SANE: A Protection Architecture for Enterprise Networks

Martin Casado, Tal Garfinkel, Aditya Akella, Michael J. Freedman
Dan Boneh, Nick McKeown, Scott Shenker
{*casado,talg,mfreed,dabo,nickm*}*@cs.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 network architecture. This leads to enterprise networks that are inflexible, fragile, and difficult to manage.

To address these limitations, we offer SANE, a protection architecture for enterprise networks. SANE defines a single *protection layer* that governs all connectivity within the enterprise. All routing and access control decisions are made by a logically-centralized server that grants access to services by handing out capabilities (encrypted source routes) according to declarative access control policies (e.g., "Alice can access http server *foo*"). Capabilities are enforced at each switch, which are simple and only minimally trusted. SANE offers strong attack resistance and containment in the face of compromise, yet is practical for everyday use. Our prototype implementation shows that SANE could be deployed in current networks with only a few modifications, and it can easily scale to networks of tens of thousands of nodes.

1 Introduction

The Internet architecture was born in a far more innocent era, when there was little need to consider how to defend against malicious attacks. Moreover, many of the Internet's primary design goals, such as universal connectivity and decentralized control, which were so critical to its success, are at odds with making it secure.

Worms, malware, and sophisticated attackers mean that security can no longer be ignored. This is particularly true for enterprise networks, where it is unacceptable to lose data, expose private information, or lose system availability. And so security measures have been retrofitted to enterprise networks via many mechanisms, including router ACLs, firewalls, NATs, and other middleboxes, along with complex link-layer technologies such as VLANs.

Despite years of experience and experimentation, these mechanisms are far from ideal. They require a significant amount of configuration and oversight [43], are often limited in the range of policies they can enforce [45], and produce networks that are complex [49] and brittle [50]. Moreover, even with these techniques, security within the enterprise remains notoriously poor. Worms routinely cause significant losses in productivity [9] and potential for data loss [29, 34]. Attacks resulting in theft of intellectual property and other sensitive information are similarly common [19].

The long and largely unsuccessful struggle to protect enterprise networks convinced us to start over with a clean slate, with security as a fundamental design goal. The result is our *Secure Architecture for the Networked Enterprise (SANE)*. The central design goals for our architecture are as follows:

- *Allow natural policies that are simple yet powerful.* We seek an architecture that supports natural policies that are independent of the topology and the equipment used, e.g., "Allow everyone in group sales to connect to the http server hosting documentation." This is in contrast to policies today that are typically expressed in terms of topology-dependent ACLs in firewalls. Through high-level policies, our goal is to provide access control that is restrictive (i.e., provides least privilege access to resources), yet flexible, so the network does not become unusable.

- *Enforcement should be at the link layer, to prevent lower layers from undermining it.* In contrast, it is common in today's networks for network-layer access controls (e.g., ACLs in firewalls) to be undermined by more permissive connectivity at the link layer (e.g., Ethernet and VLANs).

- *Hide information about topology and services from those without permission to see them.* Once an attacker has compromised an end host, the usual next step is to map out the network's topology—to identify firewalls, critical servers, and the location of end hosts—and to identify end hosts and services that can be compromised. Our goal is to hide all such information to embrace the principle of least knowledge.

- *Have only one trusted component.* Today's networks trust multiple components, such as firewalls, switches, routers, DNS, and authentication services (e.g., Kerberos, AD, and Radius). The compromise of any one component can wreak havoc on the entire enterprise. Our goal is to rely on a central (yet potentially replicated) trusted entity where all policy is centrally defined and executed.

SANE achieves these goals by providing a single protection layer that resides between the Ethernet and IP layer, similar to the place that VLANs occupy. All connectivity is granted by handing out *capabilities*. A capability is an encrypted source route between any two communicating end points.

Source routes are constructed by a logically-centralized Domain Controller (DC) with a complete view of the network topology. By granting access using a global vantage point, the DC can implement policies in a topology-independent manner. This is in contrast to today's networks: the rules in firewalls and other middleboxes have implicit dependencies on topology, which become more complex as the network and policies grow (e.g. VLAN tagging and firewall rules) [14, 47].

By default, hosts can only route to the DC. Users must first authenticate themselves with the DC before they can request a capability to access services and end hosts. Access control policies are specified in terms of services and principals, e.g., "users in group martins-friends can access martin's streaming-audio server".

At first glance, our approach may seem draconian: All communication requires the permission of a central administrator. In practice, the administrator is free to implement a wide variety of policies that vary from strict to relaxed and differ among users and services. The key here is that SANE allows the easy implementation and enforcement of a simply expressed rule.

Our approach might also seem dependent on a single point-of-failure (the DC) and not able to route traffic around failures (because of static source routes). However, as we will argue, we can use standard replication techniques, such as multiple DCs and redundant source routes, to make the network reliable and quick to recover from failures.

The remainder of the paper is organized as follows. In Section 2, we further argue why current security mechanisms for the enterprise are insufficient and why the SANE approach is feasible. Section 3 presents a detailed design of SANE. We will see that by delegating access control and routing to a central controller, we can reduce the complexity of the forwarding elements (switches) and the degree to which we must trust them. We also show how a specific implementation of SANE could be deployed in current networks with only a few modifications (even though SANE is a radical departure from traditional network design). Section 4 covers SANE's resistance to a strong attack model. In Section 5, we present and evaluate a prototype software implementation of SANE, and Section 6 demonstrates that SANE can easily scale to networks of tens of thousands of nodes and does not significantly impact user-perceived latency. We present related work in Section 7 and conclude in Section 8.

2 What's Wrong with Existing Techniques?

Complexity of Mechanism. A typical enterprise network today uses several mechanisms simultaneously to protect its network: VLANs, ACLs, firewalls, NATs, and so on. The security policy is distributed among the boxes that implement these mechanisms, making it difficult to correctly implement an enterprise-wide security policy. Configuration is complex (for example, routing protocols often require thousands of lines of policy configuration [50]), making the security fragile. Furthermore, the configuration is often dependent on network topology, and is based on addresses and physical ports, rather than on authenticated end-points. When the topology changes or hosts move, the configuration frequently breaks, requires careful repair [50], and possibly undermines its security policies.

A common response is to put all security policy in one box and at a choke-point in the network, for example, in a firewall at the network's entry and exit point. If an attacker makes it through the firewall, they have unfettered access to the whole network.

Another way to address this complexity is to enforce protection on the end host via distributed firewalls [14]. While reasonable, this has the down-side of placing all trust in the end hosts. End host firewalls can be disabled or bypassed, leaving the network unprotected, and they offer no containment of malicious infrastructure, e.g., a compromised NIDS [8].

Our new architecture allows simple high-level policies to be expressed centrally. Policies are enforced by a single fine-grain mechanism within the network.

Proliferation of Trust. Today's networks provide a fertile environment for the skilled attacker. Switches and routers must correctly export link state, calculate routes, and perform filtering; yet over time, these mechanisms have become more complex, with new vulnerabilities discovered at an alarming rate [8, 10, 7, 11]. If compromised, an attacker can often take down the network [32, 48] or redirect traffic to permit eavesdropping, traffic analysis, and man-in-the-middle attacks.

Our new architecture replaces all these mechanisms with simple, minimally-trusted forwarding elements, reducing the number of trusted (and configured) components to just one centrally-managed controller. Our goal is to minimize the trusted computing base.

Proliferation of Information. A further resource for an attacker is the proliferation of information on the network layout of today's enterprises. This knowledge is valuable for helping to identify sensitive servers, firewalls, and IDS systems, which can be exploited for compromise or denial of service. Topology information is easy to gather: switches and routers keep track of the network topology (e.g., the OSPF topology database) and broadcast it periodically in plain-text. Likewise, host enumeration (e.g., ping and ARP scans), port scanning, traceroutes, and SNMP can easily reveal the existence of, and the route to, hosts. Today it is common for network operators to filter ICMP and change default SNMP passphrases to limit the amount of information available to an intruder.

Our new architecture hides both the network structure, as well as the location of critical services and hosts, from all unauthorized network entities. Minimal information is made available as needed for correct function and for fault diagnosis.

2.1 Threat Environment

SANE seeks to provide protection robust enough for demanding threat environments—government and military networks, financial institutions, or demanding business settings—yet flexible enough for everyday use. We assume a robust threat environment with both *insider* (authenticated users or switches) and *outsider* threats (e.g., an unauthenticated attacker plugging into a network jack). This attacker may be sophisticated, capable of compromising infrastructure components and exploiting protocol weaknesses. Consequently, we assume attacks can originate from any network element, such as end hosts, switches, or firewalls.

SANE prevents outsiders from originating any traffic except to the DC, while preventing malicious end hosts from either sending traffic anywhere that has not been explicitly authorized, or, if authorized, subjecting the network to a denial-of-service attack which cannot be subsequently disabled.

SANE makes a best effort attempt to maintain availability in the face of malicious switches; however, we do not attempt to achieve full network-layer Byzantine fault tolerance [38]. In a normal SANE network, little can be done in the face of a malicious DC, however, we discuss strategies for dealing with this and other threats in §4.

2.2 What's Special about the Enterprise?

We can exploit several properties of enterprise networks to make them more secure. First, enterprise networks are often carefully engineered and centrally administered, making it practical (and 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 servers, printers, file servers, source repositories, HTTP proxies, or ssh gateways). Therefore, we can grant relatively little privilege to clients using simple declarative access control policies; in our system we adopt a policy interface similar to that of a distributed file system.

Third, in an enterprise network, we can assume that hosts and principals are authenticated; this is already common today, given 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 security technologies due to the high cost of security failures.

3 System Architecture

SANE ensures that network security policies are enforced during all end host communication at the link layer, as shown in Figure 1. This section describes two versions of the SANE architecture. First, we present a clean-slate approach, in which every network component is modified to support SANE. Later, we describe a version of SANE that can inter-operate with unmodified end hosts running standard IP stacks.

3.1 Domain Controller

The Domain Controller (DC) is the central component of a SANE network. It is responsible for authenticating

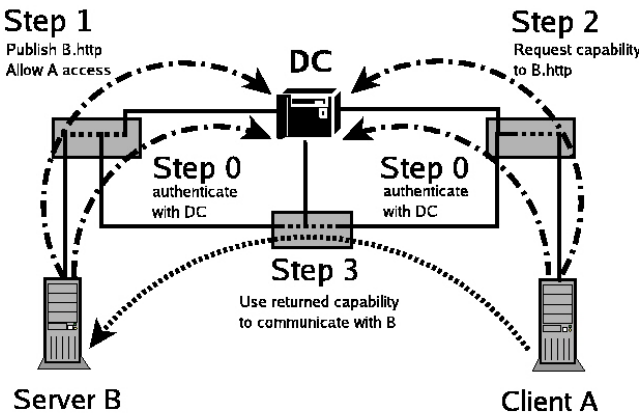

Figure 1: The SANE Service Model: By default, SANE only allows hosts to communicate with the Domain Controller (DC). To obtain further connectivity they must take the following steps: (0) Principals authenticate to the DC and establish a secure channel for future communication. (1) Server *B publishes* service under a unique name B.http in the Network Service Directory. (2) For a client *A* to get permission to access B.http, it obtains a *capability* for the service. (3) Client *A* can now communicate with server by prepending the returned capability to each packet.

.

users and hosts, advertising services that are available, and deciding who can connect to these services. It allows hosts to communicate by handing out capabilities (encrypted source routes). As we will see in Section 3.5, because the network depends on it, the DC will typically be physically replicated (described in Section 3.5).

The DC performs three main functions:

1. **Authentication Service**: This service authenticates principals (e.g., users, hosts) and switches. It maintains a symmetric key with each for secure communication.[2]

2. **Network Service Directory (NSD)**: The NSD replaces DNS. When a principal wants access to a service, it first looks up the service in the NSD (services are published by servers using a unique name). The NSD checks for permissions—it maintains an access control list (ACL) for each service—and then returns a *capability*. The ACL is declared in terms of system principals (users, groups), mimicking the controls in a file system.

3. **Protection Layer Controller**: This component controls all connectivity in a SANE network by generating (and revoking) *capabilities*. A capability is a switch-level *source route* from the client to a server, as shown in Figure 2. Capabilities are encrypted in layers (i.e., onion routes [23]) both to prove that they originated from the DC and to hide topology. Capabilities are included in a SANE header in all data packets. The SANE header goes between the Ethernet and IP headers, similar to the location VLANs occupy (Figure 3).

   The controller keeps a complete view of the network topology so that it can compute routes. The topology is constructed on the basis of link-state updates generated by authenticated switches. Capabilities are created using the symmetric keys (to switches and hosts) established by the authentication service.

   The controller will adapt the network when things go wrong (maliciously or otherwise). For example, if a switch floods the DC with control traffic (e.g. link-state updates), it will simply eliminate the switch from the network by instructing its immediate neighbor switches to drop all traffic from that switch. It will issue new capabilities so that ongoing communications can start using the new topology.

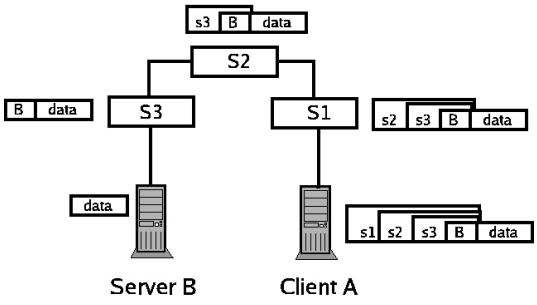

Figure 2: Packets forwarded from client A to server B across multiple switches using a source-routed capability. Each layer contains the next-hop information, encrypted to the associated switch's symmetric key. The capability is passed to A by the DC (not shown) and can be re-used to send packets to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

Figure 3: SANE operates at the same layer as VLAN. All packets on the network must carry a SANE header at the *isolation layer*, which strictly defines the path that packet is allowed to take.

All packet forwarding is done by switches, which can be thought of as simplified Ethernet switches. Switches forward packets along the encrypted source route carried in each packet. They also send link-state updates to the DC so that it knows the network topology.

Note that, in a SANE network, IP continues to provide wide-area connectivity as well as a common fram-

ing format to support the use of unmodified end hosts. Yet within a SANE enterprise, IP addresses are not used for identification, location, nor routing.

3.2 Network Service Directory

The NSD maintains a hierarchy of directories and services; each directory and service has an access control list specifying which users or groups can view, access, and publish services, as well as who can modify the ACLs. This design is similar to that deployed in distributed file systems such as AFS [25].

As an example usage scenario, suppose `martin` wants to share his MP3's with his friends `aditya`, `mike`, and `tal` in the high performance networking group. He sets up a streaming audio server on his machine `bongo`, which has a directory `stanford.hpn.martin.friends` with ACLs already set to allow his friends to list and acquire services. He publishes his service by adding the command

```
sane --publish stanford.martin.ambient:31337
```

to his audio server's startup script, and, correspondingly, adds the command

```
sane --remove stanford.martin.ambient
```

to its shutdown script. When his streaming audio server comes on line, it publishes itself in the NSD as `ambient`. When `tal` accesses this service, he simply directs his MP3 player to the name `stanford.martin.ambient` The NSD resolves the name (similar to DNS), has the DC issue a capability, and returns this capability, which `tal`'s host then uses to access the audio server on `bongo`.

There is nothing unusual about SANE's approach to access control. One could envision replacing or combining SANE's simple access control system with a more sophisticated trust-management system [15], in order to allow for delegation, for example. For most purposes, however, we believe that our current model provides a simple yet expressive method of controlling access to services.

3.3 Protection Layer

All packets in a SANE network contain a SANE header located between the Ethernet and IP headers. In Figure 4, we show the packet types supported in SANE, as well as their intended use (further elaborated below).

Communicating with the DC. SANE establishes default connectivity to the DC by building a minimum

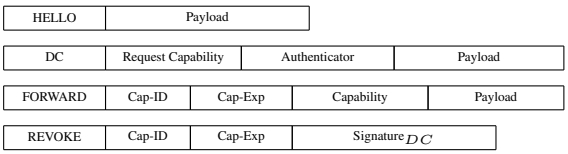

Figure 4: Packet types in a SANE network: `HELLO` packets are used for immediate neighbor discovery and thus are never forwarded. `DC` packets are used by end hosts and switches to communicate with the DC; they are forwarded by switches to the DC along a default route. `FORWARD` packets are used for most host-to-host data transmissions; they include an encrypted source route (capability) which tells switches where to forward the packet. Finally, `REVOKE` packets revoke a capability before its normal expiration; they are forwarded back along a capability's forward route.

spanning tree (MST), with the DC as the root of the tree. This is done using a standard distance vector approach nearly identical to that used in Ethernet switches [1], with each switch sending `HELLO` messages to its neighbor, indicating its distance from the root. The MST algorithm has the property that no switch learns the network topology nor is the topology reproducible from packet traces.

The spanning tree is only used to establish default routes for forwarding packets to the DC. We also need a mechanism for the DC to communicate back with switches so as to establish symmetric keys, required both for authentication and for generating and decoding capabilities. Note that the DC can initially only communicate with its immediate neighbors, since it does not know the full topology.

The DC first establishes shared keys with its direct neighbors, and it receives link-state updates from them. It then iteratively contacts switches at increasing distances (hop-counts), until it has established shared keys with all switches to obtain a map of the full topology.[3] To contact a switch multiple hops away, the DC must first generate a capability given the topology information collected thus far. Once established, keys provide confidentiality, integrity, and replay defense for all subsequent traffic with the DC via an *authenticator* header, much like IPsec's `ESP` header.

All capability requests and link state updates—packets of type `DC`—are sent along the MST. As packets traverse the MST, the switches construct a *request capability*[4] by generating an encrypted onion at each hop containing the previous and next hop, encrypted under the switch's own key. The DC uses the request capabilities to communicate back to each sender. Because these capabilities encode the path, the DC can use them to determine the location of misbehaving senders.

Point-to-Point Communication. Hosts communicate

using capabilities provided by the DC. This traffic is sent using `FORWARD` packets which carry the capability. On receipt of a packet, switches first check that the capability is valid, that it has not expired and that it has not been revoked (discussed later).

Before discussing how capabilities are constructed, we must differentiate between long-lived names and ephemeral connection identifiers. Names are known to the service directory for published services and their access control lists. Identifiers enable end hosts to demultiplex packets as belonging to either particular connections with other end hosts or to capability requests with the DC, much like transport-level port numbers in TCP or UDP. (They are denoted as `client-ID` and `server-ID` below.) So, much like in traditional networks à la DNS names and IP addresses, users use SANE names to identify end-points, while the network software and hardware uses connection identifiers to identify unique services.

The DC constructs capabilities using three pieces of information: the client's name and location (given in the capability request), the service's location (stored in the service directory), and the path between these two end-points (as calculated from the network topology and any service policies).

Each layer in the capability is calculated recursively, working backward from the receiver, using the shared key established between the DC and the corresponding switches.

1. Initialize:
   CAPABILITY $\leftarrow E_{K_{\text{server-name}}}$ (client-name, client-ID, server-ID, last-hop)

2. Recurse: For each node on the path, starting from the last node, do:
   CAPABILITY $\leftarrow E_{K_{\text{switch-name}}}$ (switch-name, next-hop, prev-hop, CAPABILITY)

3. Finalize:
   CAPABILITY $\leftarrow E_{K_{\text{client-name}}}$ (client-name, client-ID, first-hop, CAPABILITY), IV

Where, $E_k(m)$ denotes the encryption of message m under the symmetric key k. Encryption is implemented using a block cipher (such as AES) and a message authentication code (MAC) to provide both confidentiality and integrity.

All capabilities have a globally unique ID `Cap-ID` for revocation, as well as an expiration time, on the order of a few minutes, after which a client must request a new capability. This requires that clocks are only loosely synchronized to within a few seconds. Expiration times may vary by service, user, host, etc.

The MAC computation for each layer includes the `Cap-ID` as well as the expiration time, so they cannot be tampered with by the sender or en-route. The initialization vector (IV) provided in the outer layer of capabilities is the encryption randomization value used for all layers. It prevents an eavesdropper from linking capabilities between the same two end-points.[5]

Revoking Access. The DC can *revoke* a capability to immediately stop a misbehaving sender for misusing a capability. A victim first sends a revocation request, which consists of the final layer of the offending capability, to the DC. The DC verifies that the requester is on the capability's path, and it returns a signed packet of type `REVOKE`.

The requester then pushes the revocation request to the upstream switch from which the misbehaving capability was forwarded. The packet travels hop-by-hop on the reverse path of the offending capability. On-path switches verify the DC's digital signature, add the revoked `Cap-ID` to a local revocation list, and compare it with the `Cap-ID` of each incoming packet. If a match is found, the switch drops the incoming packet and forwards the revocation to the previous hop. Because such revocation packets are not on the data path, we believe that the overhead of signature verification is acceptable.

A revocation is only useful during the lifetime of its corresponding capability and therefore carries the same expiration time. Once a revocation expires, it is purged from the switch. We discuss protection against revocation state exhaustion in section 4.1.

3.4 Interoperability

Discussion thus far has assumed a clean-slate redesign of all components in the network. In this section, we describe how a SANE network can be used by unmodified end-hosts with the addition of two components: *translation proxies* for mapping IP events to SANE events and *gateways* to provide wide-area connectivity.

Translation Proxies. These proxies are used as the first hop for all unmodified end hosts. Their primary function is to translate between IP naming events and SANE events. For example, they map DNS name queries to DC service lookups and DC lookup replies to DNS replies. When the DC returns a capability, the proxy will cache it and add it to the appropriate outgoing packets from the host. Conversely, the proxy will remove capabilities from packets sent to the host.

In addition to DNS, there are a number of service discovery protocols used in today's enterprise networks, such as SLP [44], DNS SD [4], and uPNP [6]. In order to be fully backwards-compatible, SANE translation prox-

ies must be able to map all service lookups and requests to DC service queries and handle the responses.

Gateways. Gateways provide similar functionality to perimeter NATs. They 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.

Broadcast. Unfortunately, some discovery protocols, such as uPNP, perform service discovery by broadcasting lookup requests to all hosts on the LAN. Allowing this without intervention would be a violation of least privilege. To safely support broadcast service discovery within SANE, all packets sent to the link-layer broadcast address are forwarded to the DC, which verifies that they strictly conform to the protocol spec. The DC then reissues the request to all end hosts on the network, collects the replies and returns the response to the sender. Putting the DC on the path allows it to cache services for subsequent requests, thus having the additional benefit of limiting the amount of broadcast traffic. Designing SANE to efficiently support broadcast and multicast remains part of our future work.

Service Publication. Within SANE, services can be published with the DC in any number of ways: translating existing service publication events (as described above), via a command line tool, offering a web interface, or in the case of IP, hooking into the `bind` call on the local host à la SOCKS [30].

3.5 Fault Tolerance

Replicating the Domain Controller. The DC is logically centralized, but most likely physically replicated so as to be scalable and fault tolerant. Switches connect to multiple DCs through multiple spanning trees, one rooted at each DC. To do this, switches authenticate and send their neighbor lists to each DC separately. Topology consistency between DCs is not required as each DC grants routes independently. Hosts randomly choose a DC to send requests so as to distribute load.

Network level-policy, user declared access policy and the service directory must maintain consistency among multiple DCs. If the 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. (We will consider the case where DCs do not trust each other in the next section.)

Recovering from Network Failure. In SANE, it is the end host's responsibility to determine network failure. This is because direct communication from switches to end hosts violates least privilege and creates new avenues for DoS. SANE-aware end hosts send periodic probes or keep-alive messages to detect failures and request fresh capabilities.

When a link fails, a DC will be flooded with requests for new capabilities. We performed a feasibility study (in Section 6), to see if this would be a problem in practice, and found that even in the worst-case when all flows are affected, the requests would not overwhelm a single DC.

So that clients can adapt quickly, a DC may issue multiple (edge-disjoint, where possible) capabilities to clients. In the event of a link failure, a client simply uses another capability. This works well if the topology is rich enough for there to be edge-disjoint paths. Today's enterprise networks are not usually richly interconnected, in part because additional links and paths make security more complicated and easier to undermine. However, this is no longer true with SANE—each additional switch and link improves resilience. With just two or three alternate routes we can expect a high degree of fault tolerance [27]. With multiple paths, an end host can set aggressive time-outs to detect link failures (unlike in IP networks, where convergence times can be high).

3.6 Additional Features

This section discusses some additional considerations of a SANE network, including its support for middleboxes, mobility, and support for logging.

Middleboxes and Proxies. In today's networks, proxies are usually placed at choke-points, to make sure traffic will pass through them. With SANE, a proxy can be placed anywhere; the DC can make sure the proxy is on the path between a client and a server. This can lead to powerful application-level security policies that far outreach port-level filtering.

At the very least, lightweight proxies can validate that communicating end-points are adhering to security policy. Proxies can also enforce service- or user-specific policies or perform transformations on a per-packet basis. These could be specified by the capability. Proxies might scan for viruses and apply vulnerability-specific filters, log application-level transactions, find information leaks, and shape traffic.

Mobility. Client mobility within the LAN is transparent to servers, because the service is unaware of (and so independent of) the underlying topology. When a client

changes its position—e.g., moves to a different wireless access point—it refreshes its capabilities and passes new return routes to the servers it is accessing. If a client moves locations, it should revoke its current set of outstanding capabilities. Otherwise, much like today, a new machine plugged into the same access point could access traffic sent to the client after it has left.

Server mobility is handled in the same manner as adapting to link failures. If a server changes location, clients will detect that packets are not getting through and request a new set of capabilities. Once the server has updated its service in the directory, all (re)issued capabilities will contain the correct path.

Anti-mobility. SANE also trivially anti-mobility. That is, SANE can *prevent* hosts and switches from moving on the network by disallowing access if they do. As the DC knows the exact location of all senders given request capabilities, it can be configured to only service hosts if they are connected at particular physical locations. This is useful for regulatory compliance, such as 911 restrictions on movement for VoIP-enabled devices. More generally, it allows a strong "lock-down" of network entities to enforce strong policies in the highest-security networks. For example, it can be used to disallow all network access to rogue PCs.

Centralized Logging. The DC, as the broker for all communications, is in an ideal position for network-wide connection logging. This could be very useful for forensics. Request routes protect against source spoofing on connection setup, providing a path back to the connecting port in the network. Further, compulsory authentication matches each connection request to an actual user.

4 Attack Resistance

SANE eliminates many of the vulnerabilities present in today's networks through centralization of control, simple declarative security policies and low-level enforcement of encrypted source routes. In this section, we enumerate the main ways that SANE resists attack.

- **Access-control lists:** The NSD uses ACLs for *directories*, preventing attackers from enumerating all services in the system—an example of the principle of least knowledge—which in turn prevents the discovery of particular applications for which compromises are known. The NSD controls access to *services* to enforce protection at the link layer through DC-generated capabilities—supporting the principle of least privilege—which stops attackers from compromising applications, even if they are discovered.

- **Encrypted, authenticated source-routes and link-state updates:** These prevent an attacker from learning the topology or from enumerating hosts and performing port scans, further examples of the principle of least knowledge.[6] SANE's source routes prevent hosts from spoofing requests either to the DC on the control path or to other end hosts on the data path. We discuss these protections further in Section 4.1.

- **Authenticated network components:** The authentication mechanism prevents unauthenticated switches from joining a SANE network, thwarting a variety of topology attacks. Every switch enforces capabilities providing defence in depth. Authenticated switches cannot lie about their connectivity to create arbitrary links, nor can they use the same authenticated public key to join the network using different physical switches. Finally, well-known spanning-tree or routing attacks [32, 48] are impossible, given the DC's central role. We discuss these issues further in section 4.2.

SANE attempts to degrade gracefully in the face of more sophisticated attacks. Next, we examine several major classes of attacks.

4.1 Resource Exhaustion

Flooding. As discussed in section 3.3, flooding attacks are handled through revocation. However, misbehaving switches or hosts may also attempt to attack the network's control path by flooding the DC with requests. Thus, we rate-limit requests for capabilities to the DC. If a switch or end host violates the rate limit, the DC tells its neighbors to disconnect it from the network.

Revocation state exhaustion. SANE switches must keep a list of revoked capabilities. This list might fill, for example, if it is maintained in a small CAM. An attacker could hoard capabilities, then cause all of them to be revoked simultaneously. SANE uses two mechanisms to protect against this attack: (1) If its revocation list fills, a switch simply generates a new key; this invalidates all existing capabilities that pass through it. It clears its revocation list, and passes the new key to the DC. (2) The DC tracks the number of revocations issued per sender. When this number crosses a predefined threshold, the sender is removed from the service's ACLs.

If a switch uses a sender's capability to flood a receiver, thus eliciting a revocation, the sender can use a different capability (if it has one) to avoid the misbehaving switch. This occurs naturally because the client treats revocation—which results in an inability to get packets

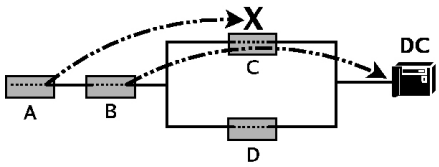

Figure 5: Attacker **C** can deny service to **A** by selectively dropping **A**'s packets, yet letting the packets of its parent (**B**) through. As a result, **A** cannot communicate with the DC, even though a alternate path exists through **D**.

through—as a link failure, and it will try using a different capability instead. While well-behaved senders may have to use or request alternate capabilities, their performance degradation is only temporary, provided that there exists sufficient link redundancy to route around misbehaving switches. Therefore, using this approach, SANE networks can quickly converge to a state where attackers hold no valid capabilities and cannot obtain new ones.

4.2 Tolerating Malicious Switches

By design, SANE switches have minimal functionality—much of which is likely to be placed in hardware—making remote compromise unlikely. Furthermore, each switch requires an authenticated public key, preventing rogue switches from joining the network. However, other avenues of attack, such as hardware tampering or supply-chain attacks, may allow an adversary to introduce a malicious switch. For completeness, therefore, we consider defenses against malicious switches attempting to sabotage network operation, even though the following attacks are feasible only in the most extreme threat environments.

Sabotaging MST Discovery. By falsely advertising a smaller distance to the DC during MST construction, a switch can cause additional DC traffic to be routed through it. Nominally, this practice can create a path inefficiency.

More seriously, a switch can attract traffic, then start dropping packets. This practice will result in degraded throughput, unless the drop rate increases to a point at which the misbehaving switch is declared failed and a new MST is constructed.

In a more subtle attack, a malicious switch can selectively allow packets from its neighbors, yet drop all other traffic. An example of this attack is depicted in Figure 5: Node C only drops packets from node A. Thus, B does not change its forwarding path to the DC, as C appears to be functioning normal from its view. As a result, A cannot communicate with the DC, even though an alternate path exists through D. Note that this attack, at the MST discovery phase, precludes our normal solution for routing around failures—namely, using node-disjoint paths whenever possible—as node A has never registered with the DC in the first place.

From a high level, we can protect against this selective attack by hiding the identities of senders from switches en-route. Admittedly, it is unlikely that we can prevent *all* such information leakage through the various side-channels that naturally exist in a real system, e.g., due to careful packet inspection and flow analysis. Some methods to confound such attacks include (1) hiding easily recognizable sender-IDs from packet headers,[7] (2) padding all response capabilities to the same length to hide path length, and (3) randomizing periodic messages to the DC to hide a node's scheduled timings.

Using these safeguards, if a switch drops almost all packets, its immediate neighbors will construct a new MST that excludes it. If it only occasionally drops packets, the rate of MST discovery is temporarily degraded, but downstream switches will eventually register with the DC.

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

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. Additionally, the isolation properties of the network are still preserved.

Note that SANE's requirement for switches to initially authenticate themselves with the DC prevents Sybil attacks, normally associated with open identity-free networks [21].

4.3 Tolerating a Malicious DC

Domain controllers 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, one can distribute trust among DCs using threshold cryptography. While the full details are beyond the scope of this paper, we sketch the basic approach. We split the DCs' secret key across a few servers (say $n < 6$), such that two of them are needed to generate a capability. The sender then communicates with 2-out-of-n DCs to obtain the capability. Thus, an

attacker gains no additional access by compromising a single DC.[8] To prevent a single malicious DC from revoking arbitrary capabilities or, even worse, completely disconnecting a switch or end host, the revocation mechanism (section 3.3) must also be extended to use asymmetric threshold cryptography [20].

Given such replicated function, 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 [18].

5 Implementation

This section describes our prototype implementation of a SANE network. Our implementation consists of a DC, switches, and IP proxies. It does not support multiple DCs, there is no support for tolerating malicious switches nor were any of the end-hosts instrumented to issue revocations.

All development was done in C++ using the Virtual Network System (VNS) [17]. VNS provides the ability to run processes within user-specified topologies, allowing us to test multiple varied and complex network topologies while interfacing with other hosts on the network. Working outside the kernel provided us with a flexible development, debug, and execution environment.

The network was in operational use within our group LAN—interconnecting seven physical hosts on 100 Mb Ethernet used daily as workstations—for one month. The only modification needed for workstations was to reduce the maximum transmission unit (MTU) size to 1300 bytes in order to provide room for SANE headers.

5.1 IP Proxies and SANE Switches

To support unmodified end hosts on our prototype network, we developed proxy elements which are positioned between hosts and the first hop switches. Our proxies use ARP cache poisoning to redirect all traffic from the end hosts. Capabilities for each flow are cached at the corresponding proxies, which insert them into packets from the end host and remove them from packets to the end host.

Our switch implementation supports automatic neighbor discovery, MST construction, link-state updates and packet forwarding. Switches exchange `HELLO` messages every 15 seconds with their neighbors. Whenever switches detects network failures, they reconfigure their MST and update the DC's network map.

The only dynamic state maintained on each switch is a hash table of capability revocations, containing the `Cap-ID`s and their associated expiration times.

We use OCB-AES [42] for capability construction and decryption with 128-bit keys. OCB provides both confidentiality and data integrity using a single pass over the data, while generating ciphertext that is exactly only 8 bytes longer than the input plaintext.

5.2 Domain Controller

The DC consists of four separate modules: the authentication service, the network service directory, and the topology and capability construction service in the Protection Layer Controller. For authentication purposes, the DC was preconfigured with the public keys of all switches.

Capability construction. For end-to-end path calculations when constructing capabilities, we use a bidirectional search from both the source and destination. All computed routes are cached at the DC to speed up subsequent capability requests for the same pair of end hosts, although cached routes are checked against the current topology to ensure freshness before re-use.

Capabilities use 8-bit IDs to denote the incoming and outgoing switch ports. Switch IDs are 32 bits and the service IDs are 16 bits. The innermost layer of the capability requires 24 bytes, while each additional layer uses 14 bytes. The longest path on our test topologies was 10 switches in length, resulting in a 164 byte header.

Service Directory. DNS queries for all unauthenticated users on our network resolve to the DC's IP address, which hosts a simple webserver. We provide a basic HTTP interface to the service directory. Through a web browser, users can log in via a simple web-form and can then browse the service directory or, with the appropriate permissions, perform other operations (such as adding and deleting services).

The directory service also provides an interface for managing users and groups. Non-administrative users are able to create their own groups and use them in access-control declarations.

To access a service, a client browses the directory tree for the desired service, each of which is listed as a link. If a service is selected, the directory server checks the user's permissions. If successful, the DC generates capabilities for the flows and sends them to the client (or more accurately, the client's SANE IP proxy). The web-server returns an HTTP redirect to the service's appropriate protocol and network address, e.g., `ssh://192.168.1.1:22/`. The client's browser can then launch the appropriate application if one such

is registered; otherwise, the user must do so by hand.

5.3 Example Operation

As a concrete example, we describe the events for an ssh session initiated within our internal network. All participating end hosts have a translation proxy positioned between them and the rest of the network. Additionally, they are configured so that the DC acts as their default DNS server.

Until a user has logged in, the translation proxy returns the DC's IP address for all DNS queries and forwards all TCP packets sent to port 80 to the DC. Users opening a web-browser are therefore automatically forwarded to the DC so that they may log in. This is similar in feel to admission control systems employed by hotels and wireless access points. All packets forwarded to the DC are accompanied by a SANE header which is added by the translation proxy. Once a user has authenticated, the DC caches the user's location (derived from the SANE header of the authentication packets) and associates all subsequent packets from that location with the user.

Suppose a user ssh's from machine A to machine B. A will issue a DNS request for B. The translation proxy will intercept the DNS packet (after forging an ARP reply) and translate the DNS requests to a capability request for machine B. Because the the DNS name does not contain an indication of the service, by convention we prepend the service name to the beginning of the DNS request (e.g. ssh ssh.B.stanford.edu). The DC does the permission check based on the capability (initially added by the translation proxy) and the ACL of the requested service.

If the permission check is successful, the DC returns the capabilities for the client and server, which are cached at the translation proxy. The translation proxy then sends a DNS reply to A with a unique destination IP address d, which allows it to demultiplex subsequent packets. Subsequently, when the translation proxy receives packets from A destined to d, it changes d to the destination's true IP address (much like a NAT) and tags the packet with the appropriate SANE capability. Additionally, the translation proxy piggybacks the return capability destined for the server's translation proxy on the first packet. Return traffic from the server to the client is handled similarly.

6 Evaluation

We now analyze the practical implications of running SANE on a real network. First, we study the performance of our software implementation of the DC and switches. Next, we use packets traces collected from a medium-sized network to address scalability concerns and to evaluate the need for DC replication.

6.1 Microbenchmarks

Table 1 shows the performance of the DC (in capabilities per second) and switches (in Mb/s) for different capability packet sizes (i.e., varying average path lengths). All tests were done on a commodity 2.3GHz PC.

As we show in the next section, our naive implementation of the DC performs orders of magnitude better than is necessary to handle request traffic in a medium-sized enterprise.

The software switches are able to saturate the 100Mb/s network on which we tested them. For larger capability sizes, however, they were computationally-bound by decryption—99% of CPU time was spent on decryption alone—leading to poor throughput performance. This is largely due to the use of an unoptimized encryption library. In practice, SANE switches would be implemented in hardware. We note that modern switches, such as Cisco's catalyst 6K family, can perform MAC level encryption at 10Gb/s. We are in the process of re-implementing SANE switches in hardware.

6.2 Scalability

One potential concern with SANE's design is the centralization of function at the Domain Controller. As we discuss in Section 3.5, the DC can easily be physically replicated. Here, we seek to understand the extent to which replication would be necessary for a medium-sized enterprise environment, basing on conclusions on traffic traces collected at the Lawrence Berkeley National Laboratory (LBL) [36].

The traces were collected over a 34-hour period in January 2005, and cover about 8,000 internal addresses. The trace's anonymization techniques [37] ensure that (1) there is an isomorphic mapping between hosts' real IP addresses and the published anonymized addresses, and (2) real port numbers are preserved, allowing us to identify the application-level protocols of many packets. The trace contains almost 47 million packets, which includes 20,849 DNS requests and 145,577 TCP connections.

Figure 6 demonstrates the DNS request rate, TCP connection establishment rate, and the maximum number of concurrent TCP connections per second, respectively.

The DNS and TCP request rates provide an estimate for an expected rate of DC requests by end hosts in a SANE network. The DNS rate provides a lower-bound that takes client-side caching into effect, akin to SANE end hosts multiplexing multiple flows using a single capability, while the TCP rate provides an upper bound.

	5 hops	10 hops	15 hops
DC	100,000 cap/s	40,000 cap/s	20,000 cap/s
switch	762 Mb/s	480 Mb/s	250 Mb/s

Table 1: Performance of a DC and switches

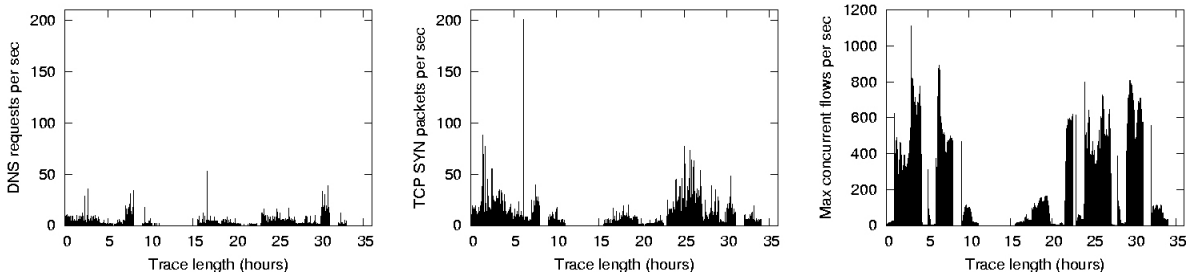

Figure 6: DNS requests, TCP connection establishment requests, and maximum concurrent TCP connections per second, respectively, for the LBL enterprise network.

Even for this upper bound, we found that the peak rate was fewer than 200 requests per second, which is 200 times lower than what our unoptimized DC implementation can handle (see Table 1).

Next, we look at what might happen upon a link failure, whereby all end hosts communicating over the failed link simultaneously contact the DC to establish a new capability. To understand this, we calculated the maximum concurrent number of TCP connections in the LBL network.[9] We find that the dataset has a maximum of 1,111 concurrent connections, while the median is only 27 connections. Assuming the worst-case link failure—whereby all connections traverse the same network link which fails—our simple DC can still manage 40 times more requests.

Based on the above measurements, we estimate the bandwidth consumption of control traffic on a SANE network. In the worst case, assuming no link failure, 200 requests per second are sent to the DC. We assume all flows are long-lived, and that refreshes are sent every 10 minutes. With 1,111 concurrent connections in the worst case, capability refresh requests result in at most an additional 2 packets/s.[10] Given header sizes in our prototype implementation and assuming the longest path on the network to be 10 hops, packets carrying the forward and return capabilities will be at most 0.4 KB in size, resulting in a maximum of 0.646 Mb/s of control traffic.

This analysis of an enterprise network demonstrates that only a few domain controllers are necessary to handle DC requests from tens of thousands of end hosts. In fact, DC replication is probably more relevant to ensure uninterrupted service in the face of potential DC failures.

7 Related Work

Network Protection Mechanisms. Firewalls have been the cornerstone of enterprise security for many years. However, their use is largely restricted to enforcing coarse-grain network perimeters [45]. Even in this limited role, misconfiguration has been a persistent problem [46, 47]. This can be attributed to several factors which SANE tries to address; in particular, their low-level policy specification and very localized view leaves firewalls highly sensitive to changes in topology. A variety of efforts have examined less error prone methods for policy specification [13], as well as how to detect policy errors automatically [33].

The desire for a mechanism that supports ubiquitous enforcement, topology independence, centralized management, and meaningful end-point identifiers has lead to the development of distributed firewalls [14, 26, 2]. Distributed firewalls share much with SANE in their initial motivation but differ substantially in their trust and usage model. First, they require that some software be installed on the end host. This can be beneficial as it provides greater visibility into end host behavior, however, it comes at the cost of convenience. More importantly, for end hosts to perform enforcement, that end host must be trusted (or at least some part of it, e.g., the OS [26], a VMM [22], the NIC [31], or some small peripheral [40]). Furthermore, in a distributed firewall scenario, the network infrastructure itself receives no protection, i.e., the network is still "on" by default. This design affords no defense-in-depth if the end-point firewall is bypassed, as it leaves all other network elements (e.g., switches, middleboxes, and unprotected end hosts) exposed.

Weaver et al. [45] argue that existing configurations of coarse-grain network perimeters (e.g., NIDS and multiple firewalls) and end host protective mechanisms (e.g. anti-virus software) are ineffective against worms, both when employed individually or in combination. They advocate augmenting traditional coarse-grain perimeters with fine-grain protection mechanisms throughout the network, especially to detect and halt worm propagation.

Finally, commercial offerings from Consentry [3] introduce special-purpose bridges for enforcing access control policy. To our knowledge, these solutions require that the bridges be placed at a choke point in the network so that all traffic needing enforcement passes through them. In contrast, SANE permission checking is done at a central point only on connection setup, decoupling it from the data path. SANE's design both allows redundancy in the network without undermining network security policy and simplifies the forwarding elements.

Dealing with Routing Complexity. Often misconfigured routers make firewalls simply irrelevant by routing around them. The inability to reason about connectivity in complex enterprise networks has fueled commercial offerings such as those of Lumeta [5], to help administrators discover what connectivity exists in their network.

In their 4D architecture, Rexford et al. [41, 24] 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 data path. Although 4D centralizes routing policy decisions, they retain the security model of today's networks. Routing (forwarding tables) and access controls (filtering rules) are still decoupled, disseminated to forwarding elements, and operate the basis of weakly-bound end-point identifiers (IP addresses). 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.

Predicate routing [43] attempts to unify security and routing by defining connectivity as a set of declarative statements from which routing tables and filters are generated. SANE differs, however, in that users are first-class objects—as opposed to end-point IDs or IP addresses in Predicate routing—and thus can be used in defining access controls.

Expanding the Link-layer. Reducing a network from two layers of connectivity to one, where all forwarding is done entirely at the link layer, has become a popular method of simplifying medium-sized enterprise networks. However, large Ethernet-only networks face significant problems with scalability, stability, and fault tolerance, mainly due to their use of broadcast and spanning-tree-based forwarding.

To address these concerns, several proposals have suggested replacing the MST-based forwarding at the link-layer with normal link-state routing [39, 35]. Some, such as Myers et al. [35], advocate 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 it eliminates the inefficiencies of broadcast. Similarly, SANE eliminates broadcast in favor of tighter traffic control through link-state updates and source routes. However, we eschew the use of persistent end host identifiers, instead associating each routable destination with the switch port from where it registered.

Capabilities for DDOS prevention. Much recent work has focused on DoS remediation through network enforced capabilities on the WAN [12, 51, 52]. 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 vice versa. 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 operating assumptions (no common administrative domain).

8 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. However, providing strong protection is difficult, and it requires some tradeoffs. There are clear advantages to having an open environment where connectivity is unconstrained and every end host can talk to every other. Just as clearly, however, such openness is prone to attack by malicious users from inside or outside the network.

We set out to design a network that greatly limits the ability of an end host or switch to launch an effective attack, while still maintaining flexibility and ease of management. 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 gives the least possible privilege and knowledge to all parties, except to a trusted, central Domain Controller. We believe that this would be an acceptable practice in enterprises, where central control and restricted access are common.

Yet SANE remains practical: Our implementation shows that SANE could be deployed in current networks with only a few modifications, and it can easily scale to networks of tens of thousands of nodes.

9 Acknowledgements

We would like to thank Mendel Rosenblum, Vern Paxson, Nicholas Weaver, Mark Allman and Bill Cheswick for their helpful comments on this project. We also like to thank the anonymous reviewers for their feedback and especially our shepherd, Michael Roe, for his guidance. This research was supported in part by the Stanford Clean Slate program, the 100x100 project and NSF. Part of this research was performed while on appointment as a U.S. Department of Homeand Security (DHS) Fellow under the DHS Scholarship and Fellowship Program, a program administered by the Oak Ridge Institute for Science and Education (ORISE) for DHS through an interagency agreement with the U.S Department of Energy (DOE). ORISE is managed by Oak Ridge Associated Universities under DOE contract number DE-AC05-00OR22750. All opinions expressed in this paper are the authors' and do not necessarily reflect the policies and views of DHS, DOE, ORISE, or NSF. This work was also supported in part by TRUST (The Team for Research in Ubiquitous Secure Technology), which receives support from the National Science Foundation (NSF award number CCF-0424422).

Notes

[1]A policy might be specified by many people (e.g, LDAP), but is typically centrally managed.

[2]SANE is agnostic to the PKI or other authentication mechanism in use (e.g. Kerberos, IBE). Here, we will assume principals and switches have keys that have been certified by the enterprises CA.

[3]To establish shared keys, we opt for a simple key-exchange protocol from the IKE2 [28] suite.

[4]Request capabilities are similar to network capabilities as discussed in [12, 51]

[5]We use the same IV for all layers—as opposed to picking a new random IV for each layer—to reduce the capability's overall size. For standard modes of operation (such as CBC and counter-mode) reusing the IV in this manner does not impact security, since each layer uses a different symmetric key.

[6]For example, while SANE's protection layer prevents an adversary from targeting arbitrary switches, an attacker can attempt to target a switch indirectly by accessing an upstream server for which it otherwise has access permission.

[7]Normally, DC packet headers contain a consistent sender-ID in cleartext, much like the IPSec ESP header. This sender-ID tells the DC which key to use to authenticate and decrypt the payload. We replace this static ID with an ephemeral nonce provided by the DC. Every DC response contains a new nonce to use as the sender-ID in the next message.

[8]Implementing threshold cryptography for symmetric encryption is done combinatorially [16]: Start from a t-out-of-t sharing (namely, encrypt a DC master secret under all independent DC server keys) and then construct a t-out-of-n sharing from it.

[9]To calculate the concurrent number of TCP connections, we tracked srcip:srcport:dstip:dstport tuples, where a connection is considered finished upon receiving the first FIN packet or if no traffic packets belonging to that tuple are seen for 15 minutes. There were only 143 cases of TCP packets that were sent after a connection was considered timed-out.

[10]This is a conservative upper bound: In our traces, the average flow length is 92s, implying that at most, 15% of the flows could have lengths greater than 10 minutes.

References

[1] 802.1D MAC Bridges. http://www.ieee802.org/1/pages/802.1D-2003.html.

[2] Apani home page. http://www.apani.com/.

[3] Consentry home page. http://www.consentry.com/.

[4] DNS Service Discover (DNS-SD). http://www.dns-sd.org/.

[5] Lumeta. http://www.lumeta.com/.

[6] UPnP Standards. http://www.upnp.org/.

[7] Cisco Security Advisory: Cisco IOS Remote Router Crash. http://www.cisco.com/warp/public/770/ioslogin-pub.shtml, August 1998.

[8] CERT Advisory CA-2003-13 Multiple Vulnerabilities in Snort Preprocessors. http://www.cert.org/advisories/CA-2003-13.html, April 2003.

[9] Sasser Worms Continue to Threaten Corporate Productivity. http://www.esecurityplanet.com/alerts/article.php/3349321, May 2004.

[10] 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.

[11] ICMP Attacks Against TCP Vulnerability Exploit. http://www.securiteam.com/exploits/5SP0N0AFFU.html, April 2005.

[12] T. Anderson, T. Roscoe, and D. Wetherall. Preventing Internet Denial-of-Service with Capabilities. *SIGCOMM Comput. Commun. Rev.*, 34(1):39–44, 2004.

[13] Y. Bartal, A. J. Mayer, K. Nissim, and A. Wool. Firmato: A novel firewall management toolkit. *ACM Trans. Comput. Syst.*, 22(4):381–420, 2004.

[14] S. M. Bellovin. Distributed firewalls. *;login:*, 24(Security), November 1999.

[15] M. Blaze, J. Feigenbaum, and A. D. Keromytis. Keynote: Trust management for public-key infrastructures (position paper). In *Proceedings of the 6th International Workshop on Security Protocols*, pages 59–63, London, UK, 1999. Springer-Verlag.

[16] 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.

[17] M. Casado and N. McKeown. The Virtual Network System. In *Proceedings of the ACM SIGCSE Conference*, 2005.

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

[19] 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.

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

[21] J. R. Douceur. The Sybil attack. In *First Intl. Workshop on Peer-to-Peer Systems (IPTPS 02)*, Mar. 2002.

[22] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the 19th Symposium on Operating System Principles(SOSP 2003)*, October 2003.

[23] 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.

[24] 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.

[25] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, R. N. Sidebotham, and M. J. West. Scale and performance in a distributed file system. *ACM Trans. Comput. Syst.*, 6(1):51–81, Feb. 1988.

[26] S. Ioannidis, A. D. Keromytis, S. M. Bellovin, and J. M. Smith. Implementing a distributed firewall. In *ACM Conference on Computer and Communications Security*, pages 190–199, 2000.

[27] G. C. S. Jian Pu, Eric Manning. Routing Reliability Analysis of Partially Disjoint Paths. In *IEEE Pacific Rim Conference on Communications, Computers and Signal processing (PACRIM' 01)*, volume 1, pages 79–82, August 2001.

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

[29] 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.

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

[31] T. Markham and C. Payne. Security at the Network Edge: A Distributed Firewall Architecture. In *DARPA Information Survivability Conference and Exposition*, 2001.

[32] G. M. Marro. Attacks at the data link layer, 2003.

[33] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *SP '00: Proceedings of the 2000 IEEE Symposium on Security and Privacy*, page 177, Washington, DC, USA, 2000. IEEE Computer Society.

[34] 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.

[35] A. Myers, E. Ng, and H. Zhang. Rethinking the Service Model: Scaling Ethernet to a Million Nodes. In *ACM SIGCOMM HotNets*, November 2004.

[36] R. Pang, M. Allman, M. Bennett, J. Lee, V. Paxson, and B. Tierney. A first look at modern enterprise traffic. In *ACM/USENIX Internet Measurement Conference*, Oct. 2005.

[37] R. Pang, M. Allman, V. Paxson, and J. Lee. The devil and packet trace anonymization. *ACM Comput. Commun. Rev.*, 36(1), Jan. 2006.

[38] R. Perlman. *Network layer protocols with Byzantine robustness*. PhD thesis, Massachussets Institute of Technology, 1988.

[39] R. J. Perlman. Rbridges: Transparent Routing. In *INFOCOM*, 2004.

[40] V. Prevelakis and A. D. Keromytis. Designing an Embedded Firewall/VPN Gatweway. In *Proc. International Network Conference*, July 2002.

[41] 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.

[42] 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.

[43] 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.

[44] J. Veizades, E. Guttman, C. Perkins, and S. Kaplan. Service location protocol. RFC 2165, july 1997.

[45] N. Weaver, D. Ellis, S. Staniford, and V. Paxson. Worms vs. Perimeters: The Case for Hard-LANs. In *Proc. Hot Interconnects 12*, August 2004.

[46] A. Wool. A quantitative study of firewall configuration errors. *IEEE Computer*, 37(6):62–67, 2004.

[47] A. Wool. The use and usability of direction-based filtering in firewalls. *Computers & Security*, 26(6):459–468, 2004.

[48] S. Wu, B. Vetter, and F. Wang. An experimental study of insider attacks for the OSPF routing protocol. October 1997.

[49] 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.

[50] 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.

[51] 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.

[52] 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.