The STRONGMAN Architecture*

Angelos D. Keromytis, Sotiris Ioannidis, Michael B. Greenwald and Jonathan M. Smith†

Abstract

The design principle of restricting local autonomy only where necessary for global robustness has led to a scalable Internet. Unfortunately, this scalability and capacity for distributed control has not been achieved in the mechanisms for specifying and enforcing security policies. This shortcoming must be overcome if end-to-end security mechanisms (such as IPsec or TLS) are to ever replace solutions of short-term convenience such as firewalls.

The STRONGMAN (for Scalable TRust Of Next Generation MANagement) system offers three new approaches to scalability, applying the principle of local policy enforcement complying with global security policies. First is the use of a compliance checker to provide great local autonomy within the constraints of a global security policy. Second is a mechanism to compose policy rules into a coherent enforceable set, e.g., at the boundaries of two locally autonomous application domains. Third is the "lazy instantiation" of policies to reduce the amount of state that enforcement points need to maintain.

We demonstrate the use of these approaches in the design, implementation, and measurements of a distributed firewall. Our experiments show that, under certain circumstances, performance can improve over the traditional-firewall approach.

1 Introduction

Much of the Internet's scalability has been achieved as a byproduct of intelligent application of the end-to-end design principle ([20, 6]), where properties that must hold end-to-end are provided by mechanisms at the end points. The resulting design keeps the network simple and allows great local autonomy in implementing these mechanisms.

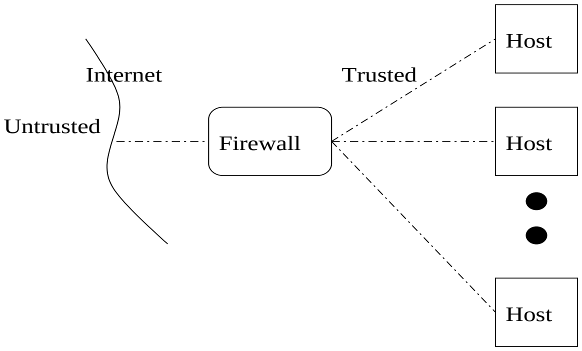

Figure 1. A firewall's bottleneck topology.

Security for distributed applications is arguably an end-to-end property. By the end-to-end argument hosts *should be* responsible for the perceived security of "the internet." However, several factors currently argue against this placement of functionality. First, policies must typically be specified at the granularity of administrative domains (*e.g.,* a corporate network), and not only at the granularity of individual hosts. Second, some operating systems have been designed under the assumption that network security is mostly handled by third parties (firewalls), thus lacking enforcement mechanisms. Third, many security policies adopt the "hard shell, soft interior" approach, by granting more rights to "local" (and, by implication, trusted) machines and entities.

This situation has led, for example, to the pervasive use of firewalls, which enforce a single security policy at network boundaries to protect multiple hosts behind the boundaries from certain classes of security problems. To implement the policy globally, the network topology must be restricted to pass all traffic through the firewall, as shown in Figure 1. Unfortunately, these firewalls have many negative consequences for Internet routing, flow control, and performance. Furthermore, when the firewall fails or is otherwise bypassed, the entire internal network is at the mercy of the intruder (as was evidenced by the recent cases of corporate-network infections by multi-vectored worms).

Any alternative that attempts to avoid the performance

*This work was supported by DARPA under Contract F39502-99-1-0512-MOD P0001.

†Angelos D. Keromytis is with the CS Department, Columbia University, Email: `angelos@cs.columbia.edu`. Sotiris Ioannidis, Michael B. Greenwald, Jonathan M. Smith are with the CIS Department, University of Pennsylvania, Email: `{sotiris,mbgreen,jms}@dsl.cis.upenn.edu`

bottleneck of a centralized firewall must support a simple (and *consistent*) specification of security policy for an entire administrative domain. In other words, there must be means of ensuring that the local enforcement actually conforms to the larger ("global") policy. Since manual or semi-automatic configuration of nodes and protocols to conform to a global policy has been shown to be problematic and error-prone [13], automatic techniques relying on a single method of specification are desirable.

To further complicate matters, experience has shown that no single mechanism exists that can address the security requirements of all applications and protocols. Therefore, multiple security mechanisms (with overlapping scopes, such as IPSec and SSL) are typically in use simultaneously in many networks. These multiple security mechanisms must present a single consistent system image to the administrator, else complexity of configuration will again result in errors.

It may seem natural to generalize the solution proposed by distributed firewalls ([2, 14]) and design a "universal" high-level policy specification language. Such a language would, ideally, specify global policies that must be enforced across multiple heterogeneous domains. However, security policies are often application-dependent. "Universal" high-level policy languages tend to be feature-rich and complex, and are therefore clumsy and lead to mistakes. Furthermore, such languages often presume homogeneity, and cannot handle mixtures of multiple mechanisms/languages for different parts of the same network.

Therefore, we argue that the correct approach is an architecture that ties together multiple security mechanisms within a single system image, that supports many application-specific policy languages, that automatically distributes and uniformly enforces the single security policy across all enforcement points, and that allows enforcement points to be chosen appropriately to meet both security and performance requirements. Further, this architecture must scale with the growth of the network in several dimensions (number of users, hosts, protocols/applications, and security policies tying all these together).

In this paper we propose an architecture, STRONGMAN, and argue that it meets these requirements. The main components of our architecture are the use of a policy compliance checker to provide great local autonomy within the constraints of a global security policy, a mechanism for composing policy rules into a coherent enforceable set, and "lazy instantiation" of policies to reduce the amount of state that enforcement points need to maintain.

In the following sections we describe these three components and their use in the STRONGMAN architecture in more detail, discuss its instantiation in the form of a distributed firewall, and present some preliminary measurements which show that performance can improve in certain scenarios, relative to the traditional firewall approach. We then compare our approach with other work, and conclude the paper with some discussion on future directions.

2 Our Approach

Following our previous discussion, we have set certain requirements for our proposed system. First, it must handle growth in the number of users, applications, enforcement points, and rules pertaining to these. A corollary to this is that the most common operations (*i.e.*, policy updates) must be very cheap. Second, security policies for a particular application should be specifiable in an application-specific language or application-specific extension. Third, administrators should be able to independently specify policies over their own domain: this should be true whether the administrator manages particular applications within a security domain, or manages a sub-domain of a larger administrative domain. In other words, the system must support privilege delegation and hierarchical management.

These requirements shape our design of the STRONGMAN architecture. An overview of the policy flow in our architecture is shown in Figure 2. It should be immediately clear that there is a distinction between high and low level policy. In particular, we envision a multiplicity of high-level policy specification mechanisms (different languages, GUIs, *etc.*), all translating to the same lower-level policy expression language. A powerful, flexible, and extensible low-level mechanism that is used as a common "policy interoperability layer" allows us to use the same policy model across different applications, without mandating the use of any particular policy front-end. This architecture has an intentional resemblance to the IP "hourglass", and resolves heterogeneity in similar ways, *e.g.,* the mapping of the interoperability layer onto a particular enforcement device, or the servicing of multiple applications with a policy *lingua franca*.

As the figure also implies, policy is enforced in a decentralized manner. STRONGMAN shifts as much of the operational burden as possible to the end users' systems because traditional enforcement points are generally overloaded with processing requests and mediating access. In our architecture, we can have an arbitrary number of enforcement points, deployed at the granularity necessary to enforce very fine-grained access control. This, however, can lead to excessively large numbers of policy rules (in the worst case, the cross-product of the number of users, number of nodes, and number of services per node). In order to minimize the resources consumed by policy storage and processing at each enforcement point, the low-level policy system supports "lazy instantiation" of policy. In other words, an enforcement point should only learn those parts of the global policy that it actually has to enforce as a re-

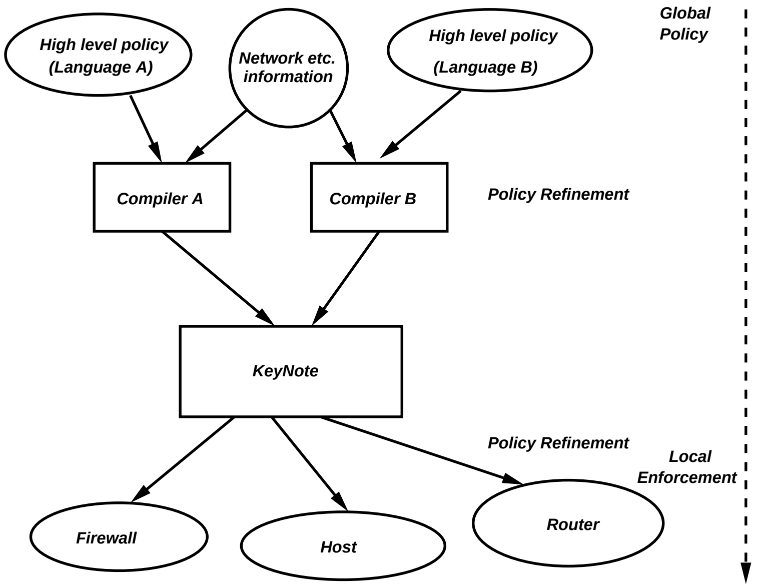

Figure 2. **KeyNote used as a policy interoperability layer. Policy composition in STRONGMAN does not depend on using the same compiler to process all the high-level policies.**

sult of user service access patterns. A further benefit of this approach is that policy may be treated as "soft state," and thus be discarded by the enforcement point when resources are running low, and recovered when space permits or after a crash.

Other important aspects of our architecture, not shown in Figure 2, include:

- Independent policy specifications can be composed in a manner which does not violate any of them, because multiple independently-specified policies may be managed at a single enforcement point.
- Users are identified by their public keys (each user may have multiple keys, for different purposes/applications). These public keys are used in the context of various protocols to authenticate the users to specific services. This also helps prevent malicious users from tampering with policies provided to enforcement points via "lazy policy instantiation".
- The low-level policy system allows for decentralized and hierarchical management and supports privilege delegation to other users. Note that delegation allows any user to be treated as an "administrator" of her delegatees; conversely, administrators in such a system can simply be viewed as users with very broad privileges. This permits both decentralized management (different administrators/users are made responsible for delegating and potentially refining different sets of privileges), and collaborative networking (by treating the remote administrator as a local user with specific privileges she can then delegate to her users). Limited privileges can be conferred to administrators of other domains, who can then delegate these to their users appropriately; this allows for Intranet-style collaborations.

Our architecture implements these design principles by using the KeyNote [3] trust-management system as a basis for expressing and distributing low-level security policy. In the next few subsections we give an overview of KeyNote, describe the policy translation and composition mechanisms, and discuss how policy is distributed (and how "lazy instantiation" is implemented) in our system.

2.1 KeyNote

KeyNote is a simple trust-management system and language developed to support a variety of applications. Although it is beyond the scope of this paper to give a complete tutorial or reference on KeyNote syntax and semantics (for which the reader is referred to [3]), we review a few basic concepts to give the reader a taste of what is going on.

The basic service provided by the KeyNote system is *compliance checking;* that is, checking whether a proposed *action* conforms to local *policy.* Actions in KeyNote are specified as a set of name-value pairs, called an *Action Attribute Set.* Policies are written in the KeyNote *assertion language* and either accept or reject action attribute sets

```
permit KEY1 if
    using strong encryption and
    target in 192.168.1.0/24

permit USERGROUP4 if
    using authentication and
    origin in LOCALNET and
    target in WEBSERVERS
```

Figure 3. A high-level IPsec policy, enforced at the network layer.

```
allow USERGROUP5 if file "/foo/bar.html"

allow ANGELOS if
    directory "/confidential" and
    source in LOCALNETWORK
```

Figure 4. A high-level web access policy, enforced by the web server.

presented to it. Policies can be broken up and distributed via *credentials*, which are signed assertions that can be sent over a network and to which a local policy can defer in making its decisions. The credential mechanism allows for complex graphs of trust, in which credentials signed by several entities are considered when authorizing actions. Users have a variety of credentials, for the different services and nodes they need to access.

Each service that needs to mediate access, queries its local compliance checker on a per-request basis (what constitutes a "request" depends on the specific service and protocol). The compliance checker can be implemented as a library that is linked against every service, as a daemon that serves all processes in a host, or as a network service (this latter case requires provisions for secure communications between the policy enforcer and the compliance checker).

2.2 Policy Translation and Composition

In our architecture, policy for different network applications can be expressed in various high-level policy languages or systems, each fine-tuned to the particular application. Each such language is processed by a specialized compiler that can take into consideration such information as network topology or a user database and produces a set of KeyNote credentials. At the absolute minimum, such a compiler needs a knowledge of the public keys identifying the users in the system. Other information is necessary on a per-application basis. For example, knowledge of the network topology is typically useful in specifying packet filtering policy; for web content access control, on the other hand, the web servers' contents and directory layout are probably more useful. Our proof-of-concept languages (examples are shown in Figures 3 and 4) use a template-based mechanism for generating KeyNote credentials.

This decoupling of high and low level policy specification permits a more modular and extensible approach, since languages may be replaced, modified, or created without affecting the underlying system.

Our architecture requires each high-level language or GUI to include a "referral" primitive. A referral is simply a reference to a decision made by another language/enforcement point (typically lower in the protocol stack). This primitive allows us to perform policy composition at enforcement time; decisions made by one enforcement mechanism (*e.g.,* IPsec) are made available to higher-level enforcement mechanisms and can be taken into consideration when making an access control decision. An example of this is shown in Figure 5. The only needed coordination between two policy domains is determining what kind of information (encoded in the referrals) needs to be generated and consumed respectively.

To complete the composition discussion, all that is necessary is a channel to propagate this information across enforcement layers. In our system, this is done on a case-by-case basis. For example, IPsec information can be propagated higher in the protocol stack by suitably modifying the Unix `getsockopt(2)` system call; in the case of a web server and SSL, the information is readily available through the SSL data structures (since the SSL and the web access control enforcement are both done in the context of a single process address space). This approach is sufficient for policy interaction across network layers, but would not work for arbitrary policy domain interaction.

2.3 Credential Management

Following our design decision of shifting as much as possible of the operational burden away from the enforcement points and to the users' systems, we make the users responsible for presenting the necessary credentials to the enforcement points they access. Thus, the enforcement points dynamically "learn" those parts of the global policy that are relevant to a particular request. It is in the interest of the user to present the correct credentials, in order to obtain service.

Compiled credentials are available to users through policy repositories. These credentials are signed by the administrator's key and contain the various conditions under which a specific user (as identified by her key in the credential) is allowed to access a service. The translation of the policy rule in Figure 5 is shown in Figure 6.

Users who wish to gain access to some service first need to acquire a fresh credential from one of the repositories. It

```
allow USER_ROOT if
  directory "/confidential" and
  source in LOCALNETWORK and
  (application IPsec says
         "strong encryption" or
   application SSL says
         "very strong encryption")
```

Figure 5. Web access policy taking into consideration decisions made by the IPsec and SSL protocols. The information on USER_ROOT and LOCALNETWORK are specified in separate databases, which the compiler takes into consideration when compiling these rules to KeyNote credentials.

```
Authorizer: ADMINISTRATOR_KEY
Licensees: USER_ROOT_KEY
Conditions: app_domain == "web access" &&
  directory ~= "^/confidential/.*" &&
  (source_address <= "192.168.001.255" &&
   source_address >= "192.168.001.000") &&
  (ipsec_result == "strong encryption" ||
   ssl_result ==
     "very strong encryption");
Signature: ...
```

Figure 6. Translation of the policy rule from Figure 5 to a KeyNote credential. The public keys and the digital signature are omitted in the interests of readability.

is not necessary to protect the credentials as they are transferred over the network, since they are self-protected by virtue of being signed[1]. Users then provide these credentials to the relevant service (web server, firewall, *etc.*) through a protocol-specific mechanism. For example, in the case of IPsec, these credentials are passed on to the local key management daemon which then establishes cryptographic context with the remote firewall or end system. In the case of firewalls in particular, the user's system can either depend on a signaling mechanism (as is being developed at the IETF IP Security Policy Working Group) to detect their existence, or can statically analyze the KeyNote credentials to determine what actions need to be taken when trying to access specific services, networks, or end-systems.

It is also possible to pass KeyNote credentials in the TLS protocol. For protocols where this is not possible (*e.g.,* SSHv1), an out-of-band mechanism can be used instead. We have used a simple web server script interface for submitting credentials to be considered in the context of an access control decision; credentials are passed as arguments to a CGI script that makes them available to the web server access control mechanism. To avoid DoS attacks, entries submitted in this manner are periodically purged (in an LRU manner).

Since policy is expressed is terms of credentials issued to users, it need not be distributed synchronously to the enforcement points. As noted above, enforcement points do not need to store all credentials and rules; rather, they learn rules through "lazy policy instantiation" as users try to gain access to controlled resources. If needed credentials were discarded because of resource scarcity, the affected users will simply have to re-submit them with their next access.

Adding a new user or granting more privileges to an existing user is simply a matter of issuing a new credential (note that both operations are equivalent). The inverse operation, removing a user or revoking issued privilege, can be more expensive: in the simple case, a user's credentials can be allowed to expire; this permits a window of access, between the time the decision is taken to revoke a user's privileges and the time the relevant credentials expire. For those cases where this is adequate, there is no additional overhead. This argues for relatively short-lived credentials, which the users (rather, software on their systems) will have to re-acquire periodically. While this may place additional burden on the repositories, it is possible to arrange for credentials to expire at different times from each other, thus mitigating the effect on the infrastructure of multiple users (re-)acquiring their credentials at the same time, if the credentials are relatively long-lived. Given that a large number of digital signatures will have to be computed as a result of periodically issuing credentials, this is also desirable from a policy-generation point of view.

For more aggressive credential revocation, other mechanisms have to be used. Although no single revocation mechanism exists that can be used in all possible systems, we note that any such mechanism should not increase the load or storage requirements on enforcement points. Thus, the most attractive approach is proofs of validity (acquired by the user from a "refresher" server, and provided to the enforcement point along with the credentials). The proofs of validity can be encoded as KeyNote credentials that are injected in the delegation chain, as shown in Figure 7. While this approach is architecturally attractive, it places high load on the refresher servers. The validity verification mechanism may be specified on a per-credential basis, depending on the perceived risk of compromise and the potential damage done if that occurs.

Finally, since KeyNote allows arbitrary levels of delegation (through chains of credentials), it is possible for users to act as lower-level administrators and issue credentials to others. In this way, we can build a hierarchical and decentralized management scheme wherein the corporate

[1] It is possible to provide credential-confidentiality by encrypting each credential with the public key of the intended recipient.

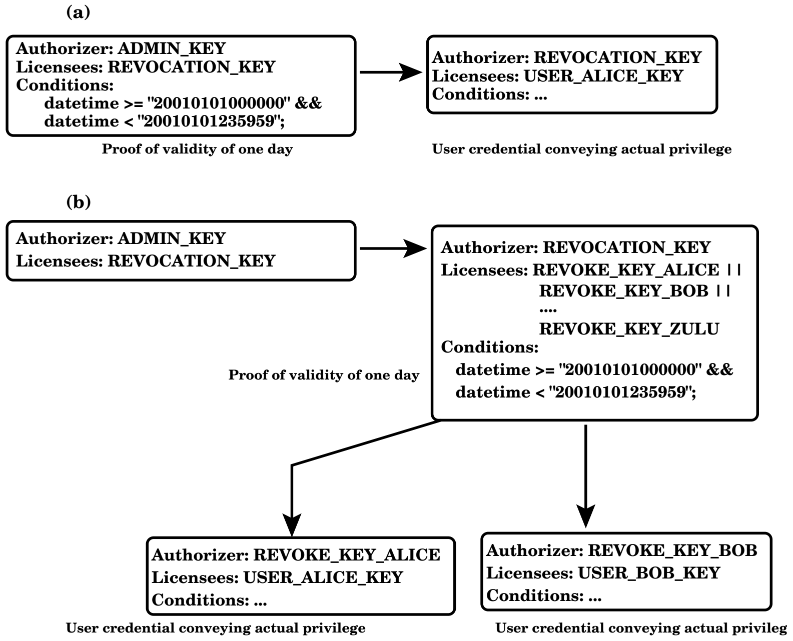
(a)
Authorizer: ADMIN_KEY
Licensees: REVOCATION_KEY
Conditions:
datetime >= "20010101000000" &&
datetime < "20010101235959";
Proof of validity of one day
Authorizer: REVOCATION_KEY
Licensees: USER_ALICE_KEY
Conditions: ...
User credential conveying actual privilege
(b)
Authorizer: ADMIN_KEY
Licensees: REVOCATION_KEY
Authorizer: REVOCATION_KEY
Licensees: REVOKE_KEY_ALICE ||
REVOKE_KEY_BOB ||
....
REVOKE_KEY_ZULU
Conditions:
datetime >= "20010101000000" &&
datetime < "20010101235959";
Proof of validity of one day
Authorizer: REVOKE_KEY_ALICE
Licensees: USER_ALICE_KEY
Conditions: ...
User credential conveying actual privilege
Authorizer: REVOKE_KEY_BOB
Licensees: USER_BOB_KEY
Conditions: ...
User credential conveying actual privileg

Figure 7. **Proof of validity in the form of KeyNote credentials that delegate to the actual user, shown in (a). This approach requires no changes in the compliance checking mechanism or credential distribution. Furthermore, by using a proof of validity that applies to large numbers of users simultaneously, as shown in (b), we can greatly reduce the number of credentials that need to be periodically re-issued.**

network administrator authorizes branch administrators to manage their networks under some constraints. More interestingly, it is possible to view the administrator of another network as a local user; that administrator can handle access to the shared resources for the remote network users, under the constraints specified in their credential, making easy the formation of so-called "extranets."

3 The Distributed Firewall

To validate our design choices and experiment with the different aspects of our architecture, we implemented it in the context of a distributed firewall. A distributed firewall (as described in [14]) enforces a single central security policy at *every* endpoint. The policy specifies what connectivity, both inbound and outbound, is permitted. This policy is distributed to all endpoints where it is authenticated and then enforced, thus making security an end-to-end property.

Distributed firewalls do not rely on the topological notions of "inside" and "outside" as do traditional firewalls. Rather, a distributed firewall grants specific rights to machines that possess the credentials specified by the central policy. A laptop connected to the "outside" Internet has the same level of protection as does a desktop in the organization's facility. Conversely, a laptop connected to the corporate net by a visitor would not have the proper credentials, and hence would be denied access, even though it is topologically "inside."

In the example STRONGMAN distributed firewall, endpoints are characterized by their public keys and the credentials they possess. Thus, the right to connect to the `http` port on a company's internal Web server is only granted to those machines having the appropriate credentials, rather than those machines that happen to be connected to an internal wire. With the advent of wireless LANs, such considerations are becoming extremely relevant.

In our prototype, end hosts (as identified by their IP address) are also considered principals when IPsec is not used to secure communications. This allows local policies or credentials issued by administrators to specify policies similar to current packet-filtering rules. Such policies or credentials have no option but to implicitly trust the validity of an IP address as an identifier. In that respect, they are equivalent to standard packet filtering. The only known solution to this is the use of cryptographic protocols to secure communications.

We should point out that the notions of a traditional and a distributed firewall are not incompatible. Traditional firewalls have an advantage over the distributed firewall in that they offer convenient aggregation points for network traf-

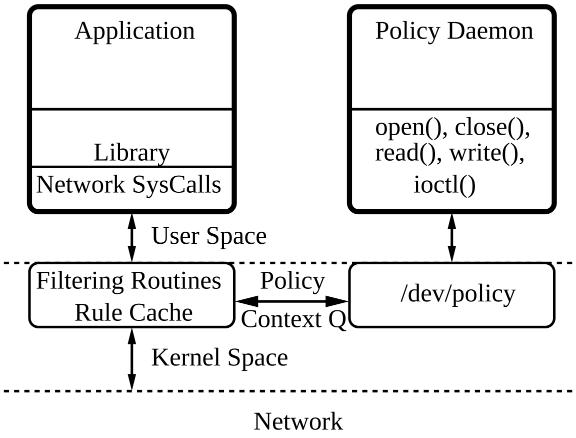

Figure 8. Block diagram of the distributed firewall implementation.

fic, on which services such as denial of service detection (or, more generally, intrusion detection) are easier to deploy and operate. Furthermore, a combination of traditional and distributed firewalls offers "defense in depth", a well-established principle in physical security and the military world.

3.1 Implementation

Our OpenBSD-based implementation is composed of three components: *(1)* a set of kernel extensions, which implement the enforcement mechanisms; *(2)* a user level daemon process, which implements the distributed firewall policies; and *(3)* a device driver, which is used for two-way communication between the kernel and the policy daemon. Our prototype implementation totals approximately 1150 lines of C code, split equally among the three components.

Figure 8 shows a graphical representation of the system, with all its components. The core of the enforcement mechanism lives in kernel space and comprises the filtering routines and the rule cache. The policy specification and processing unit lives in user space, inside the policy daemon process. Any incoming or outgoing IP packets go through the filter and are subject to the policy rules. If none of the rules match, a request is generated and inserted in the *policy context queue.* From there, via the device driver, the policy daemon can get the request and respond accordingly.

In the following three subsections, we briefly describe the various parts of the architecture, their functionality, and how they interact with each other.

3.1.1 Kernel Extensions

In the UNIX operating system, users create outgoing and allow incoming connections using a number of provided system calls. Since any user has access to these system calls, some "filtering" mechanism is needed. This filtering should be based on a policy that is set by the administrator, and any incoming or outgoing packet should be subject to it.

In order to enforce our policy over every packet and yet have a simple and elegant design, we decided to filter IP traffic. To achieve this we added hooks in the `ip_input()` and `ip_output()` routines of the protocol stack (so policies can be enforced on both incoming and outgoing traffic) that will execute our filtering code. We created two data structures to assist us in this process.

The first data structure, the *rules cache,* contains a set of rules that packets are compared against. If a match is found, the rule is followed to either accept or drop the packet. The second data structure is the *policy context queue*. A policy context is a container for all the information related to a specific packet. We associate a sequence number to each such context and then start filling it with all the information the *policy daemon* will need to make an access control decision. A request to the policy daemon comprises the following fields: a sequence number uniquely identifying the request, the ID of the user the connection request belongs to, the number of information fields that will be included in the request, the lengths of those fields, and finally the fields themselves. This can include source and destination addresses, transport protocol and ports, *etc*. Any credentials acquired through IPsec may also be added to the context at this stage. There is no limit as to the kind or amount of information we can associate with a context. We can, for example, include the time of day or the number of other open connections of that user, if we want them to be considered by our decision–making strategy.

Every packet is intercepted at the IP layer and checked against the *rules cache.* If a match is found, the rule is enforced. If no match is found, we enqueue a new request to the *policy context queue*. If we have already enqueued a request for the same class of packets, no further action is necessary. Each entry in the context queue also contains the last packet from that packet flow; if a positive decision is received from the policy daemon, the packet is re-queued for processing by the IP stack.

3.1.2 Policy Device

To maximize the flexibility of our system and allow for easy experimentation, we decided to make the policy daemon a user level process. To support this architecture, we implemented a *pseudo device driver*, `/dev/policy`, that serves as a communication path between the user–space policy daemon, and the modified system calls in the kernel.

The policy daemon reads the device for pending requests in the policy context queue. It then handles the request and returns a new rule to the kernel by writing it to the device, as a result of which the appropriate entry is entered in the rules cache.

It is possible to flush the rules cache. This is useful when the policy that needs to be enforced by the policy daemon is reloaded by the administrator; once the kernel cache is flushed, the new policies will take affect as applicable traffic (incoming or outgoing) is encountered.

3.1.3 Policy Daemon

The last component of our system is the policy daemon. It is a user-level process responsible for making decisions on whether to allow or deny connections. These decisions are based on policies that are specified by an administrator and credentials retrieved remotely or provided by the kernel.

Local policies are initially read in from a file. Policies can be added and removed dynamically. The daemon can simply flush one or more entries from the rules cache in the kernel. This way subsequent packets will not match the existing rule set and the policy daemon will be queried for the new policy. In typical configurations however, the local policies will simply specify the public key(s) of the administrator(s); any specific policies will have to be provided by the user, or (optionally) retrieved from a remote repository.

The daemon receives each request from the kernel by reading the `policy` device. The request contains all the information relevant to that connection. The daemon acts as a front-end for the KeyNote library, which is used to decide whether a request should be granted or not (as well as the "referral"). The decision is sent to the kernel, and the daemon waits for the next request. While the information received in a particular message is application-dependent (in our case, relevant to the distributed firewall), the daemon itself has no awareness of the specific application. Thus, it can be used to provide policy resolution services for many different applications, literally without any modifications.

The "referral" can be provided through the `getsockopt(2)` API to any applications (such as a web server) that may need to make a decision based on the network layer's security properties. We have implemented a module for Apache that does per-HTTP request access control, based on a different set of policies and credentials (issued by the web administrator), demonstrating the feasibility of the "referral" approach.

3.2 Experimental Evaluation

While the architectural discussion is largely qualitative, some estimates of system performance are useful. We performed several experiments, both of comparable node software (using IPF, a packet-filtering package implemented completely inside the kernel, used in many open-source systems) and of varied topologies which demonstrate the value of maintaining consistent global security properties.

Our test machines are x86 architecture machines running OpenBSD, and interconnected by 100 Mbps ethernet. More specifically, in the two-host tests (source to sink), Alice is an 850 Mhz PIII and serves as the traffic source. Bob, the traffic sink, runs the distributed firewall (DF) code and is a 400 Mhz PII.

In the following tables, *insecure* means there is neither DF nor IPF running, IPF means we have IPF activated, *cold cache* means that we have DF running but the rules cache is empty and we must go to the daemon every time to get the rules; this last scenario is useful in determining the cost of cache misses such as might be experienced in the case of a highly utilized service (*e.g.,* an intranet web server with a small ratio of packets per independent user request). *Warm cache* means that the rules are in the cache (except for the first reference).

Insecure	50.4 ms
Cold cache	61.7 ms
Warm cache	51.8 ms
IPF	63.1 ms

Figure 9. Average connection overhead for 100 TCP connections between Alice and Bob.

Insecure	109.1 ms
IPF	134.2ms

Figure 10. Average connection overhead measured for 100 TCP connections between hosts through a firewall.

Insecure	0.273 ± 0.091 ms
Cold cache	0.283 ± 0.089 ms
Warm cache	0.282 ± 0.077 ms
IPF	0.283 ± 0.124 ms

Figure 11. Average roundtrip time for 200 ICMP ECHO_REQUEST messages.

In Figure 9 we have a server application running on Alice; Bob runs a client which connects to the server 100 times using different TCP ports. This generates 200 rules (2 per connection, for incoming and outgoing packets). In the IPF case, those 200 rules are pre-loaded in the filter list. In the second experiment, Bob sent 200 ICMP ECHO_REQUEST messages to Alice; the results are shown in Figure 11. We include the standard deviation, as the measurements did vary slightly. These two experiments show us that the cost of compliance checking in our architecture is very small (within 3% of an insecure system, except for the TCP cold cache case which is 20% more expensive), and typically *better* than IPF. This means that an architecture with decentralized enforcement does not unduly affect end-system latency.

The measurements of Figure 12 have a server application running on Alice; a client running on Bob connects to Alice and transfers 100MB. It is clear that our system does not significantly affect network throughput (the difference is on the order of 0.5%).

Insecure	11,131 ms
Cold cache	11,196 ms
Warm cache	11,178 ms
IPF	11,151 ms

Figure 12. 100MB file transfer over TCP.

In the experiment of Figure 13, we configured 4 (300 MHz) PII systems interconnected via a 100Mbps ethernet hub. One of the four machines is connected to the "outside world" with 100 Mbps ethernet. In the outside world there is an 850 MHz machine (Alice). The "inside" 3 machines run a simple server accepting connections. The outside machine, through the gateway, makes 100 connections in a round robin fashion to the 3 machines. Measurements are given in the table of Figure 10.

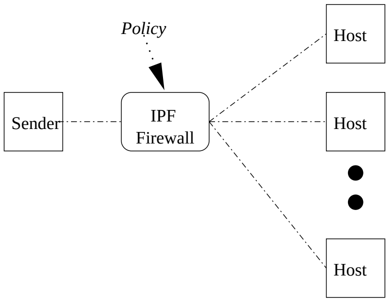

Figure 13. Test topology with intermediate firewall.

Using the same end-hosts, we eliminate the gateway machine, with each of the client machines running the distributed firewall and enforcing policy locally (see Figure 14). The ethernet hub is connected directly to the outside world; the rest of the configuration remains as in the previous experiment. To test the scalability of the distributed firewall we varied the number of hosts that participate in the connection setup. As in the previous experiment, we formed 100 connections to the machines running the distributed firewall in a round robin fashion, each time varying the number of participating hosts. We make the assumption that every protected host inside a firewall contributes roughly the same number of rules, and in the classic centralized case the firewall will have to enforce the sum of those rules. Therefore individual machines will have a smaller rule base than a central control point.

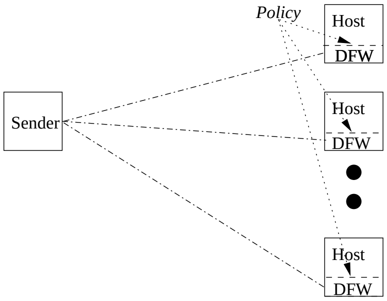

Figure 14. Test topology without intermediate firewall.

The measurements and the percentile overheads are given in Figures 15 and 16. We have kept the total number of rules constant as in the IPF case, and spread them over an increasing number of machines. This experiment clearly demonstrates the benefit of eliminating intermediate enforcement points, and pushing security functions to the endpoints: a *two-fold improvement* in performance compared to the centralized approach, in addition to the increased flexibility and scalability offered by our architecture.

	1 Host	2 Hosts	3 Hosts
Insecure	56.1 ms	53.1 ms	48.6 ms
Cold cache	84.3 ms	62.1 ms	53.7 ms
Warm cache	66.3 ms	58.0 ms	50.5 ms

Figure 15. Average connection overhead for 100 TCP connections spread over one, two and three hosts respectively, using the distributed firewall.

	1 Host	2 Hosts	3 Hosts
Cold cache	50.3%	17.0%	10.4%
Warm cache	23.0%	9.3%	3.8%

Figure 16. Reduction of processing overhead of the distributed firewall as the number of hosts increases. The percentages represent the additional cost of the distributed firewall over the insecure case and are derived from Figure 15.

In the IPF firewall experiments, the rules must be preloaded;

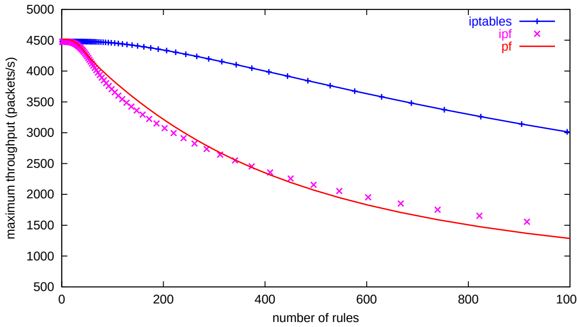

Figure 17. Performance degradation of traditional packet–filtering firewalls as the number of rules increases.

in an experimental configuration such as we described (with *ca.* 200 rules) this is a non-issue. In large installations however, the number of rules can easily reach 4,000 - 5,000 (*e.g.,* for a financial institution we are familiar with). In an environment where simple IP address checking is insufficient, each such rule has other information associated with it (*e.g.,* user public keys, acceptable encryption/authentication algorithms, other conditions for access). Thus, the storage requirements for network layer security policy could vary from 4MB to 100MB or more. This requirement would be imposed on all enforcement points of the same network, which would then be required to have persistent storage (so the policy survives crashes or power cycling). Furthermore, the enforcement points would have to sort through a large number of policies in trying to determine the access rights of any particular user.

Figure 17 shows the performance degradation of IPF and PF (two other packet-filtering packages) as the number of total rules increases. This degradation is independent of the number of *active* rules in traditional firewalls, and occurs because all the policies have to be present at the firewall. In STRONGMAN, the number of rules the enforcement point has to consider at any time is independent of the number of rules it may potentially have to enforce.

The key observation here is that not all users can (or do) access the same enforcement points at the same time; our architecture takes advantage of this fact, by only instantiating rules as-needed at an enforcement point. The rules are limited in our system to those needed to grant access to users actually requesting access. Thus, the security-related expended resources follow more closely the actual communication and transaction patterns of the network. Furthermore, only a small subset of rules (those provided by the user) need to be considered with each independent request, making processing cheaper than otherwise.

4 Related Work

Traditional firewall work ([5, 17, 21, 16, 7, 19]) has focused on nodes and enforcement mechanisms rather than overall network protection and policy coordination.

In OASIS[11], policy coordination is achieved with a role-based system where each principal may be issued with a name by one service, on the condition that it has already been issued with some specified name of another service. Event notification is used to revoke names when the issuing conditions are not satisfied, thus revoking access to services that depended on that name. Credentials are limited to verifying membership to a group or role, and OASIS uses delegation in a very limited way, limiting decentralization.

Firmato's[1] "network grouping" language is locally customized to each managed firewall. The language is portable, but limited to packet filtering. It does not handle delegation or different, interacting application domains. Policy updates force complete reloads of the rulesets at the affected enforcement points, and the entire relevant policy ruleset must be available at an enforcement point. This causes scaling problems with respect to the number of users, peer nodes, and policy entries. A similar system [12] covers additional configuration domains (such as QoS). Differences are the policy description language and the method by which the rule set is pruned for any particular device. Other work in the same vein is described in [8] and [18].

Another approach to policy coordination [9] proposes a ticket-based architecture using mediators to coordinate policy between different information enclaves. Policy relevant to an object is retrieved by a central repository by the controlling mediator. Mediators also map foreign principals to local entities, assign local proxies to act as trusted delegates of foreign principals, and perform other authorization-related duties. Coordination policy must be explicitly defined by the security administrator of a system, and is separate from access policy.

[4] proposes an algebra that allows combination of authorization policies specified in different languages and issued by different authorities. The main disadvantage is the assumption that all policies and (more importantly) all necessary supporting information is available at a single decision point, a difficult proposition even within the bounds of an operating system. Our observation here is that in fact the decision made by a policy engine can be cached and reused higher in the stack. Although the authors briefly discuss partial evaluation of composition policies, they do so only in the context of their generation and not on enforcement.

The NESTOR architecture [15] defines a framework for automated configuration of networks and their components. NESTOR uses a set of tools for managing a network topology database. It then translates high-level network configuration directives into device-specific commands through an adaptation layer. Policy constraints are enforced by dedicated manager processes, which pose scaling problems. This approach has difficulty with decentralized administration and separation-of-duty concerns, due to its view of the network through a central configuration depository.

5 Concluding Remarks

STRONGMAN is a new security policy management architecture. Its approach to scaling is local enforcement of global security policies. The local autonomy provided by compliance checking permits the architecture to scale comfortably with the Internet infrastructure.

Our distributed firewall implementation on OpenBSD was used to quantify some benefits of STRONGMAN. As we have shown in Section 3.2, this implementation has higher throughput and better scalability than a baseline firewall constructed using IPF. It accommodates considerable complexity in policies: the policy compliance checker composes policy rules into a coherent enforceable set for each enforcement point, and lazy instantiation reduces the state required at enforcement points. The removal of topological constraints in firewall placement facilitates other Internet protocols and mechanisms.

STRONGMAN is the first architecture for providing scalable access control services. Security enforcement is pushed to the endpoints, consistent with end-to-end design principles. Since the enforcement points are coupled only by their use of a common global policy, they possess local autonomy which can be exploited for scaling.

Among our goals for future work are experiments with a larger scale deployment, validating lazy evaluation on real traffic, and extending the uses of our system with new application-specific policy languages.

Acknowledgements

We would like to thank Daniel Hartmeier for permission to use Figure 17, which was taken from [10].

References

[1] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: a novel firewall management toolkit. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 17–31, May 1999.

[2] S. M. Bellovin. Distributed Firewalls. *;login: magazine, special issue on security*, November 1999.

[3] M. Blaze, J. Feigenbaum, J. Ioannidis, and A. D. Keromytis. The KeyNote Trust Management System Version 2. Internet RFC 2704, September 1999.

[4] P. Bonatti, S. D. C. di Vimercati, and P. Samarati. A Modular Approach to Composing Access Policies. In *Proceedings of Computer and Communications Security (CCS)*, pages 164–173, November 2000.

[5] W. R. Cheswick and S. M. Bellovin. *Firewalls and Internet Security: Repelling the Wily Hacker*. Addison-Wesley, 1994.

[6] D. D. Clark. The Design Philosophy of the DARPA Internet Protocols. In *Proc. SIGCOMM 1988*, pages 106–114, 1988.

[7] M. Greenwald, S. Singhal, J. Stone, and D. Cheriton. Designing an Academic Firewall. Policy, Practice and Experience with SURF. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, pages 79–91, February 1996.

[8] J. D. Guttman. Filtering Postures: Local Enforcement for Global Policies. In *IEEE Security and Privacy Conference*, pages 120–129, May 1997.

[9] J. Hale, P. Galiasso, M. Papa, and S. Shenoi. Security Policy Coordination for Heterogeneous Information Systems. In *Proc. of the 15th Annual Computer Security Applications Conference (ACSAC)*, December 1999.

[10] D. Hartmeier. Design and Performance of the OpenBSD Stateful Packet Filter (pf). In *Proceedings of the USENIX Annual Technical Conference, Freenix Track*, pages 171–180, June 2002.

[11] R. Hayton, J. Bacon, and K. Moody. Access Control in an Open Distributed Environment. In *IEEE Symposium on Security and Privacy*, May 1998.

[12] S. Hinrichs. Policy-Based Management: Bridging the Gap. In *Proc. of the 15th Annual Computer Security Applications Conference (ACSAC)*, December 1999.

[13] J. D. Howard. *An Analysis Of Security On The Internet 1989 - 1995*. PhD thesis, Carnegie Mellon University, April 1997.

[14] S. Ioannidis, A. Keromytis, S. Bellovin, and J. Smith. Implementing a Distributed Firewall. In *Proceedings of Computer and Communications Security (CCS)*, pages 190–199, November 2000.

[15] A. Konstantinou, S. Bhatt, S. Rajagopalan, and Y. Yemini. Managing Security in Dynamic Networks. In *Proceedings of the 13th USENIX Systems Administration Conference (LISA)*, November 1999.

[16] B. McKenney, D. Woycke, and W. Lazear. A Network of Firewalls: An Implementation Example. In *Proceedings of the 11th Anual Computer Security Applications Conference (ACSAC)*, pages 3–13, December 1995.

[17] J. C. Mogul. Simple and flexible datagram access controls for UNIX-based gateways. In *Proceedings of the USENIX Summer 1989 Conference*, pages 203–221, 1989.

[18] A. Molitor. An Architecture for Advanced Packet Filtering. In *Proceedings of the 5th USENIX UNIX Security Symposium*, June 1995.

[19] D. Nessett and P. Humenn. The Multilayer Firewall. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, pages 13–27, March 1998.

[20] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in System Design. *ACM Transactions on Computer Systems*, 2(4):277–288, November 1984.

[21] D. Sherman, D. Sterne, L. Badger, S. Murphy, K. Walker, and S. Haghighat. Controlling network communication with domain and type enforcement. In *Proceedings of the 18th National Information Systems Security Conference*, pages 211–220, October 1995.