

Automatic Management of Network Security Policy¹

J. Burns, A. Cheng, P. Gurung
S. Rajagopalan, P. Rao, D. Rosenbluth
A.V. Surendran
Telcordia Technologies, Inc.

D. M. Martin, Jr.
University of Denver
dm@cs.du.edu

Abstract

This paper describes work in our project funded by DARPA Dynamic Coalitions program to design, develop, and demonstrate a system for automatically managing security policies in dynamic networks.. Specifically, we aim to reduce human involvement in network management by building a practical network reconfiguration system so that simple security policies stated as positive and negative invariants are upheld as the network changes. The focus of this project is a practical tool to help systems administrators verifiably enforce simple multi-layer network security policies. Our key design considerations are computational cost of policy validation and the power of the enforcement primitives. The central component is a policy engine populated by models of network elements and services that validates policies and computes new configuration settings for network elements when they are violated. We instantiate our policy enforcement tool using a monitoring and instrumentation layer that reports network changes as they occur and implements configuration changes computed by the policy engine.

1. Introduction

Lack of security is one of the primary obstacles in fielding many technologies in both commercial and DoD networks. The piece-meal and ad hoc way in which firewalls and other security elements are typically administered makes it difficult to manage networks in such a

way that desired security policies are upheld as the network changes. Moreover, the scope of management is rapidly exceeding human capabilities because of the acceleration of changes in technology and topology. Network management tools are needed to automate management of networks containing many firewalls in dynamic environments. It is becoming necessary to enable network elements to adapt to change by reconfiguring as appropriate without human intervention. The challenge is for these network elements to know the right reconfiguration so that the appropriate security policies are upheld preventing illegitimate users from gaining access. This project focuses on management of configurations of network elements so that stated policies can be upheld.

1.1. Security Policy Administration and Network Management

While technologies for building large-scale networks and network services have advanced dramatically, creating new vulnerabilities and opportunities for complex attacks, no significant new ideas or principles have emerged for network management, and especially not for security management. Existing tools have been designed for static security and are inadequate to meet the current demands of user mobility and diversity requiring frequent and error-prone reconfigurations. Furthermore, there are no tools to verify the correctness or composability of scripts. Administrators, balancing the demand of users for new services with the security vulnerabilities that the new services can cause, must make decisions based on uncertain and

¹ This material is based upon work supported by the Air Force Research Laboratory under Contract F30602-99-C-0182. Contact: S. Rajagopalan, sraj@research.telcordia.com
©2001 Telcordia Technologies, Inc.

rapidly changing information about the networking environment. This generally leads either to over or under management of resources.

One of the specific goals of this work is management of security configurations in networks that span multiple administrative domains. This has become a real need in today's world of temporary coalition between (otherwise adversarial) corporations. This is also true within networks of large multi-national organizations that are often administered by different people with different views of privileges and responsibilities. A paradigmatic example is the situation of two connected firewalls, each of which has a local security policy (administered by a human perhaps). Even if each firewall correctly implements the local policy, the interconnection of the two firewalls may violate a global security policy that neither firewall can detect by itself. We need to be able to reason about a large network to verify whether the totality of the local configurations upholds or violates global security goals.

Simply put, our goal is to be able to answer questions such as "Can these two sub-networks be connected to each other without violating either security policy?". Such questions cannot be answered today with any reasonable degree of certainty. Indeed, in most cases, even the principles that should guide such important decisions are not clear.

1.2. Overview of the project

The challenges that we are addressing in this project include:

- (a) systematic methods for evaluating and monitoring security properties of large-scale networks,
- (b) tools for managing configurations of network elements such as firewalls, switches, routers, application servers, etc. in a large network, and
- (c) the technology for networks to self-reconfigure automatically in response to changes in the network while maintaining global security properties.

Efforts to develop more sophisticated firewalls and to ensure the protection of end-hosts and the privacy and integrity of end-to-end communications, lack the ability to address network wide security policy considerations. We propose to develop and demonstrate ways to automate the enforcement of network-wide policy by making security configuration

management dynamic and responsive. Our approach builds upon a recent paradigm shift within the security policy community from expressing policy questions as unstructured combinations of access policies and ad hoc prescriptions for network elements, to expressing network security policy questions purely in terms of access (both positive and negative) to applications and network services. Given a network, we want to verify that the desired access is enabled and the undesired access is verifiably denied. By building a computationally efficient framework for checking whether security policy is being upheld, avoiding the state explosion problem with traditional formal verification approaches, we make the automation of policy administration feasible. Finally, we will localize the problem of maintaining policy using automatically generated primitive constraints on configuration parameters of network elements that can be checked locally and efficiently.

1.3. Scope of the project

The ultimate long term goal of this project is to create a practical infrastructure that enables the network to become self-configuring so that necessary network-wide reconfiguration is initiated by the change within the system itself, and not by the coordinated actions of teams of human operators. To allow us to achieve tangible results in reasonable time, the present project focuses on some specific issues that we feel are key to success in the larger plan and for which there is a clear path to generalization. The particular problem we consider is the one that systems administrators face on a daily basis: how do we rapidly determine that a given network configuration is compliant with specified security policies and how do we provide a management aid to manage change correctly in the face of ever-growing complexity. We have several systems administrators on our team who are guiding the development toward a prototype which is both usable to systems administrators immediately and also serves as a substantial basis for further work.

Our highest priority has been minimizing the "gap" between stated policy goals and the actual enforcement of these policies. Many of our design decisions were influenced by the lack of verifiable enforcement mechanisms for certain security phenomena. An example of this is Denial of Service attacks. Since we do not have in our project any viable mechanisms to prevent

such attacks, we do not allow specification of policies that would require such enforcement mechanisms.

In this project, we focus on policies for abstract interconnection of network services between sub-networks and we will later discuss how our policy language might be extended into other aspects of security policies like authentication and user-level access of fine-grained objects such as files. Our reason for not addressing these issues directly in our project is that, in our opinion, most of the existing work does in fact focus on these aspects (see, for example [12]). Another reason for our choice is that security mechanisms such as authentication and object access control are predicated on the assumption of a secure basis. For example, the standard password-based mechanisms for logging into servers is completely vulnerable to attacks that gain root access on the server using some subtle interactions between common protocols. We believe that any security infrastructure should be consistent all the way from the bottom up to the application. While our project does not address the absolute bottom layer of the infrastructure (i.e. physical security), our security universe encompasses all layers above physical and we simply deal with as many issues as we can. In particular, while we deal with dynamic phenomena such as loss of physical or link layer connectivity, configuration changes in network devices, and formation of coalitions, we do not deal with sessions, users, authentication, encryption protocols, browser policies, etc. that cannot be controlled statically. Wherever our work intersects with these other security policy management artefacts, we provide natural interfaces that would allow anyone to add new capabilities to the basis that we provide. These interfaces are in the form of models that can represent solutions to these problems. By this design decision, all stateful process & applications are beyond the scope of this work. Because we do not model stateful behavior (because we cannot do not do and do not want to do model checking) we instead make conservative stateless extensions of potentially state-ful policies such as “Browsers should not access culturally diverse material.” It is most important to point out that our stateless approach also means online handling of Denial-of-service type attacks is beyond our scope. Another goal of this work is to build a platform that allows us to use as much of the existing work as possible. IPSec is an important example of this. Since IPSec is a well-defined protocol

we treat IPSec (and by implication VPNs based on IPSec) as another service that needs to be modeled. IPSec policies become parameters of their models. Effectively, our design allows us to VPNs as virtual edges in the network. This paper is organized in seven sections. Section 2 describes the general problems regarding security policy and the contexts in which these problems arise. Section 3 describes our view of the desirable features that any security policy specification language should have. Section 4 describes the policy engine and Section 5 gives an overview of the network instrumentation tool that we use. Section 6 compares our work with other related work. Section 7 gives the status of the project and our work plan for the immediate future.

2. System Architecture

The large size, heterogeneity, and distributed nature of networks give administrators a large number of degrees of freedom to consider when configuration changes need to be made. The proper functioning of the network as a whole is a result of the coordinated configuration of multiple network elements whose interaction gives rise to the desired behaviors. The number of options to consider and the complexity of the interactions and interdependencies between various protocols and applications make it a very hard task to reason about configuration changes. Nevertheless, large networks due to their dynamic nature demand that such configuration changes be made frequently and that such changes be implemented both quickly and correctly.

2.1. A new notion of Policy

Often, the type of configuration changes that need to be made to networks are homeostatic in nature, by which we mean they are responses, triggered by changes of network state, that are intended to maintain or recover some network property. We call such a property that has to be maintained an *invariant*. Due to the high dimensionality and complexity of configuration space, it is much easier for administrators to state the invariant to be maintained, than it is for them to identify or even recognize those network configurations which satisfy the invariant. Security invariants are the policy goals that we want to enforce in networks. Notwithstanding syntactic similarities between our invariants and

many contemporary notions of security, we will show in this work that the meaning we ascribe to our “policies” is quite different.

One of the main sub-goals of this work is to simplify the network administrators’ task of performing these homeostatic changes to the network. We do this by separating the definition of invariants (policy *specification*) from the task of finding the changes to network configuration that need to be implemented in order to maintain the policy (policy *management*). Policies should define the intent of the administrator independently of the mechanisms used to implement the policy. Policy-based network management should implement policy by reconfiguring the network appropriately. Separating policy specification from policy management offers two interrelated benefits: robustness of policies with respect to changes in technology and network topology; and the ability to automate significant parts of policy management. These two are discussed below in detail.

Use of high level description of policy, divorced from implementation, creates stability both with respect to underlying changes in technology used within networks in general, and with respect to topological changes in the specific network. Both of these aspects of networks are highly dynamic. The growing use technologies such as layer 3 switches, wireless lans, Virtual Private Networks, and Network Address Translators has been accompanied by a great change in the way networks are used and, hence, the scope of security policy concerns. The increased use of dynamic coalitions both within and between organizations brings to the forefront difficult issues regarding composition of networks. The need for policy negotiation between collaborators who do not want to reveal details of their internal network topologies implies topology-independent policy definitions. Multi-national corporations are frequently making use of networks consisting of multiple sites connected via the Internet. Networks in which new logical sub-networks are not necessarily topologically related to the parent network, cannot use policy inheritance based on topological relationships, yet these networks still require a common policy to be effective over these mutually disconnected networks. Currently, the most commonly used tools for implementing security policies, including packet-filtering rules for firewalls, make explicit use of IP addresses in policy specification.

Security policy specifications that are linked to the details of network topology in this manner, are rendered meaningless in light of the technological developments which create new flexibility in network topology. By providing a language in which the intent of security policy can be expressed, independence of both the methods available for changing network state, and the specific topology of the network, can be achieved.

2.2. Automation of Policy Management

Two areas in which automation tools would be of great use to administrators of security policies are: reasoning about policies where the goal might be, for instance, detecting inconsistencies between a new security policy rule and an existing policy; and in finding network configurations which satisfy a set of policies. Security policies expressed in terms of firewall configuration are complicated and difficult to reason about, both for man and machine. Information that might aid this reasoning process such as the history of firewall configurations, the rationale and the intent of the administrator for past changes, is typically unrecorded. The ad-hoc fashion in which new rules are introduced into firewall configurations, accommodating individual cases often without consideration of policy implications for the larger network, results in the firewall configuration being an obscure representation of policy, but the only one available to the administrator at a later date. By using policies that explicitly express the intent of the administrator we eliminate the difficult problem of inferring the current policy intent from the network state, which greatly simplifies reasoning about policy changes. The number and diversity of configuration options that can be explored will be greatly expanded by automating the search of configuration space and the consistency checking functions, which are very time consuming when done manually.

2.3. Our Approach

In any practical approach to design of network administration tools, what we can and cannot control in the network must be considered. We must avoid the assumption that our automation tool or the administrator has complete control of the network under consideration. While policy may be satisfied by many configurations, not all of these configurations may be implementable

even in an automated fashion. In cases where the control available is insufficient to implement the given security policy, we would like to know this fact too, something that is beyond most current approaches. Our approach is to capture all the elements of configuration that are beyond the control of the policy administrator as part of the network topology, and recast the policy implementation problem as one of finding configurations that satisfy both the policy and the topology. This allows us to smoothly merge what can be controlled with what cannot.

A practical approach demands that we recognize that there is a tradeoff between the computational cost of search and the optimality of configurations satisfying policies. At one extreme we could maintain the state of every application and service on the network so that we can check every composite state of the network for violation of policy. This approach suggested by Schneider [16] and others becomes rapidly untenable as the network and the number of applications grows. On the other extreme, having a fixed network with pre-checked applications is unreasonably restrictive in today's environment of dynamic networks. The crucial aspect of our approach which permits us to avoid the combinatorial explosion of the stateful approach and yet allows enough flexibility, is the distinction we draw between configurations of network elements, which are bounded in number and can be known at compile time, and application state, which need not be bounded. By modeling network elements as having a list of configuration parameters that can be assigned values from a bounded list, it becomes possible to pre-compute some or all the configuration settings for the network, making the system responsive and expediting the process of policy verification. The most important result of this approach is that it enables us to meaningfully reason about the consequences of configurations of all the network elements together rather than on a per-element basis. Within our framework, policy administration is equivalent to finding the "right" configuration settings for all the network elements relative to the given security policy and topology.

Even though we have reduced the search space by restricting ourselves to configurations rather than states, this might still be a very large space that may be infeasible to search in practical terms. The practical feasibility of our approach also depends in part on segregating out those security issues which are sufficiently orthogonal to the problems of

maintaining security policy, that they can be dealt with independently by other programs. We avoid issues of extremely high dynamism, leaving out of our consideration aspects of the network, such as routing protocols, which give rise to extremely dynamic behavior in the network, aiming only to build a system that can respond in a reasonable amount of time to changes in the network. We do not consider the possible presence of covert channels or the problem of discovering defects in implementation or undocumented behavior such as viruses though these can be entered into the system via the model. This project is not aimed at creating an intrusion detection (ID) device though some part of its functionality (monitoring) has some ID capabilities. We imagine however that ID information can be used in our system using models of attacks but that will not be the focus of our work. Finally, we do not address quality of service or reliability issues.

2.4. Architectural overview

We now describe the main architectural components of our research prototype. We point out that all these components may exist either on one machine or distributed over a network.

2.4.1 Management Console

The **Management Console** is the central component that interacts with all the other components of the system. To maintain compatibility with industry standards, the management console and all other components talk to each other in a dialect of XML. The functions of the console are to:

- report on the current state of the network being administered: Apart from displaying network information in a human-comprehensible form, it also displays network changes that have been detected, and displays warnings as necessary.
- accept control information from systems administrators: Leveraging graphical user interface (GUI) techniques where possible the management console provides an intuitive input interface, resorting to table entry only where this is not feasible or convenient. Through this interface, the user can also attach comments to network elements where appropriate.

- store persistent information regarding network topology: We have an XML variant capturing the above information.

2.4.2 Topology Interface

The **Topology Interface** component allows the system administrator to specify the connectivity characteristics of the network. As noted earlier, valid network configurations are constrained by both network security policy and network topology. We define the topology input to be the aspect of connectivity that *cannot* be changed by the automatic reconfiguration mechanisms. For example, IP address blocks for enterprise networks are pre-assigned and cannot be changed easily. Some topological facts can be discovered automatically by systems providing network monitoring and control, such as the NESTOR system developed at Columbia University. Other topological facts will require human input. The information required from the user to specify the network topology can be roughly divided into four sections:

1. Physical Connectivity, deals with how the elements in the network are physically connected (analogous to level two in the standard network hierarchy). Connection at this level is between pairs of interfaces, normally on different network elements.
2. IP address mapping specifies how IP addresses are associated with network element interfaces (level three).
3. Routing Tables, one or more for each router, specify the rules for directing transit packets (again level three). Each entry in the routing table consists of an input interface, packet information (source and destination addresses and ports) and output interface. Each router can also have an optional mode variable enabling a switch in router behavior associated with a change in the operating mode.
4. Element type, describing what type of network element each node is, e.g. whether it is a router or switch or workstation.

2.4.3. Security Policy Specification Interface

The **Security Policy Specification Interface** is where security policies are stated. The job of the interface is to parse policy statements for syntactic correctness and forward them to the Management console. The syntax and accompanying semantics of the policy language are discussed in detail in Section 3.

2.4.4 Policy Engine

The **Policy Engine** is the “brain” of the management platform. The policy engine’s job is to (a) discover inconsistencies in the given set of policy statements, (b) verify whether the policy statements, network topology, and models of network elements are mutually consistent (implying that security policies are being upheld), and (c) compute reconfiguration actions that would restore the policies in the event of a change in the network. We conceive the policy engine to be a purely logical component that has no data of its own but rather derives consequences of the facts as specified in the topology and models and checks if they contradict the goals as embodied in the policy inputs. One of the greatest challenges in the design of the policy engine will be performance: how do we ensure that the policy engine can respond in a reasonable amount of time to a request to, say, form a dynamic coalition? The **Model Repository** which is the most important sub-component of the policy engine stores the representations of the various network elements in the network being administered. It provides interfaces to add, edit, and delete models. The policy engine and model repository are described in section 4. The job of reconfiguration would be incomplete without a means to monitor and configure network elements based on the output of the policy engine. We use Columbia University’s NESTOR platform for this purpose. Section 5 describes the architecture of NESTOR and our use of it in our system.

3. The Security Policy Language SPL

This section describes the major features of our approach to policy specification.

3.1. Separation mechanisms in SPL

The unique aspects of our project are reflected in SPL as intentional separations between concepts that are usually combined in current practice.

3.1.1 Separating topology from identity

In practice, the security policy of a group of nodes does often coincide with their common IP subnet number, but it is important to allow a policy author to express divergence from this pattern without having to maintain independent

policy statements that are related only in the author's mind. In order to separate network topology from identity, the author may specify a "group" consisting of a set of arbitrary identities which can then be used as an endpoint in a flow specification. This encourages the author to express policy about the group of interest, rather than forcing the author to think of policy as a function of a specific addressing scheme.

3.1.2 Separating filtering goals from filtering mechanisms

In order to specify that a flow should be blocked or permitted, the author identifies the flow by specifying endpoints and other flow-specific characteristics such as a protocol, port number, or endpoint user. The network agent responsible for enforcing the blocking rule (i.e., a firewall, router, or end host) need not be explicitly associated with the flow, but can be inferred

The authors' interests in policy specification can vary widely. Alice may be interested only in providing web connectivity between two network areas, while Bob may wish to ensure that his internal hosts are unable to participate in the election of a new 802.3 spanning tree root. To accommodate this disparity, flow specifications are coupled to network layers. Alice would specify "allow application/web connections between these areas", while Bob would specify "disallow Ethernet/spanning tree updates between those areas." Rather than limit all flow specifications to properties visible at the IP layer, the author targets the specification to the layer of interest.

Access control policies such as "Alice should be able to log on to the Accounting workstation" are also expressed as flows in our system. In this example, the policy permits a login flow between the identities "Alice" and "Accounting". Forming more complex identities such as "Alice at home" and "Alice at her office" involves the straightforward logical combination of user-defined groups: Alice at home is the intersection of the Alice identity with the network location of her home system.

3.1.3 Separating security policy specification from the network model

We fully expect that any logical model of a network constructed today will be soon obsolete: network servers mature, attackers become more sophisticated, and the network topology changes. We want to avoid the situation where a "policy"

in the field says too much or too little about the author's intent. For example, specifying that inbound TCP SYN packets should be dropped is an antiquated way of enforcing a policy forbidding unsolicited inbound connections. This example actually specifies both too much detail and too little intent. The policy language encourages the specification of intent by the author by decoupling the policy language from logical models of the network. When new attacks are discovered, the correct response is to inform the logic engine of the attack (by distributing new network models) and run the new models on the unchanged policy. In contrast, current practice relies on the policy author to learn about the new attack and add new filtering rules that describe it.

3.1.4 Separating goals from facts

A written policy distinguishes between policy goals and facts about the system being described. A goal is a request that configurable elements be set to accommodate the desired outcome, while a fact is a simple declaration of some property.

This distinction is important for two reasons. First, no analysis engine would actually be able to account for all existing and yet-to-be-invented networking devices. The policy author can use facts as an escape hatch to describe the behavior of unknown network devices in terms that are relevant to the policy engine. Tunneling is the prototypical example of this. Any application with sufficient programmability may end up (unintentionally) acting as a tunnel. When the author notices a tunnel, only the facts regarding the tunnel's network effect need to be registered. Secondly, the goal/fact dichotomy provides a natural framework for making "what-if" analysis.

3.1.5 Separating along administrative boundaries

Large networks are administered hierarchically. Accordingly, a policy module can delegate (import) independently maintained subpolicies. The importing operation is safe because of two precautions. First, the subpolicy and superpolicy occupy different namespaces. It is therefore impossible for an import operation to accidentally change the meaning of names defined in the superpolicy (such as the membership of a named group). If such capability is desired, then the superpolicy must explicitly allow it, *and* the subpolicy must be written to reach outside of its own namespace.

Second, the superpolicy can cause certain rewritings of the subpolicy during import. This gives the superpolicy authority over interpretation of subpolicies.

3.1.6 Separating the known from the practical

We allow an author to label a flow specification with a defcon level so that it only influences engine decisions while running at the same or greater defcon level to capture the perceived threat.

Such transitions in the face of perceived threat between global operating modes of a network are thus representable as a matter of policy. This mechanism also allows highly sophisticated attacks to be represented at high defense levels without forcing the security administrator to constantly ignore unlikely threats.

The inputs to our tool are a description of the network and the administrator's goals. The tool's output is:

- *At least*: an assurance that the supplied description is consistent with the administrator's intention, or an explanation of an inconsistency.
- *Ideally*: an assurance of consistency along with settings for network components that enable the network to behave in the intended manner, where these settings are chosen from those left unbound in the input; or, an explanation of an inconsistency.

For lack of space, we give a small illustrative example of security policy. Suppose our network has sub-networks called Corporate, Research, and Guest. An example security policy for Guest could be "All *guest* nodes have access to *mail service* but should not access *database applications* in the *Corporate sub-network*." The terms *guest*, *mail service*, *database applications*, and *Corporate sub-network* are abstract tags that can be mapped to actual network entities at policy validation time. Each sub-network can have its own policies in addition to any policies inherited from its parent network. In the next section, we describe the policy engine.

4. The Policy Engine

In this section, we describe the use of a policy engine for the administration of security policy in a large heterogeneous network. The policy engine needs to handle in an automated fashion the large space of possible states, and respond to

the continuous changes in the network configuration quickly and correctly. This automated management uses the security policy to determine *good* states of the network. Using declarative reasoning the policy engine avoids exhaustive enumeration of good states. As the policy for a large network is jointly upheld by the network elements, adapting to change requires the policy engine to find new configurations for all these elements which together satisfy the policy. We use models of network elements to capture the normal functions and flaws in design and implementation of devices and software. Models aid in detecting security breaches caused by discovery of previously unknown defects or by introduction of new network elements. A change in a model will require verification and possible reconfiguration by the policy engine. A two step process of configuration followed by validation is employed by the policy engine. The configuration phase takes a partially configured network and fills in the missing parts of the configuration. The combination of the completed configuration and the network topology (henceforth termed as Network State) is then validated against the policy. In section 4.1 we describe models, configurations and the working of the policy engine and in section 4.2 we formalize device models and composition, network configuration and policy validation. The policy engine handles feature interaction caused by interconnected networks of devices by model composition. The policy language and policy engine are not limited to network descriptions. However in order to easily illustrate the means of policy enforcement we use examples gleaned from the network management world.

4.1. Topology, Models, and Configurations

The policy engine has to uphold the security policy in response to changing network states. A change in topology, or element configuration or a change in the network element or some service provided by an element could result in some rule in the security policy being violated. A rule in the policy may be violated in either of two ways. First, a rule may be contradicted by another policy rule. For example, let *Patent Lawyers* be a sub *tag* of *Lawyers*. Consider a policy definition that has the following two rules: (i) *Lawyers* should not have access to the *Research* database server and (ii) *Patent Lawyers* should have access to the *Research* database. (i) and (ii)

intrinsically contradict each other as every *Patent Lawyer* is a lawyer. Every network state will have to violate at least one of them. The network administrator has to be informed that the violation is due to intrinsic inconsistency of the rule set. This consistency check is undecidable in general, and our policy engine tries to do a good job of detecting as many of the inconsistencies as it practically can.

Second, the network cannot uphold a given policy rule. This may happen in three different ways.

- (a) A service mandated by it is unavailable because the network cannot provision it. The policy engine could discover no path between two machines to provide a mandated telnet and report it. We rely on the systems administrator to remedy these violations.
- (b) A service mandated by it is unavailable because a device such as a router, switch or a firewall is misconfigured. The policy engine may find a remedy for some such instances—such as relaxation of some firewall rules to provide connectivity while making sure that policy rules currently upheld are not violated. For other cases we rely (as in case (a)) on the systems administrator.
- (c) A service forbidden by it is available. Here the policy engine must discover every way in which the service is provided, and secondly block the service, for instance by tightening a few firewall rules till the rule is upheld.

The policy engine, based on the current network state consisting of a description of network element interconnections (topology) and configuration, determines for each policy rule the availability of the service mandated/forbidden by it. The policy engine needs “descriptions” of these network elements and service—called *models* and notions of device and service interaction. These models should be policy independent, and capture the intrinsic properties of devices and services. Some desirable properties of such models are:

- conciseness for easy management.
- they should be formal and machine-readable to **allow automated reasoning**.
- **composability** of devices .
- **vendor-independence**.
- **human readability** for sanity checks, and to avoid the need for separate

consistent set of human readable descriptions

- **extensibility** to allow technological innovations or new exploits.

Often, printed manuals and human experience are the only ways of understanding devices. Typically, devices have functionality that is unknown or ignored. Interconnected suspicious (but functional) devices could pose security risks by violating policy in subtle ways due to interaction between unknown/ignored features. We need precise knowledge of the properties of devices, and tools to represent and precisely reason about networks of devices to prevent such security holes. This knowledge, represented as a set of tunable parameters offered by a model of a device is called its *configuration*. Based on the security policy, network topology and configuration and device models the policy engine achieves the following dual goals:

- Given a complete network state, it checks the network for policy compliance. For a non-policy-compliant network it reports all violations. For e.g., paths allowing forbidden telnets.
- Given a partially defined state it derives a complete policy compliant state.

In order to generate the network state or validate the current network state from the policy, the policy engine generates representative tests for compliance with each policy rule. It analyzes results of these tests and derives new configurations to remedy non-compliance. It repeats this until it either has a policy-compliant configuration or is stuck in a vicious cycle of generating configurations that it generated before.

In order to reduce the complexity of configuration generation and to effectively handle the dynamic nature of the network the policy engine uses *abstraction*. For instance, sets of routers reconfigure themselves by communicating with one another using existing well-known protocols. Although the policy engine could replicate this process, it will *not* recreate this mechanism. We will take this ability of routers for granted and generate configurations of firewalls conservatively; i.e., we will guarantee that no router reconfiguration will turn a network from a policy compliant network to a non-compliant one. This reduces the computation that the policy engine needs to perform.

We use abstraction to deal with the complexity of the devices. The policy engine uses the abstraction provided by models to

achieve a compromise between complete knowledge of devices and a usable representation. Device vendors or certifying authorities must supply models for new devices and make existing devices conform to standards. We need techniques to test and verify models against real devices, otherwise vendors could sneak in features absent from their models into their devices.

Configuration generation is hard because devices differ significantly from one another and have very different configuration parameters. For example, IPChains firewalls and checkpoints firewall-one have different models. Abstractly a firewall is a router coupled with a packet filter. To generate routing rules in the firewall, we analyze the connectivity of the entire network and identify the subset of the network for which the firewall in question acts as a choke point. Additionally we partition this subset and associate each partition with an interface of this firewall. Now we project the set of these policy rules onto these partitions and generate the appropriate firewall rules.

The policy engine can discover different solutions to the same problem in differing circumstances. For instance, a service cannot be blocked by a firewall. First, if possible add a static route to appropriate routers to drop packets providing the service. Given a sufficiently rich model, this can be automatically derived. Second, the server can block the service to this particular client, if the model of the service permits it. Last, the network topology can be altered to block this service.

The implementation of this policy engine is based on a combination of engineering techniques and sound scientific principles. In the following subsection, we will explore the theoretical foundations underlying the policy engine.

4.2. Formal description of NEs and Services

A device can be described by a set of rules that govern packet transfer and transformation. Each such rule has the form:

$$\langle \{ \langle P_{i1}, O_{i1} \rangle, \langle P_{i2}, O_{i2} \rangle, \dots, \langle P_{in}, O_{in} \rangle \}, S \rangle \Rightarrow \langle \{ \langle P_{o1}, O'_{o1} \rangle, \langle P_{om}, O'_{om} \rangle \}, S' \rangle$$

In the above rule, P_{ik} 's and P_{ok} 's are input and output packets, O_i 's and O'_m 's are ports where packets are input and output respectively. S and S' are states of the device before and after communication. This form allows description of all kinds of packet transfers and transformations

that are not time-sensitive. This generality makes it difficult to reason about devices and provide required by policies. Often it suffices to reason about connectivity to analyze availability of services to groups of users. For instance, instead of modeling all the details of a file server, we can model file service as the ability to reach it using a file access protocol such as *nfs*. Telnet is modeled as *tcp-reachability* on Port 23.

```
available(telnet, From, To) IF
    tcp_reachable(From, AnyPort, To, 23).
```

A service can become available by composing two or more different services. At a location from which a ftp service is inaccessible, ftp can be accessed indirectly by telnetting to a different location.

```
available(ftp, From, To) IF
    available(telnet, From, Intermediate)
    AND available(ftp, Intermediate, To).
```

We can conservatively say that *telnet* allows everything by saying:

```
available(AnyService, From, To) IF
    available(telnet, From, To).
```

To study the connectivity and security properties of networks, devices such as routers and firewalls can be assumed to be in a steady state, i.e, their state is not altered by packets they consume/transfer. Such devices can be modeled as a set of edges, where the edge relation may look at the packet flowing through the device. The *edge* relation on a switch is written as:

```
edge(Device, In, Out, InPacket,
    OutPacket) IF
    devicetype(Device, switch) AND
    vlan(Device, In, Vid) AND
    vlan(Device, Out, Vid) AND
    OutPacket == InPacket.
```

The above rule says that any packet sees an edge between the interfaces *In* and *Out* on *Device* if they are on the same *vlan*. The model of the switch would consist of the *edge* rule, which is derived from the *vlan* relation. Filling in the *vlan* table configures the switch. A firewall provides a more illustrative example.

```
edge(Device, In, Out, InPacket,
    OutPacket) IF
    devicetype(Device, firewall) AND
    applicable_rule(Device, In, Out,
    Packet) AND
    allows(Rule, In, Out, Packet) AND
    OutPacket == InPacket.
```

The model of this device consists of the rules for *edge*, *applicable_rule* and *allows*. The

OutPacket and *InPacket* need not always be the same. Devices can apply transformations to packets before placing them on output interfaces. For instance, a firewall might decide to rewrite the destination of a packet requesting a service to a designated proxy. In addition to transferring packets, devices such as firewalls and routers can offer services such as logging and accounting. These services are also specified by descriptive predicates. Additionally devices can be wrapped by constraints using auxiliary software such as Nestor.

The policy engine derives the behavior of a system by putting together the models of the devices comprising the system. Let us suppose we have to analyze the TCP connectivity of a network. First we have to capture the fact that the physical connectivity present in the network enables TCP connectivity as follows: A packet can travel between the interfaces of devices if they are connected by a *lan* cable.

```
reachable(IA, DeviceA, IB, DeviceB,
         Packet) IF
lan(IA, DeviceA, IB, DeviceB).
```

Secondly, some devices (such as switches, routers and firewalls) forward packets. The *edge* rules above model devices as a set of *dynamic edges*. A packet entering the input interface of a certain device can exit from one of its output interfaces if it sees an edge between two interfaces (determined by its configuration). For a router, the routing rules define the edge relation. Firewalls are more complex and edges depend both on routing and firewall rules.

```
reachable(Port, Device, Dport, Device)
IF
edge(Device, Port, Dport, Packet,
      Packet).
```

We can answer many questions about compliance with policy rule by Computing *reachable**, the transitive closure of *reachable*. These computations should avoid the pitfalls arising from the presence of cycles of the *reachable* relation. Such computations are efficiently done using a tabling technique first used in the XSB deductive database engine (see [14]). The *reachable* relationship is dependent on the header and possibly payload information contained in the Packet, the entry interface on the device chosen by the packet, and the configuration of the device. Different packets could see different *reachable* relationship*.

Service Composition: Novel services can be composed from existing ones. The expressive

ability of the language we use to represent such compositions determines the deductive power of the policy engine. However, expressive ability comes at a computational cost. For instance, allowing *higher order logic* makes many of the questions posed to the policy engine *undecidable*. Composition of different services along different segments of an *access path* can be described easily in our syntax. For instance, we can say that *ftp* is available between two locations A and B if *telnet* is available between A and any intermediate location C, and *ftp* is available between C and B.

4.3. Implementation Considerations

An early version of the policy engine was programmed using XSB Prolog. XSB's computational power is superior to other Prologs because of its foundations in the *well-founded semantics*[14]. We implement deduction mechanisms (equivalent in power to the engine in XSB) in Java to leverage the following advantages:

- Use of Java simplified the inter-system interfaces with our GUI and NESTOR.
- Java supports threads to respond to multiple events for scalability
- Java provides distribution frameworks like RMI for de-centralization.
- Java provides tools for XML parsing.

The ease of expressing device and service models directly impacts the real world use of the policy engine. It is unreasonable to require the users or device vendors to supply definitions of devices in Predicate Logic. Fortunately, these models can be more conveniently described in XML using visual editors. Consequently, in our framework, device vendors provide models for their devices using standard XML metadata techniques. Configuration tools provide systems administrators with self-descriptive GUIs for configuring devices by metadata introspection. These techniques together make model maintenance manageable.

4.4. Policy Engine construction

The policy engine essentially is a deduction engine that computes a transitive closure over a series of dynamic edges. Accessibility to a service is represented as a path. An edge represents traversability of a link as seen by a particular packet and the service that the packet is trying to access at the destination of this edge.

For intermediate edges in paths, the service is just packet forwarding, for the terminal edge the service a service such as *ftp* or *mail_drop*. The edges are dynamic in the sense that every packet sees the network as a graph with a different set of edges, depending on parameters such as the origin and the destination of the path, and the service being accessed. Since the graph is dynamic, the transitive closure must also be dynamically computed – for positive queries, just an existential answer suffices. For a negative query, the answer is a complete reachability tree arrived at by the search of the graph that demonstrates the unreachability of the destination. The policy engine works because computationally this problem of computing the transitive closure is simple. (graphs of size of a few thousand nodes can be handled in fractions of a second, even by interpreted Prolog engines.)

By confining ourselves to stateless computations, we sidestep the complexity issues (exponentially large search spaces) faced by model checking systems. Secondly, the constraints on the expressive power of our policy language disallow potentially expensive definitions of services. Together this ensures that the search space of the policy engine is manageable. Thus, we can focus on static interdependence of services that cause unintended consequences resulting in other services becoming available – a hard problem for humans to tackle.

4.5. Why would this work?

Since logical paradigms have been proposed in the past for security policy administration without finding widespread use, a legitimate question would be what is new here that has a better chance of success. The fundamental answer to this question is that we hope to have learned from these experiences and avoid repeating or reinventing these ideas. Our policy engine is a pure deduction engine and as such all the “intelligence” is provided as input and is encapsulated in the models. This allows us to streamline the process and use as much prior knowledge as possible. The second reason is that, mindful of previous experience, we have consciously reduced the expressiveness of our policy language (for example, by not allowing stateful statements such as “Users must authenticate themselves to get service”) to avoid the need for model checking technology. This allows us to see the policy engine as a simple

search engine in a space that does not suffer from the combinatorial explosion of analyzing composed automata. The third reason is that we have tried to balance simplicity of policy language with the power of the currently available enforcement mechanisms. For example, it would be desirable to have the power to control (via policy specification) the information that is being sent out of secure networks but absent any credible mechanisms to achieve this, we want to deny specification of such properties in our policy language to the extent possible. Finally, there is always the question of how much information should be included in the models. Our answer to this hard question is that this is an inherently human-driven process. A systems administrator could evaluate the output of the policy engine for correctness and completeness which in turn is completely determined by the models provided. We imagine that whenever a policy engine shows consistently poor results that the models of network elements within would be iteratively refined to correct the policy engine’s output. We hope that such iterative refinement would even become a community activity with people with systems administrators sharing models and experiences.

In addition to the difficulties outlined above, the enforcement mechanisms that we do have suffer from the intrinsic fact that our networks are neither finely nor totally controllable. Unlike databases, we cannot impress any transaction semantics on the network and we exploit the fact that security polices are inherently conservative to sidestep this issue.

To complete the loop, we outline the interaction between the policy engine and Nestor. The policy engine controls the network by requesting Nestor to configure devices on its behalf. Conversely, Nestor informs the policy engine of the changes in the state of the actual network. In the next section, we describe the the use of Nestor in our prototype.

5. Monitoring and Instrumentation using NESTOR

The heterogeneity of hardware elements in networks coupled with the need for general policy administration tools in these networks requires a vendor neutral platform through which the network can be monitored and controlled.

NESTOR, a configuration management system developed at Columbia University, is designed to meet this requirement by providing predictable network-level instrumentation and monitoring of common network elements and services such as DHCP, DNS zones, host-based access controls, firewalls, and VLAN switches. For a general discussion of NESTOR and the details of its architecture, see [20]. Systems administrators using NESTOR can access and control managed network elements by manipulating their proxy objects in the NESTOR repository.

6. Comparison with other work

There is now a large body of work related to policy and policy-based management (see [20] for an overview). Due to space limitations, we will only provide a general outline of the differences between our work and those of others. Most efforts on policy and policy implementation ([4],[8],[9]) exhibit confusion between goal and means: they define policy in terms of configuration parameters of network elements such as rules for firewalls. As discussed in Section 3.1, these rules should not constitute policy but a particular implementation of intent. Conventional usage of the phrase “security policy” seldom conveys the intent, but almost always the means to implement and enforce the intent. We turn this on its head by having our security policy convey intent and nothing more. An important early step in policy specification is [6] which concerns itself with generating routing filter rules based on a lisp-like specification language in a logical framework. A large subset of policy-based management research is focused on linguistic issues of Policy. In the literature, policy language is almost synonymous with rule sets (see [3],[5],[22]). Rule-based (event, action) solutions [18] often are too simplistic because network phenomena are highly correlated and implications of remote change have to be derived by composition of all the other configurations that form the context. For example, whether a particular telnet may not be explicitly forbidden by the policy, but such a telnet might allow access to an application on that machine that is forbidden by the policy.

Traditionally (e.g. [4]) “security policy” has always referred to the settings of firewall rules. Firewall-based layered approaches [2][10] try to map security devices to the layers in the architectural design of IP networks. One of the most comprehensive treatments of security policy in networks with many firewalls and

distinct security policies for sub-networks (and an excellent example of presentation) is the Firmato ([2],[10]). Firmato is a firewall management toolkit with: (1) an entity-relationship model containing, in a unified form, global knowledge of the security policy and topology, (2) a model definition language, which is used as an interface to define an instance of the entity-relationship model, (3) a model compiler, translating global knowledge of the model into firewall-specific configuration files, and (4) a graphical firewall rules illustrator. Named entities directly correspond to IP address ranges and hence physical and logical topologies are mutually intertwined. Finally, the Firmato engine does not aim to respond to changes. In Firmato, the connectivity results of a change have to be computed off-line and the engine has to be re-run on the changed input. Our model of the network is dynamic and incremental, thus changes in policy are handled in a manner analogous to topological changes.

Another division of policy enforcement research can be made on the basis of the level in the network stack that the tools operate. Most firewall-based work including Firmato works at the IP level and statements about Level 2 connectivity (such as those relating to broadcast domains and VLANs) or below are not handled. Layer isolation is a good tool to design networks but since adversaries can use any possible combination of tools to compromise the system, security policy enforcement needs to account for all cross-layer vulnerabilities. As long as individual machines can be compromised at any level anywhere, all layers of the stack including Level 2 and physical layers need to be considered.

Moreover, certain aspects of security may be independent of firewalls: switches are relied upon to isolate VLANs at the network layer. Routers have to be relied upon to make sure that the firewalls are the “choke” points of network connectivity as intended. Existence of paths bypassing firewalls has to be discovered.

Network elements and services constantly undergo technological innovation. An instance of such an innovation is “intelligent switches” that are designed to do something “sensible” in response to some device with a wrong IP address plugged into it – sometimes providing connectivity where none was intended by the security policy. The stress in these devices is correct functioning rather than security, and in some contexts the two may be in conflict. Our policy engine based analysis allows us to track

such changes naturally by injecting new models of these elements and services.

Another approach towards a security policy language is SPSL (Security Policy Specification Language) [21]. SPSL is a vendor and platform independent language for specifying communication security policies, such as those controlling the use of IPsec and IKE. It allows security policies to be specified in an interoperable language, stored in databases and consumed by management systems. SPSL is more expressive than Firmato's policy language. However the policy language is geared towards expressing actions like "packet filtering" which is a firewall concept. SPSL allows individual devices to have "security policies" associated with them. These policies are enforced in points called Policy Enforcement Points (possibly firewalls). The weak link in the chain is that the so-called PEPs may not actually be able to enforce policy if they are not the choke points of the network – for instance, L2-connectivity may let traffic completely bypass the PEPs. Moreover, SPSL is very general and perhaps too expressive: one can express almost anything that is currently considered in the security literature. Cryptographic key exchange policies to properties of stateful processes in servers can all be expressed in SPSL. This makes reasoning about sub-policies very hard. In collaborative situations when two networks have to negotiate interconnections based on their policies, it would now become very difficult to know whether the policies are maintained by the interconnection. Another problem seems to be that SPSL does not explicitly address verification of compliance which is essential in automating policy management.

7. Future work and Conclusions

The goals of this project are far-reaching and involve execution of many critical sub-tasks along the way such as modeling and implementation of a monitoring and control platform. The main goal of this project is a research prototype implementing a small-scale completely automatic cross-device reconfiguration platform to maintain security policy amidst change. Such an ambitious goal needs an incremental plan for success. At the time of writing, we have a Linux test-bed with routers, firewalls (Ipchains), and hosts with a sample implementation of the Policy engine populated with models of firewalls, TCP/IP, switches and routers that are directly controlled

by the policy engine via NESTOR. This policy engine performs policy validation and rudimentary configuration synthesis. We also have a sample Graphical User Interface that allows a systems administrator to specify and validate policy statements against topology and configuration inputs. In addition to validation, the interface also supports queries

Our plan is to incrementally grow the platform in each of the main components. In the security policy space, we will cover application-level policies to move closer to higher-level needs of security policy administration. In the policy engine, we plan to continue to enhance our configuration synthesis algorithm and make the validation modular and hierarchical for scalability. In the instrumentation layer, we plan to experiment with network latencies and control loop issues. The long term goal of this project is the creation of localized policies from global policies on sub-networks so that the policy management infrastructure can be distributed. When the prototype is complete at the end of the lifetime of this project, policy violations will be handled at the most local level possible so that a scalable management tool will become possible.

8. References

- [1] J. Anderson, S. Brand, L. Gong, T. Haigh, S. Lipner, T. Lunt, R. Nelson, W. Neugent, H. Orman, M. Ranum, R. Schell, and E. Spafford. Firewalls: An Expert Roundtable. *IEEE Software* **14** (5): 60-66 (1997).
- [2] Y. Bartal, A. Mayer, K. Nissim, and A. Wool. Firmato: A Novel Firewall Management Toolkit. *IEEE Symposium on Security and Privacy* 1999: 17-31.
- [3] M. Blaze, J. Feigenbaum, and J. Ioannidis. The KeyNote Trust Management System Version 2. Network Working Group RFC 2704. <http://www.crypto.com/papes/rfc2704.txt>.
- [4] M. Carney and B. Loe. A comparison of methods for implementing adaptive security policies. In *Proceedings of the 7th USENIX Security Symposium (SECURITY-98)*, pages 1 – 14.
- [5] J. Chomicki, J. Lobo, and S. Naqvi. Axiomatic conflict resolution in policy management. Technical Report ITD-99-36448R, Bell Labs, February 1999.
- [6] J. Guttman. Filtering Postures: Local Enforcement for Global Policies.

- Proceedings of the 1997 IEEE Symposium on Security and Privacy.
- [7] S. Hambridge, C. Smothers, T. Oace and J. Sedayao. Just Type Make! Managing Internet Firewalls Using Make and other Publicly Available Utilities. USENIX Proceedings of the First Conference on Network Administration, 1999.
- [8] A. Keromytis and J. Wright. Transparent Network Security Policy Enforcement.
- [9] D. Marriott and M. Sloman. Management Policy Service for Distributed Systems. In *IEEE 3rd Int. Workshop on Services in Distributed and Networked Environments (SDNE'96)*, Macau, June 1996.
- [10] A. Mayer, A. Wool, and E. Ziskind. Fang: A Firewall Analysis Engine. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy, 2000*.
- [11] M. Miller and J. Morris. Centralized Administration of Distributed Firewalls. USENIX LISA X, 1996.
- [12] D. Mitton et al. Authentication, Authorization, and Accounting Protocol Evaluation. <http://www.ietf.org/internet-drafts/draft-ietf-aaa-proto-eval-02.txt>.
- [13] Cisco PIX firewall series and stateful firewall security. White paper 1997. http://www.cisco.com/warp/public/751/pix/nat_wp.pdf.
- [14] P. Rao, K. Sagonas, T. Swift, D. Warren and J. Freire. XSB - A System for Efficiently Computing Well Founded Semantics. *Conference on Logic Programming and Non Monotonic Reasoning (LPNMR'97)*, 1997. <http://www.cs.sunysb.edu/~prasad/papers/system.ps>.
- [15] A. Rubin and D. Greer. A survey of Web Security. *IEEE Computer* **31** (9): 33- 41 – 1998.
- [16] F. Schneider. Enforceable Security Policies. Manuscript. 1992.
- [17] M. Sloman, N. Dulay and B. Nuseibeh. SecPol: Specification and Analysis of Security Policy for Distributed Systems, EPSRC Grant GR/I 96103. <http://www-dse.doc.ic.ac.uk/projects/secpol/SecPol-overview.html>.
- [18] M. Stevens, W. Weiss, M. Mahon, B. Moore, J. Strassner, G. Waters, A. Westerinen and J. Wheeler. Policy Framework, IETF Policy Working Group, <http://www.ietf.org/internet-drafts/draft-ietf-policy-framework-00.txt>, 1999.
- [19] Cisco Systems. Cisco IOS 12.0 Network Security. Cisco Press; ISBN: 1578701600.
- [20] R. Wies. Policies in Network and Systems Management – Formal definition and architecture. *Journal of Systems and Network Management* Vol 2 no. 1., pp 63-83, 1994.
- [21] Y. Yemini, A.V. Konstantinou, and D. Florissi. Nestor: An architecture for self-management and organization. Tech. Rep. Dept. of Computer Science, Columbia University, Sept. 1999. <http://www.cs.columbia.edu/dcc/nestor/>.
- [22] J. Zao, L. Sanchez, M. Condell, C. Lynn, S. Kent. Domain Based Internet Security Policy Management. In *2000 DARPA Information Survivability Conference and Exposition*. (Now appearing in IETF as <http://www.ietf.org/internet-drafts/draft-ietf-ipspspsl-00.txt>.)
- [23] J. Zao. A Survey of Policy Problem Space. Presentation at the IA Policy workshop. Washington, D.C., October 14, 1999.

The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Air Force Research Laboratory or the U.S. Government.