SANE: A Protection Architecture for Enterprise Networks

Martin Casado, Tal Garfinkel, Aditya Akella
Dan Boneh, Nick McKeown, Scott Shenker
{casado,talg,dabo,nickm}@stanford.edu
aditya@cs.cmu.edu, shenker@icsi.berkeley.edu

Abstract

Connectivity in today's enterprise networks is regulated by a combination of complex routing and bridging policies, along with various interdiction mechanisms such as ACLs, packet filters, and other middleboxes that attempt to retrofit access control onto an otherwise permissive Internet architecture. This leads to enterprise networks that are inflexible, fragile and difficult to manage.

We offer SANE, a protection architecture for enterprise networks that overcomes these limitations. By default, hosts can only contact a logically centralized reference monitor that hands out capabilities (encrypted source routes) for services, according to declarative access control policies (e.g. *Alice can access http-proxy*). This provides flexible fine grain access control and eliminates the need for more complex ad-hoc mechanisms.

1   Introduction

The Internet architecture has its origins in a more innocent era, when the primary goal was to provide universal connectivity and the focus was on overcoming technical challenges, not defending against malicious attacks. As a result, the Internet designers opted for an open (by default, everyone can talk to everyone), decentralized, and cooperative architecture that has supported the phenomenal growth of IP networks, leading to the Internet of today.

Unfortunately, the world has changed dramatically. The prevalence of worms, malware and sophisticated attackers makes *protection* an increasingly important goal. However, the openness, decentralization, and assumption of cooperation that enabled the early success of the Internet makes protection hard, resulting in an architecture ill-suited to meeting the stringent security demands of enterprise networks.

Protection has been retrofitted to enterprise networks via many disparate mechanisms including router ACLs, firewalls, NATs and other middle-boxes, along with increasingly baroque link layer technologies, such as VLANs. However, these mechanisms are far from ideal.

They require a significant amount of configuration and oversight [26], are quite limited in the range of policies they can enforce [28] and produce networks that are complex [29] and brittle [30].

Moreover, even with these techniques, security within the enterprise remains notoriously poor. Worms routinely cause significant losses in productivity [5] and potential for data loss [18, 20]. Attacks resulting in theft of intellectual property and other sensitive information are similarly common [12].

The long and largely unsuccessful struggle to "add" protection to conventional enterprise networks has convinced us that one should start over with a clean slate, with protection as a fundamental design goal. We describe a Secure Architecture for the Networked Enterprise (*SANE*), built from the ground up with the objective of providing protection.

SANE requires the network to be *Default Off*, in the sense that every transmission in the network requires explicit authorization; for Host A to communicate to Host B, it must first request a capability from a central policy authority. The *capability* (encrypted source route) is checked by every forwarding element along the path from Host A to Host B. This allows fine-grained connectivity policies to be explicitly defined and executed in a logically *centralized* fashion.

SANE minimizes the power and information available to an attacker by adopting a conservative *Least Privilege* and *Least Knowledge* philosophy: components in our architecture are minimally trusted, and are given access to just the smallest set of network resources required for completing authorized transmissions and all network information, such as topology, forwarding paths and next-hops are disseminated on a strictly need-to-know basis.

Of course, any design with a logically centralized policy authority raises issues of reliability, overload, and vulnerability to attack. The SANE design addresses these issues explicitly by providing multiple copies of the policy authority and allowing it to cut off transmissions from misbehaving hosts. However, we should be clear that SANE is only intended for the enterprise do-

main, where there is a single, well-defined policy hierarchy, the network is actively managed and of limited scale, and protection is paramount.

In the next section we discuss the motivation behind our architecture followed by an overview of the basic design in §3. In §4 we describe the architecture in detail and how it integrates with IP. §5 discusses fault tolerance and §6 covers attack resistance. We describe our implementation in §7. We present related work in §8 and conclude in §9.

2 Motivation

As stated before, we believe that the principles of openness (default-on), decentralization, and cooperation, while critical to the early growth of the Internet, are now a significant hindrance to building effective protection in today's enterprise networks. We illustrate this by contrasting these principles and their concrete realization in today's networks with the principles set forth by Saltzer and Schroeder [27] in their classic work on the design of protection systems. While these principles were originally framed in the context of host security, we believe they are equally germane to enterprise networks.

1. The principle of "least privilege" states that an entity should be granted the minimum amount of access needed to carry out its task, thus minimizing the damage should the entity turn faulty or malicious. The principle of "fail-safe defaults" states that, by default, access should be given only with explicit permission. This ensures that overlooked or misconfigured mechanisms fail safe.

The *default-on* connectivity afforded by today's networks is in direct violation of both of these principles. Most clients in an enterprise need access to very few resources (e.g. file server, print server, mail server), yet are usually given much broader access rights. Consequently, when a machine or router is compromised, the network provides little containment.

Unfortunately, the mechanisms used to retrofit protection in enterprise networks don't give a practical way to implement least-privileged access. Solutions like NAT provide very coarse-grain boundaries. VLANs are designed to partition machines into common broadcast domains, and do not control individual access. Their small address space size (VLAN tags are only 12 bits) precludes their use for fine-grain access control.

When protection policies are inconsistent (e.g., when routes change, or configurations are modified) a component fails open; furthermore, it does so silently, which means inconsistencies are often only discovered after an attack [30].

2. The principle of "economy of mechanism" states that a design should be as simple as possible. Such simplicity is essential to providing assurance.

Today's networks use a variety of decentralized routing protocols (RIP, OSPF, EIGRP and static policies), each configured with many thousands of lines of policy [30]. Operating in conjunction with the routing protocols are a plethora of mechanisms to retrofit protection such as VLANs, ACLs, firewalls and NATs. The network-wide security policy is therefore extremely complex, difficult to configure and highly fragile. Reports of attacks on such mechanisms are widespread [4, 6, 2, 7].

3. The principle of "psychological acceptability" states that mechanisms must facilitate easy policy specification, and further that specification occur close enough to the level of policy goals to make policy coherent.

Access control policy today is a sum of host and router configurations. These are usually manually entered [26], and specify a variety of rules ranging from VLAN port mappings (link layer) to static routes (IP) and packet filter rules (application layer). The interaction among these configurations determines enterprise-wide connectivity, along with dynamic mechanisms such as NAT and learning bridges. It is not surprising, then, that policies are very difficult to reason about, and misconfigurations and conflicts are common.

5. The principle of "complete mediation" states that every access to every network entity must be checked for authority.

Since enterprises focus primarily on preventing break-ins, internal forwarding often takes place unchecked. Therefore, once a host has been compromised, an attacker has unfettered access to many other enterprise network resources. Mediation in enterprise networks is made difficult by the fact that end-host identifiers such as IP addresses and VLAN tags are trivially forged. Moreover, a single IP address might represent multiple hosts simultaneously (NAT), or over time (DHCP); VLAN tags can change based on which machine has plugged into a port.

As this quick review illustrates, today's enterprise networks are highly inconsistent with the Saltzer and Schroeder security principles, which may explain why these networks are so insecure. In an attempt to move beyond ad hoc mechanisms and develop a more systematic security architecture for enterprise networks, we slavishly followed the Saltzer and Schroeder security principles. The result is SANE, which we now describe.

3 SANE Architecture Overview

Enterprise networks have several properties that facilitate robust protection. First, enterprise networks are highly structured and centrally administered, making it practical (and probably desirable) to implement policies in a central location. [1]

Second, most machines in enterprise networks are clients that typically contact a predictable handful of local services (e.g. mail, printers, file server, source repository, an http-proxy or ssh gateway). This makes it feasible to grant relatively little privilege to clients and specify simple declarative access control policies – we adopt a policy interface similar to that of a modern distributed file systems.

Third, in an enterprise network, we can assume that hosts and principles are authenticated; this is common today, and already supported by widely deployed directory services such as LDAP and Active Directories. This allows us to express policies in terms of meaningful entities, such as hosts and users, instead of weakly bound end-point identifiers such as IP and MAC addresses.

Finally, enterprise networks—when compared to the Internet at large—can quickly adopt a new protection architecture. "Fork-lift" upgrades of entire networks are not uncommon, and new networks are regularly built from scratch. Further, there is a significant willingness to adopt new technologies due to the high cost of security failures.

3.1 Design Overview

SANE defines a *Protection Layer* for enterprise networks: a mechanism for granting fine grain access to network resources that provides low level enforcement and supports high level policy specification. This protection layer resides between the link-layer (e.g. Ethernet) and network layer (i.e. IP) in an enterprise network, similar to the place that VLANs occupy.

The SANE protection layer controls connectivity in the enterprise. IP continues to function as the "network layer" providing wide area connectivity as well as a common framing format to support the use of unmodified end hosts. However, within the enterprise, IP addresses are not used for identification, location or routing.

In Figure 1, we show important components of a SANE network. By default a network entity (switch or host) is isolated from the rest of the network (default off) and can only to communicate with a *Domain Controller* (DC). Based on an access control policy, the DC grants *capabilities* to permit communication between entities.
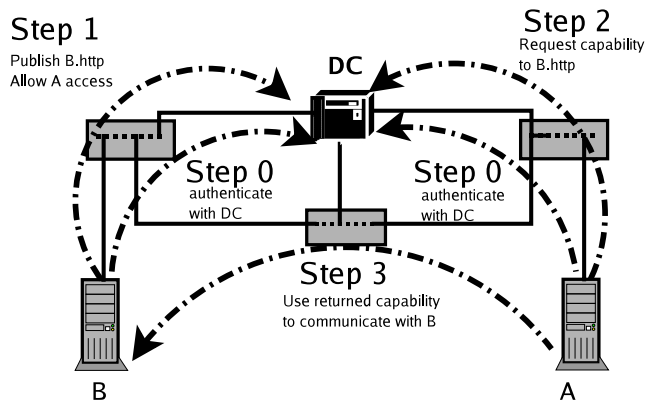

Figure 1: Components and example communication in SANE. Each end-host authenticates with the DC (step 0). The server publishes the service with associated access controls (step 1). The client requests a capability to communicate with the service (step 2). The client uses that capability to communicate directly with the client (step 3).

On joining the SANE network, every principle (e.g. *A* in Figure 1) must authenticate with the DC (*Step 0*) and establish a secure channel. A principle *B* wishing to offer a service *publishes* it with the DC under a unique name, e.g. *B.http*. The principle can also specify access controls for the service (*Step 1*). To access this service, a client *A* must request a *capability* for the service from the DC (*Step 2*). The DC consults its access control policy and determines whether to hand out the capability.

Throughout this document, we always refer to the domain controller as *the DC*. We refer to hosts that publish services as *servers* and hosts that request access to those services as *clients*.

The capabilities handed out by the DC are inserted by the client in all packets to the server (*Step 3*). Capabilities in SANE have a short lifetime (variable, typically 10 minutes) after which they must be requested again. The DC also hands out capabilities for the server to communicate with the client. These are given to the client, who hands them over to the server with the first packet.

SANE capabilities are encrypted, strict, source routes from the client to the server. Capabilities are encrypted so that each switch along the path can only see the next and previous hop on a route, and authenticated to demonstrate that the capability was generated by the DC.

Each hop in the encrypted source route consists of a tuple $< previous\_hop, next\_hop >$. Tuples are encrypted in layers working backward from the last hop (See Figure 2). For example, the three hop source-route from A to B depicted in figure 2 would be result in the capability $E_1(< A, 2 > E_2(< 1, 3 > E_3(< 2, B >)))$.
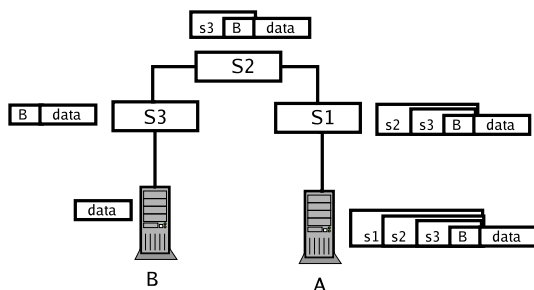
3

Figure 2: Packet forwarded from host A to host B using a hop-by-hop capability. Each layer contains the next-hop information, the previous layer and is encrypted using the associated switch's symmetric key. The capability is passed to A by the DC (not shown) and can be re-used to send packets to B until it expires.

Here $E_k(m)$ denotes encryption of message m under key k, which is a symmetric key shared by switch k and the DC. This type of layered routing is also called *onion routing* [14]. By encryption we mean authenticated encryption providing both confidentiality and integrity. [2]

At each hop, a switch removes its next-hop information from the capability. For example, if switch 2 received the onion $E_2(<1,3>E_3(<2,B>))$, it would remove $<1,3>$, check that the packet arrived from switch 1, and forward $E_3(<2,B>)$ to switch 3. Because each switch uses its key, it can only decrypt its own layer of the onion. Therefore, each switch knows no more than the next and previous hop along the path.

At the bottom layer of the capability, the DC places a *server-port*, which was specified when the service was first published. The receiving host uses the server-port to demultiplex the packet and deliver it to the correct service. Server-ports are at the lowest layer of the capability and cannot be modified en-route. In other words, the server-port ensures that only the correct service on the end-host (e.g. TCP port) will receive authorized packets.

Next, we briefly discuss some key aspects of SANE. The full details of our architecture are provided in §4:

Policy definition. Network policies are declared using high level constructs such as users, services and hosts: For example, *Alice can contact service http*. Policy definition is discussed in §4.4.

Bootstrapping. How does a host communicate with the DC to request a capability? This is accomplished by the

switches constructing a minimum spanning tree (MST). The details are provided in §4.2.

DC has full knowledge of the topology. The overall network topology is obtained from period link state updates sent by each switch to the DC. The link state update mechanism is discussed in §4.3.

Switches/hosts don't know the topology. Each switch makes a forwarding decision based on information in its own layer of the capability. Other layers are opaque and hidden by the private key in each switch. Link state updates to the DC are encrypted to hide network topology.

The DC is only logically centralized. Greater scalability and redundancy in the DC can be provided both via clustering and replication at different points in the topology. Each DC makes autonomous routing decision based its own on topology and key information. Further discussion on scaling the DC is given in§5.1.

3.2 SANE Architecture Properties

In terms of the principles set forth by Saltzer and Schroeder, SANE has the following properties.

Fail-Safe Default. By default, a SANE element can only forward packets based on access that has been explicitly granted via capabilities, or send packets on the default route to the DC.

Complete Mediation. Access control is enforced at every point in the network. Policy is applied uniformly regardless of how the topology changes, or where new elements are added (e.g. wireless access points). In contrast to today's enterprise networks, SANE networks actually benefit from large, highly redundant, complex topologies since the number of enforcement elements scales with the size of the network. Adding switches and path redundancy does not undermine existing security policy [30] but rather larger networks end up being more robust to DOS attack, more fault tolerant, more resistant to traffic analysis, and provide better defense in depth.

Economy of Mechanism. A key benefit of SANE is that it uses one central, uniform mechanism—the granting of capabilities by a DC—and so eliminates the need for many disparate protection mechanisms such as VLAN, ACLs, firewalls, and policy routing.

Least Privilege. SANE provides least privilege in two ways. First, a sender can only address the particular service for which it has been explicitly granted access, and its packets must travel over an explicitly authorized path.

Second, switching elements can only forward packets to their neighbors and talk to the DC. A switch cannot obtain capabilities of its own, and so cannot initiate new traffic to an end-host. This limits the damage that could be carried out by a compromised switch.

---

[2]Authenticated encryption can be implemented using a block cipher (such as the advanced encryption standard, AES) and a message authentication code (MAC, such as HMAC). Alternatively, authenticated encryption can be done more efficiently using the offset codebook mode [25] which provides both confidentiality and integrity in one pass over the data.

Least Knowledge. SANE end-hosts and switches only know the network elements immediately adjacent to them, which means a compromised switch reveals very little topology information to an attacker. Further, the identities of communicating end-hosts are invisible to the switches.

Psychological Acceptability. SANE policy is defined in terms of principles and service, and can be managed from a single location. SANE policy can be specified independent of topology, and uniformly applied by every element in the system. This is quite different from today's networks where policy is tied to the topology (e.g. what port you are plugged into, where you are in relationship to the firewall) and physical addresses. Consequently, complexity increases dramatically as more dynamic elements are introduced.

3.3 Are We inSANE?

The design of SANE is so far removed from that of typical IP networks, specifically of the Internet, that the rationale behind SANE's design merits further justification. It is natural to think that, in designing SANE, we push aside many issues that were paramount for the Internet—such as scalability, adapting to failures, no single point-of-failure, transparency to applications, and autonomous control. Therefore, it might appear that we are advocating the undoing of all the good the Internet has done. However, we wish to re-emphasize that the *SANE design is targeted for a specific context—enterprise networks—*where some of the above issues are less important than in the general Internet, and others can be handled effectively using special mechanisms that are not otherwise typical in the Internet-at-large.

For an example of differing priorities, we note that in carefully managed enterprise networks the fact that new applications face significant barriers is a feature, not a bug. Moreover, current security technologies, such as anomaly detection, scan suppression and signature detection and response engines, already pose a significant barrier for new application deployment. In addition, network scalability is of lesser concern in enterprise networks, since they are typically of a much smaller size (compared to the general Internet); simple replication techniques of the DC will allow SANE to scale to the necessary size systems (Section 5.1 discusses replication in greater detail). Adapting to failures is an example of an issue of real concern in the enterprise, but where different solutions can be employed. The Internet uses highly decentralized routing protocols to adapt to link failures. In SANE, the necessary adaptation can be accomplished through the use of multiple capabilities and hard time-outs (This is discussed in Section 5.2).

4 Architecture Details

In what follows, we describe each of the main components of SANE in detail. We first explain how packets are processed by switches in the network. Next, we explain how switches and end-hosts communicate with the DC. We then have the building blocks for describing how capabilities are created, routed and revoked (when necessary). Finally we explain how naming and access control works.

4.1 Forwarding Packets

All SANE packets have one of three formats:

Type: HELLO	Payload		

Type: DC	Return Capability	Authentication	Payload

Type: FORWARD	Cap-ID	Cap-Expiration	Capability	Payload

When a switch receives a packet, it processes header fields in turn. First, it must decide what type of packet it is. There are three types: (1) `HELLO` packets are for neighbor discovery, and are processed by switches locally and not forwarded; (2) `DC` packets are forwarded along the MST to the DC, They carry control information to the DC including end-host requests for capabilities, authentication messages and topology information generated by switches. and (3) `FORWARD` packets typically carry data between end-hosts and contain a capability which the switch processes to decide how to forward the packet. If the packet is of type `FORWARD`, the switch first checks the packet's integrity then it checks the Capability ID to see if the the capability is valid and hasn't been revoked by the DC. As we'll see later in this section, each switch keeps a list of revoked capabilities, so as to stop misbehaving flows. If the capability is on the list, the switch discards the packet. Next, it checks the `Capability Expiration` field to see if the capability has timed out. The network maintains a coarse network-wide synchronized clock (precise within a few seconds). If the capability has expired, the packet is silently dropped. Finally, it is ready to decrypt and process the capability.

4.2 Communicating with the DC

Bootstrapping: Getting Packets to the DC. Switches need to communicate with the DC to authenticate and send control information; and end-hosts need to communicate with the DC to authenticate and request capabilities. So as to hide the network topology, messages to the DC are sent over a minimum spanning tree (MST),

in which each switch knows only the next hop towards the DC (the root of the tree). HELLO messages are exchanged to build the MST. When a switch joins the network, it exchanges HELLO messages in the same way as the well-known spanning tree protocol used in Ethernet switches [1]. The only difference here is that the DC is always the root of the tree.

Once the MST is setup,

1. Packets can flow from switches and end-hosts to the DC.

2. None of the switches, end-hosts or the DC know the network topology.

3. Switches know the identity of their neighbors.

Bootstrapping: Sending Packets from the DC. In order to construct capabilities (encrypted source-routes), the DC needs to know the network topology. The DC calculates the topology from link-state information: Each switch sends an encrypted neighbor list to the DC. This presents a bootstrapping problem: To send the list, each switch must have established a secret shared key with the DC, yet in order to establish a key, the DC and switch must have communicated. But the DC can't send packets to a switch until it knows the topology. Furthermore, communication along this path lacks attribution— a misbehaving switch or host could send a flood of DC messages and the DC has no mechanism to determine the location of the sender.

We resolve this by having switches directly connected to the DC establish shared keys first, then those one hop away then two hops, and so on, until all switches have authenticated and established keys.

To reliably reveal the location of a sender to the DC, switches construct a "return capability" when they forward DC packets. When it forwards a packet towards the DC, a switch adds a layer containing the previous and next hop, encrypted using the secret shared key; in this manner, the switches automatically create a return capability. This provides the DC with the exact and authenticated location of the sender (topologically) as well as a return source route, and has the same properties as a normal capability in our system.

The DC sends packets back to switches and end-hosts — for example, when it is responding to a capability request — using packets of type FORWARD. The DC may use the "return capability" as the return route. However if it has already constructed the network topology, it may generate an alternate route for the return packet.

Structure of Return Capabilities. For completeness we explain how return capabilities are constructed along the MST path to the DC. As before, we use $E_k(m)$ to denote encryption of message m under the symmetric

key k. And, encryption is implemented using a block cipher (such AES) and a message authentication code (MAC) to provide both confidentiality and integrity.

Each switch on the path to DC receives a partial capability CAP from the previous hop. It adds its own data to the capability as follows:

$$\text{CAP} \leftarrow E_{SwKey} (\text{prev-hop, next-hop, CAP})$$

The switch or host initiating the message to the DC creates an initial capability as

$$\text{CAP}_0 \leftarrow E_{SwKey}(\text{nonce, next-hop})$$

where nonce is used to match the response from the DC to the outgoing message.

Authenticating and Establishing Shared Secret Keys. We do not mandate a particular public key infrastructure for establishing shared keys between the switches and the DC. For example, DCs and switches can have their own local certified public key. During the bootstrap process, each switch can use its public key to establish a shared secret key with the DC, for instance using one of the IKE2 [17] protocols.

Once a switch has a shared secret key, all subsequent packets to the DC are protected using this key. We use a mechanism similar to IPsec's ESP header (which provides confidentiality, integrity, and replay defense) for this purpose. We call this the *authentication header*. This header is required on all packets to the DC and is appended to the capability (between it and the payload).

Disconnecting Misbehaving Senders. Switches and hosts are expected to generate packets to the DC at a rate below a predetermined threshold. If the DC detects a host or switch not abiding by the specified rate limits, it will instruct the upstream switch to disconnect that entity from the network.

The "return capability" provides the DC with a reliable method for determining the location of the attacker and it can issue a request to shut off first-hop links for misbehaving senders. If the attacker is multi-homed, the DC may need to issue a request per port used for flooding. We discuss attacks of this nature in more detail in §6.2.

4.3 Routing with Capabilities

Constructing Capabilities. After a switch authenticates with the DC, it periodically sends its neighbor list, encrypted with the shared symmetric key, to the DC. The DC aggregates these messages to construct the switch-level network topology which is uses to issue capabilities (encrypted source routes). These source routes provide the only connectivity on the network, except for the initial MST.

The DC constructs capabilities using three pieces of information: The location of the requester (contained in the capability request), the location of the requested service (kept from when the service was published), and the path (calculated from the topology).

When the DC constructs a capability, it calculates each layer recursively, starting from the receiver. It encrypts each layer (hop) using the symmetric key of the corresponding switch, as follows:

1. Initialize: The inner most layer is as follows:

$$\text{CAPABILITY} \leftarrow E_{K_{\text{client-port}}} (\text{client-ID},$$
$$\text{client-port, server-port, prev-hop})$$

where client-ID is the client's unique ID, and the client- and server-ports are discussed below.

2. Recurse: For each node on the path, starting from the last node, do:

$$\text{CAPABILITY} \leftarrow E_{K_{\text{SwitchID}}} (\text{Switch ID, next-hop},$$
$$\text{prev-hop, CAPABILITY})$$

3. Finish: The completed capability is then:

$$\text{CAPABILITY} \leftarrow E_{K_{\text{client-ID}}} (\text{client-port, first-hop},$$
$$\text{CAPABILITY}), \quad \text{IV}$$

Note that the MAC computation for each layer includes both the capability ID and its expiration time. As a result these fields cannot be tampered with by the sender or en-route.

The client-ID identifies the client, and is passed by the DC to the server. It could be a well known ID, such as the client's name, or it could be opaque and therefore anonymous.

The client-port number is equivalent to the transport level source port number in TCP and UDP: the client uses it to demultiplex packets from the server. It is also used to identify replies to capability requests from the DC.

Similarly, the server-port number is equivalent to the transport layer destination port: the server uses it to demultiplex packets received from the client. The server-port number is specified by the server and registered with the DC to uniquely identify the service. Because this field is immutable, it is not possible for an end-host to attempt a brute force service enumeration scan (similar to a port scan) unless it has permissions to contact every service.

The initialization vector (IV) provided in the outer layer is the encryption randomization value used for all layers of the onion. It prevents an eavesdropper from linking capabilities between the same two end-points.

Using the same IV for all layers of the capability (as opposed to picking a new random IV for each layer) reduces the overall size of the capability.[3]

Capability Lifetimes. Every capability has a limited lifetime. Different capabilities can have different lifetimes at the discretion of the DC, and could - for example - depend on the service. For ongoing uninterrupted communication, a client needs to periodically request a new capability with a new lifetime.

Every capability carries a 32-bit `Capability Expiration` field in the outer layer. The value is also used in the MAC computation at each layer of the capability, and by each switch along the path. When a switch receives a packet, it checks that it hasn't expired, and then uses the value in its calculation to check the capability.

An alternative approach would be to periodically change keys and force every capability to expire at the same time [31]. We prefer to have the DC choose a lifetime that suits each capability, and carry a single expiration time in each packet; the trade-off is that we need a global timer. The DC and switches therefore need to keep synchronized and secure clocks, but these don't need to be very precise. We assume that the lifetime is typically measured in seconds or minutes, and that urgent revocations are handled as a special case with their own mechanism (see below). So clocks need only be synchronized within a second or so, and the client needs to request a new capability before the current one expires.

Revocation. The DC can revoke a capability to immediately stop flooding or other malicious behavior by hosts or switches (for example, a switch could flood the network with duplicates). If a client or server detects a misbehaving sender, it can ask the DC to revoke the capability and switch off the communication. It does this by sending the DC the received capability, including ID and expiration. The DC could then send a message directly to each switch along the path and tell them to revoke the capability, but this might not work if the switches themselves are misbehaving.

Instead, the DC sends a message to be forwarded by the server or client (whichever identified the malicious traffic); the DC constructs a revocation packet (below) that travels back on the reverse path of the offending capability. Each revocation contains a digital signature signed by the DC using its private key.

Type: REVOKE	expiration	capability ID	SIGNATURE$_{dcpk}$

---

[3]For standard modes of operation (such as CBC and counter-mode) reusing the IV this way has no affect on security, since each layer uses a different symmetric key.

The server or client forwards the revocation command to the switch it received the malicious traffic from. The switch verifies the revocation signature using the DC's public key, caches the revocation and compares it with the capabilityID of each incoming packet. If a match is found, the switch drops the packet and forwards the revocation to the previous hop switch. This hop-by-hop push-back continues until the revocation reaches the sender (either an end-host or a misbehaving switch). Revocations are removed from a switch's cache after they expire (as determined by the *expiration* field).

Mobility. Our network layer is divorced from the physical topology, thus client mobility within the LAN is transparent to servers. When a client changes its position (e.g., moves to a different wireless access point) it refreshes the capability set it holds from the DC and passes new return routes the servers. If a client moves, and another takes its place, the new client may receive unauthorized traffic from outstanding capabilities. This is no different than networks today and content privacy can be maintained through end-to-end cryptography.

Server mobility is not explicitly supported. If a server moves to a different location, it has no mechanism for passing an updated capability to the client (which has no published a service). Instead we require clients to refresh their capability set if they detect packets aren't getting through.

4.4 Naming and Access Control

The basic primitives of authenticated principles and capabilities for services can support a wide range of policy models. We opted for a very simple policy model similar to that of today's file systems.

Our naming scheme keeps directories containing services (much like a hierarchical file system contains directories of files). Directories and services posses ACLs specifying who can lookup, acquire and publish services, as well as who can modify ACLs. Names are delimited by periods to maintain compatibility with existing applications.

To give a concrete example, `martin` has a group of friends that he shares music and videos with at school. He has a directory `hpn.martin.friends` where he can publish service, and `tal`, `greg` and `sundar` can list and acquire services. When his streaming audio server comes on line, it publishes itself in the directory as `ambient-stream`, thus to connect to it, `tal` would direct his streaming audio client to `hpn.martin.friends.ambient-stream`.

Compound principles are also supported (names delimited by colon's). This is particularly useful for dealing with revocation. For example, suppose `tet`, (`tal`'s computer) began acting maliciously by flooding the server with traffic. Access for this account can be revoked using a negative ACL for `tet:tal`. Thus, `tal` will still have access to the service from another computer, `tav`, using `tav:tal`.

4.5 Legacy and Wide-Area Issues

SANE can support and integrate legacy, unmodified operating systems with standard IP stacks. Supporting them - and providing connectivity to the wide area - requires two additional network components.

1. Proxies are directly on route between a host and the DC, and translate IP naming events to corresponding SANE events. They map DNS traffic to SANE name queries, translate the responses and manage the addition, deletion and caching of capabilities. Proxies encapsulate IP packets generated by the end-host with SANE headers and decapsulate received packets.

2. Gateways are positioned on the perimeter of a SANE network and provide connectivity to the wide area. For outgoing packets they cache the capability and generate a mapping from the IP packet header (e.g., IP/port 4-tuple) to the associated capability. All incoming packets are checked against this mapping and, if one exists, the appropriate capability is appended and the packet is forwarded.

Publishing a Service. In today's networks offering a service consists of binding to a port on the end-host and often changing configuration in the network such as modifying firewall rules or enabling port-forwarding in NAT.

SANE services can be with the DC in any number of ways – via a command line tool, offering a web interface from the proxy, or hooking into the bind call on the local host ala SOCKS [19].

Capability Refresh on Timeout. It is the responsibility of proxies to refresh capabilities during timeouts and link failures. The latter is more difficult as it requires transport-level semantics to determine if sent packets are going unacknowledged. While it is reasonable to assume this can be done with well known protocols such as TCP, it isn't general, nor desirable to embed per-protocol logic. Instead, we suggest that the proxies themselves issue periodic, acknowledged requests to independently test whether a capability is able to traverse the network. Link-state probes must not be recognizable by an attacker en-route or she can selectively let them pass while continuing to drop data. To accomplish this we use a reserved service ID (only visible to the receiver) to mark probe capabilities.

Non-DNS Traffic. In today's network, it is not necessary for an IP flow to first issue a DNS request. Raw IP addresses can be used in place of hostnames.

When local resolution is used (such as that provided by `/etc/hosts`) the host doesn't issue a DNS request and therefore the lookup isn't exposed to the proxy. It may therefore require some local configuration to ensure that initial connections are preceded by a DNS request. One solution, used in our implementation, is to register services as the text representation of the IP address and port (e.g. "192.168.1.23:80") and have the proxies issue requests for traffic that it does not have a cached capability for.

5 Scalability and Fault Tolerance

SANE is different from the traditional Internet model in two significant ways: It relies on a central entity (DC) for all connectivity, and there is no explicit support for adaptive routing in the network. These design choices have implications with respect to scalability and fault tolerance. As the network grows, the DC's bandwidth and computational power become limiting resources; and failures in a large network might overwhelm the DC.

In this section we discuss how to replicate the DC, so as to reduce its load and make it fault-tolerant. We reiterate that a single SANE domain is not designed to scale to Internet size networks. Rather, we briefly discuss how larger networks can use a hierarchy of multiple domains. We then describe how connectivity is maintained in the wake of network faults, and how to dampen the size of the response through switch level redundancy and issuing multiple or randomized paths.

5.1 Replicating the Domain Controller

The DC is centralized in the sense that routing and access control decisions are not executed in a distributed manner. But this is only logical centralization — it does not preclude clustering (physical decentralization) or replicating the domain controller (topological decentralization) as a means of providing greater scalability and fault tolerance.

The DC can be replicated at a single point in the topology via clustering. Capabilities can easily be generated independently and in parallel, so long as topology and access control updates are atomic.

Replicating the DC at different points in the topology is possible too. Basic connectivity to multiple DCs can be provided by multiple MSTs, one rooted at each DC. Switches must authenticate and send their neighbor lists to each DC separately. Consistency is not required as each DC grants routes independent of other DCs. Principles randomly choose a DC to send requests to in order to distribute load. If DCs all belong to the same enterprise - and hence trust each other - service advertisements and access control policy can be replicated between DCs using existing methods for ensuring distributed consistency in a name service. (We will consider the case where DCs can't trust each other in the next section.)

SANE can be extended to support multiple domains. One approach we are investigating is to allow domains to operate in a hierarchy, similar to DNS. One domain is selected as the root of the hierarchy and arbitrates communication between the children domains, much like a DC does for end-hosts. There is a great deal of additional detail required to give multiple domains an adequate treatment, thus we consider it beyond the scope of this paper and a topic for future work.

5.2 Recovering from Network Failure

In SANE it is the end-host's responsibility to determine network failure.[4] End-hosts will typically detect network failure because packets get lost and traffic goes unacknowledged. Alternatively, SANE-aware hosts can send periodic probes or keep-alive messages using the capability (We discussed legacy hosts in §4.5). Either way, as soon as an end-host suspects a path has failed, it can request a new capability. The DC receives regular link-state information, and so can provide a new capability over a new path (if there is one).

An alternative approach (that we are exploring) is that when a switch receives a capability to forward a packet to a failed link, it passes the packet to the DC instead, so it can push a new capability to the sender.

If a link fails, a lot of capabilities might need replacing, which could swamp the DC with requests. This could be solved by: (1) Providing greater link connectivity (and therefore reducing the number of capabilities per link), and/or (2) Issuing multiple capabilities per-request. We consider each method in turn.

Providing greater link connectivity is easy with SANE: A good property of SANE is that additional switches and links can be added without compromising existing protection policy. Redundancy in network paths reduces the average numbers of flow traversing a given path.

Issuing multiple capabilities means two end-hosts can have available several ways to reach each other – ideally over disjoint paths. If a capability fails, it simply switches to another capability without contacting the DC. Even two or three alternate routes can provide a high degree of fault tolerance [16]. In addition, to provide greater path diversity, the DC can randomly select n of the k shortest paths to send to the client. Another advantage of handing out multiple capabilities is that the

---

[4]This is because direct communication from switches to end-hosts violates least privilege and creates new avenues for DoS.

end-host can set aggressive time-outs to detect link failures. This offers quick fail-over, and will only result in minor packet re-ordering at the receiver, in the worst case. Modern TCP stacks can gracefully handle such modest amounts of reordering [33].

6   Attack Resistance

A SANE network should degrade gracefully in face of attacks from malicious hosts, switches or DCs. In what follows, we present potential attacks on SANE and (optional) architectural extensions that mitigate them.

6.1   Revocation State Exhaustion

As discussed in Section 4.3, to support revocation SANE switches must maintain revocation lists. Such lists are typically stored using a fast content addressable memory (CAM), which is a scarce resource. Thus an attacker can exhaust it by hoarding capabilities and then abusing them to induce revocation.

To allow a SANE network to gracefully recover from such revocation state exhaustion attacks, we use the following approach:[5] Each switch contains a small CAM (e.g. 100 entries) to store the IDs of revoked capabilities. If an attacker exhausts this state, the switch simply generates a new key invalidating all pre-existing capabilities that pass through it, clears its revocation list, and passes its new key to the DC. In addition, the DC (or the server) tracks the number of revocations issued per sender. When this number crosses a predefined threshold, the sender is removed from the service's access control list or, potentially, from all access control lists.

Using this approach a SANE network can quickly converge to a state where attackers hold no valid capabilities and cannot obtain new ones. However, many well behaved senders may have to refresh their capabilities, temporarily degrading performance.

A threshold allows senders to have a "second chance" to prove they are not being framed by a misbehaving switches on route: If a switch uses a sender's capability to flood a receiver (eliciting a revocation) the sender can use a different capability to the a misbehaving switch. Since the sender gets several chances, no punitive action is taken.

6.2   Tolerating Malicious Switches

A malicious switch can attempt to sabotage the network by selectively dropping packets, advertising false topol-

---

⁵Another approach would be to use resource limits (e.g. per principle capability limits) to bound the damage a given malicous principle can inflict. However, if many nodes are malicous, this approach is insufficient.
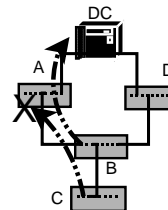

Figure 3: Attacker **A** can deny service to **C** by selectively dropping its packets and letting its parent's (**B**) through. As a result **C** cannot communicate with the DC even though a alternate path exists through D.

ogy information, or manipulating traffic traversing it. We consider such attacks next.

Sabotaging MST Discovery. To attack the MST construction process, a switch can advertise a false distance to attract traffic destined for the DC to itself. There are two attacks to consider here:

A malicious switch can falsely advertise its position as being closer to the DC than it really is. This can nominally create a path inefficiency or more seriously cause a momentary denial of service by dropping all through traffic. In the latter case, the switches adjacent to the malicious switch will treat this as a link failure and pick a different neighbor, thus routing around the problematic switch.

A Malicious switch may inflict a more subtle attack by selectively allowing packets from its neighbors and dropping all other traffic. An example of this attack is depicted in Figure 3: node A only drops patckets from node C. Since node A appears to be functioning normally to node B, it does not change its forwarding path to the DC. Therefore, C cannot communicate with the DC even though an alternate patch exists through D.

We combat this by modifying the header in packets to the DC to mask the identity of the sender from switches en-route. Normally, the header on packets destined for the DC (i.e., the IPSec ESP header) contains a consistent originating NodeID in cleartext, so that the DC knows which key to use to authenticate and decrypt the payload. We replace this static NodeID with an ephemeral nonce provided by the DC. Every response from the DC will contain a nonce to use as the NodeID in the next message. To further obscure the identity of the sender, all messages are padded and the timing associated with periodic messages to the DC should be randomized.

With this approach, if a switch drops all packets it will inevitably drop packets belonging to its nearest neighbors as well, causing traffic to be diverted away from it. If a switch drops random packets, connectivity is only temporarily degraded, but not lost.

Bad Link State Advertisement. Malicious switches can try to attract traffic by falsifying connectivity information in link state updates. A simple approach to safeguard against such attacks is for the DC to only add edges to its network map when the switches at either end have both advertised it.

This safeguard does not prevent colluding nodes from falsely advertising a link between themselves. Unfortunately such collusion cannot be externally verified. Notice that such collusion can only result in a temporary denial of service attack when capabilities containing a false link are issued (When end-hosts are unable to route over a false link, they immediately request a fresh capability). However, the isolation properties of the network are still preserved.

Malicious Data Injection. SANE packet payloads are not bound to headers by an integrity check. Therefore, a malicious switch en-route to the destination can silently modify the payload. End-to-end secrecy and integrity checks, while useful, offer little protection for components that examine the payload before (e.g. middleboxes) or during the integrity check (e.g. OpenSSL). Thus, they still leave open an attack vector to an on-path attacker . Exploits in end-host data-integrity mechanisms (IPsec, SSL), are not unheard of [3] and vulnerabilities in complex middleboxes (SpamFilters, VirusScanners, Firewalls, IDS) are a relatively common occurrence [4, 6, 2]. If a malicious switch modifies existing payload to include an exploit, the attacker is protected from discovery because the capability implicates the legitimate sender.

To address this threat, we incorporate an optional integrity checksum into packet headers. These checksums are appended to the capability by the sender and checked by designated switches along the path, as well as the recipient. Apart from preventing malicious switches from hijacking connections, this mechanism provides accountability: an administrator can now inspect traffic logs to identify the exact attack origin and propagation path. To implement this features, we can piggyback on the existing shared keys between the DC and the switches:

- Before a DC returns a capability to an end-host, it designates specific switches along the path to check data integrity. Suppose switches ID_1 and ID_2 are chosen.

  When constructing the capability, the DC adds a field in the onion layers of switches ID_1 and ID_2 indicating that they should validate the data integrity checksum. We let k_1, k_2 denote the shared key between the DC and ID_1, ID_2 respectively. The DC derives MAC keys from its shared keys with these switches by computing: $k_{mac,i} \leftarrow$ $hmac_{k_i}(\text{CAP-ID}_i)$ for $i = 1, 2$. The DC sends $k_{mac,1}$ and $k_{mac,2}$ to the sender along with the capability.

- For each packet, the sender computes the two tags $t_i \leftarrow HMAC_{k_{mac,i}}(m)$ for $i = 1, 2$ where m is the packet payload. It appends the the tags t_i to the capability.[6]

- Switch ID_1 derives the MAC key $k_{mac,1}$ from the key k_1 and the CAP-ID. When processing a packet which contains an indication that ID_1 should validate the checksum, it (optionally) validates the MAC t_i and drops the packet if it fails to verify. If successful, t_i is removed from the packet before forwarding.

6.3 Tolerating a Malicious DC

DCs are highly trusted entities in a SANE network. This can create a single point of failure from a security standpoint since the compromise of any one DC yields total control to an attacker.

To prevent such a take-over, we distribute trust among DC's using threshold cryptography, as follows (The full details are beyond the scope of this paper. We only sketch the basics):

We split the DC's secret across a few server (say $n < 6$) such that two of them are needed to generate a capability. The sender then communicates with two out of the n DC's to obtain the capability. Thus, an attacker gains no additional access by compromising a single DC.[7] Access control policy and service registration must be done independently with each DC by the end-host using standard approaches for consistency such as two-phase commit. When a new DC comes online, or when a DC re-establishes communication after a network partition, it must have some means of re-syncing with the other DCs. This can be achieved via standard byzantine agreement protocols [11].

Even with this approach, a DC must be restricted to shut off ports only the MST that it is a root of; Otherwise, rogue DCs can shut off all communication. Similarly, if we allow DC to issue revocations, a rouge DC can shut down the entire network. To deal with this, the revocation mechanism can be extended using asymmetric threshold cryptography [13]. We omit the details from this paper.

---

[6] Note that hashing m with a collision resistant hash prior to applying HMAC makes it possible to compute both tags at about the same time that it takes to compute a single tag.

[7] Implementing threshold cryptography for symmetric encryption is done combinatorially [9]—start from a t-out-of-t sharing (namely, encrypt a DC master secret under all DC server keys) and then construct a t-out-of-n sharing from it.

7   Implementation

We implemented a SANE network consisting of switches, IP proxies, a gateway to the Internet and a DC within our group LAN. We describe our experiences developing and using SANE within an operational network environment.

Development and Test Environment. All development was done in C++ within an in-house user-space networking environment based on the Virtual Network System (VNS) [10]. Working outside the kernel provided us with a flexible development, debug and execution environment. It also simplified running and testing over multiple varied and complex network topologies.

All network components (switches, proxies, the gateway and DC) are implemented as user-space processes that have the ability to directly process raw Ethernet frames and communicate with physical hosts. VNS provides that ability to run the processes within user-specified topologies as well as to interface with physical machines. We used point-to-point Ethernet to provide connectivity between network components.

Development was done on Fedora Linux, kernel 2.6. We tested the implementation by running the network components on Linux workstations interconnecting 7 physical hosts used daily as office workstations. All hosts were interconnected via 100Mb Ethernet.

Switches. Our switch implementation supports all core functionality including neighbor discovery, MST construction, and link-state forwarding. We use OCB-AES [25] for capability construction and decryption. Switches were preconfigured with the DC's public key, and they generated local IDs randomly.

We configured the switches to send HELLO packets on startup, during detection of a link change and on 15 second intervals. Link state packets were sent subsequent to authentication with the DC, on detection of a link change and on 30 second intervals.

The only dynamic state maintained on each switch was a hash table of capability revocations which contained a capability ID and the associated expiration time.

Domain Controller. In our implementation the DC consists of four separate modules, topology construction, name hosting and resolution, capability construction, and, authentication and registration. For authentication purposes, the DC was preconfigured with the public keys of all switches.

For end-to-end path calculations, we used a bidirectional search from both the source and destination. All computed routes were cached to speed up computation for capability requests between the same end-hosts. All cached routes were checked against the current topology for correctness before re-use.

Services are stored in a map that matched text strings to the switch and port the service is connected to. A separate map is used to store the access control list for each service.

For capabilities we use 8bit port identifiers and 64bit MACs. Switch and principal IDs are 32bits and service IDs are 16bits. The first layer of the capability requires 24bytes while each additional layer uses 14bytes. The longest path on our test topologies was 10 switches resulting in a 164byte header. Capabilities are given a 15 minute lifetime.

The registration component serves an extensible interface for new switches and users to register with the network. Registration is the setup required prior to joining the network so a switch or principle has the appropriate information needed to authorize with the DC.

End Hosts. We were able to integrate end-hosts running both Windows and Linux into the network with little modification. The only configuration necessary was to reduce the MTU size of the host interfaces to provide space for SANE headers. Reducing the MTU to 1300bytes was sufficient.

We developed a command line utility to allow the publication of services and setting of access controls from an end-host.

Testing. Our goal in implementing SANE was to understand the implications of running SANE within a real network. We ran our implementation within our local network for two weeks, supporting local connectivity of 7 workstations running NSF, HTTP, IMAP and SMTP. In additional we ran a SANE gateway that handles all traffic to the WAN. We tested over multiple switch-level topologies containing 4 to 20 switches.

For encryption, our implementation used a an 128 bit AES key. Given pre-computed routes, on a commodity 2.3 GHz PC, our DC was able to generate 10-hop capabilities at a rate of 20,000 per second.

Our implementation was not optimized for performance. In practice SANE switches would be implemented in hardware and the DC, proxies and gateway would run on dedicated machines. However, a quick back-of-the-envelope estimate of load on the DC suggest that even our modest implementation could handle a moderate-sized enterprise with a single DC:

Assuming that a DC is responsible for 10000 end-nodes (in a moderate-sized enterprise), and that each end-node initiates 10 new transfers per minute with other unique end-hosts, the DC receives about 1667 capability requests per second. If we assume an average transmission lifetime of 2 minutes, and a capability timeout interval of 1 minute (requiring one refresh per transmission lifetime, on average), the request rate at the DC doubles to 3334 per second. This is one third of our

simple implementation's capacity for generating capabilities.

We now estimate the bandwidth requirements to support the above network. Assuming a 10-hop network, responses to capability requests (which will include a capability for client-to-server communication and one for server-to-client) are at most 0.4KB in size, assuming 10-hop network paths. This results in 13Mbps of traffic on the network for issuing (or reissuing) capabilities.

As part of our future work, we plan to continue exploring the performance implications of running SANE in large networks with attack-level traffic conditions.

8   Related Work

SANE shares much in common with predicate routing [26], both proposals argue for unified policy specification and enforcement. Predicate routing unifies security and routing by defining connectivity as a set of declarative statements from which routing tables and filters are generated. Our work is distinct in that users (as opposed to end-point IDs or IP addresses in Predicate routing) are first class objects and can be used in defining access controls. Secondly we do not require out-of-band control elements, but rather provide default connectivity to the DC. Finally, predicate routing was not designed to support *least*-knowledge. Routers maintain full topology information and end-hosts are easily identifiable from network flows.

Weaver et. al [28] argue that existing configurations of course-grained network perimeters (e.g. NIDS, multiple firewalls) and end-host protective mechanisms (e.g. anti-virus software) are ineffective against worms, both when employed individually, or in combination. Further, they advocate augmenting traditional coarse grain perimeters with fine grain protection mechanisms throughout the network.

Much recent work has focused on DoS remediation through network enforced capabilities [8, 31, 32] on the WAN. These systems assumes no cooperation between network elements nor do they have a notion of centralized control. Instead, clients receive capabilities from servers directly (and servers from clients), Capabilities are constructed on route by the initial capability requests. This offers a very different policy model than SANE as it is designed to meet different needs (limiting wide area DoS) and relies on different assumptions (no common administrative domain).

Raghavan et. al in Platypus [23] uses authenticated source routes to enforce ISP policy compliance and identify the billable network entities. The goal of Platypus is use tracking and not attack resistance: unlike our scheme in which issued routes are opaque and strictly

immutable, they use visible, loose source routes which are composable. This approach is unsuitable for our purposes. For example, a sender capable of collecting capabilities would be able to construct arbitrary routes (and thus connectivity) within the network.

Other proposals have suggested replacing link-layer distributed spanning tree based routing with link state routing [22, 21] to achieve better scalability, stability and fault tolerance.

Myers et. al. [21] facilitate this by changing the Ethernet model to provide explicit host registration and discovery based on a directory service, instead of the traditional broadcast discovery service (ARP) and implicit MAC address learning. This provides better scalability and transparent link-layer mobility and eliminates the inefficiencies of broadcast Similarly, we eliminate broadcast in favor of tighter traffic control through link state updates. However, we eschew the use of persistent end-host identifiers, instead associating each routable destination i.e., services with the switch port where it registered from.

In their 4D architecture Rexford et. al. [24, 15] argue that the decentralized routing policy, access control and management has resulted in complex routers and cumbersome, difficult to manage networks. Similar to SANE, they argue that routing (the control plane) should be separated from forwarding, resulting a very simple forwarding path.

Although 4D centralizes routing policy decisions, they retain the security model of today's networks, where routing (forwarding tables) and access controls (filtering rules) are still decoupled, disseminated to forwarding elements and executed on the forwarding path, on the basis of weakly-bound end-point identifiers (IP address). In our work, there is no need to disseminate forwarding tables or filters as forwarding decisions are made *a priori* and encoded in source routes.

9   Conclusion

We believe that enterprise networks are different from the Internet at large, and deserve special attention: Security is paramount; centralized control is the norm, and uniform, consistent policies are important.

Providing strong protection is difficult, and always requires some tradeoffs: On one hand there are clear advantages to having an open environment where connectivity is unconstrained and every end-host can talk to every other. But just as clearly, such openness is prone to attack by malicious users from inside or outside the network.

We set out to understand what it takes to design a network so secure that it is almost unthinkable for an

end-host or switch to mount an effective attack. Drastic goals call for drastic measures, and we understand that our proposal - SANE - is an extreme approach. SANE is conservative in the sense that it enforces a philosophy of *Least Privilege* and *Least Knowledge* to all parties, except a trusted, central Domain Controller. We believe that this is typical and acceptable practice in enterprises, where central control, and restricted access to information is common.

We suspect that our approach would share much in common with any approach that has the same goals. For example, if we want switches to forward packets, but we want to hide from them the identity of both end-hosts, the service, and the path packets take, then it requires significant effort and mechanism. Our approach - encrypted source-routes - has the interesting property of hiding all unnecessary information, without requiring state in the network, and we believe it is both practical and scalable.

References

[1] 802.1d mac bridges. http://www.ieee802.org/1/pages/802.1D-2003.html.

[2] Cisco security advisory: Cisco ios remote router crash. http://www.cisco.com/warp/public/770/ioslogin-pub.shtml, August 1998.

[3] Cert advisory ca-2002-23 multiple vulnerabilities in openssl. http://www.cert.org/advisories/CA-2002-23.html, July 2002.

[4] Cert advisory ca-2003-13 multiple vulnerabilities in snort pre-processors. http://www.cert.org/advisories/CA-2003-13.html, April 2003.

[5] Sasser worms continue to threaten corporate productivity. http://www.esecurityplanet.com/alerts/article.php/3349321, May 2004.

[6] Technical cyber security alert ta04-036aarchive http parsing vulnerabilities in check point firewall-1. http://www.us-cert.gov/cas/techalerts/TA04-036A.html, February 2004.

[7] Icmp attacks against tcp vulnerability exploit. http://www.securiteam.com/exploits/5SP0N0AFFU.html, April 2005.

[8] T. Anderson, T. Roscoe, and D. Wetherall. Preventing internet denial-of-service with capabilities. *SIGCOMM Comput. Commun. Rev.*, 34(1):39–44, 2004.

[9] E. Brickell, G. D. Crescenzo, and Y. Frankel. Sharing block ciphers. In *Proceedings of Information Security and Privacy*, volume 1841 of *LNCS*, pages 457–470. Springer-Verlag, 2000.

[10] M. Casado and N. McKeown. The virtual network system. In *Proceedings of the ACM SIGCSE Conference*, 2005.

[11] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.

[12] D. Cullen. Half life 2 leak means no launch for christmas. http://www.theregister.co.uk/2003/10/07/half_life_2_leak_means/, October 2003.

[13] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In *Advances in Cryptology - Crypto '89*, 1990.

[14] D. M. Goldschlag, M. G. Reed, and P. F. Syverson. Hiding Routing Information. In R. Anderson, editor, *Proceedings of Information Hiding: First International Workshop*, pages 137–150. Springer-Verlag, LNCS 1174, May 1996.

[15] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A clean slate 4d approach to network control and management. In *In ACM SIGCOMM Computer Communication Review*, October 2005.

[16] G. C. S. Jian Pu, Eric Manning. Routing reliability analysis of partially disjoint paths. In *IEEE Pacifi c Rim Conference on Communications, Computers and Signal processing (PACRIM' 01)*, volume 1, pages 79–82, 2001.

[17] C. Kaufman. Internet key exchange (ikev2) protocol. draft-ietf-ipsec-ikev2-10.txt (Work in Progress).

[18] A. Kumar, V. Paxson, and N. Weaver. Exploiting underlying structure for detailed reconstruction of an internet-scale event. In *to appear in Proc. ACM IMC*, October 2005.

[19] M. Leech, M. Ganis, Y. Lee, R. Kuris, D. Koblas, and L. Jones. Socks protocol version 5. RFC 1928, 1996.

[20] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver. Inside the slammer worm. *IEEE Security and Privacy*, 1(4):33–39, 2003.

[21] A. Myers, E. Ng, and H. Zhang. Rethinking the service model: Scaling ethernet to a million nodes. In *ACM SIGCOMM HotNets*, November 2004.

[22] R. J. Perlman. Rbridges: Transparent routing. In *INFOCOM*, 2004.

[23] B. Raghavan and A. C. Snoeren. A system for authenticated policy-compliant routing. In *Proc. ACM SIGCOMM '04*, 2004.

[24] J. Rexford, A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Myers, G. Xie, J. Zhan, and H. Zhang. Network-wide decision making: Toward a wafer-thin control plane. In *Proceedings of HotNets III*, November 2004.

[25] P. Rogaway, M. Bellare, J. Black, and T. Krovetz. OCB: A block-cipher mode of operation for efficient authenticated encryption. In *ACM Conference on Computer and Communications Security*, pages 196–205, 2001.

[26] T. Roscoe, S. Hand, R. Isaacs, R. Mortier, and P. Jardetzky. Predicate routing: Enabling controlled networking. *SIGCOMM Comput. Commun. Rev.*, 33(1):65–70, 2003.

[27] J. Saltzer and M. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9), 1975.

[28] N. Weaver, D. Ellis, S. Staniford, and V. Paxson. Worms vs. perimeters: The case for hard-lans. In *Proc. Hot Interconnects 12*, August 2004.

[29] G. Xie, J. Zhan, D. Maltz, H. Zhang, A. Greenberg, G. Hjalmtysson, and J. Rexford. On static reachability analysis of ip networks. In *IEEE INFOCOM 2005*, March 2005.

[30] G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. Greenberg, and G. Hjalmtysson. Routing design in operational networks: A look from the inside. In *Proc. ACM SIGCOMM '04*, pages 27–40, New York, NY, USA, 2004. ACM Press.

[31] A. Yaar, A. Perrig, and D. Song. Siff: A stateless internet flow filter to mitigate ddos flooding attacks. In *In Proceedings of the IEEE Security and Privacy Symposium*, May 2004.

[32] X. Yang, D. Wetherall, and T. Anderson. A dos-limiting network architecture. In *Proc. ACM SIGCOMM '05*, pages 241–252, New York, NY, USA, 2005. ACM Press.

[33] M. Zhang, B. Karp, S. Floyd, and L. Peterson. Rr-tcp: A reordering-robust tcp with dsack. In *Proc. of IEEE International Conference on Networking Protocols (ICNP 2003)*, 2003.