Hierachical Formal Modeling and Verification of Router Policies with an Applied Case Study to Cisco Router Configurations

A Thesis
Presented in Partial Fulfilment of the Requirements for the
Degree of Master of Science
with a
Major in Computer Science
in the
College of Graduate Studies
University of Idaho
by
Matthew Brown

Major Professor: Daniel Conte de Leon, Ph.D.
Committee Members: Michael Haney, Ph.D.; Axel Krings, Ph.D.
Department Administrator: Frederick Sheldon, Ph.D.

December 2016

Authorization to Submit Thesis

This thesis of Matthew Brown, submitted for the degree of Master of Science with a major in Computer Science and titled "Hierachical Formal Modeling and Verification of Router Policies with an Applied Case Study to Cisco Router Configurations," has been reviewed in final form. Permission, as indicated by the signatures and dates given below, is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor: ______________________________ Date______________
Daniel Conte de Leon, Ph.D.

Committee
Members: ______________________________ Date______________
Michael Haney, Ph.D.

______________________________ Date______________
Axel Krings, Ph.D.

Department
Administrator: ______________________________ Date______________
Frederick Sheldon, Ph.D.

Abstract

Today's economy and society's well-being are dependent on secure information technology systems and networks. Securing enterprise-size technology systems with thousands of interconnected devices in hundreds of networks has proven a grand challenge. Within this environment, network administrators and cybersecurity personnel need a method for verifying, with a high degree of accuracy and efficiency, that security policies are being correctly implemented throughout the enterprise's network. In this thesis, we describe a formal model, and associated developed tools, for policy verification of network routing policies. We also describe the practical application of this model and tools to a an enterprise-class case study for a Cisco-based network. The specific contributions are: formal modeling of router policies, high-level querying of enterprise router policies, formal router policy verification, and toward formal routing policy concatenation. This work demonstrates that it is possible to formally model and verify real router policies in an enterprise network.

Acknowledgements

I would like to thank Dr. Daniel Conte de Leon for guiding me through my graduate education. With out his patience and wisdom I would not be where I am today. I would also like to thank my parents for being my rock even in the hardest of times. I would also like to thank the current and previous researches on the HPol team.

Table of Contents

List of Figures

List of Code Listings

CHAPTER 1

Introduction

Formal policy modeling plays an important role in the field of information assurance. The ability to formally model a policy can increase the effectiveness of information security personnel. Policy models allow professionals to better organize their domains, whether that be corporate infrastructure or a small business network. More specifically, router policy modeling is an critical asset in industry.

Information security personnel need to be able to use a piece of software that can gather information about the current configuration of a given router, model that information in an intuitive way, and allow the querying of the model for information about the router.

1.1 Problem

Information security managers have to implement policies, standards, procedures and guidelines to ensure the availability, integrity and confidentiality of resources and information [6]. Success is measured by the ability of a system, network, etc. to insure the integrity, availability, and confidentiality of data both in transit and at rest. Security policies are an essential ingredient of an organization's operations plan and the starting point of security-related operations including the configuration of system components.

In a corporate or industry network, it is hard to keep track of the configurations of different components of an enterprise network and its components. In a typical corporate level network there are hundreds if not thousands of components that have to work synchronously. Most networks don't have the same technologies throughout the network, meaning system administrators have to learn each type of technology in order to be effective at their job.

Currently, there is no high-level way, known to the authors, to gather information about a device's configuration and query information on the configuration. There is also a need for a complete model of a system's security policy that can be used to formalize concrete policies while offering a high-level of abstraction that enables verification of a security policies' correctness. Also, the initial and ongoing assurance that an organization's security policies are being completely and correctly implemented by all systems and devices is a difficult task. This is due to the number of different systems, sub-systems, and devices within an industry's network and due to the current impossibility to derive a complete system policy model from a system or device.

In order for security personnel to create and maintain a secure network, configurations of various different components in the network need to be edited. While there are technologies that can edit the configurations of many instances of a single type of component, such as Microsoft's Group Policy editor, there is no way to configure multiple different components simultaneously nor is there a way to query the current configurations for details across all devices in the network.

From the point of view of security policies and their enterprise implementation, there is still a need to view an enterprise infrastructure as a whole versus as an isolated set of individual devices and applications that are configured independently. Hence, there is a need for a model that formalizes security policies throughout a system infrastructure from users and roles, through applications and devices, and into the network. Such a model must enable security policies to be organize and to represent all necessary detail within system device configurations. In addition, this model must also be able to provide policy representation, visualization, and analysis at a high level of abstraction.

1.2 Approach

This problem was tackled in three phases. First, a formal model had to be created. The model created uses layered directed acyclic graphs (DAG)s to model both the components of a system and the policies that are active in the system. The first layer of DAGs define a set of nodes and edges that model the components or elements of a system. This DAG orders these elements in a hierarchical format. We chose this format because information about a system can be implied through the hierarchy of the graph. The second DAG layer uses the same nodes as the first layer but with two additions: a start and end node. These nodes combined with a new set of edges define the polices in effect in the system. Each policy defined by the model is converted in to a series of edges all starting at the policy-start node and ending at the policy-end nodes; this is known as a policy path. These policy paths connect the component nodes in such a way that describes how components interact with each other and with foreign devices and components.

The second phase of our approach was to choose a sample set of systems and devices to model. The set we chose was SELinux, Openstack and Cisco routers. These systems and devices were chosen because they represent a large subset of devices found in an enterprise network. The process of modeling a Cisco router scenario is described in chapter 5 of this thesis.

The last phase of the approach was to create a query engine that can be used to gather information about the model generated in phase 2. In order for us to proceed, two things were needed. We needed an engine that could organize large amounts of data quickly and produce results in a timely manner, and we needed a way to communicate with the query engine. For the first part, Prolog was used to store and query data through a series of procedures that were written for this project. For the second part, an intermediate language was developed that allows the structure of the models to be loaded into the Prolog engine. This language would also enable security professionals to write human-readable queries to the engine. The details and processes behind model querying are described in chapter 6 of this thesis.

This graph-based policy model is known as the Hierarchical Policy (HPol) formal model. HPol uses a layered directed acyclic graph to organize the structure of a device and the polices enabled in the device. The first layer in HPol is a directed tree graph that uses hierarchy to define and model the structure of components in a device. This structure creates a common single root node between all pieces of the model. The second layer of

HPol uses another directed tree that models all polices enabled by the device. The typical HPol model has three subgraph DAGs. Each one of these subgraphs has a label that is unique to the contents of the DAG. The three labels are `Subject`, `Action` and `Object`. A `Subject` is defined as an entity, component, or subsystem that can preform an action. An `Action` is defined as any action, or more specifically the permission to preform an action that a node in the `Subject` DAG can preform. An `Object` is defined as an entity, component, or subsystem that is the recipient of an action preformed by a node in the `Subject` DAG. These definitions were part of the original model and are used to describe policies with the structure: *Who* [`Subject`] can preform *What* [`Action`] on *Which* [`Object`] resource. This is further explained in chapter 4 in this thesis.

High-Level Easy-to-Use Reconfigurable Machine Evironment Specification (HERMES) is a language developed at the University of Idaho and was originally used for web-browser policy specification [7]. HERMES was chosen to specify the structure of an HPol formal model and the policies that run through the model. HERMES was chosen for three reasons: it is fully parameterized, multi-platform and easy for humans to use. In HERMES there are no keywords, just a generalized syntax structure that allows for the efficient organization of data. The contents of a HERMES file can be stored in a simple text file, allowing it to be opened and edited on any platform. HERMES was originally designed to be used by humans, in addition to being processed by computers. HERMES is fully described in chapter 4.

1.3 Overview of Thesis

In this thesis it will demonstrated that it is possible to model a Cisco router using a formal policy model and it is possible to query the model for information about the current configuration of the router. Chapter 2 describes the contributions to the HPol project written up in this thesis. Chapter 3 gives a brief introduction to previous components of the HPol project. Chapter 4 gives full details on both the HPol formal model and the HERMES language. Chapter 5 gives a detailed account of the first contribution made to the HPol project: formal router policy modeling using Cisco routers. Chapter 6 gives a detailed description of the second contribution to the HPol project: formal policy model verification and querying. Chapter 7 applies the concepts of chapter 6 to a Cisco router model that is covered in chapter 5. Chapter 8 proposes possible future work for the HPol project. Finally, chapter 9 concludes this thesis.

CHAPTER 2

Contributions

2.1 Research Questions and Objectives

The research for the HPol project is primarily driven by a single research question: Can the HPol formal model be used to organize policy specifications for all devices? This question can only be answered once every type of device on the planet has been modeled by the HPol formal model. This thesis is the answer of a subset of that question. The research for this thesis was driven by two more specific research questions:

1. Can we use the HPol formal model to accurately formulate a Cisco router policy?

2. Can we use the previously generated formal model to verify and query a specific Cisco device routing and security policy?

Chapter 5 answers the first question and provides details to support the answer. Chapters 6 and 7 answer the second question. Chapter 6 describes the query mechanism used for HPol formal models and chapter 7 presents a case study using the model generated for chapter 5.

2.2 Contribution 1: Formal Router Policy Modeling

The HPol formal model was previously developed before router policy modeling and verification research started. My contributions to the project were Cisco router policy modeling using HPol and formal model querying. Cisco router modeling is described in chapter 5 and HPol model querying is described in chapter 6, in addition to some improvements to the HPol model.

It took many iterations of Cisco router policy modeling to get the HPol model in a form that is both intuitive and informative. The final model follows the typical `Subject`, `Action`, `Object` structure that all HPol formal models follow.

2.3 Contribution 2: Formal Policy Verification

HPol formal model querying is a new addition to the project. There were multiple points of research involved in this portion of the project. The first portion involved selecting an engine that would be used for organizing model data. XSB Prolog was selected for this task. Second, a way was needed to load the data from an HPol model into the Prolog engine. HERMES was selected for this task. The HERMES language was originally used for web-browser policy specification [7]. The HERMES language was close to being what was required for the HPol project; however, modifications were required to make it fully suitable for its new role. These modification required the development of a new parser and translator to compile HERMES into Prolog while the original specification of the HERMES language remained the same.

Once the HERMES language was ready for use, a series of Prolog procedures were created to evaluate queries. Queries for the HPol formal model are also written in HERMES and are loaded into the Prolog engine the same way that graph model data is. The procedures were created for evaluating queries search through the model data to find policies that satisfy queries. Full details on the query engine are described in chapter 6 of this Thesis.

CHAPTER 3

Background

3.1 Past and Concurrent HPol Research

In order to answer the primary research question of the HPol project, device parsers must be implemented to show that policies in different types of components can be modeled using HPol. There is also research being conducted on formal model merging. This will allow security personnel to take the formal models of multiple devices and combine them into a single formal model that enables the direct comparison of multiple devices.

3.1.1 Access Control Policy Modeling

One of the initial areas of interest was the modeling of access control mechanisms. SELinux was chosen for modeling due to its verboseness. Clear access control rules allow for a direct translation between the SELinux configuration and an HPol model.

Development for the SELinux parser is still in progress. The HPol graph for SELinux consists of three separate subgraphs: `Subjects`, `Actions`, and `Objects`. For the SELinux graph, all roles and user types are being categorized as `Subjects`, all permission types are being categorized as `Actions`, and file types are being categorized as `Object`. The basic principle is that a `Subject` node can preform an `Action` node on an `Object` node [18].

3.1.2 Database Permissions Policy Modeling

Another initial area of interest was the modeling of database permissions. The Keystone database application in the Openstack framework was chosen for this task. Each user in the database has a clear set of permissions that can be easily modeled using HPol.

The goal is to model which users had access to the database and what kinds of rights they had. The typical `Subject-Action-Object` format of the HPol model fits well with the type of rights that are present in Keystone. This work is still in progress.

3.1.3 HPol Formal Model Merging

Formal model merging is a resent addition to the HPol framework. Formal modeling allows a security professional to take the HPol formal model of multiple devices of the same type and visually compare the polices by combining the policy models into a single model. This method will enable the verification of policies through multiple devices simultaneously.

3.2 Research Related to HPol

Security policy modeling has been attempted in the past. Different projects use different types of modeling and graph schemes to model systems and the security policies that are implemented in the systems. This section lists a few relevant types of modeling schemes as well as how they differ from the HPol formal model.

3.2.1 Mathematical Formalisms

Set-Based Modeling: Policy modeling and querying is not a new concept to the information assurance community. Guttman, Herzog, and Ramsdell investigated the ability to verify information flow and access control within Security-Enhanced Linux (SELinux). They used a set-based modeling scheme to verify that security policies were being enforced [5]. Our model instead uses a hierarchical model based on a Forest of Directed Acyclic Graphs (DAGs) enhanced with policy links. This graph can then be used to construct more abstract system-level policies by using graph operations.

Strand Spaces: Strands and Strands Spaces are another formal approach used for proving security policies correct [14] [15]. Strands are defined as “a sequence of events” [16]. These sequences are defined through the casual interaction of a system from both authorized and unauthorized parties. Strands are then tied to a graph to form strand spaces. Strands and strand spaces have been used to enforce packet protection [3]. Strand spaces are used to analyze the allowed traffic of a network and define policies that restrict unauthorized traffic. Policies generated are only as complete as the interactions that reveal themselves at the time of modeling. New interactions can appear after modeling is completed that can cause false results in this type of model. In contrast HPol uses the configuration files of devices to construct policy models. This method creates a more definite policy graph that can predict all types of interactions.

Bipartite Graphs: Network security automation modeling through the use of graph construction has also been researched [4]. This modeling scheme uses *paths* through bipartite graphs to define network policy. In contrast, HPol uses a directed graph set up in a hierarchical format to represent the relationship of different components in a model as well as allow for policy abstraction.

Petri-Nets: Shafiq *et al.* [13] expand on the work of others to create a generalized temporal role based access control (GTRBAC) model. They use a colored Petri-Net based framework for verifying event-driven role based access control (RBAC) in real time. To prevent unauthorized access, they designed a set of rules for detecting undesired traits in a system derived from flaws in the policy specification [13]. They do not have a system set up for querying the current status of a system. Instead of a Petri-Net model, HPol uses a directed acyclic graph for managing data in a system and can then use the data gathered to answer questions about queries. The HPol framework increases functionality through its querying mechanism while also organizing data hierarchically

Model Checking: Kotenko and Polubelova created a verification model of networks and distributed firewalls using model checking. They proposed an approach that describes a network containing a limited number of addresses and detects anomalies in the network [9]. In an effort to validate their proposal, they devised a series of tests to determine the correctness of a distributed firewall configuration. HPol converts model data into Prolog knowledge-base. Then this knowledge-base is used to verify the correctness of device configurations against a given set of polices.

3.2.2 Policy Specification Languages

Security Policy Language: Bernabé *et al.* developed an XML based High-Level Security Policy Language (SPL) that is used to translate security policies from human readable descriptions to configuration or machine-level code of given entities [12]. Similar to our own research SPL is designed to allow a system administrator to configure machine settings through the use of a policy language. Similarly to SPL, the HPol framework uses HERMES to specify model data. The HPol tool-set also extends the SPL approach to use a hierarchical structure to define policies that will allow a system administrator to set policy on both a system and network level.

3.2.3 Policy Specification Tools

Margrave: Fisler *et al.* are developing an access control policy verification tool called Margrave. Given a property of a system and a policy for that system, Margrave checks to make sure the property satisfies the policy. Margrave also detects the impact of a policy change on a system [2]. The Margrave poject was expanded to include the verification of firewall polices as well [10]. Margave, however, does not verify policies over multiple devices. HPol's objective is to verify that a policy is being enforced not only in a single system but in all systems across a network or infrastructure. The HPol model uses XSB Prolog to evaluate whether or not a query is satisfied by the polices in the model.

CHAPTER 4

The HPol Formal Model and The HERMES Language

4.1 The HPol Formal Model

HPol is a framework that reads low-level security configuration files and creates high-level abstractions of those files to create a model of security policies. This enables an automatic verification system that checks that each low-level configuration file is correct and satisfies high-level security policies required for the correct and successful operation of an organization's network. HPol is designed to answer questions based on four parameters: *Who* can access *Which* resources in *What* manner and in *Which context*?

The HPol formal model represents policy subjects, actions, and objects through a system of Direct Acyclic Graphs (DAGs). Policy graphs are then added by superimposing a second DAG on top of the original graph. Policy paths travel through the nodes of original graph. Each policy path represents a policy that is allowed in the system. The structure and details of this model are described in this chapter. Figure 4.1 is an example of an HPol model that represents the access control of a file system. In this model the `Alice` user is allowed the `read` and `write` permissions on the `/home/alice` directory by polices 1001 and 1002 respectively.

Also, note that a given node can have more than one parent node. For example, the `Oscar` node has one primary parent (dark blue) and two secondary parents (light blue). The dark blue hierarchical line states that Oscar is primarily a child of the `Users` node just like the users `Alice` and `Bob`. The light blue hierarchical lines state that the `Oscar` node is also a member of the `Role_User` and `Role_Admin` groups or subgraphs.

One of the primary advantages of the HPol approach to modeling security policies is that it sorts hierarchical organizational structures into their corresponding graph hierarchies through the use of Directed Acyclic Graphs (DAGs).

This approach enables the model to formalize these organizational hierarchies and define

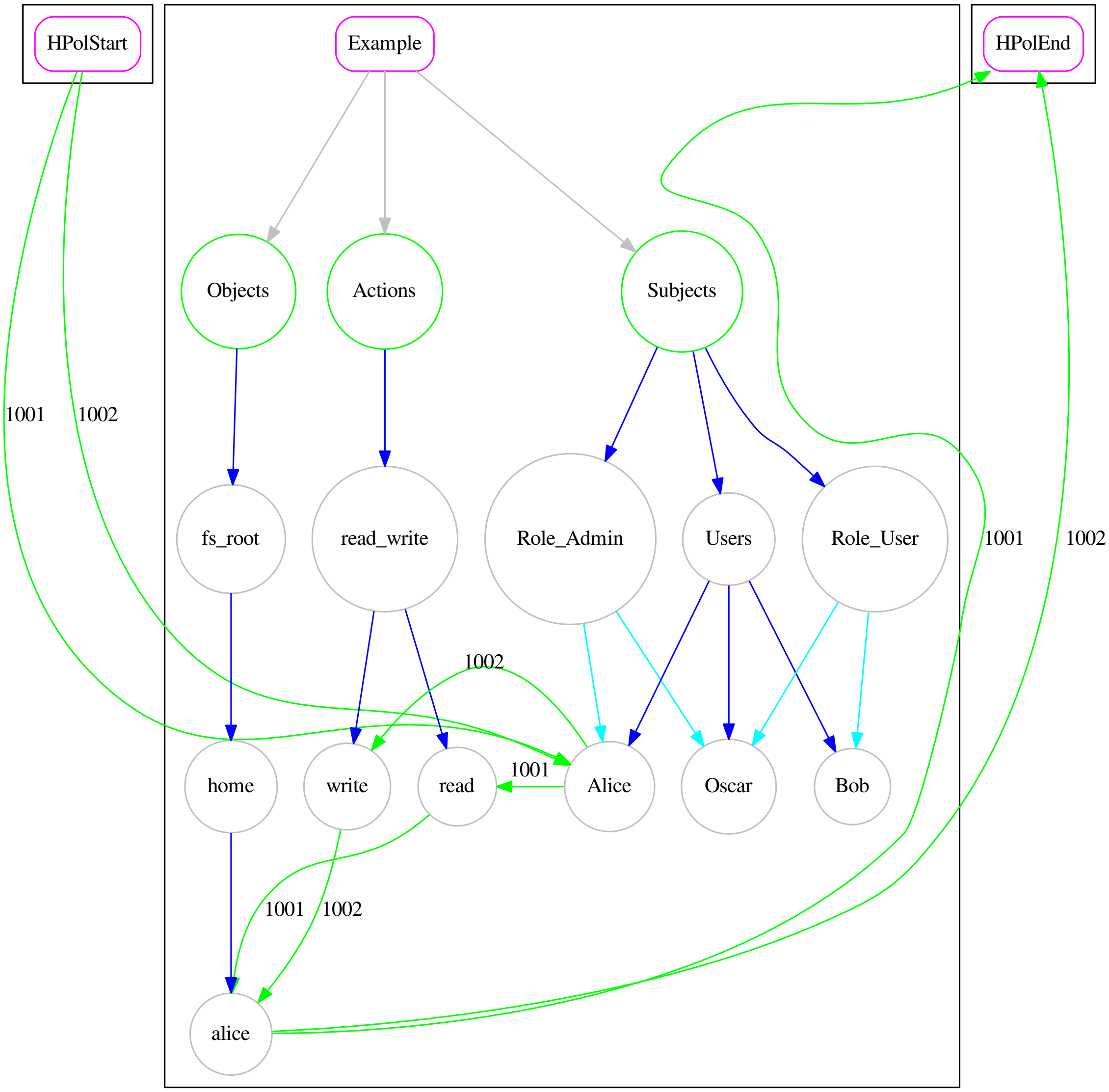

Figure 4.1: A file system permission example HPol model. This model contains three different users, two permission types and a single directory.

formal policy abstraction operations. For example, given the permissions of a file system, if a user has `write` permissions on all the files within a given sub-directory and the same user has `read` permissions on the same files, then it can be inferred that the user has both `read and write` permissions on all files in the subdirectory. If there are two separate policies allowing the `read` and `write` permissions on a file, then an abstraction can be made from these policies into a single policy that allows both reading and writing on the file. The HPol model enables a straightforward and formal method for discovering these types of abstractions. In this example, a DAG would be created to represent the file permissions hierarchy. Then in such a DAG a `read_write` node would be the parent node of nodes `read` and `write`, hence allowing the abstraction to be inferred and represented by the formal HPol model.

Figure 4.1 shows a graph representation of the example described above. Observe that in the figure, there are two paths from the `HPolStart` node to the `HPolEnd` node; both paths are drawn with green arrows. The paths in figure 4.1 are described below in detail:

1. Path in which all the links are labeled with the *1001* identifier composed of the following green arrow links:
   (a) Starting at the `HPolStart` node going to the `Alice` node in the `Subjects` DAG.
   (b) Moving to the `read` node under the `Actions` DAG from the `Alice` (Subjects) node.
   (c) Moving to the `alice` node under the `Objects` DAG from the `read` (Actions) node.
   (d) Ending at the `HPolEnd` node coming from the `alice` (Objects) node.
2. Path in which all the links are labeled with the *1002* identifier composed of the following green arrow links:
   (a) Starting at the `HPolStart` node going to the `Alice` node in the `Subjects` DAG.

   (b) Moving to the `write` node under the `Actions` DAG from the `Alice` (Subjects) node.

   (c) Moving to the `alice` node under the `Objects` DAG from the `write` (Actions) node.

   (d) Ending at the `HPolEnd` node coming from the `alice` (Objects) node.

Note that the only difference between these two policy paths is the `read` node in the first path (1001) as a file permission and the `write` node in the second path (1002) as a file permission. Also note that the same subject node, `Alice`, and the same object node, the `alice` home directory node, are equal in both policy paths.

With this formalization of a policy using policy paths plus the fact that the permissions are organized in a hierarchical manner, the information and inference rules needed to make the abstraction are all present. The information needed to make this abstraction is given by the policy paths and the hierarchy in the DAGs; in this case, the permissions DAG in which the `read_write` node is the parent of the `read` and `write` nodes.

The two policy paths described above could be abstracted by one unique hypothetical policy path as follows:

1. Path in which all the links would be labeled with the *1001-1002* identifier composed of the following links:

   (a) Starting at the `HPolStart` node going to the `Alice` node in the `Subjects` DAG.

   (b) Moving to the `read_write` node under the `Actions` DAG from the `Alice` (Subjects) node.

   (c) Moving to the `alice` node under the `Objects` DAG from the `read_write` (Actions) node.

   (d) Ending at the `HPolEnd` node coming from the `alice` (Objects) node.

This type of policy abstraction operation would also be applicable to other areas of policy DAGs. A similar abstraction could be made in the subject DAG of figure 4.1. Suppose the user `Oscar` had a similar read-write policy that `Alice` has with the only difference being the subject of the policy, `Oscar` instead of `Alice`. Also, note that both Oscar and Alice are secondary children of the `Role_Admin` node. With the combination of the similar policy paths and the common parent of both the `Oscar` and `Alice` node, an abstraction policy could be formed that states that the Admin Role has the permission to read and write on the `alice` directory. Also, note that this abstraction cannot be applied to the `Users` node. This is because the `Bob` node would not have this policy. Only when all of the children of a parent node have a similar policy can a full abstraction be made. Abstractions in the opposite direction are possible as well. If a policy states that the Admin Role can `read` and `write` on the `alice` directory then it can be inferred that all children of the `Role_Admin` node also have the ability to `read` and `write` on the `alice` directory.

Our goal is to allow the model and its computational implementation to represent the low-level policy paths and to be able to represent and infer the high-level abstract policies. This would enable security policy analysis to choose the level of abstraction to visualize and verify the security policies within and across a system.

4.2 The HERMES Language

High-Level Easy-to-Use Reconfigurable Machine Evironment Specification (HERMES), is a specification language that allows cyber security personnel to describe an organization or industry's infrastructure security policies using entity sets. Each entity set can classify a variety of components such as Domains, Groups of Users, Roles, Application and so on. HERMES allows entity sets to be defined using a hierarchical structure. Because of this structure, an HPol model can be represented as a series of entity sets. The flexibility and uniqueness of HERMES lies in the following features:

```
Policy: ID_001
{
  Description: "Disabling JavaScript.";
  Rationale: "Security Vulnerability";
  Status: "Enabled";
  Field: (JavaScript, "Disabled");
  ApplyTo: "ALL";
}
```

Listing 4.1: Example of HERMES usage (to disable Javascript in all web browsers for all machines connected to a server).[7][8]

Platform Independent: HERMES is a text-based language interpreted by a Prolog engine. Because Prolog is platform independent, HERMES is as well. Also, HERMES is capable of specifying security policies for any kind of application, on any platform or operating system and in any context. This is because all parameters of a HERMES entity set are flexible. There are no keywords required to define entity sets. All a system administrator would have to do is follow the very simple syntax defined by HERMES.

Human-Centered and Ease of Use: HERMES is designed to be written and read by humans, not computers. HERMES is similar to YAML, but is not based on XML or any other similar verbose markup languages. XML-based languages are difficult and obscure to read and write for humans. HERMES is designed to be easy to read and write by humans. Listing 4.1 contains an entity set written in HERMES. As shown HERMES entries are written using words and very little special characters when compared to other specification languages such as XML.

Versitile Specification Capacity: HERMES allows the specification of an organization's infrastructure in a hierarchical manner. For example, a company's organizational chart can be specified in HERMES with the top most entity set being the CEO and the next level of branches being the children of this entity set, such as the marketing and IT departments. HERMES can easily define tree and graph structures through its entity sets. It is because of this that HERMES was chosen to represent HPol formal models. Its parameterized nature and its ease of use make it ideal for use in both an IT environment and in the HPol framework.

HERMES is designed to accommodate the automatic generation of entity sets based on high-level policy specifications. Because of this HERMES was chosen to be an intermediate language for the HPol framework. HERMES was originally used for the specification of security policies in web-browsers. This project is called HiFiPol:Browser [7][8]. HERMES was then later adapted for use in the HPol project.

CHAPTER 5

Formal Router Policy Modeling

In this chapter, formal modeling of a Cisco Site-to-Site VPN policy with the HPol model is described. Two complete router configurations will be modeled in this chapter. A policy will also be traced through the resulting HPol formal model graph to demonstrate the models effectiveness.

5.1 Cisco Site-to-Site VPN Policy

The scenario which is parsed and modeled was developed by Cisco for training purposes and is described in detail on the Cisco site-to-site VPN training website [1]. Figure 5.1 shows a similar network architecture for this site-to-site VPN example and corresponds to Figure 3-8 in the Cisco example which is available in the online Cisco example.

In the example presented in Figure 5.1 the following can be observed:

1. VPN: There are two routers, left, hq-sanjose and right, ro-rtp, connected by an encrypted VPN tunnel.

2. LAN(s) Left: The left router has two internal VLANs, DMZ and private, which are connected to it with assigned IPv4 addresses: 10.1.6.0/24 and 10.1.3.0/24, respectively.

3. LAN(s) Right: The right router has one internal VLANs which is connected to it with assigned IPv4 addresses: 10.1.4.0/24.

4. WAN: The left side of the tunnel has been assigned the IPv4 address of 172.17.2.4; the right side of the tunnel has been assigned the IPv4 address of 172.24.2.5. These are mapped to a serial interface in each router.

5. Tunnel: The IPv4 addresses assigned to the tunnel are 172.17.3.3 and 172.24.3.6 for the left and right routers respectively. These are mapped to a tunnel interface in each

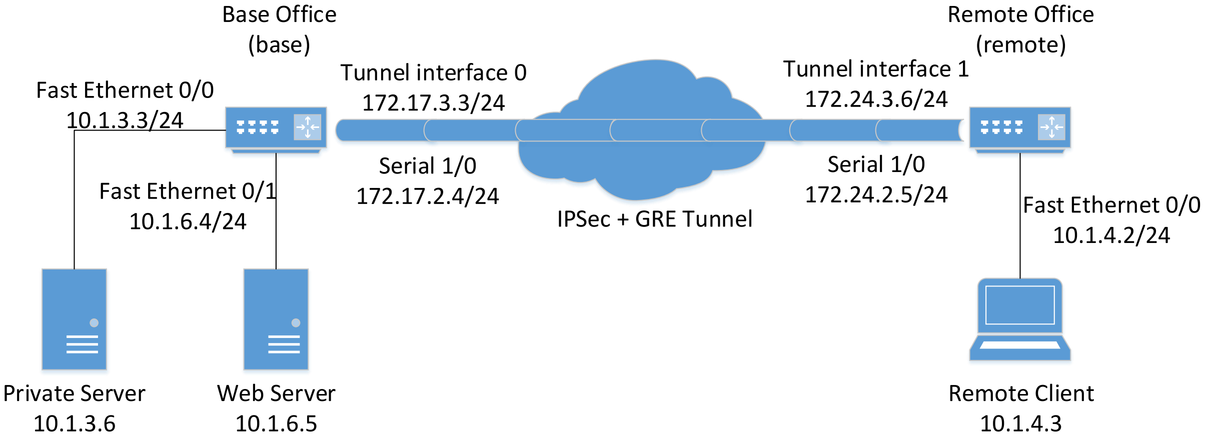

Figure 5.1: Site-to-Site VPN Scenario (based on [1]).

router. These two interfaces are virtual interfaces. The tunnel's encapsulation protocol is GRE+IPSec.

6. Host: A public Web server in the DMZ VLAN has been assigned the 10.1.6.5 IPv4 address.

7. Host: A private server in the internal (left) VLAN has been assigned the 10.1.3.6 IPv4 address.

8. Host: A private client in the internal (right) VLAN has been assigned the 10.1.4.3 IPv4 address.

9. Note: As indicated by the last two entries, the external IPv4 addresses assigned to the two routers are actually non-usable reserved IPv4 addresses used only as an example. A production configuration will configure here instead actual usable external IPv4 addresses.

10. Note: In this case both of the routers would be configured to perform Network Address Translation (NAT) between the external and the internal VLANs.

5.2 The Path of a Packet in the Example VPN Tunnel

In order to validate that our HPol model and tool-set are accurately modeling the routing policy established by the VPN Tunnel example described in the previous section, the steps followed by an IP packet in the given site-to-site VPN configuration will be manually traced. Section 5.4 will show how the HPol formal model also models the same forwarding and security policy.

1. Let's assume that a user in PC A, IPv4 10.1.4.3, (right side of figure 5.1) is accessing services provided by the private corporate server at 10.1.3.6 through the Site-to-Site VPN Tunnel.

2. A given IP packet would be initiated at the PC A client, IPv4 address 10.1.4.3, and sent to the gateway at the router endpoint of IPv4 address 10.1.4.2, which is within the same subnet and VLAN.

3. The *ro-rtp* router (right) would then see that the packet has a destination IPv4 address of 10.1.3.6 corresponding to the private corporate server on the other side of the VPN tunnel. Hence, the router would encapsulate the packet using the VPN tunnel logical addresses: source, IPv4 172.24.3.6, and destination, IPv4 172.17.3.3, and send it through the tunnel interface, IPv4 172.24.2.5. In order to accomplish this, multiple encryption and encapsulation steps would need to be carried out by the router given the router configuration. These steps are described in the next subsection.

4. When receiving the packet on its tunnel serial interface, IPv4 172.17.2.4, the *hq-sanjose* router (left) would then unwind the tunneling and encryption steps performed by the *ro-rtp* router.

5. Lastly, the packet whose source IPv4 address is 10.1.4.3 and whose destination IPv4 address is 10.1.3.6 would be send via the Private VLAN to the Private corporate server from the gateway interface, IPv4 10.1.3.3.

5.3 The Cisco Configuration Parser

A parser was created in Python that reads a Cisco configuration file and by using the HPol framework creates a graph representation of a router's policy. Figures 5.2 and 5.3 show the policy graphs resulting from this process. The configuration parsed for this example may be found on the Cisco site-to-site VPN training website [1].

The generalized router HPol format is as follows:

- Numbers above links represent a policy number.
- A link illustrates a step in the policy.
- Subnet nodes verify that a packet contains an acceptable source address.
- Interface nodes verify that a packet was received at a particular interface.
- Action nodes represent a transform that is applied to a packet, e.g. cryptography.
- Action nodes also represent actions routers can take, e.g. send.

To verify a policy, the starting policy number must match throughout the policy. There can be no breaks in the links; however, there are wildcard links that allow all policies to traverse a given link. These wildcard links allow for easy readability and policy abstraction when generating a policy graph. All policies must start at the `HPol Start` node, have a continuous path throughout the graph, and end at the `HPol End` node. Once these conditions are met a policy is considered complete.

The Cisco configuration parser runs in two general steps: step 1, parse input file to gather all necessary data; step 2, generate an HPol model out of all gathered data. For the first step, parse input file, the ciscoconfparse library was used to make parsing configuration files easier. A series of functions were then created to extract the information out of the ciscoconfparse library [11]. Listing 5.1 is an example of one of these functions. The function in listing 5.1 gets all IP route information. As shown, the ciscoconfparse library allows a

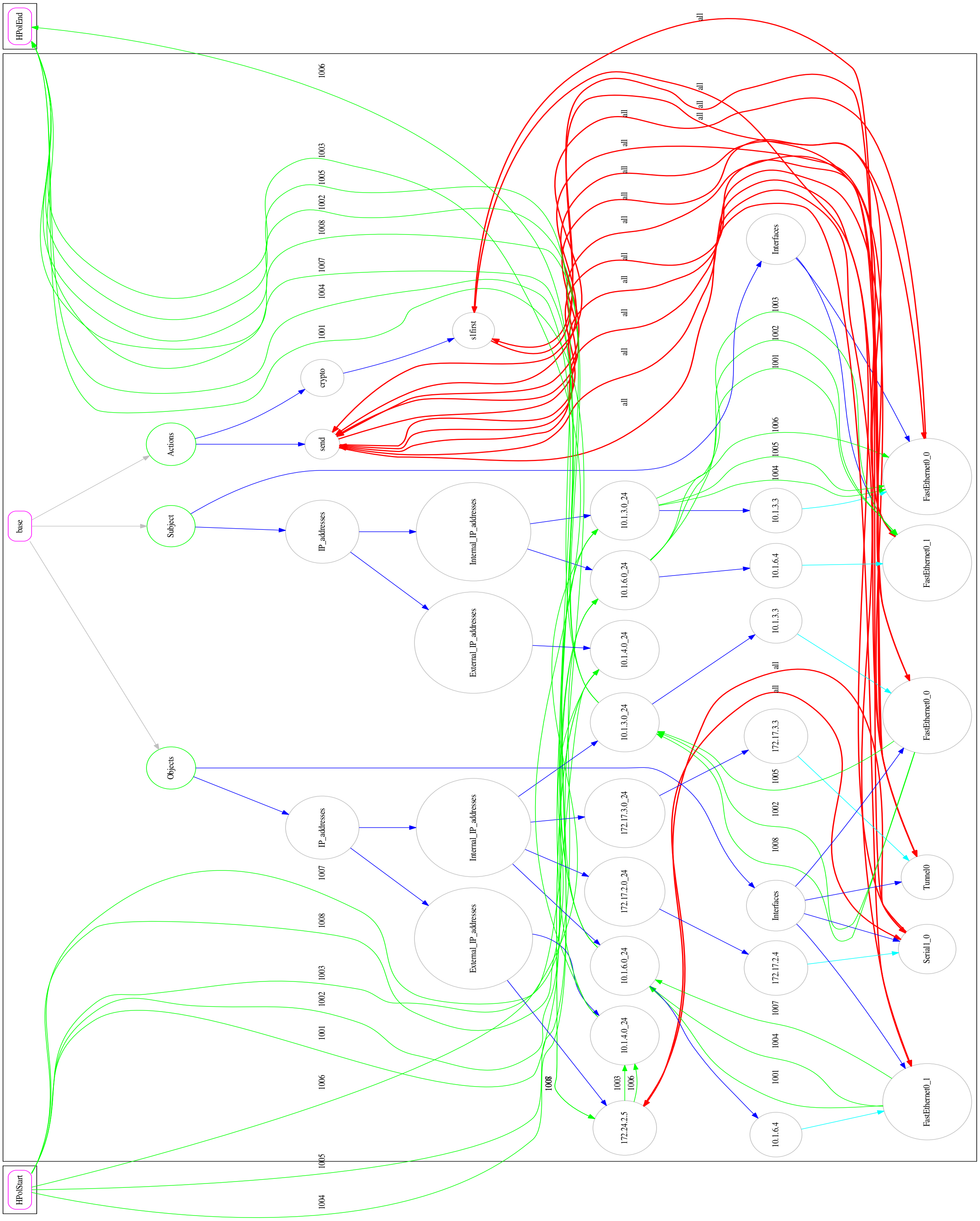

Figure 5.2: Cisco VPN HPol model: Graph View for Left Router, *hq-sanjose*

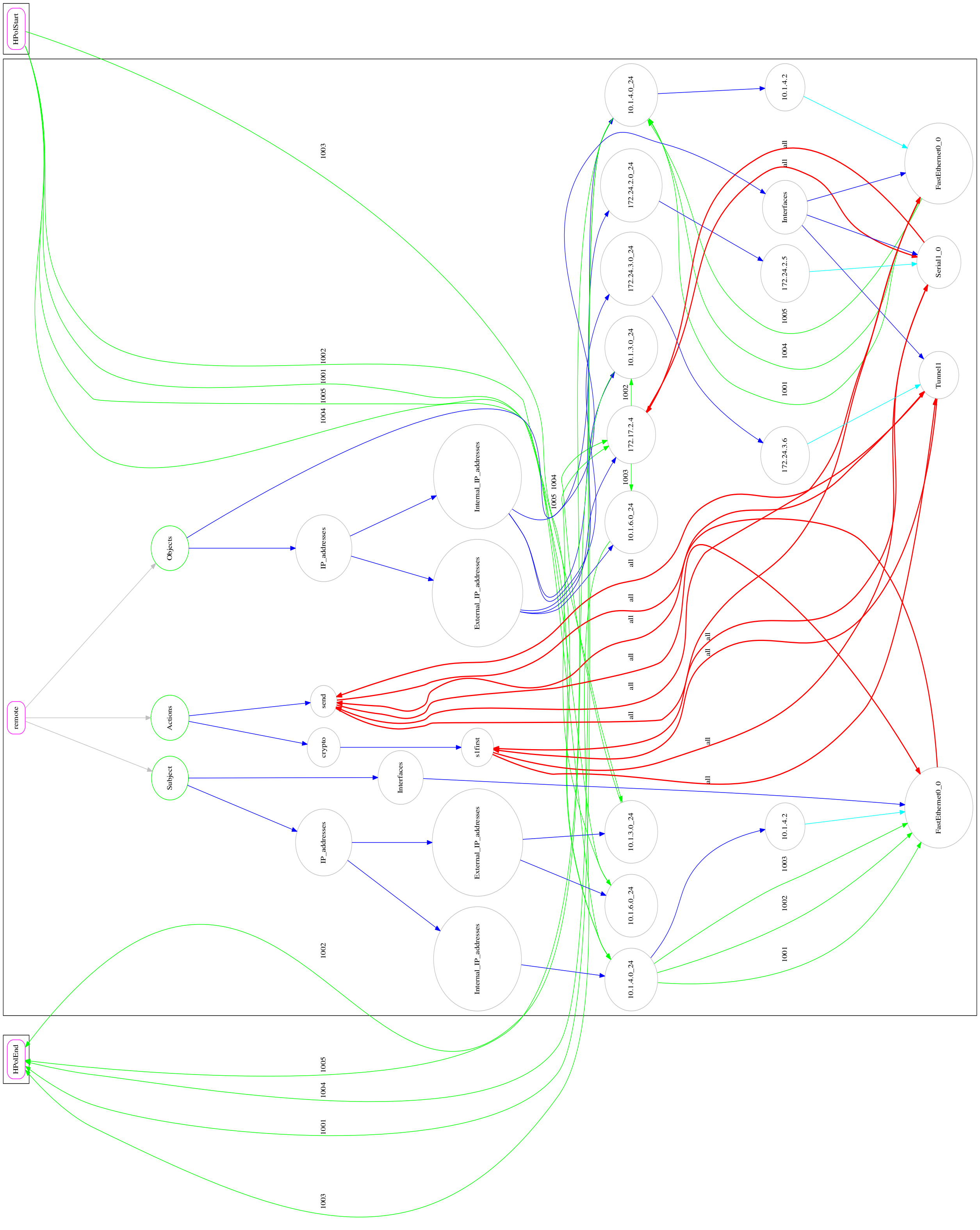

Figure 5.3: Cisco VPN HPol model: Graph View for Right Router, *ro-rtp*

user to search for configuration objects by name. In this function, all configuration objects containing the header “ip route” were searched for and were returned in a list to be used at the time of modeling.

```python
def getIPRoutes(parse):
    """
    This function gathers all predefined IP routes in the config file and stores them in a list. I chose a list for this data because there are no unique IDs for IP Routes. The Rule is just defined.
    Input:
        parse, cisco parse object of type ciscoconfparse
    Returns:
        iproutes, a list of defined IP routes
    """
    # make the list
    iproutes = []
    # find the ip route objects
    routes = parse.find_objects('ip route')
    # loop directly of the object since they have no children objects
    for entry in routes:
        # remove the 'ip route' part of the object
        temp = entry.text.replace('ip route ', '').split()
        # collect the ip of first object
        ip = getIPRange(temp[0], temp[1])
        # collect the name of the interface
        interface = temp[-1]
        # store
        iproutes.append((ip.replace('/','_'), interface))
    # and return
    return iproutes
```

Listing 5.1: Code snippet: getIPRoutes function of the Cisco configuration parser.

Once information on all points of interest were collected, a model was constructed. A description of the model is as follows. The Object DAG of the HPol model is populated using data from the interface, ipRoute, and Access Lists entries of the configuration file. Listing 5.2 shows how the IP Route information collected in listing 5.1 is used to populate the External_IP_addresses subgraph of both the Subjects and Objects DAGs. The Subject

DAG of the HPol model is populated by duplicating the objects that can also act as a subject. For example, the `FastEthernet0_0` object in the configuration represents a subnet that has devices on it. Theses devices are allowed to send information which means the `FasterEthernet0_0` node is duplicated under the `Subjects` DAG. The `Action` DAG is populated with transforms and actions that apply to packets. For example, cryptographic transforms are applied to packets that travel through `Tunnel1` so all cryptographic transforms are placed under the `Actions` DAG. The `send` action is also placed under the `Actions` DAG and represents the ability of a packet to be sent.

```
    #use ipRoutes to find external ipaddresses
    for item in ipRoutes:
        #collect the name of the ip/subnet
        nodeName = item[0]
        #add it to the Objects DAG
        graphPaths['Objects']['IP_addresses']['External'][nodeName] = createNode(hpol, nodeName, graphPaths['Objects']['IP_addresses']['External']['path'], 'Object')
        #add it to the Subjects DAG
        graphPaths['Subjects']['IP_addresses']['External'][nodeName] = createNode(hpol, nodeName, graphPaths['Subjects']['IP_addresses']['External']['path'], 'Subject')
```

Listing 5.2: Code snippet: How IP Route data is used to populate the formal model

Development of the Cisco router parser was completed in May of 2016 for this research. For the Cisco router parser, IP address and subnets are considered both Subjects and Objects. This makes parsing policies more complicated because an IP address can be the subject of one policy and the object of another. A real world scenario[1] was used in order to create a proof-of-concept parser. This scenario consists of a VPN tunnel set up between two routers. On one side of the tunnel is a client and on the other side is a server that the client is trying to access. The structure of the HPol model for the router scenario consists of the typical `Subjects`, `Objects` and `Actions` DAG. Subject nodes, in this model, are always the sender and Object nodes are always the receiver.

5.4 The Path of a Network Packet in the HPol Formal Policy

In this section shows how the HPol formal policy model matches the routing and security policy implemented by each of the VPN routers. The cisco router HPol model's current limitations are also described. For this demonstration, the router modeled in figure 5.3 will be used. Figure 5.4 shows only the relevant links in this demonstration with all other links manually removed for purposes of clarity.

1. **Router (right side):** Assume that a packet initiates from device PC A with IPv4 address 10.1.4.3 on the remote office. It is important to note that individual IPv4 address are not defined in router configuration files. Hence, packets appear to originate in the subnet the packet was sent from.

   HPol:

   We start on the remote office policy model, shown in Figure 5.3, and begin in the `HPolStart` node.

2. **Router (right side):** First, the packet with source IPv4 address of 10.1.4.3 and destination IPv4 address of 10.1.3.6, the latter corresponding to the Private corporate server, is sent to the Gateway through the corresponding connected interface, designated FastEthernet0/0. The excerpt of the router configuration that enables this route is:

```
interface FastEthernet0/0
ip address 10.1.4.2 255.255.255.0
```

   Note: For this to happen, first the gateway IPv4 address 10.1.4.2 and subnet mask 255.255.255.0 must also be configured in the client device PC A. Our current implementation of the HPol tool-set does not parse client network configurations, for example from MS Windows. However, if such a policy is parsed the corresponding model would be able to be concatenated to the overall HPol system policy model.

HPol:

(a) The link with label 1002 from the `HPolStart` node into the corresponding subnet `10.1.4.0_24` node within the `Internal_IP_address` subtree of the `Subject` DAG.

(b) The link with label 1002 from the `10.1.4.0_24` to the configured interface node, `FastEthernet0_0` residing under the `Interfaces` node of the `Subject` DAG.

(c) The wildcard Link, designated with *all*, from node `FastEthernet0_0` to the `send` node residing under the `Actions` subtree. The `send` node represents the ability for that subnet interface pair to send to a given destination.

Note: In this policy path, only packets belonging to the `10.1.4.0/24` subnet will be accepted. All other packets will be dropped. Because of the way Cisco routers are configured it is impossible to know every device connected to the router by examining the configuration file. Because of this all policies start with the corresponding subnet of the IP addresses in question. Also note that there are duplicates of some of the nodes. These nodes are under the `Objects` and `Subject` DAGs. This allows the policy to state the context of the node when paths are drawn. In this case both the subnet and interface are the subject of this policy.

3. **Router:** Because the destination of the router does not reside in a subnet attached to an interface of the router, a secondary source must be examined for this route. In this case there is an IP route set up for the destination's subnet. The IP route indicates that within this router all packets with an IPv4 destination address in the 10.1.3.0/24 subnet must be sent via VPN Tunnel, `Tunnel1` in this case. The excerpt of the router configuration that enables that routing policy is:

```
ip route 10.1.3.0 255.255.255.0 Tunnel1
```

HPol:

(d) The wildcard link from the send node to the Tunnel1 node within the Interfaces subtree of the Objects DAG.

Note: It important to notice that the only reason this router knows about the 10.1.3.0/24 subnet is because of the IP route statement. Most routers won't know about external subnets or IP addresses unless a special rule has been set up for the router.

4. **Router:** The router configuration specifying the Tunnel1 virtual interface is indicated immediately below. The last configuration line indicates that the contents should be encrypted using the rules defined by the s1first crypto map. Under this rule the packet will be encrypted using the specified rules, then encapsulated using the specified IPv4 source and destination addresses, and then sent to the physical source of the Tunnel:

```
interface Tunnel1
ip address 172.24.3.6 255.255.255.0
tunnel source 172.24.2.5
tunnel destination 172.17.2.4
crypto map s1first
```

HPol:

(e) The wildcard link from the Tunnel1 node into the s1first node within the crypto subtree.

5. **Router:** The router configuration specifying the s1first encryption scheme is indicated immediately below. The interface Serial1_0 is attached to this encrypted tunnel, the encryption configuration is stored in the attributes value within the Serial1_0 node, the allowed hosts are given by access list number 101, and the peer IPv4 address is 172.17.2.4. The excerpt of the router configuration that specifies this encryption policy is:

```
crypto map s1first local-address Serial1/0
crypto map s1first 1 ipsec-isakmp
set peer 172.17.2.4
set transfrom-set proposal1
match address 101
```

Note: There is a secondary permission evaluated by the parser known as an access list. The access list that allows communication between the two physical interfaces of the tunnel is indicated below. Here we see that the host at 172.24.2.5 is permitted to communicate with the host at 172.17.2.4 using the gre protocol. By examining the hierarchical structure of the tree we can see that node `Serial1_0` has the parent 172.24.2.5. This relationship shows that the Serial1/0 device operates with the IP address 172.24.2.5. This allows the communication between the Serial1/0 interface and the device at 172.17.2.4:

```
access-list 101 permit gre host 172.24.2.5 host 172.17.2.4
```

HPol:

(f) The wildcard link from the `s1first` node into the `Serial1_0` node within the `Interface` subtree in the `Objects` DAG.

(g) The wildcard link from the `Serial1_0` node to the `172.17.2.4` node residing under the `External_IP_address` subtree also in the `Objects` DAG.

6. **Router:** It is important to note that information about what is on the other side of the tunnel is very limited. Information about the destination for a packet is gathered through iproute statements within the router configuration. We can assume that the subnet 10.1.3.0/24 exists and is on the other side of tunnel1 by evaluating the iproute statement:

```
ip route 10.1.3.0 255.255.255.0 Tunnel1
```

HPol:

(h) The link with label 1002 from `172.17.2.4` to `10.1.3.0_24` also residing under the `External_IP_address` subtree in the Objects DAG.

(i) The link with label 1002 from the `10.1.3.0_24` subnet node in the `External_IP_address` subgraph of the objects DAG to the `HPolEnd` node.

Note: All nodes associated with the tunnel and the destination nodes fall under the `Objects` DAG. This is because in this context these nodes are objects.

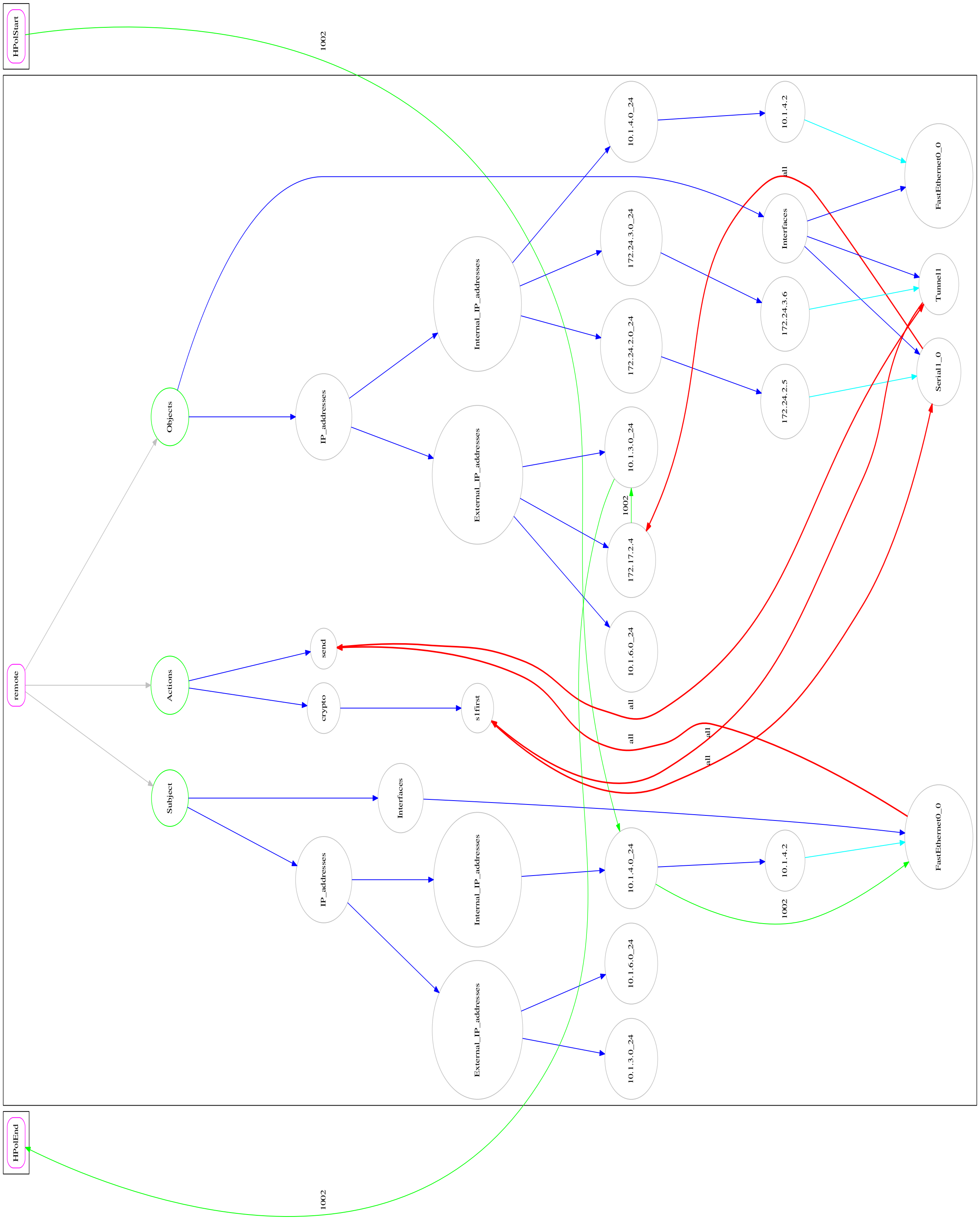

Figure 5.4: Cisco VPN HPol Policy Model: Graph View for Right Router showing only policy 1002

CHAPTER 6

Applied Formal Policy Verification

If a system administrator wanted to check if a policy is in effect he or she would have to manually check each node of the network. After checking the network the administrator would have to check the computers the policy applies to as well. This is an inefficient use of a system administrator's time and should be augmented with a framework. HPol can parse the configuration files of various components in a network and prepare the data in such a way that a system administrator can write queries to gain information about the current state of the network.

6.1 Modifications to HERMES

As mentioned in chapter 1, the HERMES language had to enhanced to make it fit for its role in the HPol project. While the overall effect of the modification was minor, the HERMES compiler had to be written to successfully allocate these changes. The changes to the HERMES language were as follows:

1. Modified the lexical analyzer
2. Added syntax for the following attribute types:
   (a) Lists
   (b) Tuples
   (c) List of Tuples, aka Dictionaries
3. Added Full parameterization

The lexical analyzer was modified to allow more types of special characters in identifiers. The original HERMES only allowed identifiers to have the same regular expression as Prolog terms. In HERMES identifiers are used to identify entry types and names as well as attribute key-value pairs. The original regular expression that defined an identifier is [A-Za-z][A-Za-z0-9_]*, which only allows for a very restricted set of possible identifiers. The new lexical analyzer allows for a greater range of symbols in an identifier including dots, hyphens and underscores. The new identifier regular expression is [A-Za-z0-9][A-Za-z0-9._-]*. The regular expression was modified to allow for identifiers that take the form of IP address and subnets. The new lexical analyzer can be expanded as needed as well. Because all identifiers are stored in Prolog as strings the number possible of characters is only limited to the size of the ASCII table. However, because we only needed the symbols in the new regular expression this is all that was included.

New attributes were added to allow greater flexibility with HERMES entries. The original HERMES compiler only allowed for single-word key value pairs and key string pairs. The new HERMES parser also allows for list, tuples, and list of tuples, or dictionaries. Listing 6.1 shows a HERMES entry that includes all attribute types. Lists were also added to list all nodes in a policy. Tuples and lists of tuples were because they are common data types in languages and can be useful in an unforeseen project.

```
Entry: example
{
  key:value;
  key:"string";
  key:[list, of, values];
  key:(value, tuple);
  key:[(list, of), (value, tuples)];
}
```

Listing 6.1: HERMES entry with all types of attributes.

The last modification to HERMES was full parameterization. A new compiler was developed to enable true parameterization. The HERMES language is now truly parameterizable for all identifiers. There are on longer any keywords in the HERMES language. The only required piece of HERMES is the general syntax structure.

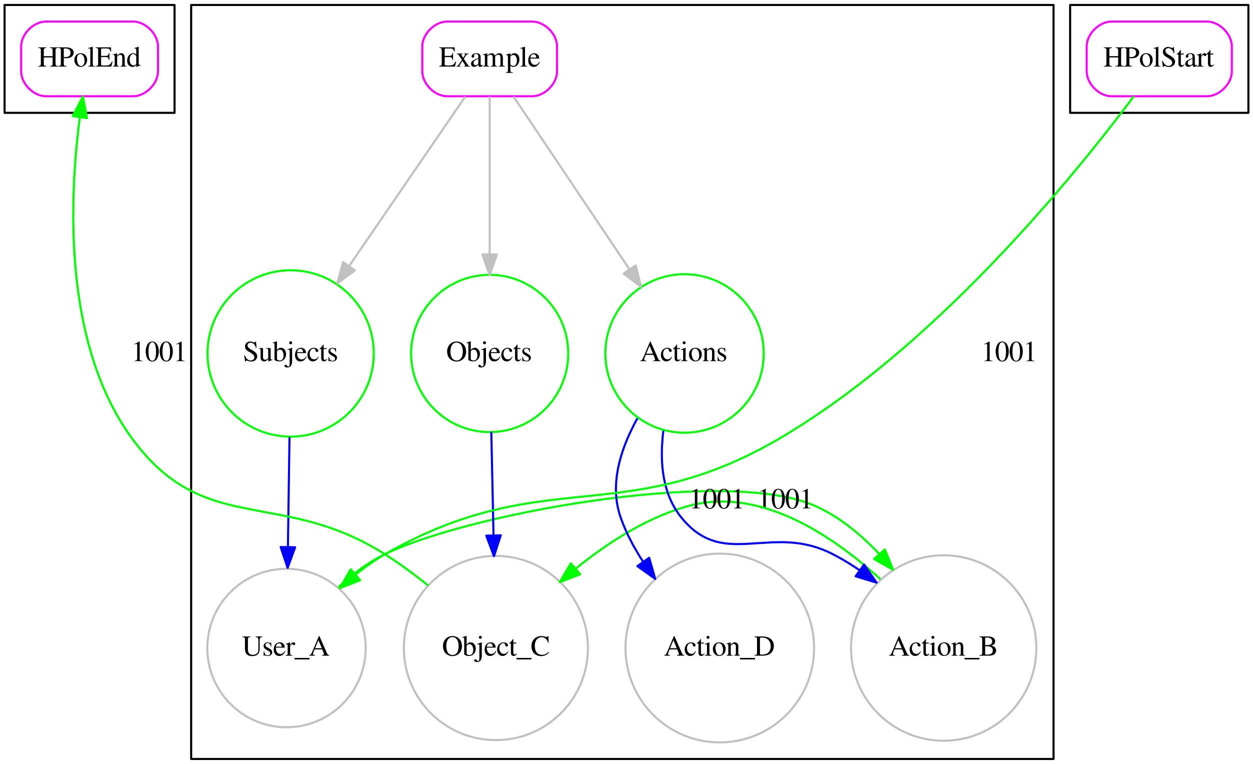

Figure 6.1: Basic Example of a HPol Graph.

6.2 A Formal Model for Policy Querying

This section will refer to figure 6.1 and the scenario that is modeled by the figure. The scenario for this experiment models the access control of a file system found in a system such as Linux. This model contains a user named `User_A`, an action called `Action_B` and an object called `Object_C`. The names chosen for this experiment are arbitrary but are used to simplify tracing each component throughout this example. In the model there is also a policy with the identifier 1001. Policy 1001 states that `User_A` can preform `Action_B` on `Object_C`. This can be compared to a user named Bob being able to read a file called documents.

Once the structure of the graph and the policies that traverse through the graph have been created, the model needs to be translated into an intermediate language that can be used in the next stage. This is where the HERMES policy language comes in. Because of its full parameterization and its flexibility as described in section 4.2, the HERMES language is a perfect intermediate step when converting from the HPol formal model. By using a series

```
SubDomain:Subjects
{
  Description: "Subjects";
  Path: "Example/Subjects";
  Type: Subjects;
  Children: ["User_A"];
}
```

Listing 6.2: Subject node written out in HERMES.

of graph tracing algorithms the structure of the HPol model is translated into a series of entries in the HERMES format. After that the policies of the model are then traced as well and are converted into HERMES as well. Listing 6.2 contains an example of the Subject node in HERMES form.

Figure 6.2 is data flow representation of how data is processed in the HPol framework. Data starts as device configuration files. This data is run through a custom configuration parser for each type of device. The output of this parser is HERMES data and a visual representation of the HPol Graph. This transformation can be seen in figure 7.1. An example of the HPol graph can be seen in figure 6.1. Next the data is run through the HERMES parser where it is turned into a Prolog knowledge-base. Listing 6.8 contains an example snippet of the created Prolog knowledge-base. Simultaneously, queries are created in HERMES and are fed into the querying mechanism to produce results. Listings 6.4 and 6.5 show queries written in HERMES and 6.10 the results of the queries.

As listing 6.2 shows, each node and policy in an HPol model is contained in a HERMES entry. This entry is composed of two parts, a header and a body. The header of an entry contains the type of the entry and the name of the entry separated by a colon. In listing 6.2 there exists a HERMES entry with the type `SubDomain` called `Subject`. All descendants of type `SubDomain` are of type `Node`. The domain and sub domain keywords are used to identify the root of the graph and its direct children. The body of an entry resides in curly braces. Inside the curly braces exists the properties of the graph. Properties in HERMES

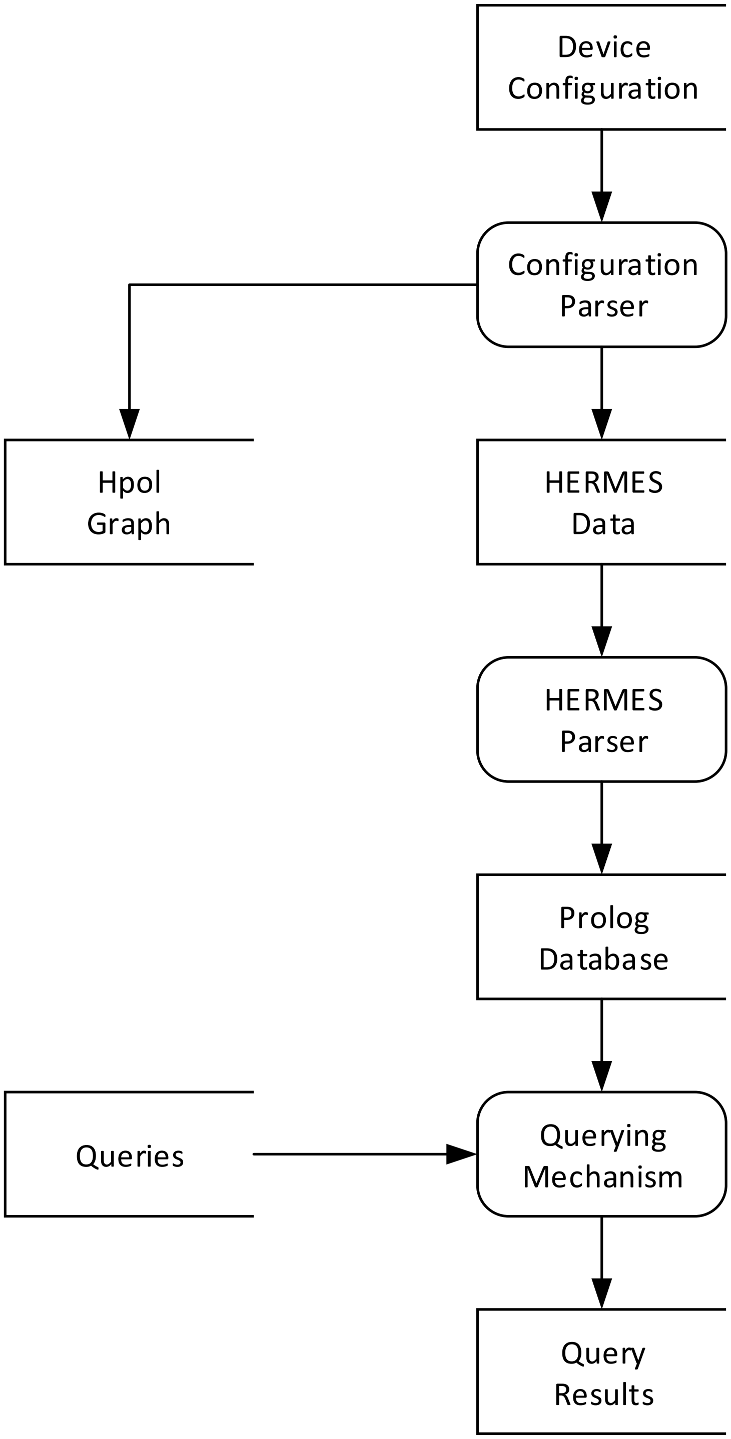

Figure 6.2: Data flow diagram showing the processing of the data from system configuration to query mechanism.

```
Policy: 1001
{
  Description: "HPol Policy";
  Status: Enabled;
  Path: [HPolStart , User_A , Action_B , Object_C , HPolEnd]
}
```

Listing 6.3: HERMES form of Policy 1001.

have a key value pair that are separated by a colon. Every property entry that is formed from by the HPol model will have the entries `Description`, `Path`, `Type`, and `Children`. The `Description` entry is the name of the node reiterated as a string so invalid characters can be used. The `Path` entry is the path of the given node starting at the root of the tree. Currently this field has no purpose and might be removed. The `Type` entry states the node's type. Finally, the `Children` entry identifies the children of the node. As seen in the graph this nodes path is `Example/Subject` and the only child of this node is `User_A`. All of this information will be extremely useful when it is time to reconstruct the graph from the HERMES source.

6.3 Policy Specification in HERMES

Polices are also modeled with the HERMES language. Listing 6.3 shows the HERMES form of Policy 1001. As listing 6.3 shows, policies are contained by the same type of entry as nodes are. There are, however, some differences. The entry-type and identifier syntax are still present as required by the HERMES syntax, but the properties of the entries are altered. Instead of `Type` and `Children` entries, there is now `Order`. Also, Path has a different meaning when referring to a policy. When a policy is represented in HERMES, `Path` is the path the policy takes through the graph and `Order` is the order of the graph nodes in the policy. There is also a status entry present. This states whether or not the policy should be active. When being generated from configuration files, all polices will receive the active

```
Query: 1
{
  Allow: yes;
  Subject: "User_A";
  Action: "Action_B";
  Object: "Object_C";
}
```

Listing 6.4: Example is allowed query written in HERMES.

status because the parsers can only find active polices. The Path entry contains a list of tuples. These tuples are type ID pairs for the nodes that are relevant to the path. The Order key also has a value of a list of tuples. These tuples contain order ID pairs to identify the place of a node in the policy.

6.4 Formal Query Structure in HERMES

Queries are also created in the HERMES format. Currently, queries enable asking allow and disallow questions. For example, if a systems administrator wanted to know if User_A could preform Action_B on Object_C, the query would look like listing 6.4. This query contains three major parts. First, the type of the entry is Query. Second, one of the key value pairs is the Allow rule. This states whether the evaluator should be checking if the action is allowed or disallowed in the HPol graph. Listing 6.5 contains an example of a query asking if the user, User_B, can preform the action, Action_D on the object, Object_C. While figure 6.1 contains an Action_D node, there is no policy that allows User_A to preform Action_D. Being able to make this distinction is a crucial part of the HPol framework. A system administrator needs to make sure an action is not possible just as much as he needs to make sure that one is. Finally, the Subject, Action, and Object keys have values that are used as the parameters of the query.

Listing 6.6 shows a second type of query. This query type only has one keyword in the body section of its entry. This keyword is Contains. The value of Contains is a list

```
Query: 2
{
  Allow: no;
  Subject: "User_B";
  Action: "Action_D";
  Object: "Object_C";
}
```

Listing 6.5: Example is not allowed query written in HERMES.

```
Query:3
{
  Contains:[User_A, Action_B, Object_C];
}
```

Listing 6.6: Query including `Contains` keyword.

containing nodes that are to be found in a policy. The purpose of the `contains` query is for searching for policies that contains nodes in the order they show up in the list. Every node does not have to be present in a `Contains` query for a result to be produced. Listing 6.6 shows the same nodes in question as listing 6.4 to show the similarities between the nodes. However, if a security professional just wanted to know a piece of the puzzle such as which polices involve `Object_C` then he or she would write a query similar to 6.7. Also, this form of query does not give regard to the context of the nodes it is searching for, this allows a faster searching through the policies when finding results.

```
Query:4
{
  Contains:[Object_C];
}
```

Listing 6.7: Query to search for polices containing `Object_C`.

```
entityDef(idhermes_out_Subjects, "Subjects", "hermes.out", "SubDomain").
entityField(idhermes_out_Subjects, "Description", "Subjects").
entityField(idhermes_out_Subjects, "Path", "Example/Subjects").
entityField(idhermes_out_Subjects, "Type", "Subjects").
entityField(idhermes_out_Subjects, "Children", ["User_A"]).
```

Listing 6.8: Prolog facts about the `Subject` node.

6.5 Loading HPol Formal Models into Prolog

Once all entries have been created in HERMES, the data needs to be stored in a Prolog knowledge-base. XSB Prolog is used as the Prolog engine during this process [17]. In order to complete this process, a translator is created to convert the HERMES format files into a Prolog knowledge base. The compiler for this task is written in Prolog as well. For each entry in a HERMES file, a series of Prolog facts are created. When a new entry is encountered, a new defining fact about the entry is created and is contained in the predicate `entityDef`. Next each key value pair of an entry is created into a series of facts contained within the predicate `entityField`. Each entry in HERMES is given a unique identifier that is composed of the input file name and the identifier it was originally assigned in HERMES. Listing 6.8 contains the entire set of Prolog facts stored about the Subject node referred in figure 6.1 and listing 6.2.

Because all data needs to be captured in order to recreate the configuration files, there is sometimes more data than needed contained within the Prolog facts. For the `Subject` node, there are many facts that are generated. First, an `entityDef` predicate is used to initially declare the existence of the `Subject` node. Next, `entityField` predicates are used to define the various fields found in the HERMES example in listing 6.2. The conversion between HERMES properties and Prolog facts is one-to-one except in the case when a dictionary is defined in the HERMES. In this case, each entry of the dictionary is created into its own `entityField` fact. Both types of predicates use a unique identifier as the first argument of

```
eval_query(ID, Policy):-
  entityDef(ID, _, _, "Query"),
  entityField(ID, "Subject", Subject),
  entityField(ID, "Action", Action),
  entityField(ID, "Object", Object),
  entityField(ID, "Allow", "yes"),
  atom(ID),!,
  policy_containing(Subject, Action, Object, Policy).
```

Listing 6.9: `eval_query` procedure written to evaluate queries.

the fact. This unique identifier is used to bind all facts about a given node or policy together.

6.6 Querying an HPol Formal Model

Once the Prolog knowledge-base is created, a series of Prolog procedures are created in order to evaluate a query. Listing 6.9 contains an example of one the procedures used to evaluate queries. Listing 6.10 contains the results of the two queries in figures 6.4 and 6.5. The first set of results represents the allow query in listing 6.4. This result is stating that the query with the identifier `idquery_herm_1` is being satisfied by the policy `idhermes_out_p1001` or policy 1001 from figure 6.1. It is important to note that the identifiers used internally by Prolog are generated at the time of compilation to facts. The original names of the policies and queries are contained in `entityDef` predicate of the Prolog knowledge base.

```
| ?- eval_query(QueryName, PolicyName).

QueryName = idquery_herm_1
PolicyName = idhermes_out_p1001;

QueryName = idquery_herm_2
PolicyName = _h180
```

Listing 6.10: Results of Queries 1 and 2.

The second set of results shows that there are no policies satisfying query `idquery_herm_2`. This is still a positive answer because query 2 is testing to make sure the action is not possible. Because there are no policies that allow the action in question, the query is correct. The `_h180` in place of the policy name is Prolog's way of stating that it can not find a valid result for the variable `PolicyName`.

This section demonstrates that through the use of an example it is possible to model a system and the policies that are in effect in that system. Further, this section illustrates that it is possible to convert the model into a database which can be queried for details about the system. The next section of this thesis demonstrates this concept with a real-world scenario.

CHAPTER 7

Case Study: Cisco Router Querying

The case study to test whether the HPol framework mechanism can model a real world scenario is done on the configurations of a pair of Cisco routers connected by a VPN tunnel. The configuration files of the two routers were parsed and converted into two separate HPol models as described in chapter 5. The model is similar to the `Subject`, `Actions`, `Objects` format as the previous example except that subnets and interface names are placed under the `Objects` subtree. Also, the send and encryption actions are placed in the `Actions` subtree. Because subnets can be treated as both subjects and objects, all subnets and interfaces were placed under the `Objects` subtree.

7.1 Querying LAN Policy

Policies in this model state which subnets or interfaces can send packets to a given destination, also a subnet or interface name. If encryption is needed, the policies pass through the appropriate encryption nodes to signify that encryption takes place when packets are sent to the destination. The model is designed in a way that only allowed actions are generated as policies in the model. A policy for the Cisco router HPol model states which subnets and interfaces can communicate with each other. For example, referring back to figure 5.2: policy 1004 of the headquarter's router starts at the policy start node, `HPolStart` then makes its way to the `10.1.3.0_24` node. Next, policy 1004 goes to the `FastEthernet0_0` node then the `Send` node and makes its way to the `10.1.6.0_24` node through the `FastEthernet0_1` node. Finally, the policy makes it to the `HPolEnd` node; this is where all policy paths must end. This policy reads as an IP address from the `10.1.3.0_24` subnet can send packets to an IP addresses in the `10.1.6.0_24` subnet.

Once the HPol model has been generated, the model is converted into HERMES. This action is the same as its counterpart in the previous section. Figure 7.1 shows the transition

```
interface FastEthernet0/0
ip address 10.1.3.3 255.255.255.0
no ip directed-broadcast
no keepalive
full-duplex
no cdp enable
```

```
Node: FastEthernet0_0
{
Description: "FastEthernet0_0";
Path: "base/Objects/
...Interfaces/FastEthernet0_0";
Type: object;
}
```

Figure 7.1: This figure contains the before (left) and after (right) when data is converted from a Cisco router configuration file to a HERMES entry.

of the Cisco configuration file into the HERMES representation of the same data. Much of the data from this node is stored in different areas of the model graph. For example the IP address and subnet of this interface is a parent of this node. Data is stored in this manner so other devices that fall under the same subnet can be placed in their proper place without having to copy or recreate data. The result in figure 7.1 shows the description of the node as well as the node's path and type. Because of how the path works in the HPol model, all forward slashes are replaced with underscores. Each parser contains its own caveats when transforming configurations into HERMES.

Once in HERMES the data is parsed. The data is then loaded into the Prolog knowledge base and is ready to query. Queries of each type are created to test the system. The first query asks if the `10.1.3.0_24` subnet can `send` to the `10.1.6.0_24` subnet. The second query wants to check that the `10.1.9.0_24` subnet cannot `send` to `10.1.6.0_24` subnet. Both of these queries return yes as a result. The first query result returns that policy `idhermes_out_p1004`, or policy 1004, satisfies the query. This means that according to the configuration that is parsed, it is possible for a device in the `10.1.3.0_24` subnet to send a packet to a device in the `10.1.6.0_24` subnet. It is also not possible for a device in the `10.1.9.0_24` subnet to `send` the `10.1.3.0_24` subnet.

7.2 Querying Tunnel Policy

In chapter 5 policy 1002 was traced with reasoning and evidence for each node chosen. Listing 7.1 shows the HERMES entry for every node involved with policy 1002 including the HERMES entry for policy 1002.

```
Node: 10.1.4.0_24
{
  Description: "10.1.4.0_24";
  Path: "remote/Subject/IP_addresses/Internal_IP_addresses
     ↪ /10.1.4.0_24";
  Type: Subject;
  Children: [10.1.4.2];
}

Node: FastEthernet0_0
{
  Description: "FastEthernet0_0";
  Path: "remote/Subject/Interfaces/FastEthernet0_0";
  Type: Subject;
}

Node: send
{
  Description: "send";
  Path: "remote/Actions/send";
  Type: Action;
}

Node: Tunnel1
{
  Description: "Tunnel1";
  Path: "remote/Objects/Interfaces/Tunnel1";
  Type: Object;
}

Node: s1first
{
  Description: "s1first";
  Path: "remote/Actions/crypto/s1first";
  Type: Action;
}

Node: Serial1_0
```

```
{
  Description: "Serial1_0";
  Path: "remote/Objects/Interfaces/Serial1_0";
  Type: Object;
}

Node: 172_17_2_4
{
  Description: "172.17.2.4";
  Path: "remote/Objects/IP_addresses/External_IP_addresses
     ↪ /172.17.2.4";
  Type: Object;
}

Node: 10_1_3_0_24
{
  Description: "10.1.3.0_24";
  Path: "remote/Objects/IP_addresses/External_IP_addresses
     ↪ /10.1.3.0_24";
  Type: Object;
}

Policy: p1002
{
  Description: "HPol Policy";
  Status: Enabled;
  Path: [HPolStart, 10.1.4.0_24, FastEthernet0_0, send, Tunnel1,
     ↪ s1first, Serial1_0, 172.17.2.4, 10.1.3.0_24, HPolEnd];
}
```

Listing 7.1: HERMES entries for all nodes associated with policy 1002.

Referring back to policy 1002, we know that this policy allows the 10.1.4.0/24 subnet to send to the 10.1.3.0/24 subnet through Tunnel1. The listing below shows the query that is required to ask if this is possible:

```
Query: 1
{
  Allow: yes;
  Subject: "10.1.4.0_24";
  Action: "send";
  Object: "10.1.3.0_24";
}
```

As shown above and in the previous section, the Subject, Action and Object query is used to verify that subnet 10.1.4.0/24 can send to the 10.1.3.0/24 subnet. The listing below shows this query being evaluated in the XSB Prolog interactive shell:

```
| ?- eval_query_by_name(1, A).

A = 1002

yes
```

In this query example the procedure `eval_query_by_name` is used to allow a user to evaluate the queries by the name given to the query while in HERMES form instead of the internal identifier that is used to separate queries from different files with the same name. This procedure also returns the identifier of the policy as it appears in the original HPol diagram. As the above listing shows, policy 1002 allows subnet 10.1.4.0/24 can send to subnet 10.1.3.0/24; just as was traced in chapter 5.

7.3 IP to IP Query

This thesis thus far has covered the process behind router-policy formal modeling and verification, which, in short, is as follows:

1. Model a router configuration file.

2. Convert the model into HERMES.

3. Create queries for the model in HERMES.

4. Load both model and queries into Prolog.

5. Use procedures created for this research to evaluate queries.

This thesis has shown the process is possible for both dummy examples and for real router configurations. However, everything that has been discussed to this point applies to

```
ip_send_query(IP1, IP2, Policy):-
  policy_containing(SubjectCodes, "send", ObjectCodes, Policy),
  fmt_write_string(Subject, "%s", args(SubjectCodes)),
  fmt_write_string(Object, "%s", args(ObjectCodes)),
  ipINsub(Subject, IP1),
  ipINsub(Object, IP2).
```

Listing 7.2: `ip_send_query` procedure of the IP to IP query evaluator

all HPol formal models for all devices because the previous query types referred to entities that explicitly existed on an HPol model. For example, subnets and interfaces explicitly exist on Cisco router formal models. To round out the research for this thesis, one more procedure was written that applies to only router policy models. Prolog procedures were written that could evaluate whether one IP address can send to another. This query is special because an extra layer of processing is required to evaluate it.

In the previous sections, queries were discussed regarding whether one subnet can communicate with another; however, individual IP addresses were never used, just subnets. In the IP to IP query, an evaluation is made to determine if source and destination IP addresses belong to subnets that exist in HPol router polices. These queries are then evaluated as `send` queries where the source IP address belongs to the `Subject` subnet, the action is `send`, and the destination IP address belongs to the `Object` subnet. Listing 7.2 contains the procedure that is at the center of the IP to IP query evaluator.

In section 5.4 an HPol policy was traced to verify that IP 10.1.4.3 could send to IP 10.1.3.6. That policy trace had to start at the subnet level due to limitations of the HPol formal model's ability to model routers and the lack of specific device IP address information in the router's configuration. Because of these setbacks, the policy had to trace from subnet 10.1.4.0/24 to subnet 10.1.3.0/24. For this last piece of research, a new type of query and new evaluation procedures were created in Prolog to allow a security specialist to write queries that pertain to individual IP addresses.

Listing 7.3 contains an example of an IP to IP query. In this query the same basic

```
Query: 1
{
  Allow: yes;
  Src: 10.1.4.3;
  Dest: 10.1.3.6;
}
```

Listing 7.3: Final type of query, IP to IP.

```
| ?- eval_query_by_name(1, A).

A = 1002

yes
```

Listing 7.4: Result of the IP to IP query.

HERMES structure remains. However, there is a variation in the key value pairs in this type of query. In this query type there is still the `Query` keyword for the type of HERMES entry and the `Allow` keyword remains. This is where the similarities end. There are now `Src` and `Dest` keywords that denote the source and destination IP addresses, respectively. The source and destination IP addresses in this query are 10.1.4.3 and 10.1.3.6, which are the same IP addresses being verified in section 5.4's policy trace.

Listing 7.4 shows the result of the IP to IP query. The policy that was returned in this query was policy 1002, the same policy traced in section 5.4 manually. This final type of query is significant because it demonstrates that the HPol formal model and the related framework can parse the configuration of a (Cisco) router and have the results semi-automatically verified through the use of queries. This provides security personnel a more refined way of determining if router configurations comply to the policies that are meant to define them.

7.4 Performance

Converting the Cisco HERMES output from the HPol model into a Prolog knowledge base takes about 5 seconds running XSB Prolog in single core mode with a 2.00 GHz computer running with 12 GB of RAM. However when running the XSB process takes less than 50 MB of RAM. Once the HERMES data is converted into a Prolog knowledge base and loaded, it takes a negligible amount of time to evaluate each query. Fortunately, unless the HERMES data changes, it only needs to be converted once.

7.5 Discussion

The results of the queries demonstrate it is possible to successfully query the configuration of a network component such as a Cisco router. For both allow and deny queries, the Prolog query system can identify queries and all policies that satisfy any given query. The system can also take a query identifier as an argument and return all policies that satisfy the query. It has also been discovered due to the nature of Prolog that the reverse is acceptable; a policy can be given and all queries that the policy satisfy will be returned.

In this chapter three examples of a case study were presented where the HPol formal models of two Cisco routers were queried. For both the LAN query and the tunnel query, accurate results were produced. In the final example, it was shown queries that allow the verification of IP address communication can be used to further verify the correctness of router policies.

CHAPTER 8

Future Work

The HPol framework is still within its infancy. There are many different components that need to be combined or created in order for it to reach its full potential. Parsers need to be generated for additional types of devices such as firewalls and managed switches. Policy concatenation also needs to be implemented. This will allow queries to be generated across a variety of components of an enterprise network. The ability to configure components through HPol also needs to be implemented. Finally, a GUI should be set up to graphically organize all of the data presented by an HPol policy.

8.1 Additional Device Configuration Parsers

In order to prove HPol's flexibility, parsers need to first be created for every type of component found in an enterprise network. Firewalls and managed switches are a large part of the networking side of an enterprise network. File system access control is another field that needs to be researched to increase HPol's flexibility. Once these components have HPol parsers, the model for each type can be formalized. Next the different brands and models would have to have parses created for them to match the model created for the general component template.

8.2 Formal Policy Concatenation

In order for HPol to evaluate an enterprise network as a whole policy, models have to be linked together. The hypothesis is that if the start and end nodes of multiple policies are connected, then it is possible to query a policy on a larger scale. For example, if two router policies are linked together, then it is possible to query whether an IP address in subnet X on router A can send a packet to the IP address of a subnet in router B. However, it is not possible to determine if the policy is correct when looking at each router's configuration separately. When looking at Router B's configuration in isolation, it is only possible to determine whether router A can send packets to a subnet in router B. By look at only this configuration, there is no way of knowing the origin of the packet from router A. Furthermore, when looking at router A's configuration separately, it can be seen that an IP in subnet X can send a packet to router B. This, however, is still not enough information to determine whether the policy is in effect. Only when the two models are combined is it possible to see that packets from a specific IP address in a subnet of router A can be sent to one of router B's subnets. This is enough information to determine that the policy is correct.

For example, figures 5.2 and 5.3 show the policy graphs for the two routers used in the given VPN configuration. However, the overall system policy for this VPN Tunnel cannot be observed by analyzing the configurations of the two routers separately.

In order to be able to analyze a network tunnel system policy in a holistic way, a complex system security policies needs to be built starting from the policies of the composing components of the system. The HPol formal model enables this by the introduction of the concatenation of security policies formal operation. The Cisco VPN Tunnel policy model is used to demonstrate this operation.

By superimposing the *End* node in the *hq-sanjose* (Left) router to the *Start* node in the *ro-rtp* (Right) router, a system wide and formal network policy that models the complete VPN Tunnel configuration is created.

This formal Policy Concatenation Operation, in conjunction with other additional operations, would enable the formal construction and analysis of system-wide and holistic security policies. In this case, the visualization and analysis of the complete VPN Tunnel system policy is possible by concatenating the policies of the two integrating routers.

While formal policy concatenation operations seem trivial, there is a greater more complex problem in this operation. Currently, HPol formal models have no since of self and are unable to identify themselves in another model. Once HPol models are able to identify themselves in other models then policy concatenation can be implemented.

8.3 Policy-Based Device Configuring

Once parsing and policy concatenation are implemented, policy-based configuration functionality can be developed. This functionality enables a system administrator to state a policy and have the HPol framework evaluate that policy, determine which configurations need to be changed to place the policy in effect, and then apply the changes to the configurations. This functionality is critical to the HPol model. When configurations are parsed, every detail from the old configurations needs to be saved in order to create the new configurations based on both old and newly assigned policies.

CHAPTER 9

Summary and Conclusions

In this thesis, the process involved in converting a Cisco router configuration into an HPol formal model has been demonstrated. Also introduced was policy querying of an HPol formal model with evidence provided using both an example and a case study.

When converting a Cisco router policy to an HPol formal model there are some key elements. First, all subnets and interface names are placed under the `Objects` DAG. Any node that can also act as the subject of a policy is then duplicated under the `Subjects` DAG. Second, the `send` node and all encryption types are placed under the `Actions` DAG. Finally, all policies in this model begin with either a subnet or interface node continue to a send or an encryption node and finish at an interface or subnet node.

Policy queries can be broken down into easy steps. First, all HPol models are converted to the HERMES language. Second, once in HERMES, the data is read into a Prolog knowledge base. Also part of the second set, queries about the model are written in HERMES and are loaded into the Prolog knowledge base. Finally, Prolog evaluates the queries with the help of some procedures written for this project.

Both of these operations are described in this thesis using simple examples and Cisco Router configurations. The conclusion of these operations is that it is possible to both model Cisco routers, or routers in general, and then query those models for information about the router's configuration.

Through the use of the HPol formal model and the associated framework, security personnel also have the ability to verify the correctness of a Cisco router's configuration. The IP to IP query enables security personnel to write queries for specific IP addresses to further increase the granularity of router policy correctness verification.

Modeling and querying different types of devices is important in industry because system administrators need to be able to know the current status of devices without having to go out and collect and analyze the data manually. A modeling system such as HPol can shift the responsibilities of system administrators from worrying about configuring devices correctly to worrying more about the policies that the configurations enforce.

References

[1] Cisco IOS VPN Configuration Guide: Site-to-Site and Extranet VPN Business Scenarios. http://www.cisco.com/c/en/us/td/docs/security/vpn_modules/6342/vpn_cg/6342site3.html, 2014.

[2] Kathi Fisler, Shriram Krishnamurthi, Leo A Meyerovich, and Michael Carl Tschantz. Verification and Change-impact Analysis of Access-control policies. In *Proceedings of the 27th international conference on Software engineering*, pages 196–205. ACM, 2005.

[3] J. D. Guttman. Security goals: Packet trajectories and strand spaces. In R. Focardi and F. Gorrieri, editors, *Foundations of Security Analysis and Design - Tutorial Lectures*, pages 197–261. Springer, 2001.

[4] Joshua D. Guttman and Amy L. Herzog. Rigorous automated network security management. *Int. J. Information Security*, 4(1-2):29–48, 2005.

[5] Joshua D. Guttman, Amy L. Herzog, John D. Ramsdell, and Clement W. Skorupka. Verifying information flow goals in security-enhanced linux. *Journal of Computer Security*, 13:115–134, 2005.

[6] Susan Hansche, John Berti, and Chris Hare. *Official ISC2 Guide To The CISSP Exam.* Auerbach Publications, 2004.

[7] Ananth Jillepalli and Daniel Conte de Leon. An architecture for a policy-oriented web browser configuration management system - HiFiPol: Browser. In *Proc. IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, Atlanta, Georgia, USA, 2016.

[8] Ananth A. Jillepalli, Daniel Conte de Leon, Stuart Steiner, and Frederick Sheldon. HERMES: a high-level policy language for high-granularity enterprise-wide secure browser

configuration management. In *Proc. 2016 IEEE Symposium Series on Computational Intelligence (SSCI-2016)*. IEEE, December 2016.

[9] Igor Kotenko and Olga Polubelova. Verification of security policy filtering rules by model checking. In *Intelligent Data Acquisition and Advanced Computing Systems (IDAACS), 2011 IEEE 6th International Conference on*, volume 2, pages 706–710. IEEE, 2011.

[10] Timothy Nelson, Christopher Barratt, Daniel J Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The margrave tool for firewall analysis. In *LISA*, 2010.

[11] David Michael Pennington. Welcome to ciscoconfparse's documentation! `http://www.pennington.net/py/ciscoconfparse/index.html`, 2015.

[12] Jesús D Jiménez Re, Jorge Bernal Bernabé, Félix J García Clemente, Juan M Marín Pérez, Jose M Alcaraz Calero, Gregorio Martínez Pérez, and Antonio F Gómez Skarmeta. Security policy specification. *Network and Traffic Engineering in Emerging Distributed Computing Applications*, page 66, 2012.

[13] Basit Shafiq, Ammar Masood, James Joshi, and Arif Ghafoor. A role-based access control policy verification framework for real-time systems. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*, pages 13–20. IEEE, 2005.

[14] F. J. Thayer Fabrega, J. C. Herzog, and J. D. Guttman. Honest ideals on strand spaces. In *Proceedings of the 11th IEEE Computer Security Foundations Workshop*, 1998.

[15] F. J. Thayer Fabrega, J. C. Herzog, and J. D. Guttman. Strand spaces: Why is a security protocol correct? In *Proceedings of the 1998 Conference on Security and Privacy (S&P-98)*, pages 160–171. IEEE Press, May 1998.

[16] F. J. Thayer Fabrega, J. C. Hezog, and J. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(2-3):191–230, 1999.

[17] Theresa Swift and David S. Warren. The xsb system. `http://xsb.sourceforge.net/manual1/manual1.pdf`.

[18] Jared Zook. High-level modeling and verification of mandatory access control policies across multiple security-enhanced linux devices. Master's thesis, University of Idaho, December 2016.