UiO : Department of Informatics
University of Oslo

Towards an unified policy for Next-Generation Firewalls

Creating a high-level language for NGFWs

Aslak Gaaserud
master thesis spring 2013

Towards an unified policy for Next-Generation Firewalls

Aslak Gaaserud

23rd May 2013

Abstract

Computer applications are becoming more and more advanced, pushing the evolution of security mechanisms in computer networks. For two decades one of the most widespread and efficient security mechanisms has been the network firewall. To keep up with the ever changing threat landscape, more security features than ever before has been implemented into the firewall, creating a new generation of firewalls named the next-generation.

The increasing amount of next-generation firewalls hitting the commercial market, shows that most vendors have their own definition of what features a next-generation firewall should hold. While most traditional firewalls generally operate by utilizing the same properties, such a common platform has yet to be established among next-generation firewalls.

By gathering features from various next-generation firewall products, the possibilites for a common platform is investigated. This platform forms the basis for a universal high-level language, a language designed to build and deploy security policies across vendor platforms.

To show how this universal language can be used in a real world setting, an expandable software prototype tool has been developed, designed to convert policies written in the universal language to operational policies used in firewalls. Investigation of language qualities, as well as significant differences between the prototype and vendor vendor tools, are measured and analysed through a series of experiments.

Acknowledgements

I would like to express my thanks to my supervisor Kyrre Begnum for his support, valuable input and contributions to the programming. It is certain that his influence has affected the end result of this thesis for the better.

A huge thanks to all the helpful people at mnemonic AS, especially Jon-Finngard Moe, for all the guidance and quality assurance, and for providing me with necessary laboratory equipment throughout the project period.

My friend, Erling Hagen, deserves a special thanks for offering his technical expertise and help during the development and experiment phases.

Finally, I would like to extend my gratitude to Oslo University College and the University of Oslo for six rewarding years as a student, and for giving me the opportunity to attend the network and system administration masters programme.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Network security is an immense topic that embraces all aspects of securing network infrastructure, including network monitoring, management of network access and protection of network resources. The field of network security has grown tremendously over the past years as we see new threats within the realm of computer networking, often driven by individuals and groups looking for financial gain, and by some just seeking the thrill of it. This growth is undoubtedly a result of the expansion of Internet and the increased number of businesses and organizations moving their sales and information channels online. It is a constant race between people with malicious intent exploiting vulnerabilities and administrators working to secure their networks. There is a multitude of hardware and software solutions available to aid in the fight for mitigating network threats, and one traditional and widely used component is the firewall.

Firewalls for large scale applications, like corporate use, are an essential part of an organizations infrastructure. Their role is to enforce corporate security policies by controlling data flow and log events that occur in order to identify attacks and detect anomalies. A firewall normally resides at the perimeter of a network, and when properly configured it makes sure no unwanted traffic reaches or leaves the network. In segmented networks, firewalls are often used as an intermediary to separate segments, making it possible to operate with a variety of security policies within different network zones. Despite the necessary first line of defense that firewalls provide, using only firewalls to secure a network is in most cases not adequate to fully enforce a security policy.

Several components can be used in conjunction with firewalls to provide a more extensive security infrastructure. These components can be systems for intrusion detection and prevention, VPN-solutions, malware and reputation filters, anti-virus and more. Together these components form a sustainable security ecosystem, commonly found in organizations with comprehensive network security practices. Running these kinds of ecosystems can be complex. There might be numerous different systems to manage, and a number of physical boxes that needs to work together in a concerted

effort to maintain a secure network environment.

In the recent years a new network security technology has surfaced, namely the next-generation firewall. The next-generation firewall is meant to be a successor to port-based and stateful firewalls[19], consolidating a number of security features into one, unified platform. While keeping firewall functionality, additional security features include intrusion detection and prevention, reputation and malware filtering as well as user and application control[18]. This new breed of firewalls typically have all their functionality compacted into one physical box, taking up less space than traditional security infrastructure containing the same features. They gather all management operations into one unified interface, opposed to the traditional approach where security components are spread out over multiple point solutions using various interfaces.

Firewalls operate by sets of rules that determine which network traffic is to be allowed and which is to be blocked. Rules can be specified for both inbound and outbound traffic and together a collection of rules make up a rule set, or firewall policy. Just like traditional firewalls, next-generation firewalls are dependant on policies to operate. Rules in a next-generation firewall policy can include a higher number of policy defining elements compared to traditional firewall rules. In addition to typical elements found in firewall rules like source addresses, destination addresses and ports, next-generation firewalls support user- and application data, making them much more customizable in terms of user and application control. This increased complexity means many legacy firewall rules needs to be expanded in order to fit in to next-generation firewall policies.

Guidelines and recommendations on how to build optimal firewall policies regardless of vendor, have evolved for over two decades, documented in the published works of a number of reputable organizations such as NIST[24], CERT[4] and SANS[23], and there has been several successful attempts to create universal languages for firewall rule sets. As next-generation firewall technology replaces existing firewalls in the years ahead, comes the job of migrating complex and comprehensive security policies from legacy firewalls to next-generation firewalls. The migration process and development of new policies for next-generation firewalls can be greatly simplified with the help of a high-level language designed to build universal policies for all kinds of next-generation firewalls.

According to Infiniti Research[7], the prevalence of next-generation firewalls is expected to expand with an annual rate of nearly 17% in the years to come, and the need for new and updated policies for these firewalls is imminent. Current sets of firewall rules will have to be re-written in order to facilitate the features of next-generation firewalls. While traditional firewall policies are generally constructed from mostly similar properties. This does not apply for next-generation firewalls as they to date have no common structure, hence the need for a high-level language that can be used to

create policies across vendor platforms.

1.1 Problem statement

While in traditional firewall configurations one can normally find the same features and properties in most products, but when it comes to next-generation firewalls many vendors seem to have gone different ways in the development process, leading to the fact that a common platform is virtually non-existent. Next-generation firewalls all include legacy firewall capabilities, but in addition to that they include a varying degree of added functionality that can make it hard to build and compare similar policies on different next-generation firewalls. Normally, a policy will have to be especially written and customised to match exact firewall requirements. Currently there are very few, if any, methods that can help on the way towards unified policies for next-generation firewalls, which leads us to the following questions:

Q_1: *How can a common platform be made for next-generation firewall rule structure and policies?*

Q_2: *How can we make sure a universal language is a functional solution for policy management in next-generation firewalls?*

Q_3: *What can we achieve by making a universal language, and how can it be evaluated against traditional vendor tools?*

1.2 Thesis objectives

As an attempt to remedy the situation of the dispersion in next-generation firewall products, the focus of this thesis is the development of a code based, high-level language that makes it possible to build security policies that can be implemented on next-generation firewalls independent of brand and platform. Being a critical component in network security infrastructure it is crucial that such a language will not compromise security, making it a subject to thorough evaluation. After completing a language development process, quality control and proper testing will need to be performed.

Reflecting the questions Q1-Q3, the objectives of this thesis are:

O_1: *In order to create a unified policy language, overview must be gained of the next-generations difference in properties, to examine how a common platform can be made. Collecting configurations and properties from a representable number of various next-generation firewalls can form the basis for a prototype language.*

O_2: *Find a future-proof language concept to ensure expandability and both forward and backwards compatibility. Develop a language that simplifies the process*

of building security policies for next-generation firewalls that can be used on multiple firewall products. Develop or use existing tools to compile and implement policies.

O_3: *Perform realistic experiments using the universal language to uncover flaws and inconsistencies, and evaluate significant aspects of the languages qualities, both on its own and compared to native languages.*

1.3 Thesis outline

This thesis is organized into 7 chapters.

Chapter 1: Introduction provides an overview of the problems asked in this thesis, as well as goal objectives that descibe methods that might give some answers to these problems. The problem statement is defined in this chapter.

Chapter 2: Background describes internet history and firewall evolution up till the first next-generation firewall in 2007. This chapter also includes a related work section, that presents some previous attempts to build universal firewall languages.

Chapter 3: Approach concerns project methodology and planning of a prototype, and describe how experiments are to be carried out.

Chapter 4: Results from the approach chapter are presented in this chapter. Here the prototype from the development process and the experiment results are presented.

Chapter 5: Analysis deals with comparing the prototype to similar vendor tools, and the results from the previous chapter are analysed.

Chapter 6: Discussion gives a review of what has been accomplished, ideas for further development and obstacles encountered during the project.

Chapter 7: Conclusion ties together the issues raised in the discussion chapter and reflects on the introductory problem statement.

Chapter 2

Background

Next-generation firewall technology did not arise out of thin air. The next-generation firewalls are a response to a constantly evolving threat landscape, leading to a need of increasingly sophisticated security measures within computer networks. The sections ahead tell the story of some highly influential events in network history, leading to the birth of the firewall and intrusion detection systems, and finally the introduction of the first known next-generation firewall. A history summarization is illustrated in figure 2.2.

2.1 Internet History

Computer networking concepts were first described in 1962 by Joseph Carl Robnett Licklider in a series of memos were he imagined a global group of computers that gave everyone access to their resources from any location[[12], Figure 2.2:B], much like the Internet we know today. In October that year Licklider began working as head scientist for the computer research program at the Defense Advanced Research Projects Agency (DARPA), a division of the United States Department of Defense. During his employment at DARPA Licklider presented his ideas about this global network to his successors, one of them a man by the name of Lawrence G. Roberts.

The first paper published on packet switching was written by the UCLA professor Leonard Kleinrock in 1961[[10], Figure 2.2:A]. Packet switching is a way of transmitting data by splitting data streams into blocks, and it is the fundamental technology used in Internet and local area network communication today. Kleinrock presented the theoretical feasibility of using packets instead of circuits in communications to Roberts leading to a ground breaking experiment carried out in 1965, when Roberts teamed up with Thomas Merrill to build the first ever wide-area computer network(WAN)[Figure 2.2:C]. This WAN connected a computer located at MIT to a computer located at UCLA using a dedicated 1200 bps dial-up telephone line. The experiment was successful, but concluded that circuit switched networks were inadequate for such purposes[22]. Based on the

ideas of Kleinrock it was suggested that packet switching methods might be a better approach.

Using his findings Roberts started making a draft on a network concept called ARPANET[[21], Figure 2.2:D]. A paper on ARPANET was presented at a conference in 1967, along with a paper on packet networks by Donald Davies and Roger Scantlebury of UKs National Physical Laboratory. Another attendee at the same conference, Paul Baran from RAND, an organization that does research for the US Army, revealed that he had done research in the field as well, uncovering the fact that the three parties had been researching the same subject without the knowledge of each others work.

The specifications for ARPANET were completed in 1968, but one key component was missing; the packet switches, which is fundamentally used to store and forward packets. DARPA issued a request for quotation for the development of the packet switches, at the time called Interface Message Processors(IMP). Late that year the request was won by research and development organization BBN. September 1969 the IMP was ready and UCLA was connected to ARPANET as the first node[Figure 2.2:E]. Stanford Research Institute followed one month later and the first host-to-host message was sent over the network. By the end of the year four nodes were connected, the last two included UC Santa Barbara and the University of Utah.

In the years to come ARPANET grew quickly and propelled the need for more protocols and networking software. In 1972 the first email system was demonstrated to the public using basic send and read software, making it the predecessor of what we know as todays email and social networking[Figure 2.2:F]. The same year Bob Kahn from BBN, who had been heavily involved with the development of IMPs for DARPA a couple of years earlier, once again found himself working for DARPA. This time to work on a program called ”internetting”, which goal was to develop an end-to-end protocol that could withstand network drop outs and interference. The protocol was developed in collaboration with Vinton Gray Cerf and later evolved into the Transmission Control Program / Internet Protocol (TCP/IP) protocol. In 1980 TCP/IP was adopted as a defense standard, and after two years it was implemented into ARPANET[Figure 2.2:G]. Today TCP/IPis a part of the Internet protocol suite, the most common stack of protocols used for communication on the Internet.

ARPANET eventually developed into the Internet, and by 1988 it had grown into a community of organizations that used the network as a communication channel for exchanging information. On November 2. an event occurred that was to change Internet history forever[Figure 2.2:H]. A message from Peter Yee at the NASA Ames Research Center hit the TCP/IP mailing list saying:

"We are currently under attack from an Internet VIRUS! It has hit Berkeley, UC San Diego, Lawrence Livermore, Stanford, and NASA Ames."

In fact, what Yee described as a virus, was actually the first known propagating network worm, the Morris worm, unleashed by Robert Tappan Morris Jr., a graduate student at Cornell University and son of a National Security Agency scientist. It was clear that the Internet was no longer a safe haven for trusted colleagues, and security measures had to be taken from now on.

2.2 Firewall and Intrusion Detection Evolution

The concept of using physical obstacles to keep out intruders dates back thousands of years. Take the great wall of China as an example, it was built over two thousand years ago to prevent invasions from the north. The term *"firewall"* was originally used to describe a wall separating parts of a structure that were more likely to have a fire from the rest of the structure, in order to slow down the spread of the fire. To this day the term has preserved its original meaning, but it has also become a common expression in automotive and aircraft industry, buildings and constructions, electrical transformer stations and in computing.

In the field of computing, a firewall is a device or a program that controls traffic flow between computer networks. Firewalls are often mentioned when discussing Internet connectivity, but they are also commonly used for restricting traffic to and from internal networks. Organizations employ firewalls to control network traffic to and from these areas to prevent unauthorized access to systems and resources, and as a way of enforcing security policies. Firewalls are often also used as traffic loggers, keeping records over time to provide traceable logs. A security policy is a detailed specification of security properties included in a system. It defines goals for security mechanisms like firewalls, in which case these policies are often referred to as network policies.

The theory about firewalls in computer networks was first presented in 1988[[9], Figure 2.2:I] by engineers at Digital Equipment Corporation(DEC). DEC was acquired by Compaq in 1998 and is now a part of Hewlett-Packard. In the late 80s, what can be considered as early firewalls were actually routers used to split networks into smaller LANs to prevent noise spilling over from one LAN to another, not to enforce security policies or as a security precaution against attackers. With the turn of the decade routers with filtering rules emerged as the first security firewalls used to enforce security policies. Developed in 1991 and shipped in 1992 the first commercial firewall product was the DEC SEAL, an abbreviation for Secure External Access Link[[20], Figure 2.2:K]. The SEAL consisted of a system called *"Gatekeeper"* as its only link to the Internet, together with a gateway and a mail hub[Figure 2.1].

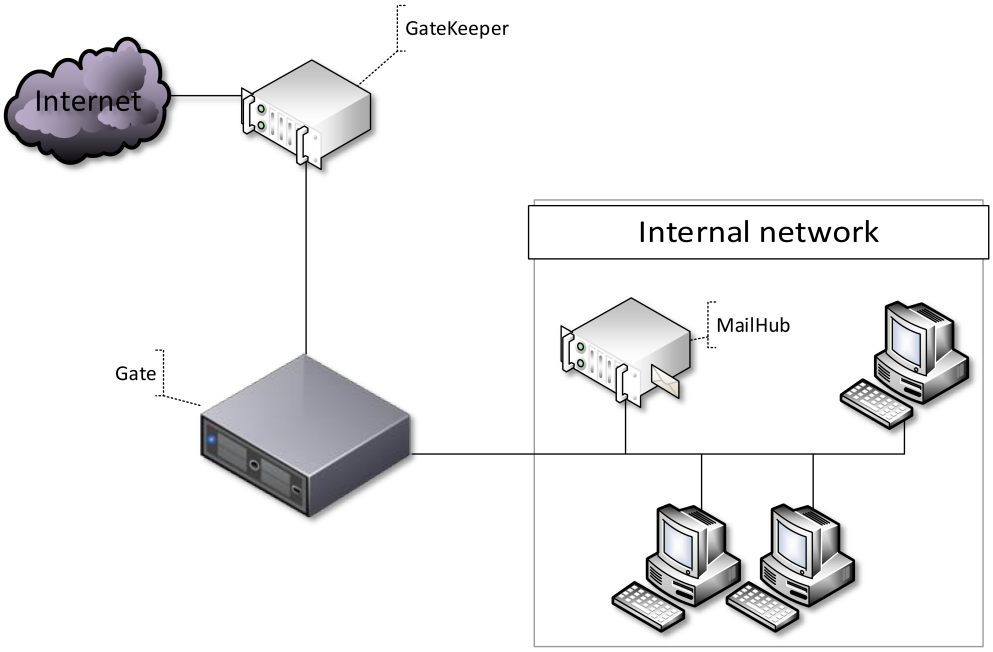

Figure 2.1: DEC SEAL

The SEAL was followed by other commercial solutions like the Ball Labs Raptor Eagle and the ANS Interlock, paving the way for more firewall products throughout the 90s.

2.3 First generation: Packet Filtering Firewalls

The first generation of firewalls are considered to be the most basic kind of firewalls, screening packets based on IP addresses, ports and services requested, hence they are often called packet filtering firewalls or screening routers. Today packet filtering is still one of the most important tasks for any kind of firewall, as the firewall is, in most cases, the first destination for any traffic travelling to or from an internal network.

A packet filter acts by inspecting each packet passing through the firewall. If a packet matches the criteria of one or more firewall rules, the filter can decide to drop, reject or accept the packet. Dropping the packet means silently discarding it, while rejecting a packet means the firewall sends an error message to the packet's source before discarding it. These decisions are made based on information found in the headers of each packet containing vital packet data.

First generation firewall are considered to be ”stateless” since they do not take into account that the packets they are inspecting might be a part of a network session consisting of a number of coherent packets.

2.4 Second generation: Stateful Packet Inspection

The concepts of the second generation of firewalls were conceived during the period 1989 to 1990 by Dave Presetto, Janardan Sharma and Kshitij Nigam at AT&T Bell Laboratories [[5], Figure 2.2:J]. Initially this new technology was called circuit level, but it was later re-named when Check Point introduced the concept of stateful packet inspection to the commercial market with their FireWall-1 in 1994[[25], Figure 2.2:M].

In addition to packet filtering, a firewall with stateful packet inspection includes the state and context of packets, thus keeping track of open connections. Any packets going out is tracked by the filter and when packets arrive the firewall can tell whether or not the incoming packet is a reply to the sent packet. A stateful firewall has the capability of examining more than just header information, it can also inspect contents of a packet up through the application layer to gather more information about the packet than its source and destination address.

The collected packet information is stored in a record of all established conversations called state table or session table. Using this method the speed of network traffic flowing through the firewall is greatly increased. As the first step in handling inbound traffic, the firewall can read from the state table rather than the more time consuming task of having to iterate through its rules.

Stateful packet inspection was not the only revolutionary innovations the FireWall-1 had on board. Up to the release of the FireWall-1, administration and configuration of firewalls was done editing ASCII files. The Firewall-1 was operated with the help of a graphical, mouse controlled user interface, greatly simplifying firewall management.

2.5 Third generation: Application Gateways

Application gateways, also referred to as proxy-based firewalls and application layer firewall, have the ability to inspect data located at application level and provide elaborate logging. Jeffrey Mogul[14] described a system working at the application level in 1991, but it was not until 1993 that Trusted Information Systems introduced an open-source firewall called the Firewall Toolkit[Figure 2.2:L], taking an important step forward in the field of application gateways. Although application level inspection should be credited to the DEC SEAL which pioneered the technology two years earlier. The Firewall Toolkit was later commercialized under the name Gauntlet Firewall, incorporating application control, anti-malware and URL filtering, greatly expanding the firewall field of operation.

Application layer firewalls do not make decision on whether to permit or block network traffic based on ports like packet filters. Decisions are made

by inspecting the data layer of the packet, making it possible to filter traffic based on application or service.

A proxy firewall acts a middleman between communicating parties. It can break the TCP/IP connection and re-establish a new connection to each system acting on the behalf of the client to obtain the requested service, making it appear like outgoing traffic is originating from the firewall, not the internal host. This feature is useful when a client and server are incompatible for direct connection and it makes the gateway capable of network address translation(NAT).

2.6 Hybrid Firewalls

While in theory, firewalls are split up into specific types, in most real-world settings firewalls are hybrids of two or more of the technologies mentioned in the previous sub-sections. The concept of the hybrid firewall dates all the way back to the DEC SEAL, which was a mix of a proxy and a packet filter illustrated in figure 2.1. Today it is possible to combine the advantages of different firewall platforms into a multilayer inspecting architecture with the ability to make use of packet filtering to provide high throughput low-risk traffic, deep packet inspection for more thorough packet content investigation and application layer filtering for data-driven attacks.

2.7 Intrusion Detection Systems

Firewalls provide excellent methods for enforcing security policies in networks, but firewalls alone are not sufficient to ward off the highly developed range of attacks observed today. With the seemingly endless amount of new Internet threats discovered daily, it is crucial striving to stay ahead of the game to avoid intruders and malware from compromising your network. Symantecs annual Internet Security Threat Report for 2011[31] revealed a surge of 81% in network attacks and an increase in malware variants by 41% from the previous year, boosting the need for network hedging solutions. No single security measure can recognize or stop all kinds of attacks, thus firewalls are often partnered with intrusion detection systems, creating an intrusion prevention systems to form a more secure platform.

Intrusion prevention systems(IPS) are extensions of intrusion detection systems. In addition to identifying network threats they have the ability to perform proactive response protection to a network by blocking malicious activity. An IPS is always situated in-line to actively detect and prevent unwanted network traffic.

The idea of intrusion detection was born in 1980, when James P. Anderson at USAF published a seminal study describing methods for monitor-

ing networks using accounting audit files[Figure 2.2:O]. This is considered to be a predecessing theory to the first intrusion detection system named Intrusion Detection Expert System(IDES)[Figure 2.2::P]. The IDES was developed by Dorothy Denning and Peter Neumann between 1984 and 1986. It was a real-time rule-based IDS that could detect known malicious activity. Throughout the 1990s the US government funded research on several intrusion detection systems[ref: history 344], and in the mid 1990s commercial products became available to the masses. Netranger[Figure 2.2:Q] and RealSecure[Figure 2.2:R] were two popular solutions, developed by respectively Wheelgroup and Internet Security Systems(ISS).

Wheelgroup was aquired by Cisco in 1998 and is now a part of Cisco's security department. Netranger still exists in a re-engineered form as the Cisco Adaptive Security Appliance. ISS released their first commercial IDS named RealSecure 1.0 in 1997. The company was bought by IBM in 2006 and currently goes by the name of IBM Internet Security Systems. In 1998 a widely used open source IDS/IPS tool called Snort saw the light of day[Figure 2.2:S]. Snort is maintained by Sourcefire which uses Snort technology in their next-generation firewall products.

Market statistic reveal that intrusion detection systems are among the top selling network security technologies, and according to Frost & Sullivan[6] they are predicted to remain in that position for years to come.

2.8 Next-generation Firewalls

In the early days of computer networking, security was relatively simple. Network traffic could be of two states, either good or bad. The approach was traditionally to permit "good" traffic and block everything seemingly "bad", and firewalls did a good job distinguishing between benign and malicious traffic. Today this mentality can cause problems due to more sophisticated evasive techniques used by applications to gain network access, and the blurring line between business- and user- applications, making it difficult to categorize traffic. As a response to the change in the threat landscape, security vendors came up with a product blending firewall technology with intrusion detection systems, making a unified network security platform known as the next-generation firewall.

The next-generation firewall collects features from all generations of firewalls and intrusion detection systems into one physical product promising:

- High throughput
- All-in-one functionality
- Single management interface
- Bulky appearance
- Lower cost ownership
- Better control, visibility and security
- Reduced rule-set complexity
- Encrypted traffic inspection

Palo Alto was the first vendor to release products falling into the next-generation firewall category with their PA-4000 series in 2007[Figure 2.2:N], two years before Gartner[18] published their definition of a next-generation firewall.

According to Gartner the following attributes must be implemented for a product to be regarded as a next-generation firewall:

> Support in-line bump-in-the-wire configuration without disrupting network operations.
>
> Act as a platform for network traffic inspection and network security policy enforcement, with the following minimum features:
>
> Standard first-generation firewall capabilities: Use packet filtering, network- address translation (NAT), stateful protocol inspection, VPN capabilities and so on.
>
> Integrated rather than merely co-located network intrusion prevention:
>
> Support vulnerability-facing signatures and threat-facing signatures. The IPS interaction with the firewall should be greater than the sum of the parts, such as providing a suggested firewall rule to block an address that is continually loading the IPS with bad traffic. This exemplifies that, in the NGFW, it is the firewall correlates rather than the operator having to derive and implement solutions across consoles. Having high quality in the integrated IPS engine and signatures is a primary characteristic. Integration can include features such as providing suggested blocking at the firewall based on IPS inspection of sites

only providing malware.

Application awareness and full stack visibility: Identify applications and enforce network security policy at the application layer independent of port and protocol versus only ports, protocols and services. Examples include the ability to allow Skype use but disable file sharing within Skype or to always block GoToMyPC.

Extrafirewall intelligence: Bring information from sources outside the firewall to make improved blocking decisions, or have an optimized blocking rule base. Examples include using directory integration to tie blocking to user identity, or having blacklists and whitelists of addresses.

Support upgrade paths for integration of new information feeds and new techniques to address future threats.

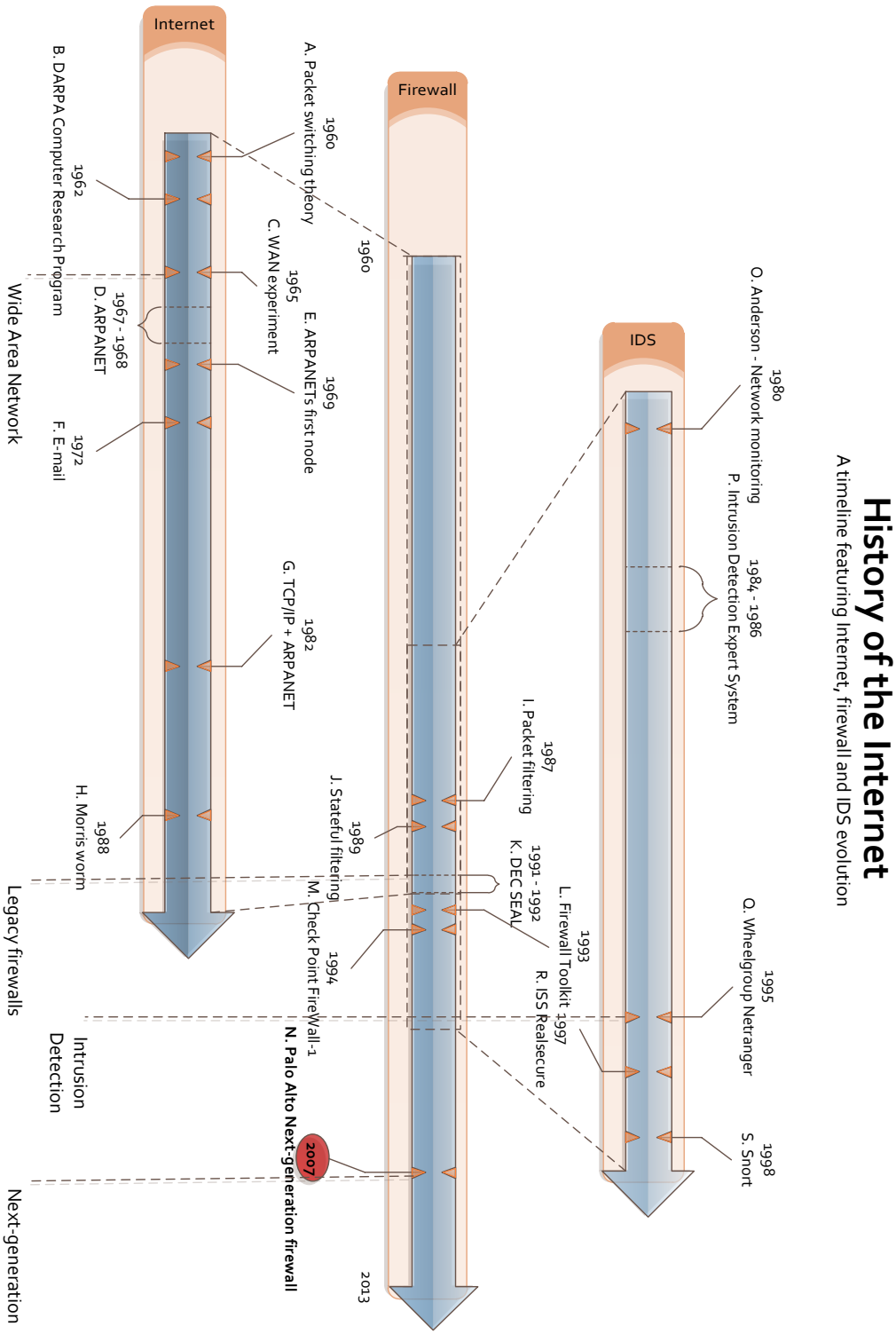

Figure 2.2: History timeline

2.9 Rules and rule sets

First and second generation firewalls operate with a focus on ports and addresses when filtering network traffic. In rules for these kinds of firewalls one may specify elements necessary for the filter to decide whether or not to forward the packet. Such elements are often referred to as *"tuples"*, and they specify a property that is used for packet filtering. A rule consisting of protocol, source and destination address and ports makes up a 5-tuple rule.

Figure 2.3: 5-tuple rule example

Generally all traditional firewall rules more or less follow a standard structure containing source and destination addresses/zones, port and protocol. Rules for next-generation firewalls might look exactly the same as traditional firewall rules, but a next-generation firewall rule can also include user and application tuples.

Figure 2.4: 7-tuple rule example

The "Application" and "User" tuples are not mandatory, they are considered meta-tuples, and they make the rule more flexible and immune to the increasingly common phenomenon of so called port hopping applications, described in Palo Alto's *"Application Usage and Risk Report"*[17].

A collection of rules are often called rule sets, and a firewall normally operate with one rule set at a time. It is very common for an organizations network to include several firewalls in order to separate zones and networks, hence the need for one rule set per firewall. A number of rule sets makes up a big part of the organizations security policy from the network security perspective. In this thesis one or more rule sets are referred to as a policy.

Gaining knowledge on how to create rule sets for next-generation firewalls can be a complex task, taken into account that no generic format exists for next-generation firewall rules. In the following chapters a possible solution to this is presented, in the form of a unified high-level language that can be used as a tool for making policies across next-generation firewall platforms.

2.10 Firewall policy languages

The first theory addressing universal firewall languages was proposed by Guttman[8] in 1997, where he describes a Lisp-like language used for specifying high-level packet filtering policies. Since then there have been several attempts to create unified languages for firewalls. The majority of these languages were designed for host-based firewalls, not for appliances, and today most of these languages are deprecated.

Although ten years old, one of the most recent projects on the matter is the HLFL(High Level Firewall Language) project[11]. The projects goal was to provide a language that could be used to write firewall rules that could be translated into usable rules for Linux host based firewalls and Cisco firewalls. The last version was released in 2003, and the project is no longer maintained.

Research done at DePaul University in 2007 resulted in a firewall policy language called FLIP[3]. FLIP is a block based language designed to be used on different configurations by abstracting itself from the network topology. As this research was performed at roughly the same time as the first next-generation firewall was released to the market, no support is provided for next-generation firwalls.

Since 2003 the commercial actor AlgoSec[1] has offered the management solutions FireFlow(AFF) and Firewall Analyser(AFA), powerful tools designed to manage firewall policies. The current versions of AFA and AFF support policy modification and deployment for a few next-generation firewalls. The AlgoSec tools are operated by a graphical interface and provides no unified language available to the user. Policy management is most likely done under the hood using vendor specific commands, hence the reason for limited for next-generation firewall support is most likely due to the fact that they a relatively new product group, and it will take time to develop full support for these products. One might try to modify or expand AlgoSecs tools to perform the same tasks addressed by this thesis, but understanding AFFs and AFAs inner workings will probably prove difficult, and modification is likely in discordance with AlgoSecs guidelines as they are commercial products. The problem statement states that the goal of this thesis is to promote an open, unified language, hence it cannot and should not be directly compared to the AlgoSec solutions.

Chapter 3

Approach

This chapter contains a development plan and describes a testing process for a prototype high-level language designed to write firewall policies for next-generation firewalls. The prototype will be used to build and implement several security policies using a real next-generation firewall appliance in a laboratory environment. If using the language for policy implementation proves to be successful, functionality tests will be performed to verify prototype functionality and to ensure that the outcome is in accordance with the objectives O_1 to O_3 from the problem statement in section 1.1.

The next-generation firewall used for laboratory testing in this thesis is a Palo Alto PA-200 appliance firewall. Steven Thomasons[28] study on next-generation firewall devices states that Palo Alto is one of the top players among next-generation firewalls, making the PA-200 a suitable candidate for the laboratory experiments in section 4.6. The firewall appliance will only be connected to a physical network through its management port for remote management. Generating network traffic to ensure enforcement of security policies is not a part of the experiments performed in this thesis, as it is expected that a successfully implemented rule set will filter traffic correctly.

All firewall products mentioned in this thesis are network based and runs a Linux based operating system. Palo Alto uses a vendor specific Linux distribution called PanOS, which incorporates command line tools, along with a web-interface and SSH support. Experiments will be performed on PanOS version 5.0.3, which is the latest available version at the time of the experiment phase. Suitable experiments will be designed to review the overall policy implementation process when using a universal language, and all these experiments will be performed on Linux based systems. Linux is also the preferred environment for software development in this project.

3.1 Policy definitions

The universal term *policy* is used vastly throughout this thesis, hence a clarification is provided to diminish any confusion. A policy describing the network security policy for an organization is referred to as policy_o. Policy_h is a policy derived from the complete or a subset of policy_o, and written by a person using the prototype language. Policy_c is compiled by the prototype tool and ready for deployment to a next-generation firewall. The policy implemented and operating in the firewall is called policy_i.

A quick reference to the policy types is provided in table 3.1.

Policy	Description
Policy_o	Network security policy for an organization.
Policy_h	Human readable policy in text form.
Policy_c	Compiled policy, ready for firewall deployment.
Policy_i	Policy implemented in firewall.

Table 3.1: Policy type definitions

A traditional firewall policy implementation process involves all of these policy types, from policy_o to policy_i, as shown in figure 3.1. Normally policy_c is produced using vendor software, and kept within the firewall itself. When using the prototype language, policy_c can also refer to a policy produced on a client computer and then transferred to the firewall.

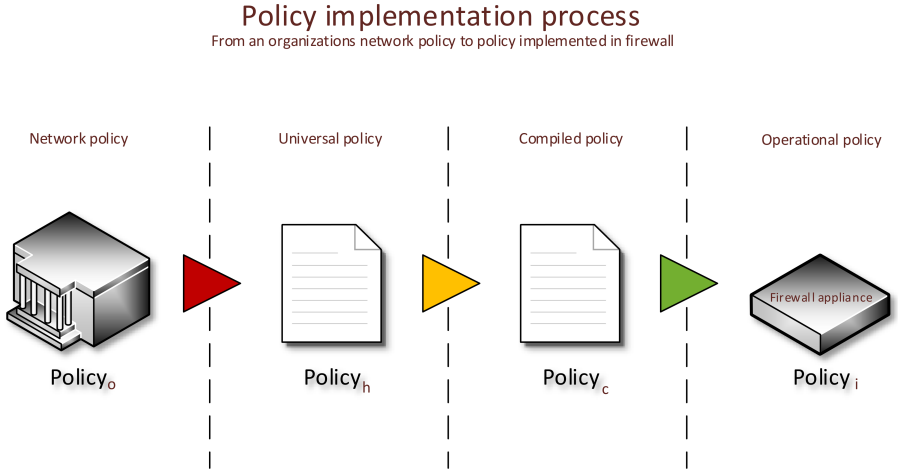

Figure 3.1: Basic implementation process

3.2 Phase 1 - Property identification

This phase reflects on the question Q_1 and the objective O_1 outlined in the introducing chapter, where we seek to investigate how a common platform can be made for next-generation firewall rule structure and policies. The process of forming a unified platform is initiated by uncovering what features and tuples are relevant for a next-generation firewall.

As mentioned in section 2.8, firewall rules are formed by tuples, or properties, that dictate how to handle network traffic based on traffic directions, ports, applications, users and addresses. A set of next-generation firewall products will be subject to analysis in order to map relevant properties, and to uncover any inconsistencies between them. Supported properties may vary from vendor to vendor, so it is essential to be certain that all possible properties are covered. Including all properties found in next-generation firewalls in the universal language will assure support for all configurations, and any excess or unsupported properties might be filtered out prior to the final policy creation.

The findings in this phase will form a basis for the next phase, where the goal is to design a prototype of a universal language.

3.3 Phase 2 - Design

Once a common standard has been established in *Phase 1*, the design of a prototype language can commence. Mainly there are two code concepts applicable for this task; tuple-based and block-based. They both have their pros and cons, and there are a number of aspects that needs to be taken into consideration when designing a language for the purpose we are seeking. To ensure that solid groundwork for future development is laid, the two concepts must be thoroughly examined before the final design is chosen.

Phase 2 seeks to answer and fulfill Q_2 and O_2 from the introduction. This phase will describe a way of creating a functional solution for policy management in next-generation firewalls, namely by creating a form of expressing firewall policies with the help of an universal language to achieve simplified policy management across next-generation firewall products.

3.4 Phase 3 - Language development

Phase 3 combines the results of *Phase 1* and *Phase 2*, hence the outcome of these two initial phases are decisive for the code syntax and language composition that is to be developed in this phase. The goal of this phase is a prototype of an easily readable language that simplifies and unites next-generation firewall rule sets. As well as *Phase 3*, this phase is also related to Q_2 and O_2 from the introduction chapter.

3.5 Phase 4 - Software design

Software tools are needed to convert policy$_h$, written in the universal language, to policy$_c$ in the respective firewalls native language. This software has to consist of a parser and a compiler. Input files containing universal policies needs to be parsed and compiled into policy$_c$ files that can be pushed directly to the firewall. For programming the parser and the compiler, Perl is the programming language of choice, as this is the language the author is most familiar with, and it should prove more than capable of dealing with such a task.

For future-proofing the compiler software needs to be expandable, in the sense that it should support add-on capabilities for future firewall languages. This functionality can be implemented by introducing module support in the compiler. Each module will compile the universal policy$_h$ into policy$_c$ in the desired vendor format, providing multi-vendor support. Introduction of new properties will be solved by utilizing a plugin architecture. Plugins can be customized to handle properties based on user needs.

Phase 4 reflects in Q_2 and O_2 defined in the introduction chapter.

3.6 Phase 5 - Implementation

This phase represents objective O_3, the evaulation of the prototype language and the software that has been developed in the provious phases. *Phase 5* will also try to give an answer the important question Q_3:
What can be achieved by creating and utilizing a universal language?

When a policy$_c$ is produced using the *Phase 4* software tools, it can be transferred to a next-generation firewall and set into production as a policy$_i$. Each vendor have their own set of control commands and interfaces that are used to perform configurations and policy deployment. Most major next-generation firewall vendors provide SSH and web interface access to their products, so an investigation will be done to decide which method proves to be the most suitable for policy transfer to the firewall. A detailed implementation process is illustrated in figure 3.2.

Figure 3.2: Detailed implementation process

3.7 Phase 6 - Experiments

Evaluation of the prototype language will be carried out by staging one artificial control scenario and one real-world like scenario, where a policy$_h$ is created from the ground up and implemented into a Palo Alto PA-200 firewall. Time restrictions and resource availability limits the possibility to perform experiments on more than one platform, although theoretically the principles conceived in this thesis should be adaptable to any next-generation firewall hardware.

To ensure a realistic outcome, a second test scenario will be designed to explore how the universal language and its complementing software perform under real-world conditions. Given the time frame for this thesis, building a policy$_h$ and deploying it as policy$_i$ in a production environment is risky due to the probability that the immatureness of the prototype language could cause unforeseen security breaches, not to mention that a policy$_o$ designed for an organization most likely would have to be kept confidential for security reasons. Because of this, the experiments will be carried out in an laboratory environment isolated from other network traffic.

The results of the experiments will hopefully unveil how the prototype language performs compared to using the native language of the PA-200 next-generation firewall appliance. It is of importance that any significant deviations and differences are uncovered, therefore a representative number of properties will be measured:

P$_X$: Complexity
Complexity is measured by number of tuples and the number of lines needed for the policies.

P$_C$: Compliance
Does the policy$_i$ implemented using the prototype tool *comply* with the desired policy design?

P$_D$: Baseline differences
Are there any *differences* in the policies when using the prototype tool compared to using the vendor tool?

P$_T$: Time
Are there any differences in *time* consumption when using the prototype tool compared to using the vendor tool?

This final phase of the project is a part of the evaluation process and should provide answers to Q_3 and fulfill O_3 described in chapter 1.

3.7.1 Experiment 1

Step A: Implementation

The first experiment will be implementing a basic policy_i from a policy_h, solely to uncover any problems or bugs in the language itself, or in the transition process from high-level policy_o through compiled policy_c to fully implemented policy_i. The policy_h will be a simple policy for one client network behind the firewall, which opens up for network traffic typical to a private home network. The prototype language will be used when writing policy_h, and the developed software will be used to convert policy_h to policy_c before it is finally compiled to policy_i by the firewalls internal software.

The implementation process should provide an answer to if the prototype software can be considered to be a functional solution to firewall policy building and management, as described in Q_2 and O_2.

Step B: Quality control

Quality control is the next natural step if the policy_o proves to be successfully implemented in *Step A* using the universal language. This means ensuring that every element in the policy_h is implemented in the next-generation firewall without errors, and that it operates as expected. First the policy will be implemented by using the PanOS web-interface, then by using the prototype language. The desired result would be if the PanOS policy_i and the policy_i generated by the prototype language proves to be identical.

The results of this step should provide some answers to how the prototype performs compared to the vendor tool, as asked in question Q_3 and described in O_3.

Step C: Timing

While performing Step A, a stop watch will be used for timing the policy creation process, both when using the web-interface when using the prototype. This step will require at minimum two test users, one novice user and one expert user, to investigate if there might be variations in the learning curve between the vendor tool and the prototype.

This last step in the experimentation process is a method for evaluation of the prototype language, answering question Q_3, as well as partially fulfilling objective O_3, the evaluation of language qualitites.

3.7.2 Experiment 2

To provide realistic testing conditions a policy_o for use at a hospital is chosen as the scenario for the second experiment. A full security policy of this proportion is usually too comprehensive for one firewall alone, so the policy will be compacted for use in a single firewall. Hopefully this will have little or no impact on the realism of the scenario and the value of the results. Hospitals are something most people can relate to, hence the threshold for understanding the policy_o should be relatively low.

Step A: Implementation

The policy_o for this experiment is heavily influenced by a production policy_i used in a real hospital. This experiment is to be considered as the proof-of-concept for the prototype language. This step will be the process of going from policy_o to policy_i, and the purpose of the following *Step B* and *Step C* is to review this process in depth.

Step B: Quality control

Quality control will be executed by first implementing policy_i through the PanOS web-interface, followed by the implementation of the policy_i using the prototype language. If the desired result is achieved the PanOS policy_i and the policy_i generated by the prototype language should be identical.

Step C: Timing

While performing Step A, a stop watch will be used for timing the implementation processes, both when using the web-interface when using the prototype. This step will require at minimum two test users, one novice user and one expert user, to investigate if there might be variations in the learning curve between the vendor tool and the prototype.

3.8 Approach summary

The problem statement for this thesis states three questions, and three corresponding objectives designed to provide answers to these questions. In short, these questions ask how a unified platform for next-generation firewalls can be used to form a language designed to write security policies, what can be achieved by such a universal language and how the language can be evaulated. The quest to fulfill these objectives is divided into 6 phases, all necessary to reach the goal, which is a working prototype of a universal language for next-generation firewall policies. All phases described in this chapter, along with the correspond elements from the problem statement are represented in table 3.2.

Question	Objective	Phase	Experiment step
$\mathbf{Q}_1$:How can a common platform be made?	$\mathbf{O}_1$: Property collecting.	**1**: Property identification	-
$\mathbf{Q}_2$: How can a functional solution be found?	$\mathbf{O}_2$: Develop language and tools.	**2**: Design **3**: Develop language **3**: Develop software	**A**: Implementation
$\mathbf{Q}_3$: What is acheived? How to evaluate?	$\mathbf{O}_3$: Evaluation process.	**5**: Implementation **6**: Experimentation	**B**: Quality control **C**: Timing

Table 3.2: Approach and problem statement affiliations

Chapter 4

Results

The findings and results from *Phases 1-6* in chapter 3 are presented here, starting with the first phase, which aims to build a unified platform for next-generation firewalls in order to be able to define features and properties for a universal language.

4.1 Phase 1 - Property identification

In order to gain a greater understanding of how next-generation firewalls operate and what are their similarities, a comparative study including six next-generation firewall products from several major vendors was carried out. The next-generation firewalls subject to comparison are shown in table 4.1.

Vendor	**Model**
Palo Alto	PA-200
Dell	Sonicwall E8500
Adyton	Network Protector
Sourcefire	NGFW
Barracuda	X600
Check Point	R75

Table 4.1: Comparison of NGFW products

The comparison of the six candidates[13][16][26][27][15] makes it clear that many of them use terminology that is somewhat similar, while a couple of the firewalls stand out by using unique properties, see comparison table 4.2. These unique properties are product specific capabilities that cannot easily be applied to all next-generation firewalls. One example is the Palo Alto's "HIP-profile" property, short for "Host Information Profile", that compiles information about client devices, a feature exclusive to Palo Alto.

Property	Adyton	Barracuda	Check Point	Dell	Palo Alto	Sourcefire
Action	∨	∨	∨	∨	∨	∨
Application	∨		∨		∨	∨
Comment		∨	∨	∨	∨	
Destination	∨	∨	∨	∨	∨	
Destination zone					∨	
Enabled	∨	∨	∨	∨	∨	∨
HIP-profile					∨	
Install on			∨			
Interface		∨				
Log			∨		∨	
Name	∨	∨	∨	∨	∨	∨
Number	∨		∨	∨		
Object		∨		∨		
Options	∨					
Profile				∨	∨	
Service		∨	∨	∨	∨	
Source		∨	∨	∨	∨	
Source zone					∨	
Type				∨	∨	
User		∨	∨	∨	∨	

Table 4.2: Property overview

The products used for the property comparison is a small selection of the products available on the market, meaning the differences are probably even bigger than this study suggests. As a result of these inequalities a major drawback is that writing a policy that is supposedly unified and applicable to all next-generation firewalls, will be lacking product specific capabilities that might make a product stand out from the rest. On the other hand, the flexibility of the prototype language actually makes it possible to define these properties *if* they are supported by the firewall. When developing a policy_h for a firewall with unique properties, these properties can be included in the policy by simply defining them as they are defined in the firewalls native language. If compiled correctly, these unique properties will be filtered out and removed if not supported by the firewall product. See listing 4.5 in section 4.3 for instructions on how to include product specific properties in rules.

If disregarding product specific properties, it is possible to gather the equalities in the next-generation firewalls to form a semi-common platform that covers the most important next-generation firewall capabilites, as shown in table 4.3.

Property	Value
Action	Allow/Deny
Application	Application object
Comment	Text
Destination	Address(es)
Enabled	Yes/No
From	Zone
Log	Yes/No
Name	Rule name
Number	Rule number
Service	Service object
Source	Address(es)
To	Zone
User	User object

Table 4.3: Unified property platform

Most of the products mentioned in this study should be able to utilize the properties in table 4.3, with the exception of the Sourcefire NGFW, which is clearly a stand-alone IPS, labeled as a next-generation firewall, but designed to operate in conjunction with a firewall.

Traffic filtering by user is possible for some next-generation firewalls, but none of these seem to have their own user database. User data is usually fetched from RADIUS, Active Directory or LDAP. This does not eliminate the universal language's ability to create custom users, but in order to benefit from this, an authentication service interface must be implemented. Such an interface is not covered in this thesis, but it might be an idea for a future extension. As a result of this, the user property is not applicable for any of the rules in the experiments.

4.2 Phase 2 - Design

Using the unified property platform created in *Phase 1* as a basis, a prototype of a universal language is developed. This section deals with design aspects of the language, and how a design conclusion was drawn after weighing several alternatives.

Blocks of code can be used to represent the structure of an organization and its network topology in a way that is fairly easily comprehensible to humans. Network components can be defined using blocks that hold information on the components, somewhat similar to a object based like language. Using blocks also opens up for using classes, overriding, loops, inheritance and variables with scope. One drawback that comes with a block based language is the level of difficulty when it comes to parsing. All blocks and their contained data have to be interpreted and parsed correctly to avoid flaws in the policy_c that can lead to corrupted policies and subsequent malfunctioning firewalls.

Figure 4.1 show how a firewall rule can be written in block format.

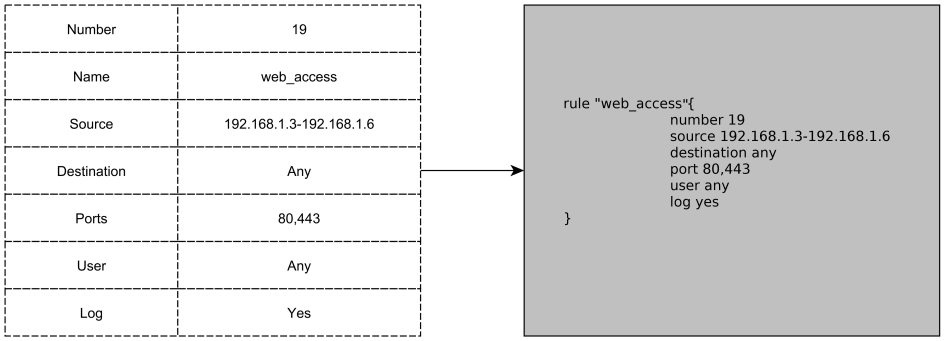

Figure 4.1: Block based code example

Sooner or later in the policy implementation process, the compiled version of the policy$_c$ will be converted into a tuple structure to make it understandable to the firewall before pushing it to an operational policy$_i$. This transition will in many cases happen inside the firewall itself, and will not be visible to the firewall administrator. An alternative to the block based approach is using tuples as a basis for a universal language, as tuples have some advantages over blocks. Tuples makes it easier to compile and implement policies, since the parser and compiler only need to do a mapping operation from policy$_h$ to policy$_c$. In fact, most modern firewall interface software display their rule sets using a tuple format.

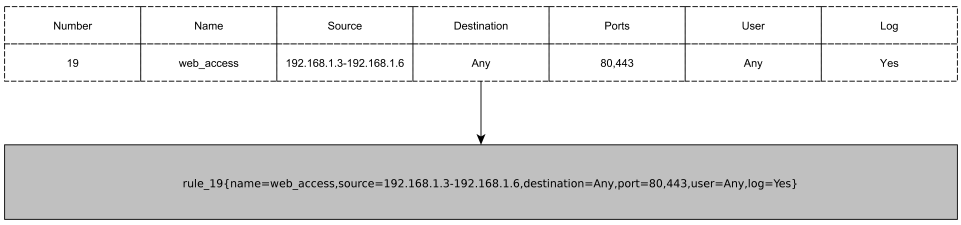

Figure 4.2: Tuple based code example

For small rule sets tuples can possess a high grade of readability for humans, but complexity is prone to increase with the number of lines in the set. One of the great disadvantages that comes with a tuple based language is the low level of expandability. With blocks, policies can be modified and expanded by seemingly doing less editing, since the structure allows for compressing the code used for writing policies. While if modifying a tuple based policy, all affected lines needs to be edited.

Due to the flexibility of a block based approach and for the sake of readability in large policies, the choice fell on using code blocks as the preferred design for the universal language.

4.3 Phase 3 - Language development

This phase presents the result of the development process, a working prototype language for next-generation firewall policies, and it gives the reader an introduction on how to use it to build objects and rules. The development of the prototype was done using the Perl programming language, and blocks and code syntax should be easliy graspable by most individuals possessing basic programming skills in any object oriented programming language.

The prototype has support for inheritance and variables. If a firewall rule is incomplete and a superclass is defined, default values will be placed in an attempt to preserve the integrity of the rule set by using inheritance. Skipping properties in rules can also be done deliberately utilizing inheritance to minimize the number of tuples in the policy.

4.3.1 Language structure and syntax

Variables are supported and prefixed with notation $, meaning defining a variable is done as follows:

```
$variable = value
```

Blocks are defined with curly brackets, "{" for begin block, "}" for end block. A block is defined by class type and must be given a name.

```
classtype name {
     property value
    }
```

Using SSH as an example,the following listing shows how a service object is made:

Listing 4.1: Service declaration example

```
service ssh {
            port 22
            protocol tcp
          }
```

As presented in the example above, the property represents an attribute that is specific to the class type, for services ports and protocols are significant properties, while when dealing with applications other properties might be used.

Listing 4.2: Application declaration example

```
application facebook {
            signature "www.facebook.com"
           }
```

The *signature* property can be used as an inspection string or to define a regular expression that trigger intrusion detection alerts based on signatures if the firewall supports deep packet inspection.

Inheritance makes it possible to define superclasses that hold general properties which can be applied to classes where these properties are not defined. A superclass block looks very similar to a normal class block, by simply adding "superclass" in front of the classtype it can be made a superclass.

```
superclass classtype name {
     property value
    }
```

In order for a class to inherit from a superclass, the superclass must be defined as a superclass within the class block.

```
classtype name {
     property value
     superclass superclassname
    }
```

For an application the superclass and class definitions might look like this:

Listing 4.3: Superclass declaration example

```
superclass application application_common {
            protocol tcp
            port 80,443
           }

application facebook {
            signature "www.facebook.com"
            superclass application_common
           }
```

Now the properties *protocol* and *port* will be added to the application *facebook*.

Rules share the exact same syntax as classes, although the rule name must be quoted to specify that the name is one single string. This is a deliberate choice of design as rule names often are text strings containing whitespaces, which might otherwise confuse the parser.

```
rule "name goes here" {
     property value
    }
```

Below is an example on how to allow access for the application *facebook* from the client network in zone *internal* to anyone in the zone *external*.

Listing 4.4: Rule declaration example

```
rule "facebook access" {
              application facebook
              from internal
              to external
              source client-network
              destination any
              action allow
             }
```

Product specific properties, meaning they are not supported by all firewalls, can be defined by simply defining them as they appear in the products native language. Using Palo Alto's "HIP-profiles" as an example, this is how it is done:

Listing 4.5: Unique property declaration example

```
superclass rule rule_common {
              hip-profiles any
             }
```

It is crucial that the property exists and appear exactly as it does in the native language of the firewall, or else it will be filtered out in the compilation process and lost.

Ranges are defined with "-", for example IP or port ranges are denoted "$beginrange-$endrange". Netmask are prefixed with the CIDR notation[30] "/", followed by the number of bits assigned to the network address. To assign multiple values to a property, the values are separated with a comma.

Listing 4.6: Range and multiple values example

```
zone internal {
              address 192.168.0.1-192.168.0.255
             }

network office {
               netmask /24
               }

service rdp {
            port 5800,5801,5900,5901
            }
```

This is a common way to specify ranges, netmasks and multiple values in firewalls. If another format is required by the firewall, re-formatting can be done in each individual compiler to accomodate other notations.

4.4 Phase 4 - Software design

To cover the entire implementation process from $policy_h$ written in the prototype language to the compiled $policy_c$, the developed software consists of a collection of elements designed with specific purposes in mind. Please refer to figure 3.2 for an overview of these elements.

The $policy_h$ can be written in any standard text editor, and it is separated into two files, one configuration file and one file containing the rule set. The configuration file defines components like services, applications, zones, networks and other objects relevant to the policy, while the rule file exclusively specifies a set of rules.

Listing 4.7: Configuration file example

```
$eth0 = 10.0.0.1
$eth1 = 192.168.1.1

superclass service service_common {
       protocol tcp
       }

superclass zone zone_common {
        netmask /24
        }

service web {
            port 80,443
            superclass service_common
            }

service ssh {
            port 22
            superclass service_common
            }

zone internal {
        iprange 192.168.1.2-192.168.1.255
        superclass zone_common
        interface $eth1
        }
```

Listing 4.8: Rule file example

```
rule "web browsing" {
        service web
        from internal
        source any
        to any
        destination any
        action allow
        log on
        }

rule "ssh access" {
        from any
        source any
        to internal
        destination any
        service ssh
        action allow
        log on
        }
```

If the firewall is already configured, only the rule file will need to be created, and configuration objects can be inserted into the rule file by using their respective names.

Utilizing superclasses for rules makes it possible to significantly decrease the amount of necessary properties in the $policy_h$. Using a basic superclass, as seen in listing 4.9, the number of properties can be reduced.

Listing 4.9: Rule superclass example

```
superclass rule rule_common {
                            action allow
                            from any
                            to any
                            source any
                            destination any
                            log on
                            }
```

This superclass allows the rule set to be written as simple as in listing 4.10.

Listing 4.10: Rule inheritance example

```
rule "web browsing" {
        service web
        from internal
        }

rule "ssh access" {
        to internal
        service ssh
        }
```

Once a configuration file and a rule file has been created, the main program can be executed, assuming a suitable compiler exists. Necessary input files are specified using command line operators, as shown in figure 4.3.

```
prototype.pl -p rulefile -c configfile -o compiler
```

Figure 4.3: Program execution

Currently supported command line options are described in table 4.4 below.

Operator	**Function**
[-p policy]	File containing rules
[-c configuration]	File containing configuration
[-o output]	Compiler
-v(verbose)	Verbose output
-h(help)	Usage description

Table 4.4: Command line options

Rule and configuration files are generated depending on how the compiler is programmed to handle the data from the main program. In the case of the Palo Alto compiler, several files are created as necessary by the internal file structure in the firewall appliance. The PA-200's complete configuration is stored in one XML-file, which contains all rules, objects and configurations, except for application and user information. The content of the XML-file includes parts that are unrelevant for this project, and some important content like application data is located within a large library file, which can not be exported or imported in any convenient way. This means that when working with the PA-200, applications are pre-defined and including applications in a rule written in the prototype language can be done by simply adding the applications name in the rule.

Listing 4.11: Use of remote desktop application in rule

```
rule "remote desktop" {
                      application ms-rdp
                      }
```

This of course requires first hand knowledge on available applications before writing the policy.

Defining users can be done by creating user objects, but a user database is usually located externally, and most next-generation firewalls are meant to connect to authentication services like LDAP, RADIUS and Active Directory to pull user data. To specify users simply add the desired user name prefixed by the property *source-user*.

Listing 4.12: External user definition

```
rule "remote desktop" {
                      application ms-rdp
                      source-user john-doe
                      }
```

Many of the next-generation firewalls are designed with external user databases in mind, hence they do not have the ability to store users within themselves. It is nevertheless possible to define user objects manually if this functionality is supported by the firewall:

Listing 4.13: Manual user definition

```
user john-doe {
                      firstname John
                      lastname Doe
                      email john-doe@example.com
                      }
```

4.4.1 PA-200 implementation process

After a policy_h has been compiled and ready to deploy, there are various ways of preparing the policy_c for implementation, differing from vendor to vendor. This section descibes the procedure used for Palo Alto PA-200.

The PA-200 configuration file is split into several parts, divided by XML tags as seen in listing 4.14.

Listing 4.14: PA-200 Configuration file example

```
<zone>
                    <entry name="Internal">
                      <network>
                        <layer3>
                            <member>ethernet1/2</member>
                        </layer3>
                      </network>
                    </entry>
</zone>
<service>
                    <entry name="http-proxy">
                      <protocol>
                            <tcp>
                              <port>8080</port>
                            </tcp>
                      </protocol>
                    </entry>
                  </service>
</service>
```

The elements relevant when dealing with firewall rules are <*zone*>, <*service*>,<*rulebase*> and <*address*>(IP-ranges or single IP-addresses) and <*address-group*>. Other elements hold various configuration options that has little or no relevance to the rule set. With such a elaborate configuration file, the file must be dissected and the output from the compiler has to be merged into the correct location in the file. This is a manual process that is performed before uploading the file to the firewall.

4.4.2 Data structure

The main program module initially does a listing of available plugins and compilers and includes their code to utilize these external files during parsing and compilation. Each data field in the configuration file and the rule file is read and parsed by the corresponding plugin, which builds up multi-level hashes containing rules and configuration. A hash is stored in whatever format the plugin dictates. The plugins provided with the prototype typically store the data structure similar to figure 4.4.

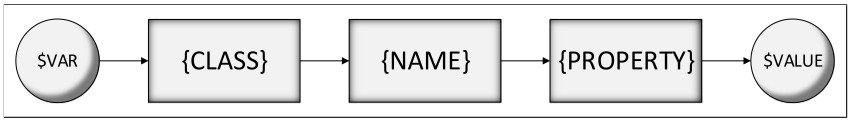

Figure 4.4: Generic hash structure

The service class is used as an example to show what a hash structure looks like in figure 4.5.

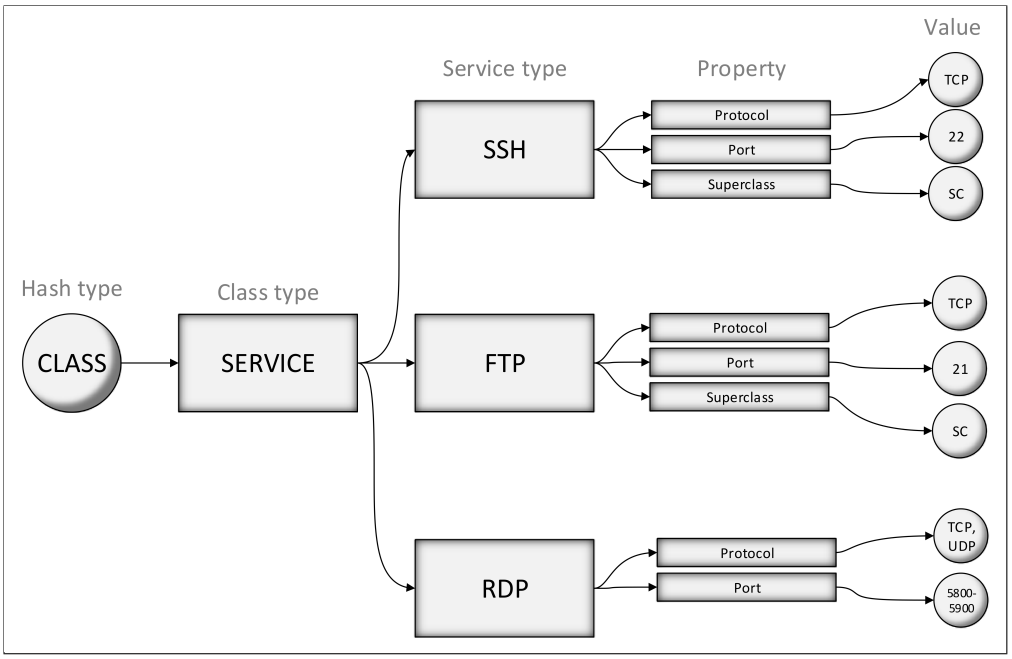

Figure 4.5: Service class hash structure

When all data is parsed and the hashes have been built, comes the process of population configuration and rules with properties from their respective superclasses. When this inheritance process is complete the compilation process can commence, resulting in a compiled policy$_c$, in the form of one or several files.

The flowchart in figure 4.6 illustrates the five stages of data parsing, inheritance and file creation through the policy building process.

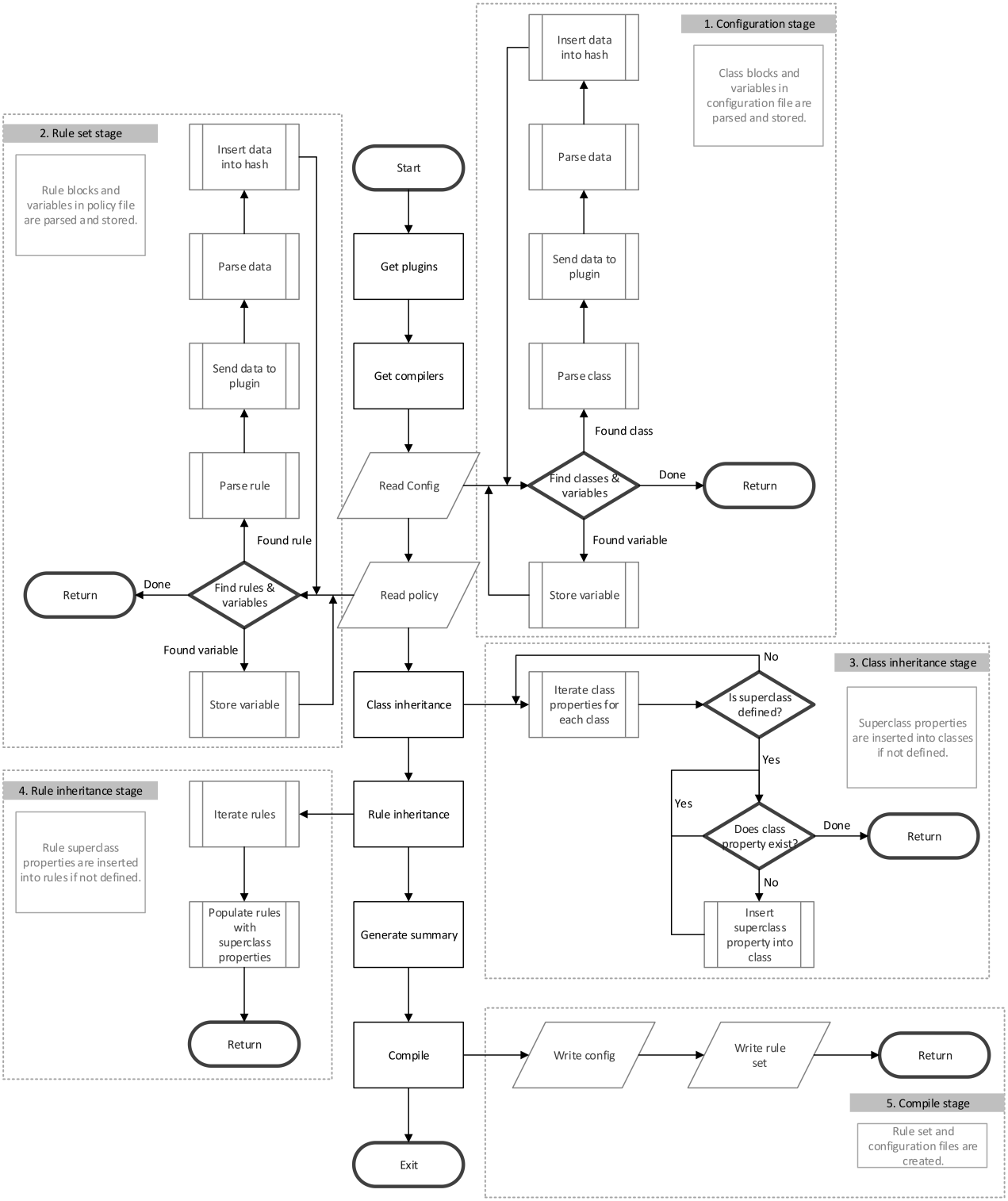

Figure 4.6: Software flowchart

A overview of currently inluded classes with their underlying properties is provided in table 4.5.

Classes		
Service	Port	Integer
	Protocol	String
	Superclass	Class hash
Application	Port	Integer
	Protocol	String
	Superclass	Class hash
	Signature	String
Group	Member	Network,Zone
Network	IP	Integer(x.x.x.x)
	Netmask	Integer(/x)
Zone	Interface	String
Rule	Interface	String
	User	(Unsupported. See section 6.4)
	Action	String
	Source	Network,IP
	Destination	Network,IP
	From	Zone
	To	Zone
	Log	Boolean
	Service	Service hash
	Application	Application hash

Table 4.5: Class overview

4.4.3 Plugins

Classes may need to be parsed in different ways, and the plugins are essential components made to parse data in a way that provides the desired output format. If the main program module finds a class block during execution, the type and name of the block is passed to the plugin corresponding to that specific type of block. The plugin will then iterate the data fields contained within the block and store them in the correct hashes before returning to the main program. If a block that contains a class type with no corresponding plugin, the class will be considered unsupported. The block will be skipped and the main program will continue to the next block. Plugins included in the prototype software correspond to the classes in the class overview table 4.5.

The plugin architecture opens up for tremendous expandability. The plugins dictate how data is parsed and stored, which gives them control of data processing before compile time. This ensures support for future features in forthcoming firewall products.

4.4.4 Generic compiler

Compilers are called from the compile sub routine in the main program, and their purpose is to output data in a format readable to the desired firewall. Data stored in hashes during the parsing process is now processed by the compiler, and the data can be manipulated and filtered to accomodate current needs. In order to build a complete rule set, a compiler must contain the following sub routines, including the passed arguments:

- beginRuleset()
- beginRule($name)
- compile($property,$value)
- endRule()
- endRuleset()

The generic process of building a rule set begins with beginRuleset(), a sub routine that defines a starting property for $policy_c$ if necessary. The beginRule() routine receives the rule name from the main program and begins the rule, before the compile() routine inserts all defined properties and their values. endRule() ends the rule if required by $policy_c$, and endRuleset() finalizes the entire rule set before returning to the main program.

4.4.5 The PA-200 compiler

Currently one compiler is provided with the prototype, namely the compiler for the Palo Alto PA-200, used for the experiments in section 4.6. As described in section 4.4, the Palo Alto configuration file $policy_c$, is split into a number of sections. This requires the rules to be processed separately from the configuration elements like zones, networks, services etc. As a result of this the PA-200 compiler generates the configuration sections before processing the rule set. In its current state, the PA-200 compiler creates 5 files, 4 configuration files and one rule file.

After generating all five files the compiler routine returns and the main program exits. Now the files must be merged manually with a Palo Alto configuration file in order to create a fully functional $policy_c$ file, which can now be uploaded and set into production as described in section 4.5.

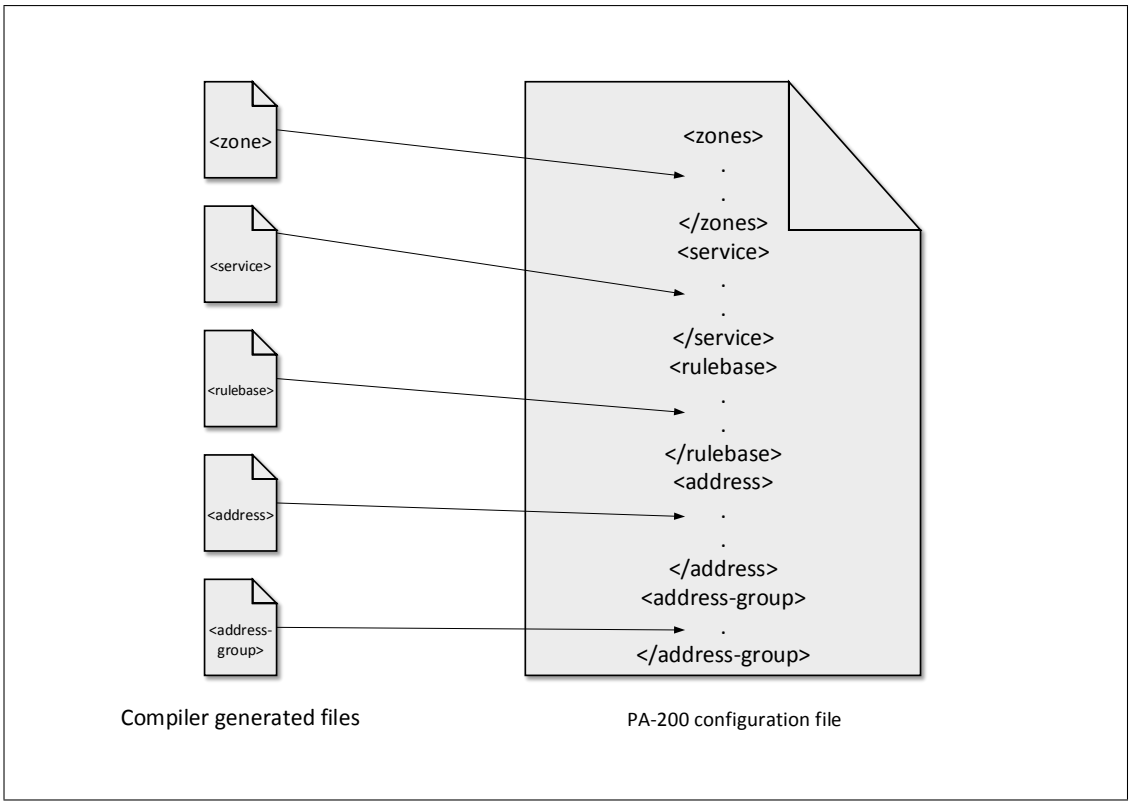

Figure 4.7: File merging

4.5 Phase 5 - Implementation

Transferring $policy_c$ can be done using SCP file transfer or through the PA-200's web interface. During the experiments only the web interface has been used for uploading files to the firewall, as in this case this proved to be the most streamlined method for file uploading to the firewall. Once $policy_c$ is uploaded it must be loaded, and changes have to be commited to achieve $policy_i$ status. Committing a rule set can take several minutes depeding on the size of the set, as this is in fact the process where $policy_i$ is created and implemented.

4.6 Phase 6 - Experiments

The Palo Alto PA-200 has four physical network interfaces. During the experiments each of these interfaces will be used as a connection to a zone. A way of expressing $policy_o$ is by using tables containing a short write-up of traffic flow that can easily be translated into rules, see table 4.9 for an example.

Both Experiment 1 and Experiment 2 initially require a copy of the firewalls running configuration to be downloaded. This configuration is used as a template and merged with $policy_c$ created by the prototype

software. After implementing the configuration and rule set into their correct sections in the original downloaded configuration file, the file can be uploaded back to the PA-200.

4.6.1 Experiment 1

The purpose of the first experiment is to test the basic integrity of the prototype language and the software components. A successful outcome will be the transition from policy_o to policy_h. The policy_h should result in a policy_c after being processed by the software. The operational policy_i should prove consistent with the corresponding policy_o, policy_h and policy_c.

The first experiment involves two zones, an external and an internal zone. Only one client network is located in the internal zone, and the external zone can be considered an Internet connection. The rule set opens up for web surfing, Bittorrent and remote desktop traffic. See table 4.6 for an overview of objects created for use in this experiment.

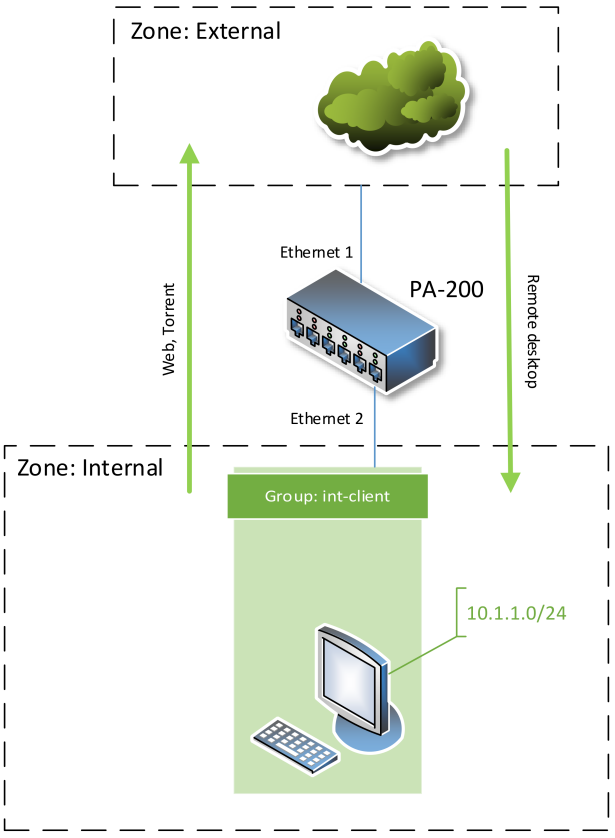

Figure 4.8: Network setup used in Experiment 1.

Objects	*Name*
Zones	Internal,External
Services	rdp
Applications	bittorrent,web-browsing
Networks	int-client

Table 4.6: Objects used in Experiment 1.

Step A: Implementation

Using the prototype software, policy_c was created without errors, uploaded to the firewall through its web interface and finally commited into policy_i. After achieving policy_i status, the first attempt proved unsuccessful. At first glance everything seemed to be in order, but after a closer look it was obvious that the order of the rules in the now operational policy_i was incorrect. This poses a problem, since the order of the rules havily impacts how the firewall operates. This could potentially lead to security breaches or too strict policies. This bug turned out to be a result of how hashes are stored i Perl, and the lack of ability to sort multi-layer hashes by values instead of key, which seems to be the default way[2]. A solution was found by automatically adding a number to each rule which made it possible to correctly arrange the order of which the rules are written to the policy_c output file.

After implementing this fix, the transition from policy_c to policy_i was successful, resulting in a perfect transition from policy_o to policy_h to policy_c, and further testing could commence in *Step B*.

Step B: Quality control($\mathbf{P}_C$ and $\mathbf{P}_D$)

There is no noticable difference when implementing policy_i using the prototype software from when using Palo Alto's web interface. Comparing the policy_c created by the compiler to the relevant sections in the file downloaded from PA-200 reveal that they are identical.

This result means a fully compliant(P_C) policy_i has successfully been implemented into a product policy in the PA-200 firewall, giving a compliance success rate of P_C=100%, and a baseline difference rate of P_D=0% after fixing the software bug described in *Step A*.

Step C: Timing($\mathbf{P}_T$)

Timing the entire process from policy_h to policy_c was done with the help of two firewall administrators, one of which had great knowledge of the prototype language as well as PanOS, while the other administrator was unexperienced, but had been given a quick theoretical introduction to both approaches. Using a video camera the policy_h writing processes was documented for further analysis.

This step in the experimentation process is divided into two sections, object creation and rule creation, object creation being the initial step. Objects are considered as class elements, for example services, zones, groups, networks etc., and these objects must exist before they can be used to build rules. Object creation using the PA-200 web-interface is comparable to creating a configuration file using the prototype language, while rule creation represents the creation of the rule set file.

The resulting time consumption differences in rule and object creation for Experiment 1 are shown in tables 4.7 and 4.8. These tables represent the combined creation time of each object type. For example the creation time for two zones are considered one time variable. Total policy$_h$ creation times can be found by adding the individual object and rule creation times.

Objects	*Prototype*	*PanOS*
Rules	148.8	229.9
Networks	29.6	33
Services	27.7	17.7
Zones	61	62.8
Total	267.1	343.4

Table 4.7: Object and rule creation times in seconds for novice user.

Objects	*Prototype*	*PanOS*
Rules	104.3	198.8
Networks	16.9	28.8
Services	17.8	15.1
Zones	36.2	44.1
Total	175.2	286.8

Table 4.8: Object and rule creation times in seconds for expert user.

Tuple comparison($\mathbf{P}_X$)

Tuple count per rule in PanOS' web interface is $\text{Tuples}_{PanOS} = 11$, and rule line count is $\text{Lines}_{PanOS} = 4$, resulting in a total tuple count Total_{PanOS}:

$$\boxed{Total_{PanOS} = Tuples_{PanOS} \cdot Lines_{PanOS} = 11 \cdot 4 = 44}$$

When using the prototype language, the number of tuples located in the rule set is $\text{Tuples}_{ruleset} = 18$.

4.6.2 Experiment 2

The second experiments is a compacted and simplified version of a real hospital network security policy. This experiment makes use of all network interfaces by using four zones, as seen in figure 4.9. Table 4.9 shows $policy_o$ in table form, infrastructural network objects are denoted *INTERFACE:ZONE:NETWORK*.

From	*To*	*Application*	*Service*	*Action*
Any	1:DMZ:email-segment	smtp	-	Allow
0:External	1:DMZ,2:Internal,3:Secure	Any	Any	Deny
1:DMZ:email-segment	0:External	smtp	-	Allow
1:DMZ:web-proxy	0:External	-	http,https	Allow
1:DMZ	0:External	Any	Any	Deny
1:DMZ:email-segment	2:Internal:backoffice	smtp	-	Allow
1:DMZ	0:External,1:Internal,3:Secure	Any	Any	Deny
2:Internal:clients	1:DMZ:web-proxy	-	http-proxy	Allow
2:Internal:clients	2:Internal:ts	ms-rdp	-	Allow
2:Internal:clients	2:Internal:backoffice	hp-jetdirect,ms-rdp,sap,ssl	-	Allow
2:Internal:ts	3:Secure:sec	scada,ftp	-	Allow
2:Internal	0:External,1:DMZ,2:Internal,3:Secure	-	-	Deny
3:Secure	0:External,1:DMZ,2:Internal	Any	Any	Deny

Table 4.9: Network policy for Experiment 2 in table form.

The table form of $policy_o$ is a very similar to how the implemented $policy_i$ appears in a firewalls graphical user interface, as displayed in figure 4.10.

Network objects created to perform Experiment 2 are shown in table 4.10.

Objects	*Names*
Zones	Internal,External,DMZ,Secure
Services	http,https,http-proxy
Applications	smtp,ms-rdp,hp-jetdirect,sap,ssl,scada,ftp
Networks	dmz-email-segment, dmz-web-proxy,int-backoffice, int-client-1,int-client-2,int-client-3, int-ts,sec-journal,sec-medequip-xray,sec-medequip-mr
Groups	dmz-email,int-client,sec

Table 4.10: Objects used in Experiment 2.

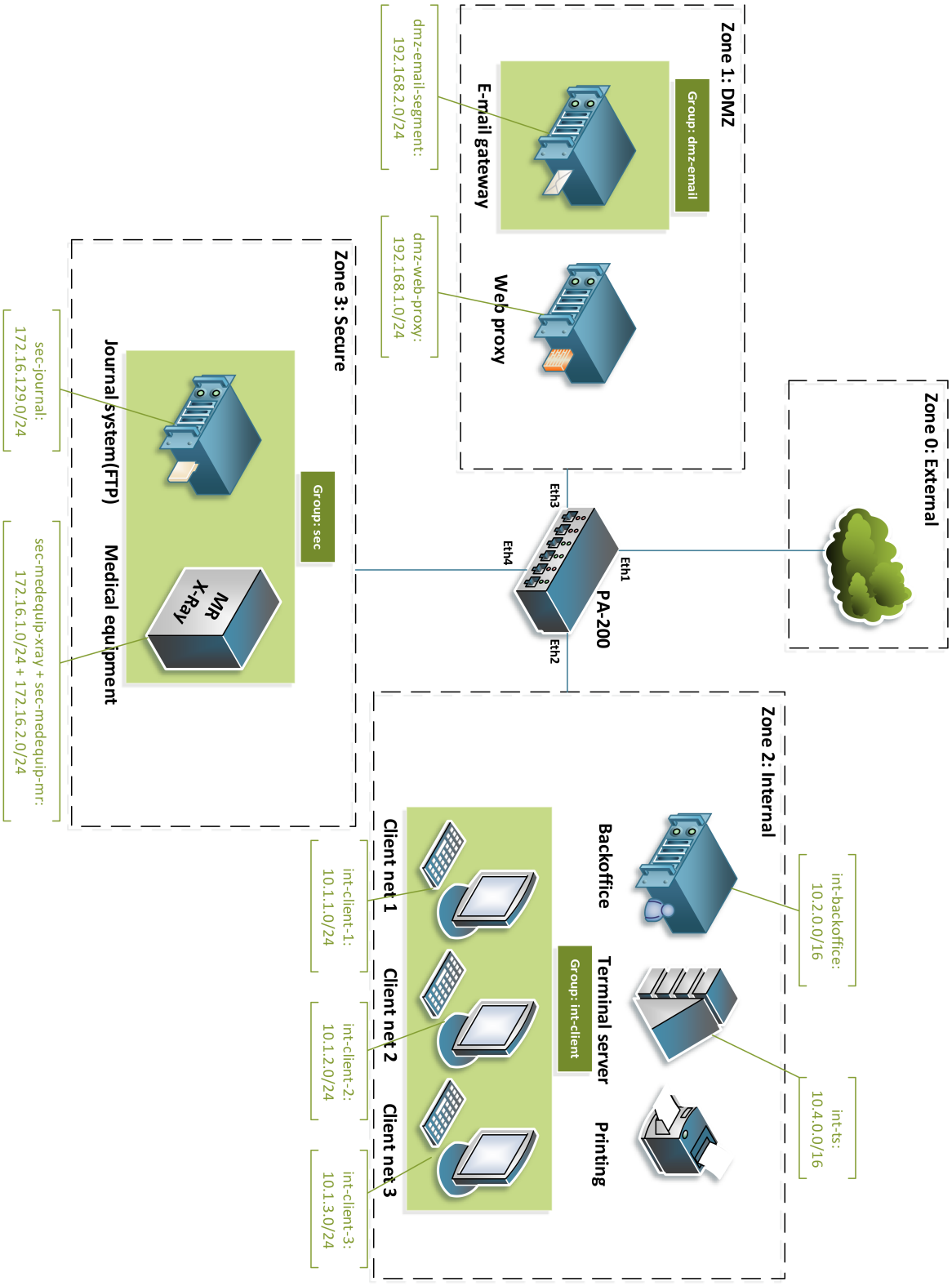

Figure 4.9: Network setup used in Experiment 2.

Step A: Implementation

No problems were encountered when making $policy_h$ and compiling it to $policy_c$, but when trying to load the $policy_c$ in order to create $policy_i$, PanOS displayed error messages stating that the application "sap" is dependant on the application "ssl". The missing element, application "ssl", was included as an allowed application in the correct rule in $policy_h$, before recompiling, uploading and loading it again. This time without encountering any errors. This comes to show that awareness of service and application dependencies is important when using the prototype language. Currently there are no control mechanisms in the prototype to ensure dependencies are met. The representation of the implemented $policy_i$ in the PanOS web interface is shown in figure 4.10.

		Source				Destination						
Name	Tag	Zone	Address	User	HIP Profile	Zone	Address	Application	Service	Action	Profile	Options
0-1 Email-receive	none	any	any	any	any	DMZ	dmz-email-seg...	smtp	any	✔	none	
0-123 deny	none	External	any	any	any	DMZ Internal Secure	any	any	any	⊘	none	
1-0 Email-send	none	DMZ	dmz-email-segment	any	any	External	any	smtp	any	✔	none	
1-0 Web-access	none	DMZ	dmz-web-proxy	any	any	External	any	any	service-http service-https	✔	none	
1-0 deny	none	DMZ	any	any	any	External	any	any	any	⊘	none	
1-2 Email-backoffice	none	DMZ	dmz-email-segment	any	any	Internal	int-backoffice	smtp	any	✔	none	
1-023 deny	none	DMZ	any	any	any	External Internal Secure	any	any	any	⊘	none	
2-1 client-webproxy	none	Internal	int-client	any	any	DMZ	dmz-web-proxy	any	http-proxy	✔	none	
2-2 client-ts	none	Internal	int-client	any	any	Internal	int-ts	ms-rdp	any	✔	none	
2-2 client-backoffice	none	Internal	int-client	any	any	Internal	int-backoffice	hp-jetdirect ms-rdp sap	any	✔	none	
2-3 ts-secure	none	Internal	int-ts	any	any	Secure	sec	cygnet-scada ftp socks2http	any	✔	none	
2-013 deny	none	Internal	any	any	any	DMZ External Internal Secure	any	any	any	⊘	none	
3-012 deny	none	Secure	any	any	any	DMZ External Internal	any	any	any	⊘	none	

Figure 4.10: Web interface view of implemented policy

Step B: Quality control(P_C and P_D)

After adding the necessary *"ssl"* element in the previous section Step A, the transition from $policy_c$ to $policy_i$ went flawlessly. There is no noticable difference when implementing $policy_i$ using the prototype software from when using Palo Alto's web-interface. Comparing the $policy_c$ created by the compiler to the relevant sections in the file downloaded from PA-200 reveales that they are identical. This gives a compliance success rate of P_C = 100% and no differences P_D = 0% after sorting out the dependency issue descibed in section 4.9.

Step C: Timing(P_T)

The timing experiment was performed by the same two administrators using the same procedure as in 4.6.1, and with a video camera the $policy_h$ writing process was documented for further analysis. The resulting time consumption differences(P_T) for Experiment 2 are shown in tables 4.11 and 4.12.

Objects	*Prototype*	*PanOS*
Rules	537.2	699.3
Groups	92.7	163.7
Networks	190	291
Services	27.7	16.5
Zones	131	88.3
Total	978.6	1258.8

Table 4.11: Object and rule creation times in seconds for novice user.

Objects	*Prototype*	*PanOS*
Rules	473.5	631
Groups	95.3	122.3
Networks	188.2	241.1
Services	14.8	15.7
Zones	97.5	65.3
Total	869.3	1075.4

Table 4.12: Object and rule creation times in seconds for expert user.

Tuple comparison(P_X)

Tuple count per rule in PanOS' web interface is $Tuples_{PanOS}$ = 11, and the line count is $Lines_{PanOS}$ = 13, resulting in a total tuple count $Total_{PanOS}$:

$$\boxed{Total_{PanOS} = Tuples_{PanOS} \cdot Lines_{PanOS} = 11 \cdot 13 = 143}$$

Using the prototype language, the number of tuples located in the rule set is $Tuples_{ruleset}$ = 64.

Chapter 5

Analysis

Analysing the prototype of the universal language and its accompanying software components can be done in a number of ways, and thorough analysis of all related aspects is an immense task. Given the scope and time fram of this thesis only a small, but vital, selection of measurements have been performed in chapter 4, and this chapter aims to analyse the results of these measurements.

Comparing next-generation firewall products in chapter 4 showed a discrepancy in supported properties and features among next-generation firewall products. As legacy firewall are more or less equal in this area, it came somewhat unexpectedly that the differences were so significant. This comes to show the technologies behind this new wave of network security products differs from vendor to vendor, probably among products originating from the same vendor as well. Despite of these obvious differences, an attempt to create a common platform for the universal language was carried out. By abstracting from the physical products and focusing on an organizational view of computer networks, a common platform was created, although product specific features, as described in 4.1, makes it hard to ascertain such a fully unified platform. These unique features are not disregarded, but rather manually applicable when using the universal language to build policies, refer to section 4.4 for further explanation.

Complexity of a universal language compared to native languages can be measured in innumerable ways. In this thesis the method for measuring complexity is by counting the amount of tuples, or properties, used to form a rule set. Tuples are what define firewall rules, for example *source*, *destination*, *port*, *application* and so forth. Refer to section 2.9 for more details on this subject. To ensure full compliance with a security policy_o, all steps through the policy stages policy_h, policy_c and policy_i have been controlled to make sure no elements are dropped or corrupted along the way. A successful transition from policy_o to policy_i using the prototype language was the aim for full compliance to be achieved. The vendor tool, PanOS, has not been measured for compliance, as it is anticipated that PanOS should perform perfectly in this area.

Uncovering any differences between policy building when using the prototype and vendor tools is crucial to ensure reliability and robustness. Baseline differences are investigated using an external software tool designed to detect inequalities in files. The time frame leaves no room to go in depth on user testing, but one property that can fairly easily be measured is the time it takes to build policies_h, when using the prototype and when using PanOS.

The following sections provide analysis of complexity, compliance, baseline differences and timing, based on the findings in chapter 4.

5.1 P_X: Complexity

When comparing the policy_h files written in the experiments in section 4.6, there are some significant differences that needs to be taken into account. The aspect of complexity in this thesis can be difficult to measure, but in this case the number of tuples and lines are two attributes that can be used for measurement.

The number of tuples counted in Experiment 1 show that $\text{Tuples}_{ruleset}$ is considerably less than for Tuples_{PanOS}.

$$\boxed{Experiment_1 : Tuples_{ruleset} < Tuples_{PanOS} = 18 < 44}$$

The same reduction in tuples apply for Experiment 2, although to a somewhat lesser extent.

$$\boxed{Experiment_2 : Tuples_{ruleset} < Tuples_{PanOS} = 64 < 143}$$

The differences are illustrated in figure 5.1.

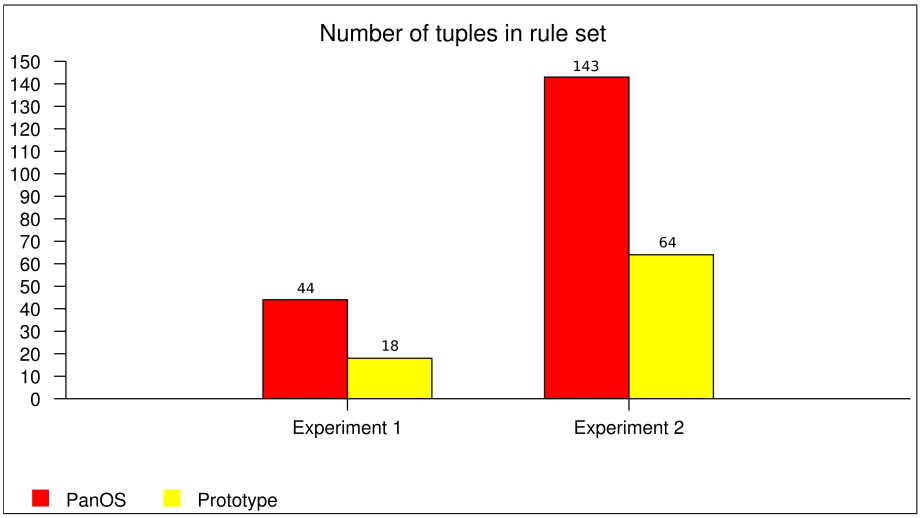

Figure 5.1: Tuple count: PanOS versus Prototype

The decrease in tuple count when using the prototype language can be credited to the prototypes inheritance capabilites. For Experiment 1 a reduction in tuples by approximately 59% was achieved.

$$Experiment_1 : 1 - \frac{Tuples_{ruleset}}{Tuples_{PanOS}} = 1 - \frac{18}{44} = 0.591 \approx 59\%$$

Experiment 2 shows an approximate reduction by 55%.

$$Experiment_2 : 1 - \frac{Tuples_{ruleset}}{Tuples_{PanOS}} = 1 - \frac{64}{143} = 0.552 \approx 55\%$$

While the number of tuples are decreased, the number of lines are prone to increase due to use of a block structure syntax. Using line count can be considered an unfair mean of measurement, since one block represents one line of tuples in a rule set, it is possible to flatten a block and achieve exactly the same amount of lines as when using a tuple based language, hence the amount of lines as a measure of complexity was disregarded.

5.2 P_C: Compliance

A vital step in the quality assurance process is checking compliance to uncover any inconsistencies in the entire policy creation process. The property P_C is used as a measure of compliance, to determine if the final implemented $policy_i$ is in accordance with $policy_o$. In both Experiment 1 and Experiment 2 the transition from network $policy_o$ to the implemented $policy_i$ proved to be successful when using the prototype software for $policy_h$ and $policy_c$ creation.

5.3 P_D: Baseline differences

As well as compliance, baseline difference control has to be performed to uncover any weaknesses in the transition process from compiled to implemented policy. When using the prototype software, it is not possible to spot any differences in the compiled $policy_c$ and the implemented $policy_i$ from the policies created using the vendor tool. By using a file content comparison tool like *diff* [29] this is proven to be correct.

5.4 P_T: Time

The entire policy creation process for both test users is analysed from video, making it possible to accurately record creation times of the individual objects in $policy_h$, as well as the time taken to build the complete $policy_h$.

Time experiments show that using the prototype is somewhat quicker for policy creation compared to PanOS. The PanOS web interface has a

periodically sluggish behaviour that can potentially slow policy and object creation times down more than necessary. Although the prototype generally performs better in terms of speed, PanOS seems to be able to create some objects faster than the prototype. Figure 5.2,5.3,5.4 and 5.5 provide bar charts to illustrate object and rule creation times all experiments.

When it comes to the creation of a service in Experiment 1, PanOS seems to be slightly faster, while all other objects and the rules are created quicker when using the prototype.

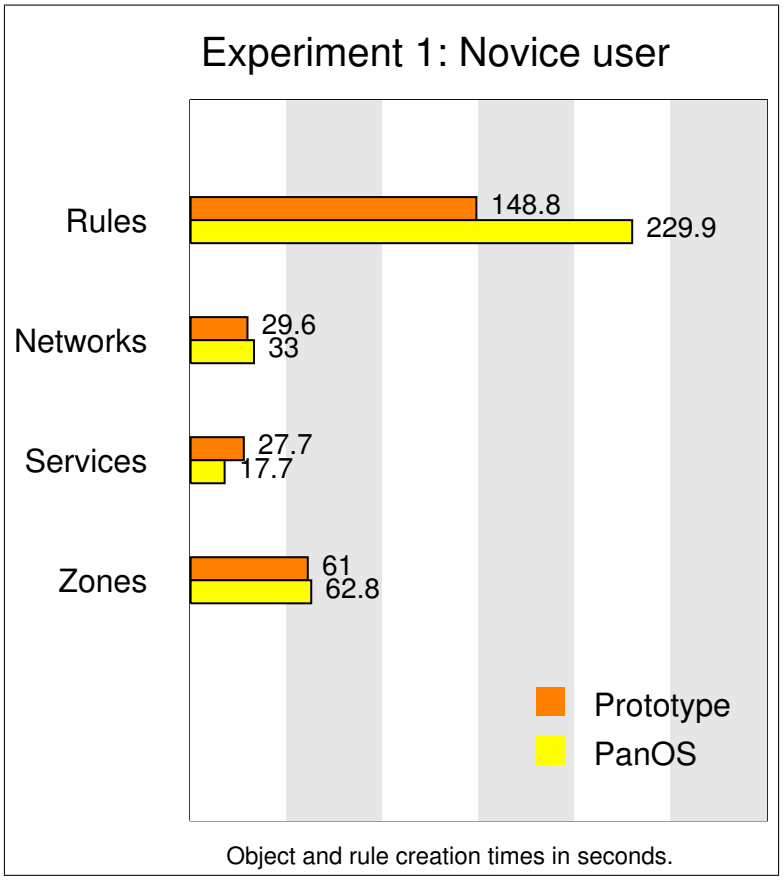

Figure 5.2: Experiment 1: Time measurements for novice user.

The expert users times from Experiment 1 are illustrated in figure 5.3. This shows the same tendencies as experienced with the novice user, service creation time is lower when using PanOS, while all other objects and the rules can be created faster with the prototype.

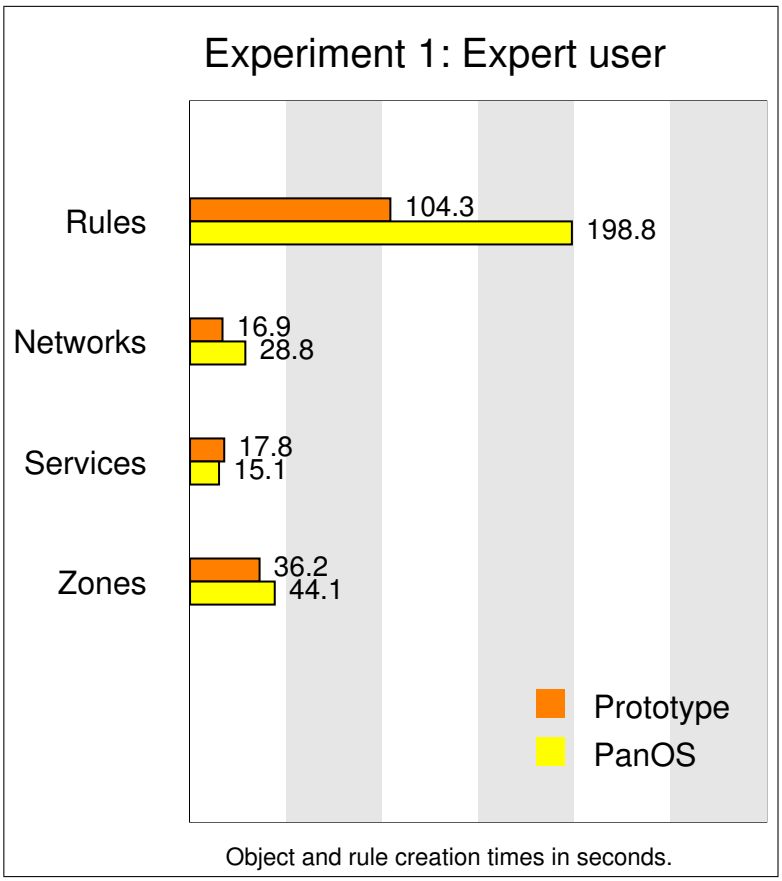

Figure 5.3: Experiment 1: Time measurements for expert user.

The novice users times from Experiment 2 are illustrated in figure 5.4. With the more advanced policy in Experiment 2 it looks like both service and zone creation times are at their lowest when using PanOS.

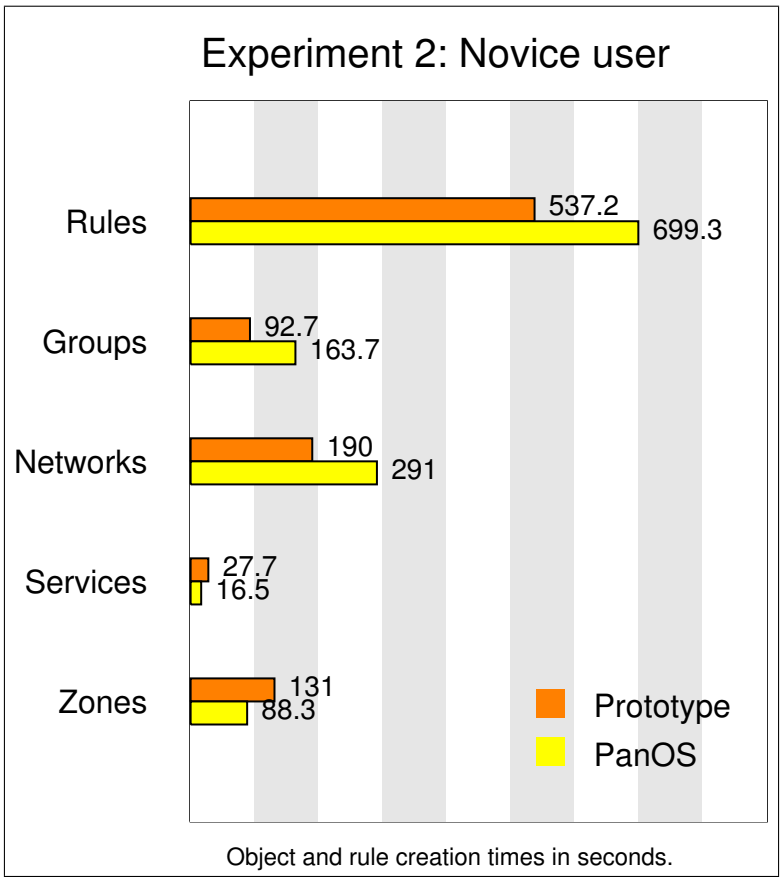

Figure 5.4: Experiment 2: Time measurements for novice user.

The expert users times from Experiment 2 are illustrated in figure 5.5, and they indicate the some of the same trend as seen in the novice user experiment in figure 5.4; zone creation takes longer when using the prototype. Service creation times are very close, and it is hard to draw any conclusion from this, since there is only one single service involved in the experiment. To properly measure service creation times, more elaborate testing must be done.

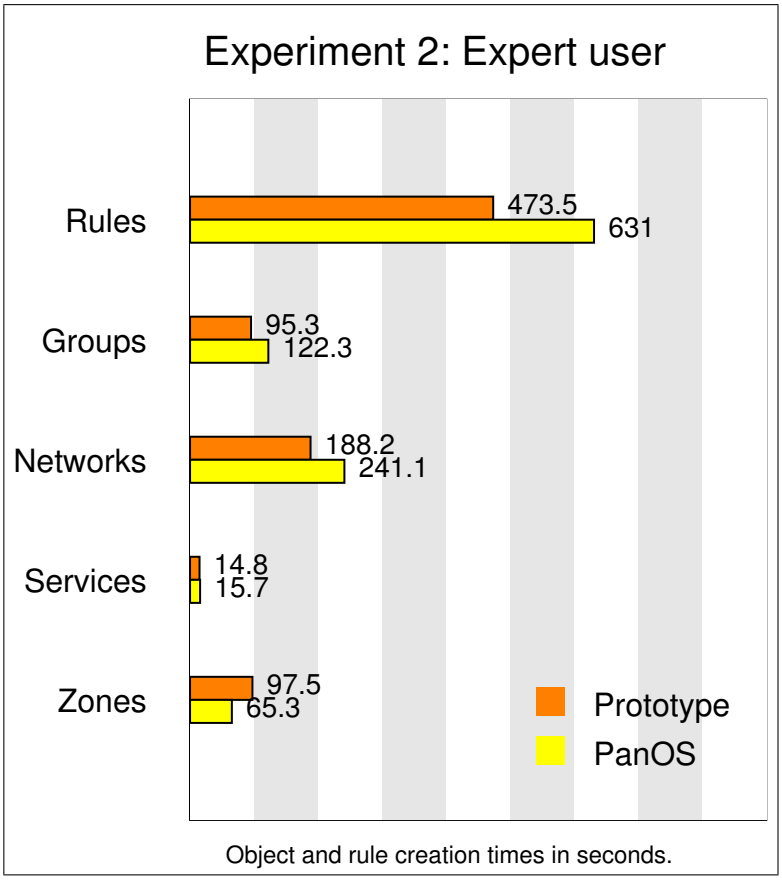

Figure 5.5: Experiment 2: Time measurements for expert user.

Looking at rule creation, one can clearly see the decrease in time consumption when using the prototype. The simple rule set from Experiment 1 experiences a reduction in creation time between approximately $\frac{1}{3}$ and $\frac{1}{2}$, depending on the user.

$$Experiment_1\ Novice\ user: 1 - \frac{RuleCreationTime_{Prototype}}{RuleCreationTime_{PanOS}} = 1 - \frac{148.8}{229.9} = 0.353 \approx 35\%$$

$$Experiment_1\ Expert\ user: 1 - \frac{RuleCreationTime_{Prototype}}{RuleCreationTime_{PanOS}} = 1 - \frac{104.3}{198.8} = 0.475 \approx 48\%$$

It appears that when creating a more advanced policy, like the policy from Experiment 2, a reduction the time difference between using the prototype and PanOS is experienced. The reduction is user dependant here as well,

but in this case the gap between user are not as significant as in Experiment 1. A reduction in rule set creation time of about $\frac{1}{4}$ is achieved.

$$Experiment_2\ Novice\ user : 1 - \frac{RuleCreationTime_{Prototype}}{RuleCreationTime_{PanOS}} = 1 - \frac{537.2}{699.3} = 0.232 \approx 23\%$$

$$Experiment_2\ Expert\ user : 1 - \frac{RuleCreationTime_{Prototype}}{RuleCreationTime_{PanOS}} = 1 - \frac{473.5}{631} = 0.25 \approx 25\%$$

The amount of time data is far too lacking in order to draw a solid conclusion from the times recorded in these experiments, but there are indications that the prototype is decreasingly faster proportionally with the size of the rule set.

By adding object and rule set creation times the result is the time used for the creation of a complete policy. Total policy creation times for both users are shown in table 5.1 and figure 5.6.

Experiment	*Prototype*	*PanOS*
1: Novice user	978.6	1258.8
1: Expert user	869.3	1075.4
2: Novice user	267.1	343.4
2: Expert user	175.2	286.8

Table 5.1: Total policy creation times.

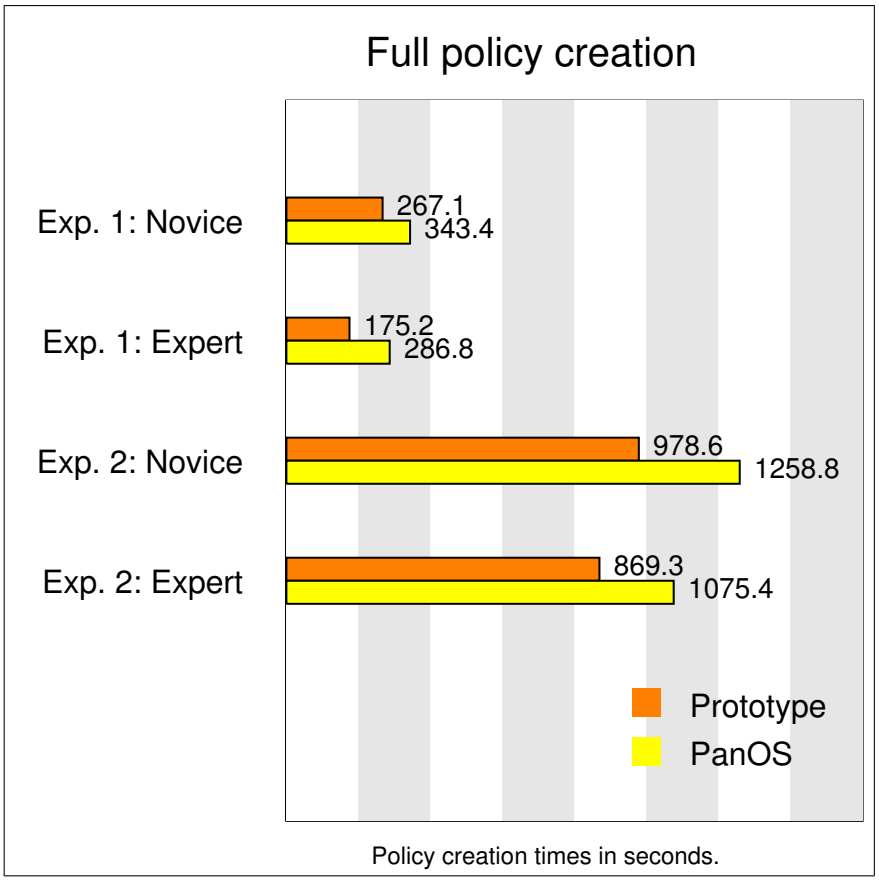

Figure 5.6: Total policy creation times.

Judging from the total time it takes to create complete policies$_h$ in Experiment 1, a reduction in creation time when using the prototype ca be seen, 22% for the novice user and 39% for the expert user.

$$Experiment_1\ Novice\ user : 1 - \frac{PolicyCreationTime_{Prototype}}{PolicyCreationTime_{PanOS}} = 1 - \frac{267.1}{343.4} = 0.222 \approx 22\%$$

$$Experiment_1\ Expert\ user : 1 - \frac{PolicyCreationTime_{Prototype}}{PolicyCreationTime_{PanOS}} = 1 - \frac{175.2}{286.8} = 0.39 \approx 39\%$$

Total policy creation times in Experiment 2 show similar numbers for the novice user, but the decrease in creation time for the expert user is 19%, under half of the reduction seen in Experiment 1.

$$Experiment_2\ Novice\ user : 1 - \frac{PolicyCreationTime_{Prototype}}{PolicyCreationTime_{PanOS}} = 1 - \frac{978.6}{1258.8} = 0.222 \approx 22\%$$

$$Experiment_2\ Expert\ user : 1 - \frac{PolicyCreationTime_{Prototype}}{PolicyCreationTime_{PanOS}} = 1 - \frac{869.3}{1075.4} = 0.192 \approx 19\%$$

These results indicate that when increasing the size of the rule set, the gap in policy building time between the prototype and PanOS is reduced, independent of user experience level.

Calculating the difference in time consumption in Experiment 1 and Experiment 2 for each test user results in a ratio that is very similar for the novice user.

$$PanOS,\ Novice\ user: \frac{PolicyCreationTime_{Experiment1}}{PolicyCreationTime_{Experiment2}} = \frac{343.4}{1258.8} \approx 0.27$$

$$Prototype,\ Novice\ user: \frac{PolicyCreationTime_{Experiment1}}{PolicyCreationTime_{Experiment2}} = \frac{267.1}{978.6} \approx 0.27$$

While for the expert user writing the bigger policy from Experiment 2, this ratio is lower, which means the difference in time can increase as the user gains experience in using the prototype language. This is a logical outcome, as the users are likely to improve efficiency the more they use the prototype, only limited by typing speed, whereas using the PanOS web interface can be a restricting factor due to some periodical lag.

$$PanOS,\ Expert\ user: \frac{PolicyCreationTime_{Experiment1}}{PolicyCreationTime_{Experiment2}} = \frac{286.8}{1075.4} \approx 0.26$$

$$Prototype,\ Expert\ user: \frac{PolicyCreationTime_{Experiment1}}{PolicyCreationTime_{Experiment2}} = \frac{175.2}{869.3} \approx 0.2$$

This might indicate that the learning curve is approximately equally steep for the prototype and PanOS, but the time spent on writing policies can be decreased more when using the prototype, supporting the outcome of previous findings in this chapter.

Chapter 6

Discussion

This chapter will try to answer Q_1, Q_2 and Q_3 from the problem statement in section 1.1 and seen when describing the approach methodology in chapter 3. It will also review if the objectives O_1,O_2 and O_3 have been fulfilled, and new possibilites for future development and obstacles encountered along the project period are also presented.

6.1 The search for an unified platform

Question Q_1 in the problem statement asks how a common platform be made for next-generation firewall rule structure and policies. A logical step in the process of trying to find an answer to this question is by examining a selection of next-generation firewalls, as done in section 4.1, to uncover similarities and differences between these firewalls. This comparative study concludes that a unified platform for next-generation firewalls can be difficult to make, as they differ from traditional firewalls in terms of features, functionality and filtering properties. For most traditional firewalls, rules are based on the properties:

- Source
- Destination
- Service:
  - Port
  - Protocol
- Log
- Name
- Number
- Action

This represents a very common recipe for creating firewall rules for the majority of traditional firewalls. For next-generation firewalls the feature sets differ to some extent, making it harder to find the same degree of similarity. A platform covering the shared properties in the next-generation firewalls compared in table 4.1 was found and is presented in table 4.3 in section 4.1. This fulfills objective O_1, examining how a common platform can be made. The level of differences in features and properties among next-generation firewalls was suprising, a more unified base platform as seen in traditional firewalls was expected.

The inequalities between next-generation firewalls probably comes as a result of the differences in firewall technology used by each vendor, technologies that have evolved over several year. The various paths the vendors have chosen weakens the hope of a unified platform anytime soon, but one possible solution would be if vendors could agree upon a common standard. This is however a highly unlikely scenario, as large sums has probably been spent developing products with unique capabilities in order to stand out from the competition. Because of this technology race, chances are security products, like next-generation firewalls, will distance themselves from each other even more in the time to come, and creating a unified platform might become increasingly difficult. There are numerous fields within computer technology that are continously evolving, and sometimes we just have to make the best of the technology currently available. This means the need or use for a universal language should not be eliminated beacuse of the lack of a common next-generation firewall platform.

Even though making a completly unified platform for next-generation firewalls proved to be difficult, the project commenced with the development of a universal language and software tool, and possible solutions to overcome the unforeseen events was investigated. A prototype was developed, and a solution to the differing tuple problem was found. Any unique properties that are not supported by all next-generation firewalls can still be defined when writing rule sets. These properties can be filtered out, by a compiler module if not supported by the current firewall. Of course these properties will thereby be lost, but since the firewall lacking in support for these firewalls anyway, the policy will still come out functional, although it may not operate quite as intended.

This ability to filter out non-supported properties also allow for legacy firewall support. User and application filtering properties in rules can be removed in a compiler module designed for a legacy firewall. Such capabilites greatly help the universality of the prototype language, although the goal of a completely universal platform could not be achieved at this point.

6.2 The prototype and its impact

The prototype software was coded entirely from scratch with the intention to save time by avoiding to learn external tools, and to remain in full control of the development process. With that in mind, the technical level of the project is not very advanced in terms of programming. For further development, the introduction of external programs could be valuable. This could save time in the development process by using already developed and tested code and programs, including more advanced parsers and compilers.

An expandable software tool was the desired result of the development process, in order to provide support for a diversity of future expansions and added features. This gave birth to the idea of a plugin architecture. There is virtually no limit to how a plugin architecture can expand functionality. Plugins are a good way of securing support for future features that may appear in future firewall products. Third parties can easily make their own plugins containing new features and properties without needing any insight to the main program's workings. The use of plugins also reduces the total size of the application, as only plugins necessary to each individual user need to be installed. In the current state of the prototype this may not be considered an issue, since the footprint of the entire software package is quite small, but for future expansion and for systems with low storage capacity, this is a valid point.

A plugin architecure can also be a way to circumvent software licencing issues. In the event of a project such as the universal language going commercial, third parties can develop functionality without having access the source code of the main program. Or seen from another angle; if the software source code remains open, vendors can develop closed plugins to protect their technology.

In order to support a multitude of firewalls, the possibility of outputting the policy in a desired format for a variety firewalls is vital. Vendors could produce their own compilers, developed for and shipped with their firewalls, making it possible for organizations to create policies before deciding on purchasing a specific firewall. Universal policies also makes it possible to exchange existing infrastructure without re-configuration and the need to build new rule sets is eliminated, potentially saving great amounts of time and money. The possibility for re-use of existing universal policy code, for either new firewall installations or firewalls that can share parts of or the entire code, is also a factor to take into consideration. One thing is certain, in order to make a universal language useable, more compilers need to be developed to add support for more firewalls.

The prototype language, along with its software tools, proved to be a success. It can be used to write policies that compiles to a format understandable to next-generation firewalls, and the software design makes the

concept future-proof. This leads to the fulfillment of objective O_2 from the problem statement, developing a future proof concept that simplifies the process of building security policies for next-generation firewalls. The simplicity of the language seems to make it comprehensible even to non-experienced users.

A universal language for firewalls will first of all influence the work of network administrators. It might help to boost their efficiency in an otherwise stressful environment. In the test results, discussed in the next section, a reduction in time spent on constructing rule sets when using the prototype is seen. This can be a major selling point to the professional market. Policy building and deployment can be done faster, reducing labour time and thereby saving money. This should make the language attractive to any organization currently using, or planning to purchase, firewall hardware that can be supported by the language.

6.3 Testing and results

Measuring various aspects of the prototype can be done in numerous ways, hence only a selection of key experiments feasable within the project time frame were conducted. The experimentation process includes a series of tests that have a high degree of realism to them, and they are designed to give answer to what is achieved with a universal language, and how can it be evaluated against traditional vendor tools, like asked in question Q_3. The prototype must be considered a proof-of-concept, hence in order to achieve release candidate status, more extensive testing should be performed. All experiments can be reproduced by replicating the $policy_h$ files in appendices section, where the necessary program code can also be found. A Palo Alto PA-200 or a device using the same version of PanOS will also be required.

Objective O_3 states that realistic experiments should be performed to measure language qualities on its own, and in comparison to vendor tools. After the initial implementations in *Step A*, complexity(P_X) measurements were carried out. Covering all aspects of complexity in a block based textual language is a tremendous task. For this reason measurements of complexity that give results that can be easily measured and interpreted was chosen. By counting properties in rule sets one can clearly spot differences between the prototype and the vendor tool, PanOS. One might argue what is the best way to present large rule sets to an administrator, but generally most people would probably agree that to the human eye less information is easier to process. This means blocks containing stripped down rules from the use of superclass inheritance can be considered less complex. Of course this would require full overview of the properties contained within the superclasses used. On the other hand, this reduction in complexity can lead to higher errors rates in policies as a result of lack of superclass insight by the administrator.

Lowering tuple count also seem to be an efficient method for reducing policy creation time, escpecially for low complexity rule sets, as seen in *Experiment 1*, section 4.6.1. Measurements of time(P_T) gave a slightly different outcome from what one might expect. Suprisingly, creating more advanced rule sets resulted in smaller time differences between the prototype and PanOS. This reduction in policy creation time might indicate that PanOS can in fact be more efficient when dealing with even big rule sets, but this can not be verified unless more extensive testing is conducted. To provide reliable results and to increase the scientific value in these experiments, the experiments should be repeated a number of times, using more testing personnel as well as a larger amount of more diverse policies.

Measuring compliance(P_C) may seem like a trivial experiment, but measuring compliance is an essential part of the testing process. Measuring comliance by matching prototype generated and vendor generated policies, shows the prototype is capable of generating policies that are fully compliant with a PanOS generated policy. This is a result of carefully studying original policy files generated by PanOS, to ensure correct output by the prototype compiler. The successful transition from $policy_o$ to $policy_i$ using the prototype show that a universal language is a functional solution for policy management in next-generation firewalls, providing an answer to question Q_2 in the problem statement in chapter 1.

6.4 Impediments and shortcomings

In retrospect, prototype development went fairly uncomplicated, no major obstacles that had any significant deal of impact on the process was encountered. Still, there has been some unforeseen circumstances that made the creation of a rule language more difficult than anticipated.

First off, the Palo Alto PA-200 stores its application database in files that are inaccesible to those using a standard administrator account. This could be a measure to avoid tampering with the database which could compromise security, or a result of the application database being very large, and combining it with the rest of the configuration files would result in a disorganized file structure. More likely it is due to the fact that this database needs regular updates, and keeping the database separately avoids frequent, time consuming changes to the rest of the configuration. Unfortunately this makes it impossible to define your own applications through the prototype language when using Palo Alto products, and one would have to know exactly the names of the applications that are to be implemented when building a rule set.

An issue similar to the lack of manual application definition, is how the user database is used by the firewall to filter traffic based on users.

It seems normal operation is to pull user account information from an authentication service, and it is not possible to create users stored locally within the firewall. This means when adding users to rules they must be defined by their username. Currently there is no way of implementing users created using the prototype language to Palo Alto firewalls, and it seems the same goes for other next-generation firewalls as well. Since user and application filtering is a great part of what makes a next-generation firewall next-generation, the inability to define custom users and applications might be considered a fatal blow to the prototype language, but this has in fact no relevance for the languages theoretical workings, it is solely linked to the way Palo Alto, among others, have implemented their user and application databases. Some vendors may have a different approach when dealing with users and applications, but in the abscence of products available for testing, this could not be investigated in this thesis.

Technical difficulties related to programming and PanOS was resolved relatively quick. Two problems that could compromise the security in the policies created using the prototype software are mentioned in section 4.6.1 and 4.6.2. The first one being a software bug that caused the rule order to get cluttered by the way Perl internally sorts hashes. As each rule is a hash stored within a hash, they proved difficult to sort, hence another way to overcome this obstacle was needed. The solution came in the form of an automated numbering function that assigns a number to each rule. This number is used to write rules to policy_c in the order they are written in policy_h. These numbers can easily be used to manually override rule order, and this functionality can be utilized as a new feature in a potential future version of the prototype software. Furthermore, a highly sophisticated rule numbering method would be a to develop a rule analysis engine with the purpose of placing rules in the correct order automatically. There are, to the author's knowledge, no such solution in existance in any firewall today, but the design of the prototype allows for this functionality to be implemented as a post-parse plugin module at a later time.

Services in PanOS rule sets might require other services in order for the network traffic filtering to operate correctly, and an unforeseen glitch that came to light in the final experiment was related to these service dependencies. If a service is dependant on one or more other services, the policy_c will still be implemented, but most likely it will not operate as wanted. This was rectified by simply adding the missing service to the affected rule. Currently there is no way to make sure dependencies are met before implementing policy_i. This would require dependency handling, and possibly a database containing services and their dependencies to other services. All of this can be implemented as a module at a later time.

6.5 Future work

The field of network security is a world of constant change and evolution, and if a universal policy language is to keep up with security technology, it will have to be continously maintained and developed. This would require access to a large amount of next-generation firewall products for development and testing, making it an extremely extensive project requiring big time and financial investments, making it an unlikely project for any private individual.

The inheritance implemented in the prototype is very functional but quite basic in its current form, it allows only for one level of superclasses. In a final product it would be desirable to have multi-level inheritance, which could be achieved by for example using recursive programming to loop through the superclasses. This would allow for superclasses to have superclasses, greatly expanding the flexibility of the language, and opening up for even more simplified policies.

In order to provide a truly universal language, product support must be greatly expanded by developing more compilers, specifically crafted for a variety of firewalls. Further, the PA-200 specific compiler could be improved by adding merging or new configuration file capabilites, that automatically writes a full configuration file. The policy building and deployment process can be fully automated by using SCP and the firewalls command line interface, to automatically send and load a policy$_c$.

To ensure full user support a piece of software that can function as a connection to an authentication service would be a welcommed addition. This can come in the for of a plugin that parses user objects. Such an integration would open up for manual user creation and the possibility to push new users to the user database of the authentication service.

Manually defining objects and writing rule sets when using the prototype for policy building, is more susceptible to typing errors than when using an interface like PanOS, and currently the prototype holds no syntax error or data validity checking mechanisms. A graphical user interface that mostly do object and rule set creation by using point-and-click operations, presents the user with a limited number of choices, an it may the lower error rate in policies. A study on firewall mis-configuration by Avishai Wool[32] show that software mechanisms can be helpful for resolving errors in configurations and policies in firewalls. This implies such error detecting mechanisms could be a valuable contribution to future development of the prototype, enhancing both user-friendliness and robustness.

Language syntax is simple, perhaps too simple. The use of operators and delimiters would likely lead to a more robust language, less prone to errors, which leads us to consider even more programming language like features. How about introducing loops like *for* and *while*, or conditionals

like *if* and *else*? Introducing such features opens up the possibility for compacting policies even more. Loop functionality can be used to create large amounts of objects like networks,groups and rules even faster, while conditionals can prove valuable as a way of checking for policy errors or dependencies. What other consequences the implementation of these ideas could lead to is hard to predict without further development. There might even more to gain in terms of reducing time consumption for policy writing and lowering tuple count, not to mention the adoption of features never before seen in any vendor tool.

Chapter 7

Conclusion

Aiming to merge next-generation firewalls into one unified platform and using this to create a universal language for building firewall policies, this thesis has resulted in a fully functional universal language prototype along with complimentary software. Together, the language and software form a tool that can be used to build universal security policies for next-generation firewalls. By introducing external module support, the prototype design opens up for expandability, and can in theory support a wide range of different firewall products, being appliances or host-based firewalls, legacy or next-generation.

In an attempt to form a basis for the universal language, a comparative study including a select number of next-generation firewalls was performed. This study uncovered significant inequalities between different firewall products, meaning there are certain product specific features that cannot be applied to all types of next-generation firewalls. To truly unleash the full potential of a universal firewall policy language, computer security vendors would have to agree upon a common standard and feature set for next-generation firewalls, thus creating a completely unified platform. Despite of this lack of a common platform, the prototype's mode of operation allows for the use of product specific properties that can be filtered out in a policy compilation process if not supported by the current firewall.

Experiments designed to measure practical usability suggest that policy building can be done in less time when using the prototype compared to using some vendor tools that come shipped with firewalls. One form of complexity, namely the number of tuples in a firewall rule set, can be reduced by taking advantage of the prototypes inheritance abilities, that can help lower tuple count, resulting in time savings in policy creation.

First steps in the quest for a universal firewall language have been taken with the work presented in this thesis. It proves that a universal language can be used to define a range of network objects, services and applications as well as their users, by using a blocks of code to build

policies for firewalls. The concept can be further refined to form a sustainable software tool that can make life easier for network administrators, as well as offering pay-offs in the form of reduced labour and cost for organizations.

Bibliography

[1] Algosec. *Firewall Management*. [Online; accessed 18-Feb-2013]. 2003. URL: http://www.algosec.com/.

[2] Jon Allen. *Perl 5 version 16.2 documentation*. [Online; accessed 19-Apr-2013]. 2012. URL: http://perldoc.perl.org/functions/sort.html.

[3] Ehab Al-Shaer Bin Zhang and Radha Jagadeesan. 'Specifications of A High-level Conflict-Free Firewall Policy Language for Multi-domain Networks'. In: *SACMAT '07 Proceedings of the 12th ACM symposium on Access control models and technologies*. DePaul University. June 2007, pp. 185–194.

[4] John Karsch Eric Byres and Joel Carter. *Firewall deployment for SCADA and process control systems, Good Practice Guide*. Center for the Protection of National Infrastructure. 2005.

[5] Rik Ferguson. *The history of the next-generation firewall*. [Online; accessed 11-Feb-2013]. 2009. URL: http://www.computerweekly.com/news/2240159432/The-history-of-the-Next-Generation-Firewall.

[6] Frost and Sullivan. *Growth of IDS Market to be Boosted by Decreasing Prices and Improved Performances of Wireless Solutions*. [Online; accessed 5-Feb-2013]. 1997. URL: http://www.afterdawn.com/news/press_releases/press_release.cfm/7732/.

[7] *Global Next Generation Firewall market report*. Market Report. Infiniti Research Limited, 2012.

[8] J.D. Guttman. 'Filtering Postures Local Enforcement for Global Policies'. In: *SP 97 Proceedings of the 1997 IEEE Symposium on Security and Privacy*. The MITRE Corporation. May 1997, p. 120.

[9] Richard F. Rashid Jeffrey C. Mogul and Michael J. Accetta. 'The Packet Filter:An Efficient Mechanism for User-level Network Code'. In: *SOSP '87 Proceedings of the eleventh ACM Symposium on Operating systems principles*. Digital Equipment Corporation. Nov. 1987, pp. 39–51.

[10] L. Kleinrock. *Information Flow in Large Communication Nets*. Quarterly Progress Report. Research Laboratory of Electronics, 1961.

[11] Arnaud Launay and Renaud Deraison. *High-Level Firewall Language*. [Online; accessed 18-Feb-2013]. 2000. URL: http://www.hlfl.org/.

[12] J.C.R. Licklider and W. Clark. 'On-Line Man Computer Communication'. In: *AIEE-IRE '62 (Spring) Proceedings of the May 1-3, 1962, spring joint computer conference*. Bolt, Beranek and Newman. May 1962.

[13] Check Software Technologies LTD. *Firewall R75 Administation Guide*. 2010.

[14] J. Mogul. *Using screend to implement IP/TCP security policies*. Tech. rep. Digital Network Systems Laboratory, July 1991.

[15] Barracuda Networks. *Barracuda Firewall*. 2012.

[16] Palo Alto Networks. *Palo Alto Networks Administrators Guide 4.1*. 2011.

[17] Palo Alto Networks. *The Application Usage and Risk Report 7th ed.* Risk Report. Palo Alto Networks, 2011.

[18] John Pescatore and Greg Young. *Defining the Next-Generation Firewall*. Research Note. Gartner, 2009.

[19] Check Point. *Firewall-1 Whitepaper*. White Paper. Check Point, 1997.

[20] M.J. Ranum. 'A network firewall'. In: *Proceedings of World Conference on Systems Management and Security*. Digital Equipment Corporation. 1992.

[21] L. Roberts. 'Multiple Computer Networks and Intercomputer Communication'. In: *SOSP '67 Proceedings of the first ACM symposium on Operating System*. Association for Computing Machinery. Oct. 1967, pp. 3.1–3.6.

[22] L. Roberts and T. Merrill. 'Toward a Cooperative Network of Time-Shared Computers'. In: *AFIPS '66 (Fall) Proceedings of the November 7-10, 1966, fall joint computer conference*. MIT, Lincoln Laboratory. Oct. 1966, pp. 425–431.

[23] SANS Institute Reading Room. *Achieving Defense-in-Depth with Internal Firewalls*. 2001.

[24] Karen Scarfone and Paul Hoffman. *Guidelines on firewalls and Firewall Policy*. Special Pulbication. National Institute of Standards and Technology, 2009.

[25] Gil Shwed. *Check Point awarded patent for stateful inspection technology*. [Online; accessed 10-Feb-2013]. 1997. URL: http://www.checkpoint.com/press/1997/patent2.html.

[26] Dell SonicWALL. *SonicOS Enhanced 5.0 Administrator's Guide*. 2008.

[27] Sourcefire. *Sourcefire 3D System Administrator Guide 4.9.1*. 2010.

[28] Steven Thomason. 'Improving Network Security: Next Generation Firewalls and Advanced Packet Inspection Devices'. In: *Global Journal of Computer Science and Technology Network, Web and Security* (2012).

[29] GNU Tools. *DIFF*. [Online; accessed 6-May-2013]. 1993. URL: http://unixhelp.ed.ac.uk/CGI/man-cgi?diff.

[30] J. Yu V. Fuller T. Li and K. Varadhan. *Classless Inter-Domain Routing (CIDR): an address Assignment and Aggregation Strategy*. RFC. 1993.

[31] Paul Wood. *Internet Security Threat Report*. Threat Report. Symantec, 2012.

[32] Avishai Wool. 'A Quantitative Study of Firewall Configuration Errors'. In: *IEEE Computer Volume:37,Issue: 6*. Tel Aviv University. June 2004, pp. 62–67.

Appendices

.1 Source code

.1.1 Main

```
1   #!/usr/bin/perl
2   use Data::Dumper;
3   use Getopt::Std;
4
5   getopts("vp:hc:o:", \%options ) or usage();
6   usage() if $options{'h'};
7   my $CONFIG = $options{'c'};
8   my $COMPILER = $options{'o'};
9   my $POLICY = $options{'p'};
10
11  if ($options{'v'}){$VERBOSE = 1};
12
13  our %VARIABLES;
14  our %SUPERCLASSES;
15  our %CLASSES;
16  our %RULES;
17  our $rulenumber;
18  our @RULELIST;
19
20  getPlugins();
21  getCompilers();
22  readConfig();
23  readPolicy();
24  inheritance();
25  ruleInheritance();
26  count();
27  compile();
28
29  sub readConfig{
30          open(CONF,"$CONFIG") or die "Unable to open $CONFIG: $!\n";
31          verbose("#Parsing configuration file\n");
32
33          while( my $line = getNextLine( ) ){
34
35                  # normal class block
36                  if ( $line =~ /^\s*(\S+)\s+(\S+)\s+{/ ){
37                          my $type = $1;
38                          my $name = $2;
39                          verbose("Found class block: Type=$type. Name=$name.\n");
40
41                          # checking if we have a parse function for type $type
42                          my $parsefunction = $type . "_parse";
43                          if ( defined(&$parsefunction) ){
44                                  verbose("Calling $parsefunction.\n");
45                                  if ( &$parsefunction($type, $name) ){
46                                          verbose("$parsefunction successful.\n\n");
47
48                                  }
49
50                          }else {
51                                  verbose("$parsefunction failed.\n");
52
53                          }
54
55                          # superclass declaration
56                          } elsif ( $line =~ /^\s*superclass\s+(\S+)\s+(\S+)\s+{/ ){
57                                  my $type = $1;
58                                  my $name = $2;
59                                  verbose("Found superclass block: Type=$type. Name=$name.\n");
60
61                                  # checking if we have a parse function for type $type
62                                  my $parsefunction = $type . "_parse_superclass";
63                                  if ( defined(&$parsefunction) ){
64                                          verbose("Calling $parsefunction.\n");
65                                          if ( &$parsefunction($name) ){
66                                                  verbose("$parsefunction successful.\n\n");
67                                          }
68                                  }
69                                  else {
70                                          verbose("$parsefunction failed.\n");
71                                  }
72
73                                  # variable declaration
74                                  } elsif ( $line =~ /^\s*(\$\S+)\s*=\s*(.*)$/ ){
75                                          my $key = $1;
76                                          my $value = $2;
77                                          chomp $value;
78                                          verbose("Variable declaration: Key=$key. Value=$value\n\n");
79                                          $VARIABLES{$key} = $value;
80                                  }
81
82          }
83
84  }
85
```

```
sub readPolicy{

        $rulenumber = 0;
        open (POLICY, "$POLICY") or die "Unable to open $POLICY: $!\n";

        verbose("\n#Parsing policy file\n");
        while( my $line = getNextLine( ) ){

                if ( $line =~ /^\s*(\S+)\s+"(.*?)"\s+{/ ){
                        my $type = $1;
                        my $name = $2;
                        $rulenumber++;
                        verbose("Found rule block: Name='$name'\n");

                        # checking if we have a parse function for type $type
                        my $parsefunction = $type . "_parse";
                        if ( defined(&$parsefunction) ){
                                verbose("Calling $parsefunction.\n");
                                if ( &$parsefunction($type, $name, $rulenumber) ){
                                        verbose("$parsefunction successful.\n\n");
                                }

                }else { verbose("$parsefunction failed.\n");}

                        }elsif ( $line =~ /^\s*(\$\S+)\s*=\s*(.*)$/ ){
                                my $key = $1;
                                my $value = $2;
                                chomp $value;
                                verbose("Variable declaration: Key=$key. Value=$value\n\n");
                                $VARIABLES{$key} = $value;
                        }
        }
}

sub count {
        verbose("#Summary:\n");
        foreach my $type ( keys %CLASSES ){
                $typecount = scalar (keys $CLASSES{$type});
                verbose("Found $typecount $type" ."s.\n");
        }
}

sub ruleInheritance {

        verbose("#Rule inheritance\n");

foreach $rule (keys %RULES) {
        foreach $rulename (keys %{$RULES{$rule}}){

                foreach $ruleproperty (keys %{$RULES{$rule}{$rulename}}){

                        # find superclass
                        foreach $supername (keys %{$SUPERCLASSES{$$rule}}){

                                foreach $rsuperproperty (keys %{$SUPERCLASSES{$rule}{$supername}}){

                                        #if ($rsuperproperty eq $ruleproperty ){}else
if ($rsuperproperty ne $ruleproperty )
{
# merge if non-existent property
if (not exists $RULES{$rule}{$rulename}{$rsuperproperty}){
$RULES{$rule}{$rulename}{$rsuperproperty} = $SUPERCLASSES{$rule}{$supername}{$rsuperproperty};

verbose("Inserting $rsuperproperty '$SUPERCLASSES{$rule}{$supername}{$rsuperproperty}'\n");

                                                }

                                        }

                                }

                        }

                }
        }
}
verbose("\n");

}


sub inheritance {
verbose("#Class inheritance\n");

foreach $class (keys %CLASSES){

        foreach $name (keys %{$CLASSES{$class}}){

                # find superclass
                my $superclass = $CLASSES{$class}{$name}{"superclass"};
                if ($superclass eq ""){next;}
                verbose("\nInherit for $name\n");
```

```
178                     verbose("Superclass is $superclass\n");
179                     if ( $superclass ){
180
181                             # get superclass tree
182                             my $temph = getSuperclassHash($class,$superclass);
183                             my %superhash = %$temph;
184
185                             # merge if non-existent property
186                             foreach $superproperty (keys %superhash ){
187                                     verbose("Checking superproperty $superproperty\n");
188
189                     print "HER: $superproperty  = $CLASSES{$class}{$name}{$superproperty}\n";
190
191                     if (not exists $CLASSES{$class}{$name}{$superproperty}){
192                             $CLASSES{$class}{$name}{$superproperty} = $superhash{$superproperty};
193                 verbose("Inserting $superproperty '$superhash{$superproperty}' into class '$name'\n");
194                                     }
195
196                             }
197
198                     }
199
200             }
201
202     }
203     verbose("\n");
204
205     }
206
207
208     sub compile {
209
210     my $name;
211     my $value;
212
213     # sort rule by number and store in hash
214
215     foreach $rule (keys %RULES) {
216
217             foreach $rulename (keys %{$RULES{$rule}}){
218
219                     $RULELIST[$RULES{$rule}{$rulename}{"number"}] = $rulename;
220             }
221     }
222
223
224                                             verbose("\n#Compiling.\n");
225
226     # send rules ordered numerically to compiler
227     beginRuleset();
228             foreach $rulenames (@RULELIST){
229
230             foreach $rule (keys %RULES) {
231
232                     foreach $rulename (keys %{$RULES{$rule}}){
233
234
235                             if ($rulenames eq $rulename){
236                                     $name = $rulename;
237                                     beginRule($name);
238
239
240                                     verbose("\nCreating rule: $name:\n");
241
242                                     foreach $property (keys %{$RULES{$rule}{$rulename}}){
243
244
245                             $value = $RULES{$rule}{$rulename}{$property};
246                             compiler($property,$value);
247                             verbose("Inserting $property = $value\n") unless ($property eq 'number');
248
249
250                                     }
251                             }
252
253                             }
254
255                     }
256                     if ($rulenames != undef){endRule();}
257
258             }
259
260             endRuleset();
261             verbose("\nDone. Wrote $rulenumber rules to file.\n");
262     }
263
264
265
266     sub getNextLine {
267
268             if ( CONF ){
269                     while ( my $line =  <CONF> ){
```

```
                        chomp $line;
                        return $line unless $line eq "";
                }
        }
        if ( POLICY ){
                while ( my $line =  <POLICY> ){
                        chomp $line;
                        return $line unless $line eq "";
                }
        }

}

sub getPlugins {
        my $plugin_path = `pwd`;
        chomp $plugin_path;
        $plugin_path .= "/";

        my @pluginlist = `ls plugins/*.plugin`;

        verbose("#Loading plugins\n");

        foreach my $plugin (@pluginlist ){
                chomp $plugin;
                verbose("Found plugin: ".$plugin ."\n");
                require $plugin_path . $plugin;
        }
        verbose("\n");

}

sub getSuperclassHash {
        my $class = $_[0];
        my $superclass = $_[1];
        return $SUPERCLASSES{$class}{$superclass};

}

sub getCompilers {
        my $compiler_path = `pwd`;
        chomp $compiler_path;
        $compiler_path .= "/";
        my @compilerlist = `ls *.comp`;

        foreach my $compiler (@compilerlist ){
                chomp $compiler;
                verbose("Found compiler: ".$compiler ."\n");
                require $compiler_path . $compiler;
        }
        verbose("#Using compiler: $COMPILER\n");
}

sub verbose {

        print $_[0] if $VERBOSE;

}

sub expandString {

        my $string = $_[0];
        $string =~ s/(\$\S+)/$VARIABLES{$1}/g;
        return $string;
}

sub usage {
print "Usage: $0 [-p policy] [-c configuration] [-o output format] -v(verbose) -h(help) \n";
exit (0);
}

1;
```

.1.2 Plugins

Application

```

sub application_parse {

    my $name = $_[1];
    verbose((caller(0))[3]." called for block: $name\n");

    while( my $line = getNextLine()){
        if ( $line eq "}" ){
            last;
        }
        $line = expandString($line);
```

```
        $line =~ /\s+(\S+)\s+(.*)$/;

        $CLASSES{"application"}{$name}{$1} = $2;
        verbose("Inserting into application:$name:$1 -> $2\n");
    }

    verbose("Block parsing complete.\n");
    return 1;
}

sub application_parse_superclass {

    my $name = $_[0];
    verbose((caller(0))[3]." called for block: $name\n");

    while( my $line = getNextLine()){
        $line = expandString($line);
        if ( $line eq "}" ){
            last;
        }
        $line =~ /\s+(\S+)\s+(.*)$/;

        $SUPERCLASSES{"application"}{$name}{$1} = $2;
        verbose("Inserting into application:$name:$1 -> $2\n");

    }

    return 1;

}


1;
```

Service

```

sub service_parse {
    my $type = $_[0];
    my $name = $_[1];
    verbose((caller(0))[3]." called for block: $name\n");

    while( my $line = getNextLine()){
        if ( $line eq "}" ){
            last;
        }
        $line = expandString($line);
        $line =~ /\s+(\S+)\s+(.*)$/;

        $CLASSES{"service"}{$name}{$1} = $2;
        verbose("Inserting into service:$name:$1 -> $2\n");
    }

    verbose("Block parsing complete.\n");
    return 1;
}

sub service_parse_superclass {

    my $name = $_[0];
    verbose((caller(0))[3]." called for block: $name\n");

    while( my $line = getNextLine()){
        $line = expandString($line);
        if ( $line eq "}" ){
            last;
        }
        $line =~ /\s+(\S+)\s+(.*)$/;

        $SUPERCLASSES{"service"}{$name}{$1} = $2;
        verbose("Inserting into service:$name:$1 -> $2\n");

    }

    return 1;

}


1;
```

Zone

```

```

```
2  sub zone_parse {
3
4      my $name = $_[1];
5      verbose((caller(0))[3]." called for block: $name\n");
6      while( my $line = getNextLine()){
7          if ( $line eq "}" ){
8              last;
9          }
10         $line = expandString($line);
11         $line =~ /\s+(\S+)\s+(.*)$/;
12         $CLASSES{"zone"}{$name}{$1} = $2;
13         verbose("Inserting into zone:$name:$1 -> $2\n");
14     }
15
16
17     verbose("Block parsing complete.\n");
18     return 1;
19 }
20
21 sub zone_parse_superclass {
22
23     my $name = $_[0];
24     verbose((caller(0))[3]." called for block: $name\n");
25
26     while( my $line = getNextLine()){
27         $line = expandString($line);
28         if ( $line eq "}" ){
29             last;
30         }
31         $line =~ /\s+(\S+)\s+(.*)$/;
32         $SUPERCLASSES{"zone"}{$name}{$1} = $2;
33         verbose("Inserting into zone:$name:$1 -> $2\n");
34
35     }
36
37     return 1;
38
39 }
40
41
42 1;
```

Network

```
1
2
3  sub network_parse {
4
5      my $name = $_[1];
6      verbose((caller(0))[3]." called for block: $name\n");
7      while( my $line = getNextLine()){
8          if ( $line eq "}" ){
9              last;
10         }
11         $line = expandString($line);
12         $line =~ /\s+(\S+)\s+(.*)$/;
13         $CLASSES{"network"}{$name}{$1} = $2;
14         verbose("Inserting into network:$name:$1 -> $2\n");
15     }
16
17
18     verbose("Block parsing complete.\n");
19     return 1;
20 }
21
22 sub network_parse_superclass {
23
24     my $name = $_[0];
25     verbose((caller(0))[3]." called for block: $name\n");
26
27     while( my $line = getNextLine()){
28         $line = expandString($line);
29         if ( $line eq "}" ){
30             last;
31         }
32         $line =~ /\s+(\S+)\s+(.*)$/;
33         $SUPERCLASSES{"network"}{$name}{$1} = $2;
34         verbose("Inserting into network:$name:$1 -> $2\n");
35
36     }
37
38     return 1;
39
40 }
41
42
43 1;
```

Group

```
1
2  sub group_parse {
3
4      my $name = $_[1];
5      verbose((caller(0))[3]."␣called␣for␣block:␣$name\n");
6      while( my $line = getNextLine()){
7          if ( $line eq "}" ){
8              last;
9          }
10         $line = expandString($line);
11         $line =~ /\s+(\S+)\s+(.*)$/;
12         $CLASSES{"group"}{$name}{$1} = $2;
13         verbose("Inserting␣into␣group:$name:$1␣->␣$2\n");
14     }
15
16
17     verbose("Block␣parsing␣complete.\n");
18     return 1;
19 }
20
21 sub group_parse_superclass {
22
23     my $name = $_[0];
24     verbose((caller(0))[3]."␣called␣for␣block:␣$name\n");
25
26     while( my $line = getNextLine()){
27         $line = expandString($line);
28         if ( $line eq "}" ){
29             last;
30         }
31         $line =~ /\s+(\S+)\s+(.*)$/;
32         $SUPERCLASSES{"group"}{$name}{$1} = $2;
33         verbose("Inserting␣into␣group:$name:$1␣->␣$2\n");
34
35     }
36     return 1;
37
38 }
39
40
41 1;
```

Rule

```
1
2  sub rule_parse {
3      my $name = $_[1];
4
5      verbose((caller(0))[3]."␣called␣for␣block:␣$name\n");
6      while( my $line = getNextLine()){
7          if ( $line eq "}" ){
8              last;
9          }
10         $line = expandString($line);
11         $line =~ /\s+(\S+)\s+(.*)$/;
12
13         $RULES{"rule"}{$name}{$1} = $2;
14         $RULES{"rule"}{$name}{"number"} = $_[2];
15
16         verbose("Inserting␣into␣rules:$name:$1␣->␣$2\n");
17     }
18
19     verbose("Block␣parsing␣complete.\n");
20     return 1;
21 }
22
23 sub rule_parse_superclass {
24
25     my $name = $_[0];
26     verbose((caller(0))[3]."␣called␣for␣block:␣$name\n");
27
28     while( my $line = getNextLine()){
29         $line = expandString($line);
30         if ( $line eq "}" ){
31             last;
32         }
33         $line =~ /\s+(\S+)\s+(.*)$/;
34
35         $SUPERCLASSES{"rule"}{$name}{$1} = $2;
36         verbose("Inserting␣into␣service:$name:$1␣->␣$2\n");
37
38     }
39
40     return 1;
41
42 }
```

```
43
44 1;
```

.1.3 PA-200 Compiler

```
1  require "parser.pl";
2  my $rules = "rules";
3  my $zones = "zones";
4  my $networks = "networks";
5  my $groups = "groups";
6  my $hosts = "hosts";
7  my $services = "services";
8
9  open(OUTPUT,'>>'.$rules) or die "\nUnable␣to␣create␣'$output'\n";
10 open(ZONE,'>>'.$zones) or die "\nUnable␣to␣create␣'$zones'\n";
11 open(NET,'>>'.$networks) or die "\nUnable␣to␣create␣'$networks'\n";
12 open(GROUP,'>>'.$groups) or die "\nUnable␣to␣create␣'$groups'\n";
13 open(SERVICE,'>>'.$services) or die "\nUnable␣to␣create␣'$services'\n";
14
15 zoneGenerator();
16 serviceGenerator();
17 groupGenerator();
18 networkGenerator();
19
20 sub beginRuleset{
21
22 print OUTPUT "\t\t\t<rules>\n";
23
24 }
25
26 sub beginRule {
27 print OUTPUT "\t\t␣␣␣".'<entry␣name="'.$_[0].'">';
28 print OUTPUT "\n";
29 print OUTPUT "\t\t␣␣␣␣<option>\n";
30 print OUTPUT "\t\t\t␣<disable-server-response-inspection>no</disable-server-response-inspection>\n";
31 print OUTPUT "\t\t␣␣␣␣</option>\n";
32 return 1;
33 }
34
35 sub compiler {
36
37         $negate_source = 0;
38         $negate_destination = 0;
39
40         if ($_[0] eq "superclass" || $_[0] eq "number"){return 1;}
41
42         if (($_[0] ne 'log') and ($_[0] ne 'action')){
43
44                 my @values = split(/,/, $_[1]);
45
46                 print OUTPUT "\t\t\t<$_[0]>\n";
47                 foreach (@values){
48
49                         if ($_ =~ s/^\!//s){
50
51                                 if ($_[0] eq 'source'){
52                                         $negate_source = 1;
53                                 }
54                                 if ($_[0] eq 'destination'){
55                                         $negate_destination = 1;
56                                 }
57                         }
58                         print OUTPUT "\t\t\t␣␣<member>$_</member>\n";
59                 }
60
61                 print OUTPUT "\t\t\t</$_[0]>\n";
62         }
63
64         if ($_[0] eq 'action'){
65                 print OUTPUT "\t\t\t<$_[0]>$_[1]</$_[0]>\n";
66         }
67
68         if (($_[0] eq 'log') and ($_[1] eq 'on')) {
69                 print OUTPUT "\t\t\t<log-end>yes</log-end>\n";
70         }
71
72         if($negate_source){
73                 print OUTPUT "\t\t\t<negate-source>yes</negate-source>\n";
74         }
75
76         if ($negate_destination){
77                 print OUTPUT "\t\t\t<negate-destination>yes</negate-destination>\n";
78         }
79         return 1;
80 }
81
82 sub endRule {
83         print OUTPUT "\t\t␣␣␣</entry>\n";
84         return 1;
```

```
}

sub endRuleset {
        print OUTPUT "\t\t</rules>\n";
        return 1;
}

sub zoneGenerator {

print ZONE "\t\t<zone>\n";

foreach $class (keys %CLASSES){

        if ($class eq 'zone'){

                foreach $name (keys %{$CLASSES{$class}}){

                        print ZONE "\t\t  '<entry name="'.$name.'">'."\n";
                        print ZONE "\t\t    <network>\n";
                        print ZONE "\t\t      <layer3>\n";

                        foreach $property (keys %{$CLASSES{$class}{$name}}){
                                if ($property eq "interface"){
                print ZONE "\t\t\t<member>$CLASSES{$class}{$name}{$property}</member>\n";

                                }
                        }
                        print ZONE "\t\t      </layer3>\n";
                        print ZONE "\t\t    </network>\n";
                        print ZONE "\t\t  </entry>\n";
                }

        }
}
print ZONE "\t\t</zone>\n";

}

sub serviceGenerator {

my $last;
print SERVICE "\t\t<service>\n";

foreach $class (keys %CLASSES){

        if ($class eq 'service'){

                foreach $name (keys %{$CLASSES{$class}}){

                        print SERVICE "\t\t  '<entry name="'.$name.'">'."\n";
                        foreach $property (keys %{$CLASSES{$class}{$name}}){
                                if ($property eq "protocol"){
                                        print SERVICE "\t\t    <protocol>\n";
                        print SERVICE "\t\t\t<$CLASSES{$class}{$name}{$property}>\n";
                                        $last = $CLASSES{$class}{$name}{$property};

                                }
                                if ($property eq "port"){
                        print SERVICE "\t\t\t    <port>$CLASSES{$class}{$name}{$property}</port>\n";
                                }


                        }
                        print SERVICE "\t\t\t</$last>\n";
                        print SERVICE "\t\t    </protocol>\n";
                        print SERVICE "\t\t  </entry>\n";
                }
        }
}
print SERVICE "\t\t</service>\n";

}

sub networkGenerator {


print NET "\t\t<address>\n";

foreach $class (keys %CLASSES){

        if ($class eq 'network'){

                foreach $name (keys %{$CLASSES{$class}}){

                        print NET "\t\t  '<entry name="'.$name.'">'."\n";

                foreach $property (keys %{$CLASSES{$class}{$name}}){
                                if ($property eq "ip"){
                        print NET "\t\t\t<ip-netmask>$CLASSES{$class}{$name}{$property}";

}elsif ($property ne "superclass") {print NET "$CLASSES{$class}{$name}{$property}</ip-netmask>\n";}
                        }
```

```
177
178                             print NET "\t\t  </entry>\n";
179                     }
180
181             }
182     }
183     print NET "\t\t</address>\n";
184
185     }
186
187     sub groupGenerator {
188
189     print GROUP "\t\t<address-group>\n";
190
191     foreach $class (keys %CLASSES){
192
193             if ($class eq 'group'){
194
195                     foreach $name (keys %{$CLASSES{$class}}){
196
197                             print GROUP "\t\t  ".'<entry name="'.$name.'">'."\n";
198
199                     foreach $property (keys %{$CLASSES{$class}{$name}}){
200
201                             my @values = split(/,/, $CLASSES{$class}{$name}{$property});
202
203                             foreach (@values){
204                                     print GROUP "\t\t\t <member>$_</member>\n";
205                                     }
206                             }
207                             print GROUP "\t\t  </entry>\n";
208                     }
209
210             }
211
212     }
213     print GROUP "\t\t</address-group>\n";
214     }
215
216     1;
```

.2 Configuration files

.2.1 Experiment 1

```
superclass rule rule_common {
        action allow
        from any
        to any
        source any
        destination any
        log on
        hip-profile any
        source-user any
        category any
}

service rdp {
        port 5800,5900
        protocol tcp
}

zone Internal {
        interface ethernet1/1
}

zone External {
        interface ethernet1/2
}

network int-client {
        ip 10.1.1.0
        netmask /24
}
```

.2.2 Experiment 2

```
superclass service service_common {
        protocol tcp
        admin aslak
}

superclass rule rule_common {
        source-user any
```

```
        category any
        hip-profiles any
        action allow
        source any
        destination any
        from any
        to any
        log on
        service any
        application any
}

superclass network network_common {
        netmask /24
}

zone External {
        interface ethernet1/1
}

zone Internal {
        interface ethernet1/2
}

zone DMZ {
        interface ethernet1/3
}

zone Secure {
        interface ethernet1/4
}


application smtp {

}

service http-proxy {
        port 8080
        superclass service_common
}

network dmz-email-segment {
        ip 192.168.2.0
        superclass network_common
}

network dmz-web-proxy {
        ip 192.168.1.0
        superclass network_common
}

network int-backoffice {
        ip 10.2.0.0
        netmask /16
        superclass network_common
}

network int-client-1 {
        ip 10.1.1.0
        superclass network_common
}

network int-client-2 {
        ip 10.1.2.0
        superclass network_common
}

network int-client-3 {
        ip 10.1.3.0
        superclass network_common
}

network int-ts {
        ip 10.4.0.0
        netmask /16
}

network sec-journal {
        ip 172.16.129.0
        superclass network_common
}

network sec-medequip-xray {
        ip 172.16.1.0
        superclass network_common
}

network sec-medequip-mr {
        ip 172.16.2.0
        superclass network_common
}
```

```
group dmz-email {
        members dmz-email-segment
}

group int-client {
        members int-client-1,int-client-2,int-client-3
}

group sec {
        members sec-journal,sec-medequip-mr,sec-medequip-xray
}
```

.3 Policy files

.3.1 Experiment 1

```
rule "web access" {
        from Internal
        to External
        application web-browsing
}

rule "allow torrents" {
        from Internal
        to External
        source int-client
        application bittorrent
}

rule "remote desktop access" {
        from External
        to Internal
        destination int-client
        service rdp
}

rule "clean up" {
        from External,Internal
        to External,Internal
        action deny
}
```

.3.2 Experiment 2

```
      rule "0-1 Email receive" {
        to DMZ
        destination dmz-email-segment
        application smtp
}

rule "0-123 deny" {
        from External
        to DMZ,Internal,Secure
        action deny
}

rule "1-0 Email-send" {
        from DMZ
        source dmz-email-segment
        to External
        application smtp
}

rule "1-0 Web-access" {
        from DMZ
        source dmz-web-proxy
        to External
        service service-http,service-https
}

rule "1-0 deny" {
        from DMZ
        to External
        action deny
}

rule "1-2 Email-backoffice" {
        from DMZ
        source dmz-email-segment
        to Internal
        destination int-backoffice
        application smtp
}
```

```
rule "1-023_deny" {
        from DMZ
        to External,Internal,Secure
        action deny
}

rule "2-1_client-webproxy" {
        from Internal
        source int-client
        to DMZ
        destination dmz-web-proxy
        service http-proxy
}

rule "2-2_client-ts" {
        from Internal
        source int-client
        to Internal
        destination int-ts
        application ms-rdp
}

rule "2-2_client-backoffice" {
        from Internal
        source int-client
        to Internal
        destination int-backoffice
        application hp-jetdirect,ms-rdp,sap,ssl
}

rule "2-3_ts_secure" {
        from Internal
        source int-ts
        to Secure
        destination sec
        application cygnet-scada,ftp,socks2http
}

rule "2-013_deny" {
        from Internal
        to DMZ,External,Internal,Secure
        action deny
}

rule "3-012_deny" {
        from Secure
        to DMZ,External,Internal
        action deny
        log off
}
```

.4 Compiler generated files

.4.1 Zones

```
<zone>
  <entry name="Internal">
    <network>
      <layer3>
        <member>ethernet1/2</member>
      </layer3>
    </network>
  </entry>
  <entry name="Secure">
    <network>
      <layer3>
        <member>ethernet1/4</member>
      </layer3>
    </network>
  </entry>
  <entry name="DMZ">
    <network>
      <layer3>
        <member>ethernet1/3</member>
      </layer3>
    </network>
  </entry>
  <entry name="External">
    <network>
      <layer3>
        <member>ethernet1/1</member>
      </layer3>
    </network>
  </entry>
</zone>
```

.4.2 Groups

```
<address-group>
                    <entry name="sec">
                            <member>sec-journal</member>
                            <member>sec-medequip-mr</member>
                            <member>sec-medequip-xray</member>
                    </entry>
                    <entry name="int-client">
                            <member>int-client-1</member>
                            <member>int-client-2</member>
                            <member>int-client-3</member>
                    </entry>
                    <entry name="dmz-email">
                            <member>dmz-email-segment</member>
                    </entry>
                  </address-group>
```

.4.3 Networks

```
<address>
                    <entry name="int-client-1">
                            <ip-netmask>10.1.1.0/24</ip-netmask>
                    </entry>
                    <entry name="dmz-email-segment">
                            <ip-netmask>192.168.2.0/24</ip-netmask>
                    </entry>
                    <entry name="sec-medequip-xray">
                            <ip-netmask>172.16.1.0/24</ip-netmask>
                    </entry>
                    <entry name="int-client-2">
                            <ip-netmask>10.1.2.0/24</ip-netmask>
                    </entry>
                    <entry name="int-ts">
                            <ip-netmask>10.4.0.0/16</ip-netmask>
                    </entry>
                    <entry name="int-client-3">
                            <ip-netmask>10.1.3.0/24</ip-netmask>
                    </entry>
                    <entry name="sec-journal">
                            <ip-netmask>172.16.129.0/24</ip-netmask>
                    </entry>
                    <entry name="int-backoffice">
                            <ip-netmask>10.2.0.0/16</ip-netmask>
                    </entry>
                    <entry name="dmz-web-proxy">
                            <ip-netmask>192.168.1.0/24</ip-netmask>
                    </entry>
                    <entry name="sec-medequip-mr">
                            <ip-netmask>172.16.2.0/24</ip-netmask>
                    </entry>
                  </address>
```

.4.4 Services

```
                  <service>
                    <entry name="http-proxy">
                      <protocol>
                            <tcp>
                              <port>8080</port>
                            </tcp>
                      </protocol>
                    </entry>
                  </service>
```

.4.5 Rules

```
                        <rules>
                     <entry name="0-1␣Email␣receive">
                      <option>
                            <disable-server-response-inspection>no</disable-server-response-inspection>
                      </option>
                          <source>
                            <member>any</member>
                          </source>
                          <destination>
                            <member>dmz-email-segment</member>
                          </destination>
                          <application>
```

```
            <member>smtp</member>
          </application>
          <source-user>
            <member>any</member>
          </source-user>
          <service>
            <member>any</member>
          </service>
          <to>
            <member>DMZ</member>
          </to>
          <from>
            <member>any</member>
          </from>
          <action>allow</action>
          <hip-profiles>
            <member>any</member>
          </hip-profiles>
          <category>
            <member>any</member>
          </category>
          <log-end>yes</log-end>
        <entry name="0-123_deny">
         <option>
           <disable-server-response-inspection>no</disable-server-response-inspection>
         </option>
          <source>
            <member>any</member>
          </source>
          <destination>
            <member>any</member>
          </destination>
          <application>
            <member>any</member>
          </application>
          <source-user>
            <member>any</member>
          </source-user>
          <service>
            <member>any</member>
          </service>
          <to>
            <member>DMZ</member>
            <member>Internal</member>
            <member>Secure</member>
          </to>
          <from>
            <member>External</member>
          </from>
          <hip-profiles>
            <member>any</member>
          </hip-profiles>
          <action>deny</action>
          <category>
            <member>any</member>
          </category>
          <log-end>yes</log-end>
        <entry name="1-0_Email-send">
         <option>
           <disable-server-response-inspection>no</disable-server-response-inspection>
         </option>
          <source>
            <member>dmz-email-segment</member>
          </source>
          <destination>
            <member>any</member>
          </destination>
          <application>
            <member>smtp</member>
          </application>
          <source-user>
            <member>any</member>
          </source-user>
          <service>
            <member>any</member>
          </service>
          <to>
            <member>External</member>
          </to>
          <from>
            <member>DMZ</member>
          </from>
          <action>allow</action>
          <hip-profiles>
            <member>any</member>
          </hip-profiles>
          <category>
            <member>any</member>
          </category>
          <log-end>yes</log-end>
        </entry>
        <entry name="1-0_Web-access">
```

```
 <option>
       <disable-server-response-inspection>no</disable-server-response-inspection>
 </option>
      <source>
        <member>dmz-web-proxy</member>
      </source>
      <destination>
        <member>any</member>
      </destination>
      <application>
        <member>any</member>
      </application>
      <source-user>
        <member>any</member>
      </source-user>
      <service>
        <member>service-http</member>
        <member>service-https</member>
      </service>
      <to>
        <member>External</member>
      </to>
      <from>
        <member>DMZ</member>
      </from>
      <action>allow</action>
      <hip-profiles>
        <member>any</member>
      </hip-profiles>
      <category>
        <member>any</member>
      </category>
      <log-end>yes</log-end>
</entry>
<entry name="1-0_deny">
 <option>
       <disable-server-response-inspection>no</disable-server-response-inspection>
 </option>
      <source>
        <member>any</member>
      </source>
      <destination>
        <member>any</member>
      </destination>
      <application>
        <member>any</member>
      </application>
      <source-user>
        <member>any</member>
      </source-user>
      <service>
        <member>any</member>
      </service>
      <to>
        <member>External</member>
      </to>
      <from>
        <member>DMZ</member>
      </from>
      <hip-profiles>
        <member>any</member>
      </hip-profiles>
      <action>deny</action>
      <category>
        <member>any</member>
      </category>
      <log-end>yes</log-end>
</entry>
<entry name="1-2_Email-backoffice">
 <option>
       <disable-server-response-inspection>no</disable-server-response-inspection>
 </option>
      <source>
        <member>dmz-email-segment</member>
      </source>
      <destination>
        <member>int-backoffice</member>
      </destination>
      <application>
        <member>smtp</member>
      </application>
      <source-user>
        <member>any</member>
      </source-user>
      <service>
        <member>any</member>
      </service>
      <to>
        <member>Internal</member>
      </to>
      <from>
        <member>DMZ</member>
```

```
        </from>
        <action>allow</action>
        <hip-profiles>
          <member>any</member>
        </hip-profiles>
        <category>
          <member>any</member>
        </category>
        <log-end>yes</log-end>
      </entry>
      <entry name="1-023 deny">
       <option>
          <disable-server-response-inspection>no</disable-server-response-inspection>
       </option>
        <source>
          <member>any</member>
        </source>
        <destination>
          <member>any</member>
        </destination>
        <application>
          <member>any</member>
        </application>
        <source-user>
          <member>any</member>
        </source-user>
        <service>
          <member>any</member>
        </service>
        <to>
          <member>External</member>
          <member>Internal</member>
          <member>Secure</member>
        </to>
        <from>
          <member>DMZ</member>
        </from>
        <hip-profiles>
          <member>any</member>
        </hip-profiles>
        <action>deny</action>
        <category>
          <member>any</member>
        </category>
        <log-end>yes</log-end>
      </entry>
      <entry name="2-1 client-webproxy">
       <option>
          <disable-server-response-inspection>no</disable-server-response-inspection>
       </option>
        <source>
          <member>int-client</member>
        </source>
        <destination>
          <member>dmz-web-proxy</member>
        </destination>
        <application>
          <member>any</member>
        </application>
        <source-user>
          <member>any</member>
        </source-user>
        <service>
          <member>http-proxy</member>
        </service>
        <to>
          <member>DMZ</member>
        </to>
        <from>
          <member>Internal</member>
        </from>
        <action>allow</action>
        <hip-profiles>
          <member>any</member>
        </hip-profiles>
        <category>
          <member>any</member>
        </category>
        <log-end>yes</log-end>
      </entry>
      <entry name="2-2 client-ts">
       <option>
          <disable-server-response-inspection>no</disable-server-response-inspection>
       </option>
        <source>
          <member>int-client</member>
        </source>
        <destination>
          <member>int-ts</member>
        </destination>
        <application>
          <member>ms-rdp</member>
```

```
        </application>
        <source-user>
          <member>any</member>
        </source-user>
        <service>
          <member>any</member>
        </service>
        <to>
          <member>Internal</member>
        </to>
        <from>
          <member>Internal</member>
        </from>
        <action>allow</action>
        <hip-profiles>
          <member>any</member>
        </hip-profiles>
        <category>
          <member>any</member>
        </category>
        <log-end>yes</log-end>
      </entry>
      <entry name="2-2_client-backoffice">
       <option>
          <disable-server-response-inspection>no</disable-server-response-inspection>
       </option>
        <source>
          <member>int-client</member>
        </source>
        <destination>
          <member>int-backoffice</member>
        </destination>
        <application>
          <member>hp-jetdirect</member>
          <member>ms-rdp</member>
          <member>sap</member>
        </application>
        <source-user>
          <member>any</member>
        </source-user>
        <service>
          <member>any</member>
        </service>
        <to>
          <member>Internal</member>
        </to>
        <from>
          <member>Internal</member>
        </from>
        <action>allow</action>
        <hip-profiles>
          <member>any</member>
        </hip-profiles>
        <category>
          <member>any</member>
        </category>
        <log-end>yes</log-end>
      </entry>
      <entry name="2-3_ts_secure">
       <option>
          <disable-server-response-inspection>no</disable-server-response-inspection>
       </option>
        <source>
          <member>int-ts</member>
        </source>
        <destination>
          <member>sec</member>
        </destination>
        <application>
          <member>cygnet-scada</member>
          <member>ftp</member>
          <member>socks2http</member>
        </application>
        <source-user>
          <member>any</member>
        </source-user>
        <service>
          <member>any</member>
        </service>
        <to>
          <member>Secure</member>
        </to>
        <from>
          <member>Internal</member>
        </from>
        <action>allow</action>
        <hip-profiles>
          <member>any</member>
        </hip-profiles>
        <category>
          <member>any</member>
        </category>
```

```
        <log-end>yes</log-end>
    </entry>
    <entry name="2-013_deny">
     <option>
          <disable-server-response-inspection>no</disable-server-response-inspection>
     </option>
        <source>
          <member>any</member>
        </source>
        <destination>
          <member>any</member>
        </destination>
        <application>
          <member>any</member>
        </application>
        <source-user>
          <member>any</member>
        </source-user>
        <service>
          <member>any</member>
        </service>
        <to>
          <member>DMZ</member>
          <member>External</member>
          <member>Internal</member>
          <member>Secure</member>
        </to>
        <from>
          <member>Internal</member>
        </from>
        <hip-profiles>
          <member>any</member>
        </hip-profiles>
        <action>deny</action>
        <category>
          <member>any</member>
        </category>
        <log-end>yes</log-end>
    </entry>
    <entry name="3-012_deny">
     <option>
          <disable-server-response-inspection>no</disable-server-response-inspection>
     </option>
        <source>
          <member>any</member>
        </source>
        <destination>
          <member>any</member>
        </destination>
        <application>
          <member>any</member>
        </application>
        <source-user>
          <member>any</member>
        </source-user>
        <service>
          <member>any</member>
        </service>
        <to>
          <member>DMZ</member>
          <member>External</member>
          <member>Internal</member>
        </to>
        <from>
          <member>Secure</member>
        </from>
        <hip-profiles>
          <member>any</member>
        </hip-profiles>
        <action>deny</action>
        <category>
          <member>any</member>
        </category>
    </entry>
  </rules>
```