion-node botnets. In *SIGCOMM*, 2008.

[53] X. Liu, X. Yang, and Y. Xia. NetFence: Preventing Internet denial of service from inside out. In *SIGCOMM*, 2010.

[54] J. López and R. Dahab. An overview of elliptic curve cryptography. Technical report, State University of Campinas, 2000.

[55] X. Lui, A. Li, X. Yang, and D. Wetherall. Passport: Secure and adoptable source authentication. In *NSDI*, 2008.

[56] X. Lui, X. Yang, D. Wetherall, and T. Anderson. Efficient and secure source authentication with packet passports. In *SRUTI*, 2006.

[57] D. Magoni and J. J. Pansiot. Analysis of the autonomous system network topology. *Computer Communication Review*, 31(3):26–37, July 2001.

[58] R. Mahajan, S. M. Bellovin, S. Floyd, J. Ioannidis, V. Paxson, and S. Shenker. Controlling high-bandwidth aggregates in the network. *Computer Communication Review*, 32(3):62–73, July 2002.

[59] U. Maheshwari, R. Vingralek, and W. Shapiro. How to build a trusted database system on untrusted storage. In *OSDI*, 2000.

[60] S. McCreary and k claffy. Trends in wide area IP traffic patterns: A view from Ames Internet Exchange. In *ITC Specialist Seminar*, 2000.

[61] D. McCullagh. How Pakistan knocked YouTube offline (and how to make sure it never happens again). `http://news.cnet.com/8301-10784_3-9878655-7.html`.

[62] R. Morselli. *Lookup Protocols and Techniques for Anonymity.* PhD thesis, University of Maryland, College Park, 2006.

[63] The OpenSSL project. `http://www.openssl.org`.

[64] B. Parno, D. Wendlandt, E. Shi, A. Perrig, B. Maggs, and Y.-C. Hu. Portcullis: Protecting connection setup from denial-of-capability attacks. In *SIGCOMM*, 2007.

[65] R. Perlman. *Network layer protocols with Byzantine robustness.* PhD thesis, MIT, 1988.

[66] Prolexic Technologies. `http://www.prolexic.com`.

[67] N. Provos. A virtual honeypot framework. In *USENIX Secrity*, 2004.

[68] K. Ramakrishnan, S. Floyd, and D. Black. The addition of explicit congestion notification (ECN) to IP. RFC 3168, September 2001.

[69] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *SIGCOMM*, 2001.

[70] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Middleware*, 2001.

[71] S. Savage, D. Wetherall, A. R. Karlin, and T. Anderson. Practical network support for IP traceback. In *SIGCOMM*, 2000.

[72] D. R. Simon, S. Agarwal, and D. A. Maltz. AS-based accountability as a cost-effective DDoS defense. In *HotBots*, 2007.

[73] E. Sit and R. Morris. Security considerations for peer-to-peer distributed hash tables. In *IPTPS*, 2002.

[74] A. C. Snoeren, C. Partridge, L. A. Sanchez, C. E. Jones, F. Tchakountio, S. T. Kent, and W. T. Strayer. Hash-based IP traceback. In *SIGCOMM*, 2001.

[75] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for Internet applications. In *SIGCOMM*, 2001.

[76] Trusted Computing Group. TCG trusted network connect: TNC architecture for interoperability, 2008.

[77] P. Wang, N. Hopper, I. Osipkov, and Y. Kim. Myrmic: Secure and robust DHT routing. Technical report, U. of Minnesota, 2006.

[78] A. Yaar, A. Perrig, and D. Song. Pi: A path identication mechanism to defend against DDoS attacks. In *Security and Privacy*, 2003.

[79] A. Yaar, A. Perrig, and D. Song. SIFF: A stateless Internet flow filter to mitigate DDoS flooding attacks. In *Security and Privacy*, 2004.

[80] X. Yang and X. Liu. Internet protocol made accountable. In *HotNets*, 2009.

[81] X. Yang, D. Wetherall, and T. Anderson. A DoS-limiting network architecture. In *SIGCOMM*, 2005.

[82] X. Yang, D. Wetherall, and T. Anderson. TVA: A DoS-limiting network architecture. *Transaction on Networking*, 16(6), Dec. 2008.

[83] A. R. Yumerefendi and J. S. Chase. Trust but verify: Accountability for network services. In *SIGOPS European Workshop*, 2004.

[84] A. R. Yumerefendi and J. S. Chase. Strong accountability for network storage. *ACM Transactions on Storage*, 3(3), October 2007.