UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

RAFAEL HENGEN RIBEIRO

A Bottom-up Approach for Extracting Network Intents

Dissertation presented in partial fulfillment
of the requirements for the degree of
Master of Computer Science

Advisor: Prof. Dr. Lisandro Zambenedetti
Granville

Porto Alegre
November 2020

CIP — CATALOGING-IN-PUBLICATION

Ribeiro, Rafael Hengen

A Bottom-up Approach for Extracting Network Intents / Rafael Hengen Ribeiro. – Porto Alegre: PPGC da UFRGS, 2020.

75 f.: il.

Dissertation (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2020. Advisor: Lisandro Zambenedetti Granville.

1. Computer Networks. 2. Network Management. 3. Intent-Based Networking. 4. Intents. 5. Programming Languages. I. Granville, Lisandro Zambenedetti. II. Título.

“Those who can imagine anything,
can create the impossible.”
— Alan Turing

ACKNOWLEDGMENTS

First of all, I would like to thank my girlfriend, Andrieli, for all the support, commitment, and love. I would like to thank my family: my father, Leonir, my grandmother, and my siblings, Raquel and Juliano. Thanks for all their support and dedication, providing me help when necessary.

I would like to thank my advisor, Prof. Dr. Lisandro Zambenedetti Granville, for all orientations, teachings, and experiences that he shared with me. I will always be grateful for his guidance.

I would like to express my gratitude for all the INF colleagues who always encouraged me to improve. Special thanks to Arthur, Luciano, and Ricardo from Lab. 210, which always help me in papers and in the dissertation.

ABSTRACT

Intent-Based Networking (IBN) is showing significant improvements in network management, especially by reducing the complexity by using intent-level languages. However, IBN is not yet integrated and widely deployed in most of the large-scale networks. Network operators may still encounter several issues deploying new intents, such as reasoning about complex configurations to understand previously deployed rules before writing an intent to update the network state. Large-scale networks may include several devices distributed in multiple hierarchical levels of its topology; each of these devices is configured using low-level, vendor-specific languages. Thus, inferring intents from these low-level configuration files can be an arduous and time-consuming task. Current solutions that derive high-level representations from bottom-up configuration analysis can not represent configurations in an intent-level. Moreover, they fail by not aggregating essential details to enhance the representation. In this work, we present a bottom-up process to extract intents from network configurations. To validate our approach, we develop a system called SCRIBE (SeCuRity Intent-Based Extractor), which decompiles security configurations from different network devices and translates them to an intent-level language called Nile. To demonstrate the feasibility of SCRIBE, we outline two case studies and evaluate our approach with dumps of real-world firewall configurations containing rules from various servers and institutions. Results show that our solution can represent configurations in intent-level and also maintain a high accuracy representing aspects of low-level configurations.

Keywords: Computer Networks. Network Management. Intent-Based Networking. Intents. Programming Languages.

Uma abordagem *bottom-up* para extração de intenções de rede

RESUMO

O paradigma Intent-Based Networking (IBN) está mostrando melhorias significativas no gerenciamento de redes, especialmente na redução da complexidade de utilização das linguagens de rede em nível de intenções (*intents*). No entanto, o paradigma IBN ainda não está totalmente integrado e amplamente implantando na maioria das redes de larga escala. Os operadores de rede ainda podem encontrar vários problemas ao implantar novos *intents*, tal como entender configurações complexas antes de escrever um novo *intent* para atualizar o estado da rede. Redes de larga escala podem incluir diversos dispositivos distribuídos em múltiplos níveis da hierarquia da topologia da rede; cada um desses dispositivos sendo configurado através de linguagens proprietárias e de baixo nível. Consequentemente, inferir *intents* de rede a partir dessas configurações de baixo nível pode ser uma tarefa árdua e consumir muito tempo. As soluções atuais que derivam representações de alto nível a partir de configurações feitas em baixo nível ainda não estão em nível de *intent*. Além disso, elas falham por não agregar detalhes para melhorar sua representação. Neste trabalho, nós apresentamos um processo *bottom-up* para extrair intents a partir das configurações de rede. Para validar nossa abordagem, nós desenvolvemos um sistema chamado SCRIBE (SeCuRity Intent-Based Extractor), que decompila as configurações de segurança de diferentes dispositivos e as traduz para uma linguagem em nível de *intent* chamada Nile. Para demonstrar a viabilidade do SCRIBE, nós utilizamos dois estudos de caso e avaliamos nossa abordagem com configurações de firewalls reais contendo regras de diversos servidores e instituições. Os resultados mostram que nossa solução pode representar configurações em nível de *intent* e ainda representar os aspectos de configuração de baixo nível mantendo uma boa de acurácia.

Palavras-chave: Redes de Computadores, Gerência de Redes, Redes Baseadas em Intenções, Intenções, Linguagens de Programação.

LIST OF ABBREVIATIONS AND ACRONYMS

ACL	Access Control List
BGP	Border Gateway Protocol
CSV	Comma-Separated Values
DMZ	DeMilitarized Zone
DTN	Data Transfer Node
EBNF	Extended Backus–Naur Form
HTTP	HyperText Transfer Protocol
HTTPS	HyperText Transfer Protocol Secure
IBN	Intent-Based Networking
ICMP	Internet Control Message Protocol
IDN	Intent-Driven Network
IP	Internet Protocol
ISP	Internet service provider
JSON	JavaScript Object Notation
MAC	Media Access Control
NAT	Network Address Translation
SSH	Secure SHell
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
WAN	Wide Area Network

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

In Intent-Based Networking (IBN), an intent is a high-level abstract declaration written by network operators to specify the desired network behavior (BEHRINGER et al., 2015). IBN, thus, helps operators configure the network by hiding unnecessary low-level details of the underlying network, inside an intent-aware network management system. In a recent effort, the employment of specification languages that are closer to natural languages has been exploited in the definition and deployment of network intents. Nile (JACOBS et al., 2018) and Jinjing (TIAN et al., 2019) are examples of such languages. Employing such solutions into the network can improve network management and also enable new features to be deployed easily. However, current efforts to support IBN focus on top-down approaches, meaning that network configurations are firstly specified as intents and then refined to low-level configuration directives — *i.e.*, the network is assumed to support intents already. Network operators may still encounter several issues deploying new intents, such as reasoning about complex configurations to understand previously deployed rules before writing an intent to update the network state.

1.1 Problem and Motivation

Network operators may encounter networks with missing documentation of previously possibly manually deployed intents, becoming hard to understand the network behavior. In such cases, especially in large-scale networks, operators must read many configuration files to understand the network intended behavior. Large-scale networks may include various devices distributed in multiple hierarchical levels of their topology, each of these devices configured using low-level, vendor-specific languages. Thus, current top-down approaches to specify intents are not straightforward to be implemented. Take, for instance, the configuration of firewall: an operator must manually check several low-level firewall rules to extract a simple intent, such as "*block p2p traffic*".

In a traditional network infrastructure, tasks of implementation, configuration, and troubleshooting require a technically skilled network operator provisioning and managing large multivendor networks (BENZEKKI; FERGOUGUI; ELALAOUI, 2016). Novel technologies arise as alternatives to provide network flexibility, reducing the complexity of network management, such as Software-Defined Networking (SDN) (FEAMSTER; REXFORD; ZEGURA, 2014). SDN provides a separation between the control plane

(which decides how to handle the traffic) from the data plane (which forwards traffic according to decisions that the control plane makes). In this way, operators may program the control plane and manage several devices in a centralized manner. However, according to Arezoumand et al. (2017), during years of continuous development and operation of the SAVI Testbed experiments, authors realize that the capabilities of programmable networks are not achievable for end-users, unless higher-level abstractions are provided. The authors are also convinced that intents can fill this gap by providing a simple yet expressive high-level abstraction over the network controller. The intent abstraction hides the unnecessary details of the underlying infrastructure from users and allows them to customize network configuration using human-readable intents.

There is a lack of solutions for interpreting low-level configuration directives and expressing them using intents. Some efforts, such as Batfish (FOGEL et al., 2015) and ARC (GEMBER-JACOBSON et al., 2016) read low-level configurations of traditional networks and generate abstract models of data and control planes. However, these solutions focus on the analysis and verification techniques, aiming to check properties of routing protocols, and do not provide a way to express the generated model in higher-level representations. In the network security context, many work efforts interpret the behavior of firewalls in the network by converting rules to high-level representations, such as tables (BODEI et al., 2018a), symbolic models (DIEKMANN et al., 2016), generic firewall languages (TONGAONKAR; INAMDAR; SEKAR, 2007), and graphs (MARTÍNEZ et al., 2012). These representations help operators understand firewall rules by capturing low-level configuration details in a bottom-up approach and displaying firewall rules in a vendor-independent form. However, these solutions show rules as allowing and blocking statements for IP addresses and transport ports, requiring operators to know the functioning of firewalls, protocols, and ports to understand the generated representation. Thus, we can observe a lack of solutions to help operators understand already deployed network rules in an intent-level. An intent-level representation can help operators by reducing the necessary effort to consolidate network rules and reason about previously deployed intents.

1.2 Main Goals and Contributions

Given the lack of solutions to express the behavior of already deployed network configurations in the form of intents, we propose a bottom-up method to represent net-

work configurations in intent-level. Our approach extracts intents from low-level network configuration files and translates them into Nile language. We developed a prototype tool called SCRIBE (SeCuRity Intent-Based Extractor) to prove the concept of our method. SCRIBE interacts with third-party tools that parse vendor-specific configurations and combines the output of these solutions with complementary information to display network rules in an intent-level language. Our system receives exported configuration files from various systems as input. SCRIBE then processes ACL and NAT rules and refines them by grouping related characteristics (*e.g.*, IP source and destination pairs, network services). Finally, we enrich intents with complementary information to translate to corresponding Nile intents to present to operators. However, the existing language constructs does not support NAT forwarding behavior. To express these constructions, we also extended the Nile grammar to support NAT actions.

We demonstrate the feasibility of our approach using case studies and an evaluation of the translation accuracy. The case studies represent realistic scenarios of a typical infrastructure of a small company and a high-speed campus network using science DMZ. With such case studies, we demonstrate that our solution is able to describe the network behavior using intents, closely resembling natural language, and represent accurately existing network configurations. We also evaluated SCRIBE with real datasets provided by (DIEKMANN et al., 2016), containing firewall rules from various servers and institutions. Our results show that SCRIBE is capable of expressing network configurations in an intent-level with high accuracy, capturing the majority of the low-level details present in the input configuration files.

1.3 Document Outline

The remainder of this dissertation thesis is structured as follows. In Chapter 2, we provide a background and review related work. In Chapter 3, we specify an end-to-end methodology to extract intents from network configurations. In Chapter 4, we provide details of the implementation of SCRIBE. In Chapter 5, we describe two case studies based on realistic scenarios and conduct an evaluation of our system using real-world configurations, as well as a discussion of our findings. Finally, in Chapter 6, we conclude this dissertation with discussions and future work.

2 BACKGROUND AND RELATED WORK

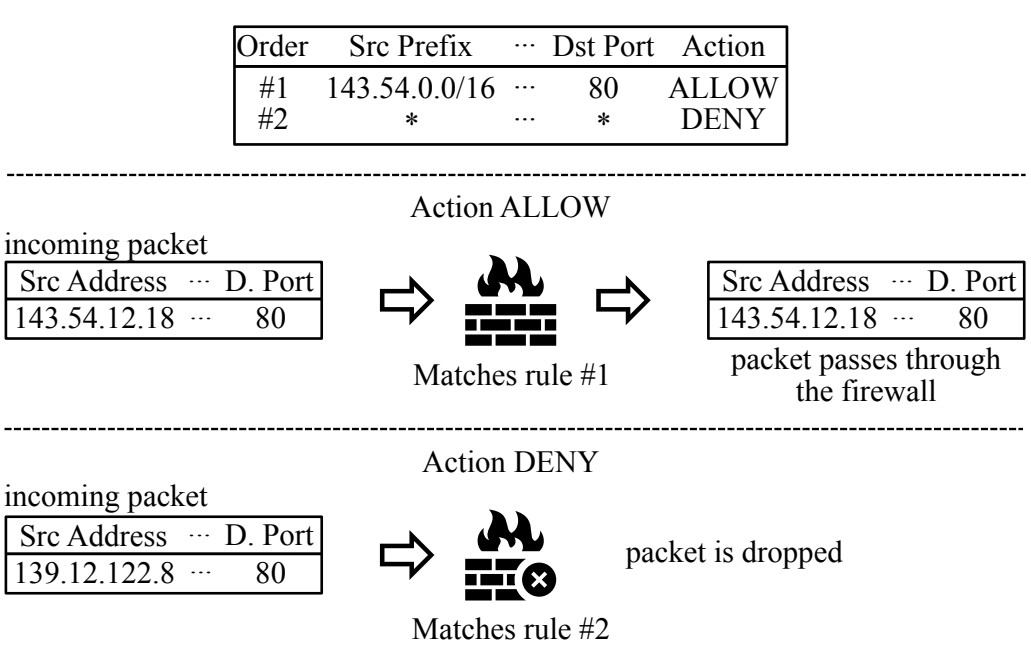
In this chapter, we present concepts laying the ground to the results of this dissertation. First, in Section 2.1, we provide a brief explanation of firewalls and Access Control Lists (ACL) concepts, as well as the complexity involved in managing these devices. Next, in Section 2.2, we explain the Network Address Translation (NAT) method and its use. We further present concepts and efforts for network configuration synthesis in Section 2.3, which give support to this work. In Section 2.4, we provide a background on intents and describe recent efforts to standardize this concept. Finally, in Section 2.5, we provide a discussion briefly listing the recent efforts that derive high-level representations from low-level configurations.

2.1 Firewalls and ACL

Firewalls are the main elements to ensure access control in networks (HACHANA; CUPPENS-BOULAHIA; CUPPENS, 2015). Firewalls use a set of rules that determines which packets can reach the different subnetworks and hosts, and how packets are modified or translated (BODEI et al., 2018b). Firewalls can be configured in various ways to guarantee network security, usually including several filtering rules. Filtering rules define the characteristics that a packet must hold to match a rule and the action that should be performed for matching packets. In the filtering context, firewalls operate by comparing the network traffic with this set of filtering rules. Therefore, firewalls compare incoming and outgoing packets with these rules sequentially until it reaches a final action like ALLOW or DENY, which will permit the packet to pass or discard it, respectively. Initially, a firewall is configured with a default action of ALLOW or DENY, meaning that if a packet matches none of the rules in the set, the default rule will be applied.

We illustrate an example of firewall ALLOW and DENY actions in Figure 2.1. In this figure, there are two firewall rules: one indicating an ALLOW action for packets with source prefix 143.54.0.0/16 and destination port 80 (HTTP), and another indicating a DENY action to all addresses and all ports. We use the wildcard '*' to denote that a field matches any possible value. In this context, the firewall will go through rules sequentially and accept the first incoming packet, indicated in Figure 2.1 by the Action ALLOW, which will match rule #1. Next, in an Action DENY, the firewall will discard the second packet, which does not match the source prefix in the first rule, then, going to

Figure 2.1: Firewall ALLOW and DENY actions



Source: Author

match the next rule (#2), which will discard the packet.

Besides firewall, ACL (Access Control List) rules can be implemented directly in network devices, such as routers, to packet filtering. ACL is a collection of "permit" and "deny" conditions and the packets are permitted (or denied) based on their source and destination IP addresses, ports, and protocol (TIAN et al., 2019). Packets that come into a network device that contains ACL rules are compared to the set of rules sequentially until it has a match. If there are no matches, a packet is denied or permitted according to the default rule.

The configuration of a firewall is typically composed of numerous rules, and generally it is hard for an operator to figure out the overall behavior of a firewall with too many rules (BODEI et al., 2018b). On enterprise networks, the complexity of management in firewalls increases because it can count with multiple firewalls deployed in the same network (AL-SHAER; HAMED, 2004). A study (WOOL, 2010) concludes that corporate firewalls are poorly configured, and the complexity of a ruleset is positively correlated with the number of detected configuration errors. Also, because of constant changes in firewall rules, many of them can be redundant or conflicting, preventing them from being actually triggered (ABEDIN et al., 2010), making appear various types of anomalies, such as shadowing and redundancy (AL-SHAER; HAMED, 2004). Network

administrators often appeal to policy refactoring to solve these issues and obtain a minimal and clean set of configurations (BODEI et al., 2018b).

Several approaches have been made to facilitate the management of firewall rules. In the verification context, efforts have been made to analyze the set of already deployed firewall rules to find inconsistencies and also guarantee a minimal set of firewall rules, eliminating the redundancy between them (JAYARAMAN et al., 2014). From a different perspective, other efforts are focusing on displaying the firewall rules in higher-level and vendor-independent representations, such as graphs or tables.

In previous efforts (TONGAONKAR; INAMDAR; SEKAR, 2007) (MARTÍNEZ et al., 2012), the authors proposed a solution to capture low-level firewall rules and represent them in a platform-independent representation. In (TONGAONKAR; INAMDAR; SEKAR, 2007), the authors obtain firewall rules and express them in a platform-independent language. In another work (MARTÍNEZ et al., 2012), authors proposed a solution to read vendor-specific firewall rules and represent them as a directed graph abstraction, where nodes correspond to network devices. Directional edges represent a statement to allow or block traffic on a specific port. These solutions can help operators understand firewall rules by displaying the rules in a platform-independent representation. However, the rules are not displayed in higher-level representations, requiring users to understand firewalls, protocols, and other low-level details of the network. Also, in these solutions, there is no support for distributed firewall configurations.

2.2 Network Address Translation (NAT)

Network Address Translation (NAT) is specified by RFC 3022 (SRISURESH; EGEVANG, 2001). Initially designed to solve the scarcity of IPv4 addresses, the concept following NAT is for the ISP (Internet Service Provider) to assign each home or business a single or very few IP addresses for Internet traffic. With NAT, computers on a network (internal to a company) started to use ranges of IP addresses that have been declared as private. In an internal network, multiple devices connect using private IP addresses, without using public IP addresses. Private addresses are designed to be used internally and are inaccessible outside of the network. The range of private addresses is defined by RFC 5735 (COTTON; MOSKOWITZ; VEGODA, 2010) and is shown in Table 2.1.

NAT allows hosts in a private network to transparently access the external network and enable access to selective local hosts from the outside (SRISURESH; EGEVANG,

2001). For this, NAT offers methods to map IP addresses from one group of private addresses to public addresses that are visible on the Internet. The NAT box is the device responsible for the mapping between private addresses and public addresses. In a typical network, the NAT box is physically coupled with a router or a firewall. When there is a request for an external address, the NAT box generates a new source port number and creates a new entry with it on the NAT translation table (KUROSE; ROSS, 2012). Table entries include NAT source and destination addresses and ports. The NAT method can select a source port that is not currently in the NAT translation table, and then it adds a new entry.

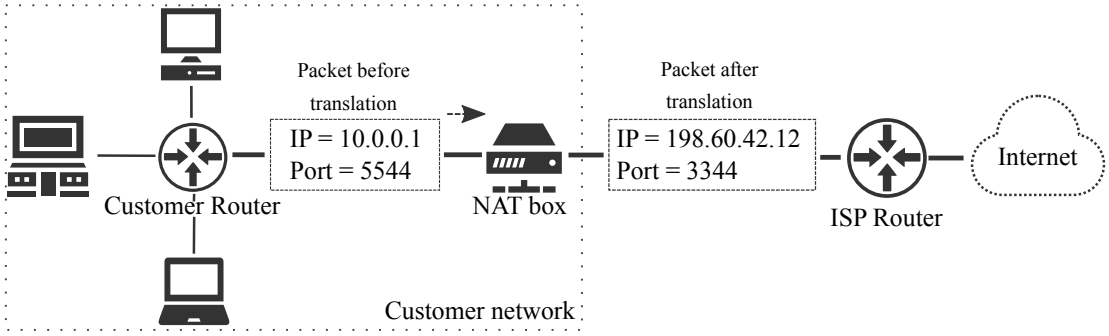
Table 2.1: Range of private IP addresses

Range	**Prefix**
10.0.0.0 - 10.255.255.255	10/8 prefix
172.16.0.0 - 172.31.255.255	172.16/12 prefix
192.168.0.0 - 192.168.255.255	192.168/16 prefix

Source: Adapted from (REKHTER et al., 1996)

An example of a NAT operation is shown in Figure 2.2. This figure shows an outgoing request from an internal host with IP address 10.0.0.1 to an address external to the network, using the NAT method. In this request, the outgoing packet passes through a NAT box before leaving the customer network. The packet has, initially, the source IP address 10.0.0.1. The NAT box maps the internal IP source address 10.0.0.1 for a public IP address 198.60.42.12, and the internal port 5544 to an externally visible port 3344. The NAT box generates a new entry containing NAT internal address, the public IP address, internal port, and the externally visible port. Next, the NAT box puts this new entry in the NAT translation table, and, in this example, it rewrites the source address to the IP address 198.60.42.12 and port to 3344 before the packet leaves the customer LAN.

Figure 2.2: NAT Example



Source: Adapted from (TANENBAUM; WETHERALL, 2011)

The NAT method also improves the network's security because an external ad-

dress cannot directly access an internal address of the network. For this, NAT actions are used together with firewall filtering in firewall implementations such as `iptables` (REDHAT, 2005b). Adopting a firewall along with a NAT, it is possible to maintain a public server externally not accessible with an internal network address (private) for security reasons. The use of NAT allows redirecting to this server only allowed traffic, such as HTTP and HTTPS, for instance. As an example, we can configure a NAT box to redirect all incoming traffic on ports 80 and 443 (for HTTP and HTTPS traffic, respectively) to a private-addressed server, and in this way, automatically discard all other unallowed incoming traffic.

2.3 Network configuration synthesis

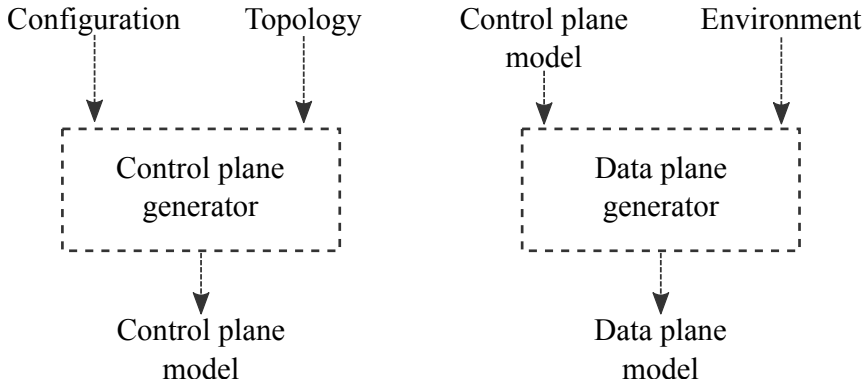
Configuration is an essential part of network management. It defines how devices run protocols, communicate with each other, and integrate within the network (NARAIN; TALPADE; LEVIN, 2010). Maintaining a network with various devices running protocols and communicating with an adequate level of trustworthiness and security is a challenging task. For an operator, it is complicated to reason about policies that affect the network globally because of the network configuration is generally made at a device-level. Hence, it may occur gaps between the operators' desired policy and the configuration that was deployed in the network. In today's networks, the gap between end-to-end requirements and configurations is manually bridged (BECKETT et al., 2016). In this way, the network is more susceptible to configuration errors, which leads to adverse effects on security, stability, and increases the maintenance and deployment costs.

Previous efforts (QUOITIN; UHLIG, 2005) (NELSON et al., 2010) focus on developing approaches to detect network configuration errors by inspecting network configuration files directly. In such works, the authors proposed approaches to find errors statically and proactively before a new configuration is applied to the network. However, real network configurations are complex and include many interacting aspects (BENSON; AKELLA; MALTZ, 2009). Because of the complexity, these approaches are limited to developing customized models for specific aspects or verifying the correctness of specific properties, such as reachability (FOGEL et al., 2015).

Recent efforts, however, can more precisely generate a logical model that reproduces the characteristics and interacting aspects from the given input configurations (FOGEL et al., 2015). These efforts can efficiently reproduce the low-level aspects of config-

urations in higher-level platform-independent representations, such as abstract topologies (BECKETT et al., 2017), abstract model of control and data planes (FOGEL et al., 2015) or a tabular representation (BODEI et al., 2018a). As an instance, Batfish (FOGEL et al., 2015) can reproduce the data and plane models. This solution, first, takes as input configuration files and generates the control plane model. After, Batfish (FOGEL et al., 2015) generates the data plane, which takes as input the control plane model generated in the previous step and the environment, which consists of the up/down status of each link in the network and a set of route announcements (for BGP synthesis). We show the process of generation of control and data planes on Batfish (FOGEL et al., 2015) in Figure 2.3.

Figure 2.3: Bafish generation of control and data planes



Source: Adapted from (FOGEL et al., 2015)

The need for abstract models arises from the fact that debugging raw configurations would require different frameworks for each configuration, which becomes unfeasible and cost expensive. With the model generated by Batfish (FOGEL et al., 2015), operators can find errors and guarantee the correctness of planned or current network configurations of traditional routers (*e.g.*, Cisco IOS, Juniper JunOS). The system allows the analysis of forwarding decisions and error checking, for instance, the absence of black holes or loops by simulating the data and control plane behavior.

In firewall and NAT contexts, there are efforts to derive high-level and platform-independent models. Previous work efforts (TONGAONKAR; INAMDAR; SEKAR, 2007) (MARTÍNEZ et al., 2012) provide a high-level representation of the filtering behavior in firewalls. More recently, FWS (BODEI et al., 2018a) provides a formal characterization of firewall and NAT configuration. The process of FWS uses first-order logic predicates that determine which packets will be accepted by the firewall and supports the NAT redirect behavior. The final result is a tabular representation of firewall and NAT rules in the form of *flattened* rules – *i.e.*, an unordered representation of firewall rules.

2.4 Intents

Intents are high-level abstract declarations that dictate how the network is expected to behave (SCHEID et al., 2017). In an Intent-Based Networking (IBN), operators should be able to specify the desired network behavior using high-level policies, without worrying about how it would be achieved (JACOBS et al., 2018). As an example, an operator can express a simple intent, such as *"block p2p traffic"*, without knowing the network context, as well as languages in which this action can be performed. Intents can assist both novice and experienced operators to deploy network-wide configurations, not requiring to think in a device-level configuration. There are research efforts concentrated on creating intent-level languages, aiming to include intents in network configurations.

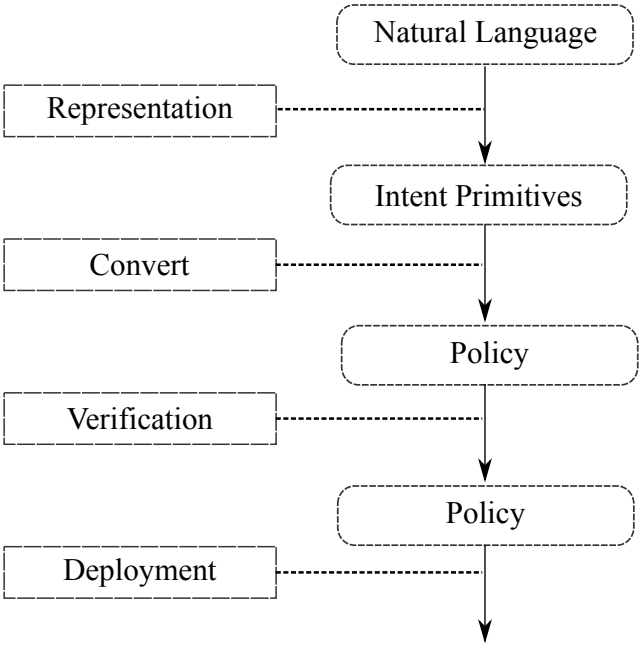
The international community is working to standardize definitions and concepts of intents. RFC 7575 (BEHRINGER et al., 2015) sets definitions and design goals for autonomic networks, and intents are included. In this document, intents are defined as "An abstract, high-level policy used to operate the network". RFC 7575 (BEHRINGER et al., 2015) also sets as a design goal that intents are defined at a level of abstraction that is much higher than typical configuration parameters. According to RFC 7575, intents must not convey low-level commands or concepts, since those are on a different abstraction level. A more precise definition of intent and Intent-Based Networking (IBN) is in a recent draft (CLEMM et al., 2019), which specifies intents as operational goals that a network should meet in a declarative way. In this manner, intents do not describe the steps to achieve these goals.

Recent efforts were made to define an architecture for Intent-Based Networking (IBN), also named Intent-Driven Network (IDN). However, proposed architectures differ from one research group to another (PANG et al., 2020). In terms of execution order, most of IBN solutions follows the intent representation, convert, verification, and deployment, as shown in Figure 2.4. Operators interact with applications using intents, or natural language, and the network must understand the inputted intent and represent it into a specific expression, using intent primitives. Intent primitives, therefore, are converted into advanced policies for network execution. In the next step, following the intent flow, verification methods can be used to guarantee the policy's reliability. Finally, the last step involves the deployment of the generated policy into the network.

Recently, several research efforts are focusing on creating intent-level languages. These languages allow operators to express intents at a very high-level of abstraction;

some are closer to the natural language. In this way, the operator can take away low-level details that increase the complexity of the deployment of network intents. These intent-level languages are top-down efforts, meaning that the operator expresses the intent at a high-level. Based on operators' desired intent, the solution applies a refinement process to generate low-level configurations for the network devices and then deploys these configurations.

Figure 2.4: Intent Flow



Source: Adapted from (PANG et al., 2020)

In a recent effort (JACOBS et al., 2018), authors provide a solution to allow operators to express intents in natural language. Authors also propose Nile, a high-level, comprehensive intent definition language, closely resembling the English language. The Nile language reduces the need for operators to learn a new policy language for each different type of network. Operators can interact with the system through a chatbot interface, and a neural network learning model is used to extract the user's intent, in a process called refinement. In the process of intent refinement, the system introduces a feedback mechanism that displays the user intent in the Nile language, and the operator can check whether the intent expresses their desired behavior.

Jinjing (TIAN et al., 2019) allows operators to declare update intents (*e.g.*, ACL migration) in a declarative language called LAI, and automatically synthesizes ACL update plans that meet their intents. Jinjing modeled the ACL configuration formally and designed an intent primitive to ensure the accuracy of the system operation. However, the

system is limited to the ACL update plans and designed for WANs. The solution cannot be applied to other networks.

In a recent effort (BIRKNER et al., 2018), the authors propose a bottom-up solution to summarize the forwarding behavior in natural language, called Net2Text. This solution provides interaction through a "chatbot", where users can query for a specific forwarding behavior and the system replies with the forwarding state for the specified query. While this work does an excellent job in summarizing the network-wide forwarding state, producing succinct reports in natural language, it is limited to forwarding behavior, not supporting ACL and NAT, for instance.

2.5 Discussion

There is a lack of solutions for representing already deployed configurations as intents. Bottom-up solutions like Batfish (FOGEL et al., 2015) and FWS (BODEI et al., 2018a) parses low-level configuration directives and represent them in a vendor-independent model. Despite using models, these solutions do not aim to express configurations in an intent-level representation. Batfish, however, aims to use the model to analyze network properties (*e.g.*, reachability) by finding inconsistencies in the deployed configurations.

For the firewall and NAT context, the solution FWS (BODEI et al., 2018a) parses firewall configurations from different systems and synthesizes them in a tabular form. To this end, authors provide a language to interact with FWS where the user can load vendor-specific rules from different firewalls and queries for specific IP addresses, ports, and networks, similar to SQL in databases. The tabular representation helps operators understand firewall rules by centralizing all configurations in a single place. However, the tabular representation requires users' knowledge of firewall tables and NAT. Also, for distributed firewall configurations, this solution loads each device configuration separated in a different table, each representing the firewall rules of a single device. Due to the use of various tables, operators can have difficulties reasoning about previously deployed intents in a network-wide context.

In the context of bottom-up approaches, there is scarse research that expresses the behavior of the network using intent-level languages. The Net2Text (BIRKNER et al., 2018), for instance, provides summaries of the forwarding behavior. However, it is designed to support only forwarding behavior. Anime (KHERADMAND, 2020), on the

other hand, infer high-level intents from the provided configuration. Anime is a framework to infer high-level intents by mining the commonalities among the forwarding behavior in the network. Although Anime can infer intents, its behavior is non-deterministic, and the context is limited to forwarding behavior. Other bottom-up approaches mentioned before are not able to represent configurations in a level of intents.

On the other hand, intents' research efforts are focused on creating intent-level languages, such as Nile (JACOBS et al., 2018) and Jinjing (TIAN et al., 2019). Using these solutions, operators then dictate the behavior of the network through intents. However, these solutions do not provide a mechanism capable of translating already deployed low-level configurations into an intent-level language. Our work arises from this lack of solutions to display existing configurations in an intent-level representation. We take advantage of previous solutions that parse low-level configurations and create an intermediate representation, such as FWS. FWS (BODEI et al., 2018a), for instance, is a solution that parses low-level configuration directives and represents them in a vendor-independent tabular model. We use the output of the synthesis solutions (e.g., FWS) as a base to create an above layer between these solutions and the user. At the final of this process, the configurations will be displayed in an intent-level language.

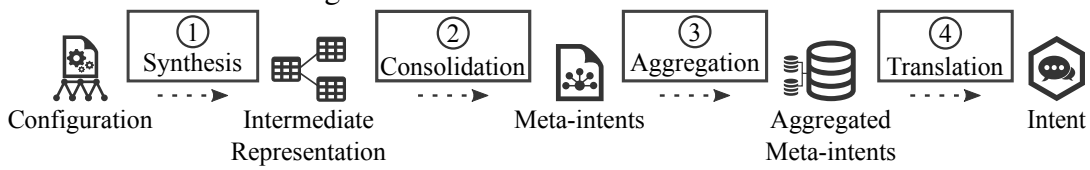
3 SOLUTION OVERVIEW

In this chapter, we describe a bottom-up solution to parse network configurations and represent them into an intent-level language. The process of extracting intents from low-level configuration directives is detailed in Section 3.1. In Section 3.2, we describe the specialization process for firewall and NAT. Finally, in Section 3.3, we detail the modifications needed to extend the Nile language to represent NAT forward actions.

3.1 Process

The process of intent extraction is divided into four steps, as depicted in Figure 3.1. The process expects configuration files exported from network devices (*e.g.*, firewalls, NAT, routers) as input. Having these configuration files, in step (1), they are stored in a centralized database and, posteriorly, synthesized. During step (2), entities that represent parts of the configuration are exported and gathered to generate a platform-independent abstract representation, called meta-intent. In step (3), meta-intents are enriched with complementary information provided by various sources. The final result of this step is an enriched representation of the network configurations called aggregated meta-intents. Finally, in step (4), the aggregated meta-intents are translated into our extended version of the Nile language, described in Section 3.3.

Figure 3.1: Process of intent extraction



Source: Author

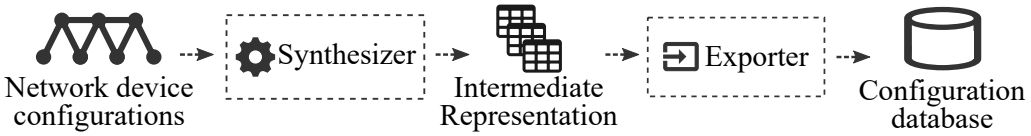
Configuration synthesis

In the first step of the process (Figure 3.1, step (1)), low-level configuration directives are collected from various configuration files, which can be distributed in multiple levels of the network hierarchy. In this step, the solution expects as input configurations exported from various devices. After, all exported configuration files are gathered along with the network topology and aliases. Each configuration file consists of various

rules and parameters. For instance, a rule may be composed of source and destination IP addresses, network ports, forwarding rules, and stateful elements. Given that network devices can be distributed in multiple levels of the hierarchy, the acquisition of rules is performed envisioning the future exportation, maintaining all configurations centralized in the same database.

The synthesis step is performed by a Synthesizer and an Exporter, as shown in Figure 3.2. The Synthesizer is responsible for collecting and gathering configuration files and synthesize configurations into an intermediate representation. The synthesis must be low-level and detailed in order to create a trustworthy representation that can reproduce all aspects of configuration and their interactions. We leverage information from multiple sources in the network and exploit existing literature solutions to extract such information in a structured format. As the focus of this work is not to construct a parser for each vendor-specific language, we use third-party synthesizers as a part of the solution. However, since there is no globally accepted single solution, our solution must be able to parse multiple input sources.

Figure 3.2: Configuration synthesis



Source: Author

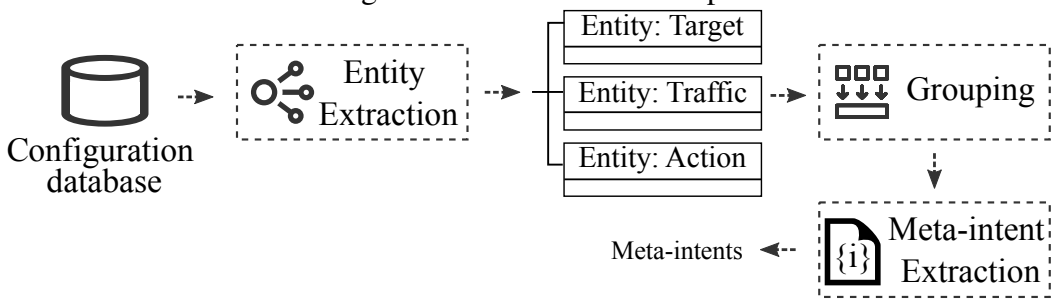
The last step of synthesis involves collecting and putting the configurations generated by the synthesizers in a centralized database. This process is made by the Exporter component that receives configurations that are the output of the Synthesizer and export them in a centralized configuration database.

Consolidation

With the representation in an intermediate format, the process starts to raise the network model's abstraction level. The purpose of the consolidation step is to (*i*) extract from the intermediate representation higher-level entities that represent parts of the configuration, (*ii*) grouping these entities by similar characteristics, and (*iii*) generating the meta-intent representation. Figure 3.3 indicates the pipeline of this step of the process. The Entity Extraction is responsible for loading configurations from the database and generating network entities. The purpose of the Grouping step is to compact the set of rules.

Finally, the meta-intent Exporter generates the output in meta-intent format.

Figure 3.3: Consolidation step



Source: Author

The consolidation step begins with entity extraction. A configuration is defined as a composition of entities, and it is divided into three parts. Each part of the configuration is represented by a different type of entity, which can be marked as Target, Traffic, or Action. Formally, we define an entity as $E = \{Target, Traffic, Action\}$, where:

- $Target$: represents the target object of a determined rule. A Target can be specialized to represent network target objects in various scenarios. A target can be composed of a set of devices in a traditional network where the rule will be applied. For instance, in a firewall context, a rule for blocking traffic contains the source and destination of the traffic that will be parameters belonging to a Target. The source and destination prefixes can be modelled by $Target :=< src.prefix, dst.prefix >$.
- $Traffic$: represents the type of traffic in the network. The type of traffic can differentiate between network services. For instance, service types can be distinguished by TCP and UDP protocol and the port used by each service. For example, in firewall and NAT contexts, the source and destination ports and the protocol (TCP or UDP) compose the Traffic entity for filtering traffic, which will discern the traffic type in the network.
- $Action$: represents an operation to be executed for a pair $< Target, Traffic >$. Operations consist of a behavior belonging to a network function, such as a forward behavior in a NAT box or a block rule in a firewall. Each action contains the needed parameters to run the desired operation. Then $P = (p_1, p_2, ..., p_n)$ is the set of parameters of an Action. As an example, a NAT forward operation can be represented by $Action = \{\texttt{forward}\}$ and include the parameters $p_1 = \texttt{DNAT}_{\texttt{IP}}$ and $p_2 = \texttt{DNAT}_{\texttt{Port}}$, indicating the destination NAT IP and port for a specified traffic.

Finally, to group the entities that produce the same intent, an additional processing phase, called grouping, is performed. In this phase, the solution searches through

the extracted configurations and match pairs of entities by common characteristics using grouping functions. We define grouping as a function of entities $g : E \times E \mapsto E$. The grouping is performed as one-level grouping and not consider the type of the entities involved in this process.

The grouping process is defined by set operations. Formally, each rule is represented by a set S of entities containing network properties, each of them represented by p_i. Given a set S, we extract a property p_i as a varying parameter v_i and a subset G containing the $length(S) - 1$ remaining properties. For a given set $S = \{p_1, p_2, ..., p_n\}$, where n is the set size, we attribute to each v_i a property of set S, as instance $v_1 = p_1$, $v_2 = p_2$, ..., and $v_n = p_n$. For each v_i, a superset V is created, containing the property v_i and the subset $G = \{S \setminus v_i\}$. In this way, $V_1 = \{v_1, S \setminus v_1\}$, $V_2 = \{v_2, S \setminus v_2\}$, ..., $V_n = \{v_n, S \setminus v_n\}$.

The grouping process consists of going through all the generated sets of V, grouping entities V with similar G subsets, and gathering them by evidencing the varying parameter v_i. The sets G for each V set are compared and grouped if they are equal. Take V_i and V_j as an instance. If the subset $G_i \in V_i = G_j \in V_j$, they are grouped in a new set I and include in the set the varying parameter, so the final grouped set is $I = \{G_i, v_i, v_j\}$.

The last step of the consolidation process is to generate the meta-intents. Meta-intents is composed of pairs of information, such as origin, destination, services, and the default action for a given rule in a well-structured format. Meta-intents are a more high-level representation of our process, much closer to an intent, and they are represented in JSON format. After grouping the sets, the Meta-intent Extractor will traverse the sets and map the values of V_i sets to an intermediate representation called Meta-intent. The meta-intent is a JSON representation that maps all values from sets in the form `{"Entity.Property" : "value"}`. For each $p_i \in E_i$, where p_i is a property and E_i an entity, the meta-intent extractor will map to an entry `{"`E_i`.`p_i`":"value"}`.

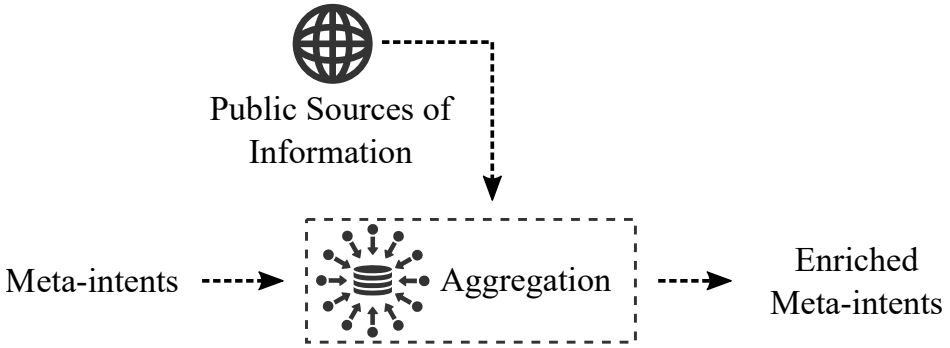
Aggregation

In the previous step, meta-intents were generated as an intermediate structured representation. However, meta-intents contains a bit of low-level information, such as IP addresses. In the third step of the process, all meta-intents generated in the consolidation step are gathered and enriched with complementary information from various sources. This step aims to display the information at a user-friendly level.

The solution collects hostnames and topology data as a piece of additional information from the network to enrich the meta-intents, featuring an option to input “friendly" names for hosts that aim to facilitate the recognizing of network devices. With all the complementary information gathered, the IP addresses are replaced with the provided names. After that, a query to the Service Name and Transport Protocol Port Number Registry[1] assist in inferring the service or protocol names according to the provided ports. This service lists the standard port for services.

This step of process aggregates the meta-intent representation with complementary information obtained from various sources, enriching the rules with additional information. The aggregation process is shown in Figure 3.4. The solution queries for complementary information to each meta-intent entry, representing a network property. For instance, if a network property represents an IP address, the aggregation process tries to discover the host with this IP address. The final result of this step is a meta-intent enriched with complementary information.

Figure 3.4: Aggregation process



Source: Author

In this step, the complementary information databases are queried for each piece of low-level information of the meta-intents. Such information is mapped to a piece of corresponding low-level information in the meta-intents and posteriorly replaces the low-level information with complementary information. If this process fails due to missing information, a bit of low-level unprocessed data may be present in the output intents.

[1] <https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml>

Intent translation

After enriching meta-intents with complementary information, the next step is to process them to translate the resulting intents to the Nile language. The Nile language, which is the target language for this step, is in an intent-level language, closely resembling the natural language. Thus, it turns easy for operators to understand the most common operations currently supported by networks.

The Nile grammar has several symbols to represent network rules, and most of the enriched intents can be directly translated to Nile symbols. Initially, the *Target* entity is mapped to Nile ⟨from_to⟩ symbol. The Nile ⟨from_to⟩ expect two *endpoints* as parameters. The arguments on *Target* entity are mapped to generate two Nile ⟨endpoints⟩, representing *source* and *destination*.

The second phase of Nile translation process involves the mapping of *traffic* entities to Nile ⟨matches⟩ symbol. Nile has support for service, traffic, protocol, and group traffic types. In this work, the Nile language was extended to allow it to represent addresses utilized to express the entity *traffic* in different levels of abstraction. In the last phase of the translation process, the representation of *Action* entities are accomplished using Nile ⟨rules⟩. The grammar symbol ⟨rules⟩ permits the representation of actions, such as allow and block, and ⟨middleboxes⟩ allowing the representation of several network functionalities supported by different types of middleboxes. In this work, we also extend the Nile language to support an operation forward, allowing the representation of NAT forwarding actions, as described in Section 4.

3.2 Specializations

Specializations allow applying the solution to specific contexts. In this section, we describe the specializations for Firewall and NAT.

Firewall

This specialization specifies how the solution can be applied to the firewall context. The firewall specialization contains functions that receive input configurations in a tabular form. For the firewall specialization, is expected as input a tabular representation

of firewall rules containing the fields *source* and *destination* for Target entity, *port* and *protocol* (TCP or UDP) for Traffic entity and an Action, which contains a function to *allow* or *deny* a packet. Given a tuple (*Target.source*, *Target.destination*, *Traffic.protocol*, *Traffic.port*) the firewall will *allow* or *deny* a packet based on the Target and Traffic fields.

The synthesis solutions output these fields in the tabular format. As this work's focus is not to generate a firewall model for each different vendor, a third-party solution was used to generate the tabular representation. To the firewall specialization, FWS (BODEI et al., 2018a) is chosen as the base solution, satisfying our intermediate representation requirements. FWS synthesizes vendor-specific firewall rules to a tabular representation. The firewall specialization follows our process's four steps to transform tabular intermediate representation into the intent-defined Nile language. After that, in this firewall specialization, the process proceeds with the Consolidation step, described in Section 3.1.

To the next step, the aggregation, the rules are divided into three categories: internal, incoming traffic, and outgoing traffic, as shown in Table 3.1. The traffic is considered internal when the source and destination addresses are inside the network prefix; for instance, source and destination addresses are included in a prefix 10.0.0.0/8. Traffic is classified as incoming when the source address is not part of the internal network, and the destination address is part of the internal network. Lastly, outgoing traffic refers to the situations where the source address is in the internal network, and the destination address is not. This classification does not consider packets created inside the network with a (possible) external source address.

Table 3.1: Traffic classification by source and destination

Source	**Destination**	**Classification**
external	internal	incoming traffic
internal	external	outgoing traffic
internal	internal	internal traffic

Source: Author

The first step of specialization for a firewall is to name the origin and destination of the target. In this step, IP addresses and prefixes are changed to a hostname or other name provided by the user. These targets denote where the traffic is input, output, or internal. The wildcard "*" denotes any address, which in firewall means a rule that matches all addresses. When this wildcard appears, it is assumed that the traffic is from the Internet, if it is incoming traffic or to the Internet, if it is outgoing traffic.

Initially, to define the source and destination addresses as friendly names, the process verifies if the source or destination is marked as any (wildcard '*') and change it to

"Internet", denoting external traffic in a friendly way. If not, the IP addresses are renamed according to the friendly names, if they are available. The function `rename` attributes friendly names to an IP address or a prefix. The rename function receives as a parameter an IP address or prefix and searches in the database for aliases to the address. For example, if a firewall rule has the source address any (‘*’) and destination 192.168.0.2, which is internally known as "Web Server", the algorithm will change the origin wildcard "*" to "Internet" and the address 192.168.0.2 to "Web Server".

NAT

This specialization specifies how the solution can be applied to the NAT context. The NAT specialization contains functions that receive input configurations in a tabular form. For the NAT specialization, is expected as input a tabular representation of NAT rules containing the fields *source* and *destination* for Target entity, *port* and *protocol* (TCP or UDP) for Traffic entity and an Action, which contains a function to *forward* a packet, and a parameter of *sourceNAT* or *destinationNAT*. Given a tuple (*Target.source*, *Target.destination*, *Traffic.protocol*, *Traffic.port*) the NAT will *forward* a packet based on these characteristics, considering the parameter *sourceNAT* or *destinationNAT*.

The synthesis step generates a tabular representation for NAT. As this work’s focus is not to generate a NAT model for each vendor, a third-party solution was used to generate the tabular representation. To the NAT specialization, FWS (BODEI et al., 2018a) is chosen as the base solution to generate the tabular representation. With the tabular representation, the first step is to classify the traffic. The NAT traffic classification follows a traffic classification algorithm, similar to the Firewall, provided in Table 3.1. However, in NAT, for incoming traffic, the final destination is inaccessible directly to the origin. Then, the traffic destination is the NAT box. In the NAT box, the destination will be changed to the correct destination, an internal address. The algorithm that defines the origin and destination for incoming traffic is presented in Algorithm 3.1. After that, we proceed with the Consolidation step, described in Section 3.1.

Algorithm 3.1: Define targets for NAT incoming traffic

```
1 if target.srcIP = '*' then
2 |   target.src ← "Internet";
3 else
4 |   target.src ← rename(target.srcIp);
5 end
6 target.dst ← NATBox;
7 target.NATdst ← rename(target.NATdstIp);
```

Source: Author

For outgoing traffic, first, the source address is processed. Initially, the NAT source address (*target.NATsrc*) is renamed to a familiar name. After, the target source (*target.src*) is set to the NAT Box address. Finally, the destination IP address is verified to set a friendly name. The algorithm that defines the origin and destination for NAT outgoing traffic is presented in Algorithm 3.2.

Algorithm 3.2: Define targets for outgoing NAT traffic

```
1 target.NATsrc ← rename(target.NATsrcIp);
2 target.src ← NATBox;
3 if target.dstIp = '*' then
4 |   target.dst ← "Internet";
5 else
6 |   target.dst ← rename(target.dstIp);
7 end
```

Source: Author

3.3 Extending Nile

Nile was chosen as the default language to represent the extracted intents. The Nile language arises as a need for an abstraction layer between natural language intentions and lower-level languages (JACOBS et al., 2018). Nile is an intermediate representation that closely resembles the natural language. The Nile language also can act as an abstraction layer for other policy mechanisms, reducing the need for operators to learn multiple policy languages for each different type of network (JACOBS et al., 2018). The Nile language has a high expressiveness and can faithfully represent the operator's intention.

For instance, an intent such as "*Allow incoming traffic from the Internet to Web Server for SSH protocol*" can be expressed in Nile in a simple syntax, as shown in Listing 3.1.

Listing 3.1: Example of allow intent in Nile

```
define intent firewallIntent:
        from endpoint ('Internet')
        to endpoint ('Web Server')
        allow protocol ('SSH')
```

Source: Author

The Nile language offers constructs that are capable of representing various network components and actions. Its grammar constructs give the Nile language representation capabilities. The entire grammar of the Nile language, with all constructs, in EBNF notation, is shown in Listing 3.2.

The Nile language can express user intentions in an intent-level representation that is close to natural language and satisfies our readability and abstraction requirements. However, Nile natively does not support NAT behavior. The Nile grammar was extended with NAT constructs to allow the representation of NAT behavior, introducing NAT capabilities to the Nile language.

The Nile extension begins with the inclusion of the `forward` keyword, which adds support for fully functional NAT forward behavior in the Nile language. The NAT forward behavior requires information of the source address or hostname, source port, and destination port. It is defined by a redirect from incoming traffic in a specific port to a specified address in a specific destination port – the protocol can be inferred from source and destination ports. For instance, incoming traffic in SSH protocol can be abstracted to "traffic on port 22". The representation of NAT port preservation is facilitated in the syntax by allowing to omit the destination port. In this case, the destination port is the same as the source port.

The `forward` keyword was added to the Nile language to represent a forward NAT action. A Nile forward action requires a ⟨target⟩ symbol. Nevertheless, the actual Nile ⟨target⟩ symbol allows the representation only of groups, services, endpoints, and traffic. It can not represent a pair of IP address and port as a target for operation. To overcome the limitation of ⟨target⟩, a new symbol ⟨address⟩ was created to express a pair of IP addresses or hosts and a port. The syntax of the address symbol is displayed in Listing 3.3.

Listing 3.2: Nile Grammar

```
<intent> ::= `define intent' <term>`:' <operations>
<operations> ::= <path> <operation> { ' ' <operation> }
<path>   ::= [<from_to>  |  <targets>]
<from_to> ::= `from' <endpoint> `to' <endpoint>
<operation> ::= (<middleboxes>
              |  <qos>
              |  <rules>)+ [<interval>]
<middleboxes> ::= [add | remove] <middlebox> { (`,'
               |  `, \n') <middlebox> }
<middlebox> ::= `middlebox('<term>')'
<qos> ::=  [set | unset] <metrics>
<metrics> ::= <metric> { (`,'
           |  `, \n') <metric>}
<metric> ::= [bandwidth | quota]('[`max' | `min' ]`,' <term>`)'
<rules> ::= [allow | block] <matches> [<matches>]
<targets> ::= `for' <target> { (`,' | `, \n') <target> }
<target> ::= [ <group> | <service> | <endpoint> | <traffic> ]
<matches> ::= [ <service>
           |  <traffic>
           |  <protocol>]
<endpoint> ::= `endpoint('<term>`)'
<group> ::= `group('<term>`)'
<service> ::= `service('<term>`)'
<traffic> ::= `traffic('<term>`)'
<protocol> ::= `protocol('<term>`)'
<date_time> ::= `datetime('<term>`)'
              |  `date('<term>`)'
              |  `hour('<term>`)'
<term> ::= [a-z0-9]+
```

Source: Adapted from (JACOBS et al., 2018)

With the forward keyword, plus address symbol, NAT actions can be represented in the Nile language. The ⟨address⟩ symbol is composed of a ⟨addrterm⟩ and ⟨port⟩ terms, as indicated in Listing 3.3. The ⟨addrterm⟩ symbol represents a target source or destination, which is a host or IP address. The symbol ⟨host⟩ can be composed only of valid characters for a hostname, and the symbol ⟨ip⟩ represents grammatically valid IP addresses.

Listing 3.3: Directives for NAT forwarding behavior in Nile Grammar

```
<forward> ::= `forward' <address>
<address> ::= `address (' <addrterm> `)'
            | `address (' <addrterm> `, ' <port> `)'
<addrterm> ::= <ip>
             | <host>
<middlebox> ::= `middlebox('<term>`)'
<ip> ::= ([0-9]\{1,3\}.)\{3\}[0-9]\{1,3\}
<host> ::= [a-zA-Z]\{1\}[a-zA-Z0-9\-\_]*
<port> ::= [0-9]\{1,5\}
```

Source: Author

The syntax of a forward action also requires two endpoints, representing the source and destination hostnames or IP addresses, and a service or a protocol (TCP or UDP) and port. For a traditional NAT forward operation for incoming traffic, the source will be an external address. The destination will represent the NAT box, which will forward the traffic for a given address and port, expressed by symbol ⟨address⟩.

Let us suppose a scenario containing a "NAT Device" at the edge of the network, managing all input traffic. Internally, there is also an "SSH Server" with IP address 10.0.0.5, listening for SSH traffic on port 2222. In this scenario, the NAT Device will receive input connections on port 22 for SSH traffic and redirect all this traffic to SSH Server. Using the extended version of the Nile language, we can represent this situation, as shown in Listing 3.4.

Listing 3.4: NAT Forward in Nile

```
define intent forwardIntent:
    from endpoint ('internet')
    to endpoint ('NAT Device')
    for protocol ('SSH')
    forward address ('SSH Server', 2222)
```

Source: Author

This example shows how the Nile `forward` command can be used together with ⟨address⟩ symbol. In this example, a NAT action was represented from incoming traffic. The incoming SSH traffic with the destination "NAT Device" will be redirected to the internal server named "SSH Server" on port 2222.

4 IMPLEMENTATION

We validate the methodology's feasibility by developing a prototype solution called SCRIBE (SeCuRity Intent-Based Extractor). SCRIBE implements specializations of the process, focusing on Firewall and NAT configurations. The system leverages different input sources to process low-level firewall configurations and to query for all configurations available on firewall filtering and NAT tables. SCRIBE has about 400 lines of Python code. The entire source code of SCRIBE is available at Github[1]. In this chapter, we discuss the implementation of SCRIBE in detail and the modifications made in third-party tools to make them interact with SCRIBE.

4.1 System architecture and use

The architecture of SCRIBE is composed of modules that follow the process steps described in Section 3.1. The synthesis step's implementation involves the creation of additional modules that interact with the synthesizer, which is used as a base solution to parse network configurations. The Query Generator module brings commands for the synthesizer. These commands include load-low-level configuration rules into a synthesizer, make the synthesizer incorporate the rules, and query results on generated tables.

The synthesizer must be able to read low-level configurations and generate the intermediate representation containing the fields for source and destination IP addresses, ports, and protocols. For this prototype implementation, FWS (BODEI et al., 2018a) was chosen to be the synthesizer because it is capable of reading low-level firewall and NAT configurations from the iptables, IPFW, PF, and Cisco IOS, and represents these configurations into a tabular intermediate representation.

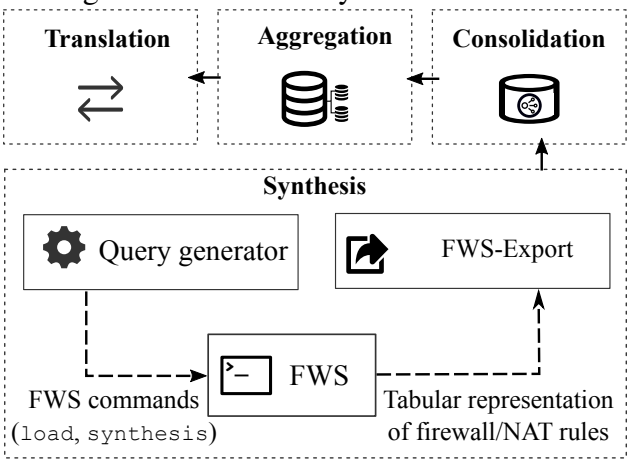
In order to make SCRIBE interact with FWS, the Query Generator was adapted to produce FWS commands. Also, a module called FWS-Export was developed to intercept the FWS's output and export it to a CSV format. The general architecture of SCRIBE, including the interaction with FWS, is shown in Figure 4.1. The FWS-Export implementation is detailed in Section 4.2.

The architecture of SCRIBE is composed of modules that follow the process steps described in Section 3.1. The general architecture of SCRIBE is shown in Figure 4.1. The synthesis step's implementation involves the creation of additional modules that interact

[1] <https://github.com/ComputerNetworks-UFRGS/SCRIBE>

with the synthesizer, which is used as a base solution to parse network configurations. The Query Generator module brings commands for the synthesizer. These commands include load-low-level configuration rules into a synthesizer, make the synthesizer incorporate the rules, and query results on generated tables.

Figure 4.1: SCRIBE system architecture



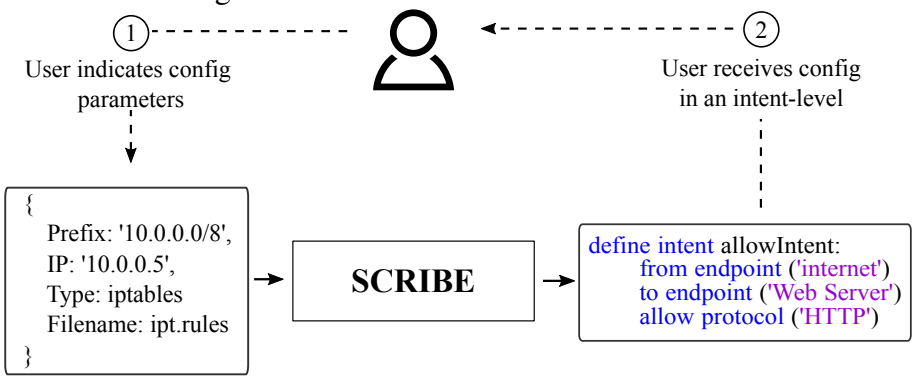
Source: Author

For the consolidation step, network devices are represented by classes. The aggregation process consists of enriching the modeled objects with additional information, and the translation process consists of functions that translate meta-intents to the Nile language. System modeling is described in Section 4.3.

User configurations drive the usage of SCRIBE. The user interaction with SCRIBE is shown in Figure 4.2. In the first step, indicated in Figure 4.2 by (1), the user (or possibly an external system) provides configuration parameters in JSON format indicating the network prefix, IP, type, and the name of the configuration file that contains the low-level firewall and NAT rules. The `Prefix` parameter indicates to which network prefix the firewall or NAT box device belongs in the user-provided topology. The `IP` parameter indicates the IP address of the firewall or NAT box device. In parameter `Type`, the user indicates the vendor of the firewall or NAT software or device. FWS dictates the support for low-level configurations, which supports `iptables`, IPFW, PF (Packet Filter), and Cisco (Cisco IOS ACL). The last parameter is Filename, indicating the file containing the rules.

In step (2), the system retrieves configurations informed by the user and process input according to the process described in Section 3.1, and returns the configurations for the user in the Nile language.

Figure 4.2: User interaction with SCRIBE



Source: Author

4.2 FWS

FWS (BODEI et al., 2018a) is used as a base solution to extract vendor-specific firewall and NAT configurations. FWS provides firewall characterization, determining which packets are accepted by the firewall applying the source configuration and represents the firewall and NAT rules in a tabular format. The output of FWS is a set of *flattened* rules, suggesting that the rules are not ordered as in traditional firewall configuration, aiding to simplify the representation. FWS uses the flattened representation designed in SecGuru (JAYARAMAN et al., 2014). For example, in a typical set of firewall rules to allow all traffic, except for a determined IP address, a rule to block the traffic for the fictitious IP address 192.168.0.15 can be defined, and next, a rule to allow incoming traffic for all hosts. Using the flattened rules this firewall policy is represented by $* \setminus \{192.168.0.15\}$, indicating an allow policy for all address, except 192.168.0.15.

FWS provides a language to interact with them. SCRIBE Query generator module brings commands in the FWS's language to load configurations and synthesize them. SCRIBE use `load_policy` command to read low-level configurations from a file, which is provided in the SCRIBE configuration. This command has the signature as `load_policy(`⟨type⟩`,` ⟨rules⟩`,` ⟨interfaces⟩`)`, where ⟨type⟩ represents the firewall type of configurations (actually supporting Cisco IOS, iptables, IPFW and PF), ⟨rules⟩ is the filename of exported configurations and ⟨interfaces⟩ is the interface description file. The `iptables` software is commonly used in GNU/Linux distributions. An input example of `iptables` rules is shown in Listing 4.1.

The input rules for `iptables` in Listing 4.1 represents a scenario where three main policies governs the firewall. The first (*i*) is to allow arbitrary traffic between internal hosts. Second, (*ii*) operators want to ensure that incoming HTTP and HTTPS traffic are

allowed to Web Server, and SSH incoming traffic is allowed in the SSH Server. Finally, (*iii*) all outgoing traffic to HTTP and HTTPS must be allowed.

```
### Filtering rules ###
*filter
# Default policy DROP in filtering chains
:INPUT DROP [0:0]
:FORWARD DROP [0:0]
:OUTPUT ACCEPT [0:0]

#1: Allow arbitrary traffic between internal hosts
-A FORWARD -s 172.16.1.0/24 -d 172.16.1.0/24 -j ACCEPT

#2: Allow SSH/HTTPS incoming traffic to the corresponding
hosts
-A FORWARD -p tcp -d 172.16.1.15 --dport 80 -j ACCEPT
-A FORWARD -p tcp -d 172.16.1.15 --dport 443 -j ACCEPT
-A FORWARD -p tcp -d 172.16.1.16 --dport 22 -j ACCEPT

#3: Allow HTTP/HTTPS outgoing traffic
-A FORWARD -s 172.16.1.0/24 -p tcp --dport 80 -j ACCEPT
-A FORWARD -s 172.16.1.0/24 -p tcp --dport 443 -j ACCEPT

COMMIT
```

Listing 4.1: Iptables input

SCRIBE executes the `load` command on FWS and passes the result to a variable using the FWS's language. After, SCRIBE runs the command `synthesis` that synthesizes configurations to a tabular format. The command `synthesis` expects as parameter the variable with the rules loaded in the previous step. SCRIBE runs `synthesis` command two times: first to synthesize firewall filtering rules and another to synthesize NAT forward rules. FWS then generates two tables for each configuration file, one for filtering and another for NAT forward rules. An example of an output table for filtering rules is shown in Table 4.1. The tables only display rules for accepted connections.

FWS, however, has a limitation that displays all the configurations only on the screen, which turns in a challenge to reuse its output. To overcome this limitation, the FWS source code was extended with a module called FWS-Export, which is publicly

Table 4.1: Example of FWS output

Source IP	Source Port	Destination IP	Destination Port	Protocol
*	*	172.16.1.16	22	TCP
*	*	172.16.1.15	443 80	TCP
172.16.0.0/24	*	172.16.0.0/24	*	*
172.16.0.0/24	*	*	443 80	TCP

Source: Author

available at Github[2].

The FWS-Export module allows the exportation of FWS tabular representation to CSV format. FWS-Export intercepts all filtering and forward rules generated by FWS and exports them to CSV files. FWS-Export generates two CSV files per device, one for filtering rules and another for NAT rules. For firewall rules, the CSV contains the fields `srcIP` and `dstIP`, representing the source and destination addresses respectively, `srcPort` and `dstPort` representing the source and destination ports, the field `protocol` indicates if the protocol is TCP or UDP and the field `state` indicates if the state of connection is new or established. NAT tables contain all the fields present in the firewall output and include the fields `srcIP´` and `dstIp´`, representing the NAT source and destination, respectively. These files are posteriorly used as input to SCRIBE. The structure of CSV files is shown in Figure 4.3.

Figure 4.3: CSV structure

Firewall

srcIP	srcPort	dstIP	dstPort	protocol	state

NAT

srcIP	srcIP'	srcPort	dstIP	dstIp'	dstPort	protocol	state

Source: Author

4.3 System modeling

SCRIBE was modeled to be extensible and to be capable of representing several network functions. This extensibility makes use of modules, each representing a different network function. Initially, the SCRIBE system model uses Target, Traffic, and Action entities explained in Section 3.1. These entities are composed of attributes relating to

[2]<https://github.com/ComputerNetworks-UFRGS/fws-export>

the parts of the configuration. The model was specialized to represent firewall and NAT through mapping from input files to SCRIBE entities.

With the structured input files, the initial approach is to map the fields from the input files to the SCRIBE classes. In SCRIBE, the network was modeled using an Objected Oriented Programming (OOP) paradigm. For the model, an object is an instance of a network component, such as an IP address. In the first moment, utility functions read the input configuration and CSV files before generating entities.

Table 4.2: Mapping FWS Firewall functions to SCRIBE Entities

Information	**FWS Column**	**SCRIBE Entity**
Source IP	srcIP	Target.SrcIP
Destination IP	dstIP	Target.DstIP
Source Port	srcPort	Traffic.SrcPort
Destination Port	dstPort	Traffic.DstPort
Protocol	protocol	Traffic.Protocol

Source: Author

The firewall filtering behavior was implemented in class Filtering. The class Filtering contains methods for generating and export entities based on the input CSV. The function `generate_entitity` generates entities based on CSV fields. It maps source and destination IP addresses and ports and the protocol fields received as input to entities. The attributes will constitute the Target and Traffic entities. The mapping attributes are displayed in Table 4.2.

The NAT behavior was implemented in class NAT. The class NAT contains methods for generating and exporting entities based on the input CSV, similar to the Filtering class. The function `generate_entitity` generates entities based on CSV fields. It maps source and destination IP addresses and ports and the protocol fields received as input to entities. Additionally, it adds the parameter `operation`, indicating the NAT operation to be executed, and the NAT forward address as a parameter for the action. The attributes will constitute the Target and Traffic entities, and action and parameter will constitute the Action entity. The mapping attributes for NAT are displayed in Table 4.3.

Filtering and NAT classes implement the method `export_entities`, which returns the Network Entities in a JSON format. An entity must have the fields `traffic`, `action` and opcionally `src`, `dst` and `param`. The function `export_entities` returns the meta-intents that are composed of entities, each of them representing a part of the configuration.

Meta-intents contains entities composed of information corresponding to the classes

Table 4.3: Mapping NAT functions to SCRIBE Entities

Information	**FWS Column**	**SCRIBE Entity**
Source IP	srcIP	Target.SrcIP
Source NAT IP	srcIP´	Target.NATSrcIP
Destination IP	dstIP	Target.DstIP
Destination NAT IP	dstIP´	Target.NATDstIP
Source Port	srcPort	Traffic.SrcPort
Destination Port	dstPort	Traffic.DstPort
Protocol	protocol	Traffic.Protocol

Source: Author

Filtering and NAT. A Filtering meta-intent will contain Target, Traffic, and Action entities, where the Action will represent a allow or block rule. For NAT, the entity Action will represent a `forward` operation and will contain a parameter, representing the address that NAT will forward to (or from). Meta-intents are represented in JSON format. An example of meta-intent for Filtering is shown in Listing 4.2. This example of meta-intent represents the first line of FWS output, a firewall policy representing an allow action from the Internet to the SSH server (172.16.1.16).

```
{
    "Target":{
        "src":"*",
        "dst":"172.16.1.16"
    },
    "Traffic":{
        "srcPort":"*",
        "dstPort":"22",
        "protocol":"tcp"
    },
    "Action":"allow"
}
```

Listing 4.2: Meta-intent

Meta-intents represent a higher-level rule than the tabular representation and include the Action — *i.e.*, the operation to be executed over the rule. The next phase is the aggregation of complementary information. The first part of aggregation is service discovery, which consists of getting the service name of an application protocol based in an L4 protocol and port number. The standardized Service Name and Transport Protocol Port Number Registry (SERVICE. . . , 2020) was used to query for services in a given port and protocol.

The aggregation process allows operators to provide complementary information. SCRIBE supports as input a host description file, indicating the hostname and IP address or prefix of a host. Users can set the `alias_file` parameter to indicate a host file. This file's structure must follow the order hostname (or name of a prefix) and the IP address or prefix separated by space, as indicated in Listing 4.3.

```
internal     172.168.1.0/24
web_server   172.168.1.15
```

Listing 4.3: Example of host file

The meta-intent representation is, then, enriched with the complementary information of the hosts' files and the service names. After this step, low-level information, such as port name and protocol, is removed to simplify the meta-intent. A result of an enriched meta-intent is shown in Listing 4.4.

```
{
    "Target":{
        "src":"Internet",
        "dst":"SSH Server"
    },
    "Traffic":{
        "service":"SSH"
    },
    "Action":"allow"
}
```

Listing 4.4: Aggregated meta-intent

The last step is the translation for an intent-level language. Nile is the default target language for translation. The translation to Nile involves going through the meta-intents and mapping meta-intent attributes to the Nile grammar symbols.

For a given meta-intent, the initial step is to map the entities `Target.src` and `Target.dst` to Nile endpoints. After, the traffic is identified by type, which for firewall and NAT may be a service or protocol. Then, SCRIBE maps the traffic from meta-intents to Nile service or protocol, composing the ⟨matches⟩ symbol. Last, the system identifies the action, which can be mapped to a Nile ⟨operation⟩ symbol. The system follows the Nile grammar to translate correctly. An example of a Filtering translation is shown at Listing 4.5. This resulting Nile code is a translation from the aggregated meta-intent from Listing 4.4.

```
define intent intent1:
    from endpoint('Internet')
    to endpoint('SSH Server')
    allow service('SSH')
```

Listing 4.5: Translation of a firewall ALLOW rule to Nile language

5 EVALUATION AND DISCUSSION

In this chapter, we present case studies that demonstrate the solution's applicability in different network contexts. To prove the concept, we also evaluated the accuracy of SCRIBE using real datasets containing firewall rules from various servers and institutions (DIEKMANN et al., 2016). Results show that SCRIBE is capable of expressing network configurations in an intent-level with high accuracy.

5.1 Case Studies

Assuming that a new operator initiates at a company and wants to understand the network configuration, the operator intends to assert if the network behaves as expected. For this, there is a need to understand the existing network configuration. The operator could use SCRIBE to understand the network state and then insert a new rule as needed. In this section, two case studies indicating the solution's applicability are described using real-world scenarios to extract intents from already deployed network configurations.

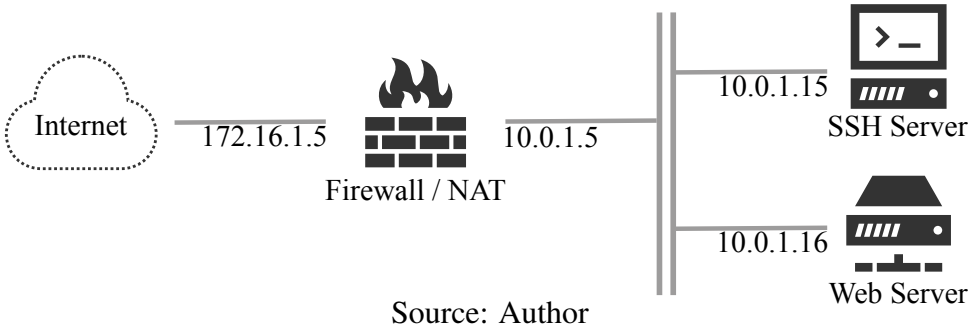
The deployment of new intents in a traditional network requires that operators discover all network devices and their respective configurations. To do this task, operators must have platform-specific expertise, besides network administrative tools knowledge. In this situation, a network operator would need to discover all devices in the network and their respective IP addresses, read the entire configuration from various network devices, and provide some documentation to help understand the network behavior for the next alteration.

Different scenarios were analyzed to prove the concept and the technical feasibility of the proposed solution. The first scenario represents a small company containing a single firewall device at the input of the network that manages ACL and translates NAT addresses. This scenario also includes servers and a workstation, isolated by NAT. The second scenario explores a Science DMZ campus network containing ACLs directly in a DMZ router, due to the high-speed characteristic in this network, and a firewall to secure the internal network and provide NAT isolation. Next, these case studies are discussed in detail.

Case Study #1 - Small company

This scenario represents a small company, with a unique firewall device managing all incoming and outgoing traffic and also managing NAT rules. The studied network contains a Web Server, an SSH Server, and a Firewall/NAT middlebox, which manages the network input. Servers can not directly be accessed externally. For access to servers, all input traffic in Firewall/NAT middlebox is redirected to the corresponding server according to the traffic type. The firewall device has two interfaces and also translate NAT addresses. For this scenario, the firewall was configured using iptables, which also manage NAT rules. The representation of this scenario is shown in Figure 5.1. The entire configuration for this case is about 40 lines of `iptables` code and is available at GitHub[1].

Figure 5.1: Corporate network with NAT



Source: Author

In this context, the solution can help by parsing the iptables code and processing all deployed network configurations and representing NAT and firewall rules in the Nile language, assisting operators in understanding the network behavior into a language closer to the natural language. At the beginning of the process, the solution queries FWS for all configurations, resulting in a table indicating all allowed services in this network. After, the solution processes the resulting table, extracts entities, and generates the model as described in Section 3. Friendly hostnames can be aggregated together with the service running in each port indicated by the firewall rule, enriching the model, providing more high-level information. Finally, the system generates meta-intents with higher-level rules and translate them to Nile. The generated Nile code is shown in Listing 5.1.

After extracting intents, the system can help the operator to comprehend network behavior. It is simple to infer the primary security behavior of this scenario by observing the extracted intents in Listing 5.1. From the first intent, it can be noted that (*i*) only protocols HTTP, HTTPS, and SSH are allowed to enter on this network. The remaining

[1]<https://github.com/ComputerNetworks-UFRGS/SCRIBE/tree/master/case_studies/1_Small_Company>

of the generated intents describe the NAT behavior of the network. The second produced intent in Listing 5.1 indicates that (*ii*) all SSH traffic will be redirected to the SSH Server, which is internally addressed by the IP address 10.0.1.15. The last two intents express that (*iii*) all HTTP and HTTPS traffic will be redirected to the Web Server, which is internally addressed by the IP address 10.0.1.16. The default behavior for this scenario is the default block; thus (*iv*), all other connections will be denied.

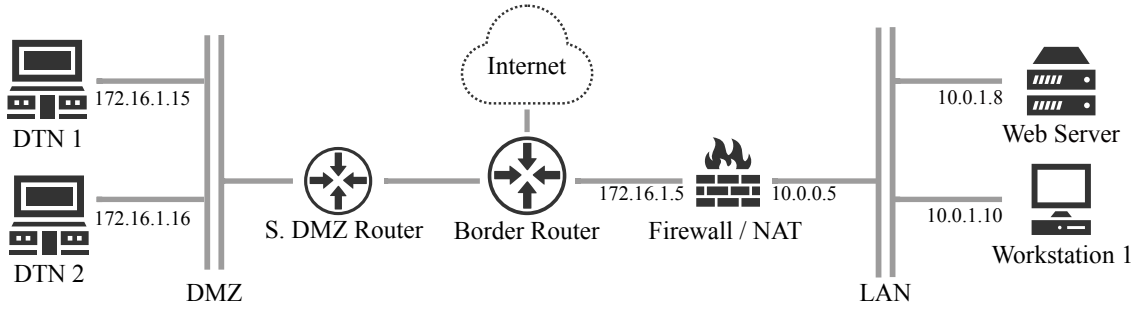
```
define intent firewallIntent:
  from endpoint ('internet')
  to endpoint ('firewall')
  allow protocol ('SSH'), protocol ('HTTPS'),
      protocol ('HTTP')

define intent natIntent:
  from endpoint ('internet')
  to endpoint ('firewall')
  for protocol ('HTTPS')
  forward address ('Web Server')
```

Listing 5.1: Nile code for firewall Intents

Case Study #2 - Science DMZ

Figure 5.2: Science DMZ network



Source: Author

In this case study, a university scenario with Science DMZ was explored. In this network, there are high-speed DTNs (Data Transfer Nodes) in a DMZ (Demilitarized Zone), a router with ACLs, and an internal firewall. Figure 5.2, shows the scenario. In this network, the DTNs are connected directly to a router because of the high-speed transfers

(DART et al., 2014). A DMZ can be used to isolate particular servers from the machines within a network. It is important to separate the public servers from the internal network so an attacker can not get to all the systems. A stateful firewall can degrade the network performance and is not suitable to realize the transfer of huge amounts of scientific data in high-speed connections. For this, the ACLs (Access-Control Lists) are inserted directly in the router in a stateless manner, aiming to maximize the transfer speeds (DART et al., 2014).

```
define intent firewallIntent1:
  from endpoint ('University A')
  to endpoint ('DTN 1')
  allow traffic ('all')

define intent firewallIntent2:
  from endpoint ('University B')
  to endpoint ('DTN 2')
  allow traffic ('all')

define intent firewallIntent3:
  from endpoint ('My Network')
  to endpoint ('My Network')
  allow traffic ('all')

define intent natIntent1:
  from endpoint ('internet')
  to endpoint ('Firewall / NAT')
  for protocol ('HTTP')
  forward address ('Web Server')

define intent natIntent2:
  from endpoint ('internet')
  to endpoint ('Firewall / NAT')
  for protocol ('HTTPS')
  forward address ('Web Server')
```

Listing 5.2: Nile code for firewall Intents

Several intents can be inferred after applying the bottom-up process to the configuration from this network. The ACL router begins with a default-blocked setting. It is possible to add network IPs and prefixes in a “allowed list", indicating which server is allowed to perform a high-speed connection with a specific DTN. NAT isolates the internal

network, and the firewall device attends to translate external addresses to NAT addresses. The firewall rules deal with internal traffic, besides blocking suspected protocols, such as UDP or TCP for some specific internal hosts.

From the intents listed in Listing 5.2, it can be inferred the base intents derivated from this scenario. The first two intents (*i*) represent an allow policy from Universities A and B for their corresponding authorized DTNs (Data Transfer Nodes). Also, these intents hide the complexity involved to configure two separate devices to manage ACLs in a high-speed DMZ and a conventional stateful firewall managing internal connections. The third intent of Listing 5.2 (*ii*) describes the behavior of allowing all types of traffic in the internal network. With the use of intuitive alias, supported by SCRIBE, as an example, the alias “My Network" can be defined to refer to the prefix 10.0.0.0/8 (prefix of the internal network). From intents `natIntent1` and `natIntent2`, it can be observed (*iii*) a NAT translation from all incoming traffic for HTTP and HTTPS protocols respectively, indicating that these types of traffic will be forwarded to the Web Server, which is only visible in the internal network.

5.2 Accuracy Evaluation

The evaluation of the bottom-up process requires a careful analysis of the intents, *i.e.*, there is a need to find an approach to identify whether the extracted intents represent the same configuration as the low-level configuration. In this section, we define and measure accuracy metrics collected from SCRIBE through several real-world network configurations.

Two main aspects were evaluated to assert the feasibility of the bottom-up process: (*i*) the trustworthiness of generated intents and (*ii*) SCRIBE’s support for the most present functionalities in real-world firewall dumps. There is a need to check whether the characteristics of the original configuration files are present in the extracted intents to evaluate the trustworthiness of generated intents.

Datasets

Measurements were performed using real-world security rules from a public collaborative repository (DIEKMANN, 2015), which contains dumps of firewall rules from

various servers and institutions. For security concerns, some public IP and MAC addresses have been anonymized. For this work, six datasets were selected . In Table 5.1, the datasets are presented, including a short description, type (firewall software), and the number of Lines of Code (LoC).

Table 5.1: Dataset name, type, description and Lines of Code (LoC)

Dataset	**Type**	**Description**	**LoC**
Unikernel	`iptables`	A Unikernel Firewall for QubesOS	65
Home	`iptables`	A typical home user's firewall	296
Kornwall	`iptables`	A script to set up a host-based firewall on a PC	92
NetArq	`iptables`	The Chair of Network Architectures and Services	5668
Memphis	`iptables`	Firewall of the memphis testbed	46
Veroneau	`iptables`	Security of the Veroneau's site	270

Source: Author

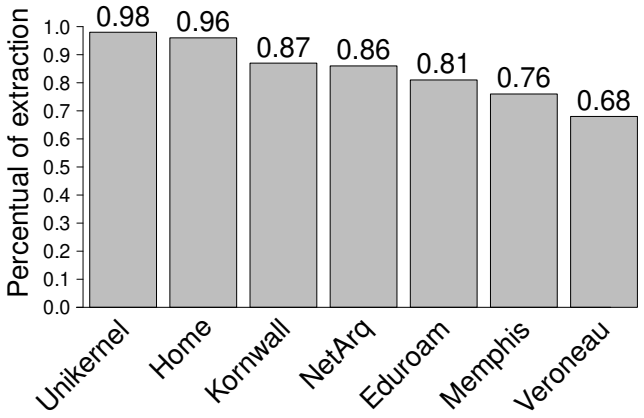
Translation Accuracy

We develop an approach to evaluate the trustworthiness of extracted intents. For this, static analysis was performed to validate whether each generated intent represents the configuration accurately. The characteristics were analyzed from various real-world firewall configuration files to perform this process. Each characteristic is a configuration element of a rule. For instance, source address, port number, and action (*e.g.*, allow) are characteristics of a traditional allow rule. SCRIBE provides a script to extract all firewall parameters of configurations. These extracted parameters were used to compare with parameters extracted from generated intents to evaluate the accuracy.

SCRIBE was used to extract intents from several real-world firewall configuration files and to evaluate accuracy. After, the system dumps the parameters of configurations present in generated intents. The parameters dumped from low-level configurations are then compared if they are also present in generated intents. Thus, the accuracy is calculated by counting how many parameters of the configuration files are also present in generated intents. Figure 5.3 presents the accuracy of the intents. The X-axis shows the firewall dump (with a compact name). Y-axis represents the rate of correctly extracted configuration parameters, *i.e.*, the percentual of configuration parameters present in low-level configurations, and present in generated intents.

Figure 5.3 shows that the accuracy of generated intents is above 80% for most of the dumps, which represent that SCRIBE can extract intents from configuration files with

Figure 5.3: Percentual of corrected extracted intents for each firewall dump



Souce: Author

high fidelity, missing very few configuration details. There is a variation in the accuracy between datasets. This variation occurs due to limitations of the representation capabilities, *i.e.*, some datasets contain features yet not supported by SCRIBE. The worst case is Veroneau (a dump of *veroneau.net*) and contains several directives of ICMP messages and some directives of rate-limiting, both not supported in SCRIBE yet. However, even for this dump, the solution can express most of the firewall configuration's functionalities. It can be observed that network configurations can be represented using high-level intents, closely resembling the natural language and maintaining high fidelity to the original configuration.

Functionality support

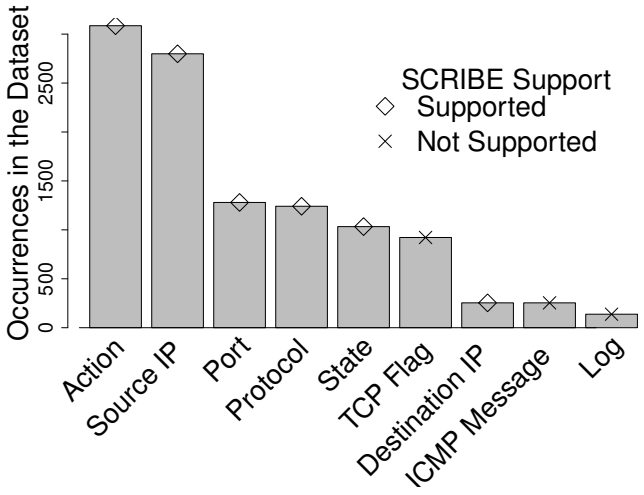
This evaluation aims to verify the level of functionality support of SCRIBE to the most used configuration in real-world firewall configurations. This evaluation represents the trustworthiness of extracted intents concerning the original configurations. For this, several configuration dumps were obtained from a public collaborative repository proposed by (DIEKMANN et al., 2016). There are various files containing dumps of `iptables` firewalls exported from various servers and institutions in this repository.

This evaluation consists of obtaining all `iptables` options exported from real-world configurations files and checking which of them are supported by SCRIBE. Most of `iptables` options can refer to command, parameter, match, target, and listing options

(REDHAT, 2005a). Command options instruct `iptables` to perform specific actions, such as append or delete a new rule. Parameter options are used to define a packet filtering rule. Match options are used for specialized matching options such as protocols and ports, and also can be extended through modules. Target options represent actions to take if a packet is matched, such as ACCEPT or DROP actions. Listing options represent options to list rules on the screen.

Functionalities are represented by `iptables` parameters, as well as matches and target options. This evaluation is performed through a script that dumps all existing parameters, as well as matches and target options existing that represent functionalities in the configuration files. In this evaluation, the number of functionalities existing in the dataset was counted and compared with functionalities supported by SCRIBE. This evaluation demonstrates the most used functionalities in datasets and SCRIBE support for them. The result of this evaluation is shown in Figure 5.4.

Figure 5.4: SCRIBE support for the most used functionalities in datasets



Souce: Author

Results show that SCRIBE supports the most used functionality parameters in real-world firewall configurations. Due to this support, SCRIBE can represent the most used firewall functionalities at an intent-level. Most of SCRIBE unsupported functionalities of firewalls refers to logging, TCP flags, and ICMP messages. For the moment, SCRIBE does not support rate-limiting functionalities either.

6 CONCLUDING REMARKS

In this dissertation, we introduced a novel bottom-up approach to intent extraction, which synthesizes and represents existing network configurations in an intent-level language. A solution was developed to validate the process's feasibility, and two case studies describing real-world scenarios were provided to apply the process. In the case studies, it was evidenced that intents can be extracted from these scenarios, and it represents the network behavior in intent-level, close to the natural language. Additionally, real-world firewall configurations were used to evaluate the accuracy of SCRIBE. Experiments demonstrate that it is possible to extract intents from real configurations using SCRIBE with high accuracy, preserving the majority of aspects of the original configurations in the translation process.

6.1 Summary of Contributions and Results

Current network management relies on human in the control loop, where the human insight is fundamental for network maintenance (BIRKNER et al., 2018). However, traditional networks are generally hard to maintain because of the gap between the high-level specifications and low-level configuration directives. In a traditional network, it is hard to infer high-level network-wide specifications from device-level configuration directives. Operators need to learn several low-level configuration languages to express configurations for each different network device. Also, the solutions that use intent-level languages have top-down approaches, *i.e.*, networks must support intent-level configuration already to use them.

This work emerged as a solution to fill the gap between the high-level specifications and device-level configurations using a bottom-up approach. The proposed solution can fill this gap by facilitating the operators' understanding of the actual network configurations by extracting intents from such low-level configuration directives, assisting both novice and experienced operators in understanding existing configurations. Also, this solution may help operators migrating from legacy networks to intent-based networks.

The implementation of the solution was made through a prototype tool called SCRIBE. In the SCRIBE tool, the bottom-up process was implemented, focusing on firewall and NAT. This implementation aims to (*i*) prove the concept of the solution and (*ii*) serve as a tool for the experiments, allowing the demonstration of case studies and the

accuracy evaluation. This tool actually supports device-level directives representing configurations of firewalls such as iptables, IPFW, Cisco IOS, etc. Moreover, the output of SCRIBE is directly a Nile code reproducing these configurations into an intent-level. After all the process of intent extraction, in some situations, operators can directly use the output of SCRIBE as input for modern tools for Intent-Based Networking, particularly the Nile language.

In this work, we demonstrated the applicability of the solution through two case studies using two different scenarios: one representing an infrastructure of a small company with a single firewall and another depicting a high-speed campus network using Science DMZ. In such cases, it was demonstrated the technical feasibility of the solution. The output of the solution through the prototype tool SCRIBE depicts that traditional configurations can be expressed in easy-to-understand Nile intents, closely resembling the English language. Also, the total number of lines for such configurations were substantially reduced.

In the evaluation, we prove the solution's concept by demonstrating (*i*) the trustworthiness of extracted intents and (*ii*) the SCRIBE support for the most present functionalities in real-world firewall configurations. Results show that SCRIBE can accurately express configurations in an intent-level language, supporting the most used functionality parameters in real-world firewall configurations.

6.2 Limitations and Future Work

Although SCRIBE extracts intents with reasonable accuracy, it still faces some limitations. In particular, it does not support firewall features of rate-limiting, logging, and ICMP messages included in some datasets. The support for these features and the inclusion of that in the network model may significantly increase SCRIBE accuracy. In future work, such functionalities may be added to SCRIBE, increasing representation capabilities and evaluation accuracy. The support for capabilities is given by (1) the synthesizer and (2) the capacity of representation of the target language. The synthesizer gives support for capabilities supported by network technologies. Thus, the accuracy is directly related to the synthesizer and can be improved by adding a new synthesizer or extending the current synthesizer (FWS). Furthermore, adding new constructs in the target language, Nile, will allow the representation of the extracted configurations in the intent-level language.

SCRIBE was developed as a prototype tool to prove the proposed solution's concept. Despite that SCRIBE is modular and can be extended, the SCRIBE functionalities are focused on NAT and Firewall contexts. From a future perspective, SCRIBE may be extended, adding support to other network functions such as routing algorithms (*e.g.*, OSPF, BGP). These modifications would require SCRIBE support from other synthesizers, rather than to FWS, as well as changing the target language or extending the Nile grammar to make it support these functionalities.

Currently, SCRIBE only supports the extraction of intents in a 1:1 mapping, i.e., an intent is extracted for each configuration/rule. Finally, as future work, the use of probabilistic methods (FRANK et al., 2009) and data mining techniques (KHERADMAND, 2020) is suggested to represent higher-level intents. These techniques will allow the inference of complex intents from the low-level configurations, deriving an intent from many low-level configurations; however, it will introduce non-deterministic behavior to the solution.

REFERENCES

ABEDIN, M. et al. Analysis of firewall policy rules using traffic mining techniques. **International Journal of Internet Protocol Technology (IJIPT)**, Inderscience Publishers, Geneva 15, CHE, v. 5, n. 1/2, p. 3–22, abr. 2010. ISSN 1743-8209.

AL-SHAER, E. S.; HAMED, H. H. Discovery of policy anomalies in distributed firewalls. In: **Proceedings IEEE INFOCOM 2004 - The 23rd Conference of the IEEE Communications Society**. New York, NY, USA: IEEE, 2004. v. 4, n. 23, p. 2605–2616 vol.4. ISSN 0743-166X.

AREZOUMAND, S. et al. MD-IDN: Multi-domain intent-driven networking in software-defined infrastructures. In: **2017 13th International Conference on Network and Service Management (CNSM)**. New York, NY, USA: IEEE, 2017. p. 1–7. ISSN 2165-963X.

BECKETT, R. et al. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations Ryan. In: **Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery (ACM), 2016. (SIGCOMM '16), p. 328–341. ISBN 978-1-4503-4193-6. Available from Internet: <http://doi.acm.org/10.1145/2934872.2934909>.

BECKETT, R. et al. Network configuration synthesis with abstract topologies. In: **Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: Association for Computing Machinery (ACM), 2017. (PLDI 2017), p. 437–451. ISBN 9781450349888.

BEHRINGER, M. et al. **Autonomic Networking: Definitions and Design Goals**. [S.l.], 2015.

BENSON, T.; AKELLA, A.; MALTZ, D. Unraveling the complexity of network management. In: **Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2009. (NSDI'09), p. 335–348.

BENZEKKI, K.; FERGOUGUI, A. E.; ELALAOUI, A. E. Software-defined networking (SDN): a survey. **Security and Communication Networks**, v. 9, n. 18, p. 5803–5833, 2016. ISSN 19390122.

BIRKNER, R. et al. Net2Text: Query-Guided Summarization of Network Forwarding Behaviors. In: **15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)**. Renton, WA: USENIX Association, 2018. p. 609–623. ISBN 978-1-931971-43-0.

BODEI, C. et al. Language-Independent Synthesis of Firewall Policies. In: **2018 IEEE European Symposium on Security and Privacy (Euro S&P)**. New York, NY, USA: IEEE, 2018. p. 92–106.

BODEI, C. et al. Transcompiling firewalls. In: **Principles of Security and Trust**. Cham: Springer International Publishing, 2018. p. 303–324. ISBN 978-3-319-89722-6.

CLEMM, A. et al. **Intent-Based Networking - Concepts and Definitions**. [S.l.], 2019. Work in Progress. Available from Internet: <https://datatracker.ietf.org/doc/html/draft-irtf-nmrg-ibn-concepts-definitions-00>.

COTTON, M.; MOSKOWITZ, B.; VEGODA, L. **RFC 5735: Special Use IPv4 Addresses**. [S.l.]: RFC Editor, 2010.

DART, E. et al. The science DMZ: A network design pattern for data-intensive science. **Scientific Programming**, Hindawi, v. 22, n. 2, p. 173–185, 2014.

DIEKMANN, C. **net-network**. [S.l.]: GitHub, 2015. <https://github.com/diekmann/net-network>.

DIEKMANN, C. et al. Verified iptables firewall analysis. In: **2016 IFIP Networking Conference (IFIP Networking) and Workshops**. New York, NY, USA: Association for Computing Machinery (ACM), 2016. p. 252–260.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The Road to SDN: An Intellectual History of Programmable Networks. **ACM SIGCOMM Computer Communication Review**, Association for Computing Machinery (ACM), New York, NY, USA, v. 44, n. 2, p. 87–98, abr. 2014. ISSN 0146-4833.

FOGEL, A. et al. A General Approach to Network Configuration Analysis. In: **Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2015. (NSDI'15), p. 469–483. ISBN 978-1-931971-218.

FRANK, M. et al. A probabilistic approach to hybrid role mining. In: **Proceedings of the 16th ACM conference on Computer and communications security**. New York, NY, USA: Association for Computing Machinery (ACM), 2009. p. 101–111.

GEMBER-JACOBSON, A. et al. Fast Control Plane Analysis Using an Abstract Representation. In: **Proceedings of the 2016 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery (ACM), 2016. (SIGCOMM '16), p. 300–313. ISBN 978-1-4503-4193-6.

HACHANA, S.; CUPPENS-BOULAHIA, N.; CUPPENS, F. Mining a High Level Access Control Policy in a Network with Multiple Firewalls. **Journal of Information Security and Applications (JISA)**, Elsevier Science Inc., USA, v. 20, n. C, p. 61–73, feb. 2015. ISSN 2214-2126.

JACOBS, A. S. et al. Refining network intents for self-driving networks. In: **Proceedings of the Afternoon Workshop on Self-Driving Networks**. New York, NY, USA: Association for Computing Machinery (ACM), 2018. (SelfDN 2018), p. 15–21. ISBN 978-1-4503-5914-6.

JAYARAMAN, K. et al. **Automated Analysis and Debugging of Network Connectivity Policies**. Redmond, WA, NY, USA, 2014.

KHERADMAND, A. Automatic inference of high-level network intents by mining forwarding patterns. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery (ACM), 2020. (SOSR '20), p. 27–33. ISBN 9781450371018.

KUROSE, J.; ROSS, K. **Computer Networking: A Top Down Approach**. Boston, MA, USA: Addison-Wesley, 2012.

MARTÍNEZ, S. et al. A model-driven approach for the extraction of network access-control policies. In: **Proceedings of the Workshop on Model-Driven Security**. New York, NY, USA: Association for Computing Machinery (ACM), 2012. (MDsec '12), p. 5:1–5:6. ISBN 978-1-4503-1806-8.

NARAIN, S.; TALPADE, R.; LEVIN, G. Network Configuration Validation. In: KALMANEK, C. R.; MISRA, S.; YANG, Y. R. (Ed.). **Guide to Reliable Internet Services and Applications**. London: Springer, 2010. p. 277–316. ISBN 978-1-84882-828-5.

NELSON, T. et al. The margrave tool for firewall analysis. In: **USENIX Large Installation System Administration Conference (LISA), 2010**. Berkeley, California, United States: USENIX, 2010.

PANG, L. et al. A Survey on Intent-Driven Networks. **IEEE Access**, v. 8, p. 22862–22873, 2020. ISSN 2169-3536.

QUOITIN, B.; UHLIG, S. Modeling the routing of an autonomous system with C-BGP. **IEEE Network**, v. 19, n. 6, p. 12–19, Nov 2005. ISSN 1558-156X.

REDHAT. **18.3. Options Used within iptables Commands**. RedHat, Inc, 2005. Available from Internet: <http://web.mit.edu/rhel-doc/4/RH-DOCS/rhel-rg-en-4/s1-iptables-options.html>.

REDHAT. **7.4. FORWARD and NAT Rules**. RedHat, Inc, 2005. Available from Internet: <http://web.mit.edu/rhel-doc/4/RH-DOCS/rhel-sg-en-4/s1-firewall-ipt-fwd.html>.

REKHTER, Y. et al. **RFC 1918: Address allocation for private internets**. [S.l.]: RFC Editor, 1996.

SCHEID, E. J. et al. INSpIRE: Integrated NFV-based Intent Refinement Environment. In: **2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)**. New York, NY, USA: IEEE, 2017. p. 186–194.

SERVICE Name and Transport Protocol Port Number Registry. 2020. Available from Internet: <https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml>.

SRISURESH, P.; EGEVANG, K. **Traditional IP network address translator (Traditional NAT)**. [S.l.]: RFC 3022, January, 2001.

TANENBAUM, A.; WETHERALL, D. **Computer Networks**. New York, NY, USA: Pearson Prentice Hall, 2011. ISBN 9780132126953.

TIAN, B. et al. Safely and Automatically Updating In-Network ACL Configurations with Intent Language. In: **Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery (ACM), 2019. p. 214–226. ISBN 9781450359566.

TONGAONKAR, A.; INAMDAR, N.; SEKAR, R. Inferring Higher Level Policies from Firewall Rules. In: **Proceedings of the 21st Conference on Large Installation System Administration Conference**. Berkeley, CA, USA: USENIX Association, 2007. (LISA'07), p. 2:1–2:10. ISBN 978-1-59327-152-7.

WOOL, A. Trends in Firewall Configuration Errors: Measuring the Holes in Swiss Cheese. **IEEE Internet Computing**, v. 14, n. 4, p. 58–65, July 2010. ISSN 1941-0131.

APPENDIX A — PUBLISHED PAPER – AINA 2020

Intent-Based Networking (IBN) is showing significant improvements in network management, especially by reducing the complexity through intent-level languages. However, IBN is not yet integrated and widely deployed in most networks. Network operators may still encounter several issues deploying new intents, such as reasoning about complex configurations to understand previously deployed rules before writing an intent to update the network state. Many networks include several devices distributed along its topology, each device configured using vendor-specific languages. Thus, inferring the behavior of devices as high-level intents from these low-level configurations can be an arduous and time-consuming task. Current solutions that derive high-level representations from bottom-up configuration analysis can not represent configurations in an intent-level. In this work, we present a bottom-up approach to extract intents from network configurations. To validate our approach, we develop a system called SCRIBE (SeCuRity Intent-Based Extractor), which decompiles security configurations from different network devices and translates them to an intent-level language called Nile. To demonstrate the feasibility of SCRIBE, we outline two case studies and evaluate with dumps of real-world firewall configurations containing rules from various servers and institutions.

- **Title:** A Bottom-up Approach for Extracting Network Intents
- **Conference:** International Conference on Advanced Information Networking and Applications (AINA)
- **Type:** Main Track (Full Paper)
- **Qualis:** A2
- **Held at:** Caserta, Italy

A Bottom-Up Approach for Extracting Network Intents

Rafael Hengen Ribeiro( ), Arthur Selle Jacobs, Ricardo Parizotto, Luciano Zembruzki, Alberto Egon Schaeffer-Filho, and Lisandro Zambenedetti Granville

Institute of Informatics – Federal University of Rio Grande do Sul, Av. Bento Gonçalves, Porto Alegre 9500, Brazil
{rhribeiro,asjacobs,rparizotto,lzembruzki,alberto,granville}@inf.ufrgs.br

Abstract. Intent-Based Networking (IBN) is showing significant improvements in network management, especially by reducing the complexity through intent-level languages. However, IBN is not yet integrated and widely deployed in most networks. Network operators may still encounter several issues deploying new intents, such as reasoning about complex configurations to understand previously deployed rules before writing an intent to update the network state. Many networks include several devices distributed along with its topology, each device configured using vendor-specific languages. Thus, inferring the behavior of devices as high-level intents from low-level configurations can be an arduous and time-consuming task. Current solutions that derive high-level representations from bottom-up configuration analysis can not represent configurations in an intent-level. In this work, we present a bottom-up approach to extract intents from network configurations. To validate our approach, we develop a system called SCRIBE (SeCuRity Intent-Based Extractor), which decompiles security configurations from different network devices and translates them to an intent-level language called Nile. To demonstrate the feasibility of SCRIBE, we outline a case study and evaluate with dumps of real-world firewall configurations containing rules from various servers and institutions.

Keywords: Intent-Based Networking · Network management · Programming languages

1 Introduction

In Intent-Based Networking (IBN), an intent is a high-level abstract declaration written by network operators to specify the desired network behavior [1,2]. IBN helps operators configure the network by hiding unnecessary low-level details of the underlying network, inside an intent-aware network management system. In a recent effort, researchers have exploited specification languages that are closer to natural language to define and employ network intents. Nile [1] and Jinjing [3] are examples of such languages. Employing such solutions into the network can

L. Barolli et al. (Eds.): AINA 2020, AISC 1151, pp. 858–870, 2020.
https://doi.org/10.1007/978-3-030-44041-1_75

improve management and also facilitate the deployment of new features. However, current efforts to support IBN focus on top-down approaches, meaning that network configurations are first specified as intents and then refined to low-level configuration directives—*i.e.*, the network is assumed to support intents already. Network operators may still encounter several issues deploying new intents, such as reasoning about complex configurations to understand previously deployed rules before writing an intent to update the network state.

Network operators may encounter networks with missing documentation of previously deployed intents, becoming hard to understand the network behavior. In such cases, especially in large-scale networks, operators must read many configuration files to comprehend the network behavior. Large-scale networks may include various devices distributed in multiple hierarchical levels of their topology, each of these devices configured using low-level, vendor-specific languages. Thus, existing top-down approaches to specify intents are not straightforward to be implemented. Take, for instance, the configuration of a firewall: an operator must manually check several low-level firewall rules to extract a simple intent, such as "*block p2p traffic*".

There is a lack of solutions for interpreting low-level configuration directives and expressing them using intents. In the network security context, many work efforts are made to interpret the behavior of firewalls in the network by converting rules to high-level representations, such as tables [4], symbolic models [5], generic firewall languages [6], and graphs [7]. These representations help operators understand firewall rules by capturing low-level configuration details in a bottom-up approach and displaying firewalls rules in a vendor-independent form. However, these solutions display as output only *allowing* and *blocking* statements for IP addresses and ports, requiring operators to know the functioning of firewalls, protocols, and ports to understand the generated representation. Thus, we can observe a lack of strategies to bridge the existing gap between low-level deployed network rules and high-level network intents. An intent-level representation can help operators by reducing the necessary effort to consolidate network rules and reason about previously deployed intents.

Given the lack of strategies to express the behavior of already deployed network configurations in the form of intents, we propose a bottom-up methodology to represent network configurations in an intent-level language. We developed a prototype tool called SCRIBE (SeCuRity Intent-Based Extractor) to validate our method. SCRIBE interacts with tools that reverse engineer vendor-specific configurations to an intermediate level and aggregates the complementary information to provide a high-level representation. Finally, SCRIBE translates this representation into an intent-defined language, called Nile. We demonstrate the feasibility of our approach using a case study and an evaluation of the translation accuracy with real-world firewall dumps [5], containing firewall rules from various servers and institutions. Our results show that SCRIBE is able to express network configurations in an intent-level language with high accuracy, capturing the vast majority of the low-level details present in the input configuration files.

The remainder of this paper is structured as follows. In Sect. 2, we review related work. In Sect. 3, we specify an end-to-end methodology to extract intents from network configurations and we detail the implementation of SCRIBE. We describe two case studies based on realistic scenarios in Sect. 4. In Sect. 5, we conduct an evaluation of our system using real-world configurations, as well as a discussion of our findings. Finally, in Sect. 6, we conclude this paper with discussions and future work.

2 Related Work

In this section, we review related work. We consider as related work previous efforts that read low-level network configurations and express configurations in an intent-level language. Given the lack of strategies in this area, we review intent languages that (*i*) allow deploying network configurations using high-level directives; and previous efforts that (*ii*) read vendor-specific configurations from a restricted context and use high-level representations to display these configurations.

Jinjing [3] automatically generates ACL update plans based on operators' high-level intent. Jinjing offers an intent language named LAI, with high-level representation, which operators can use to express their in-network ACL plan updates. Janus [8] support policies with QoS requirements, mobility, and temporal dynamics. In a recent effort [1], authors provide a solution to allow users to express intents in natural language, using Nile as an intermediate representation, closely resembling the English language. Their approach reduces the need for operators to learn a new policy language for each different type of network. However, all these languages are top-down approaches, meaning that network configurations are firstly specified as intents and then refined to low-level configuration directives.

In the security context, some efforts have been made to parse low-level firewall rules and express them using high-level representations, such as graphs or tables. In a previous work [7], authors propose a solution to read vendor-specific firewall rules and represent them as a directed graph abstraction, where nodes correspond to the network devices, and directional edges represent a statement to allow or block traffic on a specific port. This graph abstraction helps users understand firewall configurations by displaying the network topology, allow and block statements in an easy-to-understand representation. However, this solution only supports simple allow and block policies in specific ports and hosts, and there is no support for distributed firewall configurations.

More recently, in [4], the authors propose FWS, a solution that reads firewall configurations from different systems and synthesizes them in a tabular form. To this end, authors provide a language to interact with FWS, where the user can load vendor-specific rules from different firewalls and queries for specific IPs, ports, networks, as well as compare them. The tabular representation helps operators understand firewall rules by centralizing all configurations in a single place. However, for distributed firewall configurations, this solution loads each

device configuration separated in a different table, each representing the firewall rules of a single device. Due to the use of various tables, operators can have difficulty to reason about previously deployed intents in a network-wide context.

In Net2Text [9], authors propose a solution to summarize the forwarding behavior in natural language. This solution provides interaction through a "chat-bot", where users can query for a specific forwarding behavior. While this work does an excellent job in summarizing the network-wide forwarding state, producing succinct reports in natural language, it is limited to forwarding behavior, not supporting ACL and NAT, for instance.

3 Methodology Overview

In this section, we describe a bottom-up methodology to parse network configurations and represent them into an intent-level language, depicted in Fig. 1. Our process expects as input configuration files exported from network devices (*e.g.*, firewalls, NAT, routers). Having these configuration files, in the step ①, we gather them in a centralized database, synthesize these configurations and express them using an intermediate representation. During the step ②, we extract the network entities and generate a platform-independent abstract model. In the step ③, we enrich the model with complementary information provided by various sources. The final result of this step is the Aggregated Model, an enhanced representation of the network configurations, which we call meta-intents. Finally, in the last step ④, we translate meta-intents into our extended version of the Nile language, which is described in Sect. 3.4.

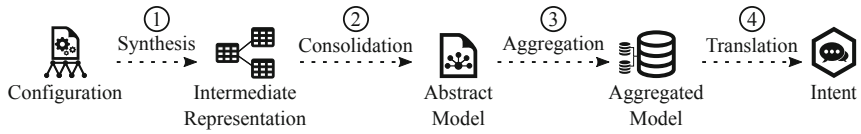


Fig. 1. Process of intent extraction

3.1 Configuration Synthesis

In the first step of our process – indicated in Fig. 1 by step ① – we collect low-level directives from various configuration files, which can be distributed in multiple levels of the network hierarchy, and synthesize them to an intermediate representation. In this step, the input our approach is composed of configuration files exported from various devices. After, we parse the configuration files and synthesize the configuration directives. This process must be low-level and accurate in order to create a trustworthy representation, which can reproduce all aspects of configuration and their interactions. As we leverage information from multiple different sources in the network, we exploit existing literature solutions

to extract such info in a structured fashion. However, as the existing solutions are not standardized, we have various representation types such as tables, graphs, or symbolic models. Thus, our solution must be able to parse multiple input sources. Aiming to parse these input sources, we design our synthesis methodology as an extensible module that can use many solutions to characterize network behavior. Our approach consists of gathering configuration files, generating input commands for the base solutions, and exporting the output of these solutions to process them in the next steps. How the base solutions are heterogeneous, we may have several outputs in this step, one for each network device. In the next step, described in Sect. 3.2, we detail how we gather these generated outputs and generate an intermediate representation.

3.2 Consolidation

After we have the intermediate representation, we can start to raise the abstraction level of the network model. This step is the most important phase of the process, once it develops a more manageable and structured view of the configuration. In this phase, we parse the intermediate representation and extract the entities needed for our reverse-engineering process.

Definition 1. *A network configuration is a composition of several entities. We define an entity as* $E = \{Target,\ Traffic,\ Actions_K\}$*, where:*

- *Target*: represents the target object of a determined rule. A Target can be specialized to represent network target objects in various scenarios. For example, if we model a firewall, we can model firewall Target entities by using source and destination prefixes ($Target :=< src.prefix, dst.prefix >$). A firewall prefix may represent a single IP (*e.g.*, $src.ip :=$ `10.0.0.1`) or a network IP prefix (*e.g.*, $src.prefix :=$ `10.0.0.0/8`).
- *Traffic*: represents the type of traffic in the network. For example, in firewall and NAT actions the Traffic entity represents the set of source and destination ports for filtering traffic, which will discern the traffic type in the network.
- *Action*: represents operations to be executed. An action will be applied to the rules, for a pair $< Target, Traffic >$. Each entity may have k actions, then $Actions = (action_1, action_2, ..., action_K)$. Each action contains parameters necessary to run the desired operation. Then $P = (p_1, p_2, ..., p_n)$ is the set of parameters of an Action i. As an example, we can represent a NAT forward operation in a $Action =$ {`forward`} and include the parameters $p_1 = \texttt{DNAT}_{\text{IP}}$ and $p_2 = \texttt{DNAT}_{\text{Port}}$, indicating the destination NAT IP and port for a specified traffic.

Finally, to group the entities which produces the same intent, we perform an additional processing phase. In this phase, we search through the extracted configurations and match pairs of entities by common characteristics using grouping functions.

Our grouping process consists of three applications of grouping functions in Target, Traffic, and Action entities, respectively. We define **grouping** as a

function of entities, $g : E \times E \mapsto E$, where we first group entities by the Target. Targets with the same origin and destination must be part of the same intent. Therefore we merge the entities that match these parameters. Formally, $\forall i, j \in E$, if $target_i \equiv target_j$ then we compute $g(i, j)$. Second, the grouping is performed by the entity Traffic, containing input and output ports. Formally, $\forall i, j \in E$, if $traffic_i \equiv traffic_j$ then we compute $g(i, j)$. In the last stage, we gather similar Action entities. If two actions have the same traffic type, we complement the traffic type by adding a parameter, indicating a variation in a set of similar characteristics of the Traffic. We consider as *similar*, a group of rules containing the same characteristics in which there is only a characteristic that differs them. This only different characteristic is considered the varying parameter. The varying parameter will be the input of traffic and stored separated from the entities, thereby reducing the total set of rules, aiming to create compact intents. Formally, $\forall i, j \in E$, if $traffic_i$ is similar to $traffic_j$ by the characteristic c, then $action.p := action.p \cup c$. All entities generated in this step will serve as input for the aggregation, which is described next.

3.3 Aggregation

In the third step of our process, we join all related entities to generate meta-intents. Meta-intents contains pairs of information, including the origin, destination, services, and the default action. Meta-intents are a more high-level representation of our process, much closer to an intent, and we represent them in JSON format.

We use complementary information from various sources to enrich the generated meta-intents and display the information at a user-friendly level. The purpose is to eliminate the redundant constructions and replace them with a smaller and clearer set of rules R. To enrich the meta-intents, we collect hostnames and topology as a piece of additional information from the network. Also, our solution provides an option to input "friendly" names for hosts, aiming to facilitate the recognizing of network devices. With all the complementary information gathered, we replace all IP addresses for the provided names. After, we infer the service or protocol names from provided ports. For this, we query the Service Name and Transport Protocol Port Number Registry [10], which lists the standard port for services.

3.4 Extending Nile

We use Nile as default language to represent our intents. The Nile language can express intents in an intent-level of abstraction, closely resembling the natural language, and satisfies our requirements of readability and abstraction. However, Nile natively does not contain symbols to support NAT behavior. In order to express NAT behavior, we extend the Nile grammar to support NAT directives. To do that, we include NAT constructs to Nile language. We add the `forward` keyword to represent a forward NAT action.

A Nile `forward` action requires a <*target*> symbol. Nevertheless, the actual Nile <*target*> symbol only allows representing groups, services, endpoints, and traffic. We can not represent a pair of IP address and port as a target for operation. To overcome this limitation, we create a new symbol <*address*> to express a pair of IP or host and a port. The syntax of the address symbol is <*address*> :: =< *IP/Host*, [*Port*] >, expecting an IP or host as a required parameter and a port as an optional parameter. If the port parameter is missing on intent, the system will assume the NAT port as the same as the input or output port.

```
define intent forwardIntent:
    from endpoint ('internet')
    to endpoint ('NAT Device')
    for protocol ('SSH')
    forward address ('10.0.0.5', 2222)
```

Listing 1. NAT Forward in Nile

In Listing 1, we provide an example using the Nile command `forward` and the *address* symbol. In this example, we represent a NAT action from incoming traffic. The incoming SSH traffic with the destination "NAT Device" will be redirected to the internal address 10.0.0.5 on port 2222.

3.5 Intent Translation

After enriching the model with complementary information, we process the generated meta-intents to translate them into our extended version of Nile. Nile grammar has several symbols to represent network rules, which can be directly translated from our generated entities. Nile grammar expects two *endpoints* as parameters, which can be hosts, middleboxes, and many other network devices. We map our arguments on *Target* entity and generate two Nile *endpoints*. The second phase of the translation process to Nile involves the mapping of *Traffic* entities to Nile *matches* symbol. The Nile language supports service, traffic, protocol and group traffic types, and, we also extend Nile allow the representation of addresses, which permit us to express our *Traffic* type in different levels of abstraction. In the last phase of the Nile translation process, we express our *Action* entities. The Nile grammar allows us to represent our actions using Nile *rules*, which permits us to represent actions like allowing, blocking, and add *middleboxes*, permitting us to represent several network functionalities supported by different types of middleboxes. We also extend the Nile language to support the operation forward, allowing us to represent NAT forwarding actions, as described in Sect. 3.4.

3.6 Implementation

We validate the feasibility of our methodology by developing a prototype solution called SCRIBE. The system leverages different input sources to process low-level firewall configurations and queries for all configurations available on filtering and NAT tables. SCRIBE has about 400 lines of Python code. In this section, we discuss the implementation of SCRIBE in detail as well as our modifications to third-party tools to make them interact with SCRIBE.

We divide the system into two different modules. In the first module (1), SCRIBE generates commands to load all configuration files in FWS [4]. We choose FWS due to the trustworthiness to generate the intermediate model. Also, FWS supports the most common firewall solutions for Linux/Unix systems, such as iptables, IPFW, packet filter (PF), and Cisco IOS ACL configurations. FWS process low-level firewall configurations files and synthesizes the network configuration using first-order logic predicates to characterize allowed policies in firewall and NAT forwards. In the second module (2), SCRIBE queries for all filtering and NAT rules in FWS. FWS then generates the result as a table, displaying all allowed connections in the network. We process the rules table generated by FWS and register them into our database. Then, we process these rules and complement with additional data to consolidade in our intermediate representation format, as explained in Sect. 3. Finally, we translate the meta-intents to the Nile language.

4 Case Study

Let us suppose that a new operator initiates at a company and wants to assert if the network behaves as expected and makes some changes. For this, the operator needs to discover all devices of the network and understand their respective configurations. In this situation, the network operator would need to (*i*) discover all devices in the network and their respective IP addresses, (*ii*) read the entire configuration from various network devices and (*iii*) provide some documentation to help understand the network behavior for the next alteration. To do these tasks, the operator needs to have platform-specific expertise, besides knowledge of network administrative tools.

Instead, the operator could rely on Scribe to understand the network state and then insert a new rule as needed. In this section, we analyze a Science DMZ case study indicating the applicability of our solution to extract intents from already deployed network configurations and demonstrate the effectiveness and technical feasibility of our proposed solution. For this purpose, we explore a Science DMZ scenario. Next, we discuss this scenario in detail.

4.1 Scenario - Science DMZ

In this case study, we explore a university scenario with Science DMZ. In this network, there are high-speed DTNs (Data Transfer Nodes) in a DMZ (Demilitarized Zone), a router with ACLs and an internal firewall. A DMZ can be used to isolate particular servers from the machines within a network. Figure 2, shows the scenario. In this network, the DTNs are connected directly to a router because of the high-speed transfers [11]. A stateful firewall can degrade the network performance and is not suitable to realize the transfer of huge amounts of scientific data in high-speed connections. For this, the ACLs (Access-Control Lists) are inserted directly in the router in a stateless manner, aiming to maximize the transfer speeds [11].

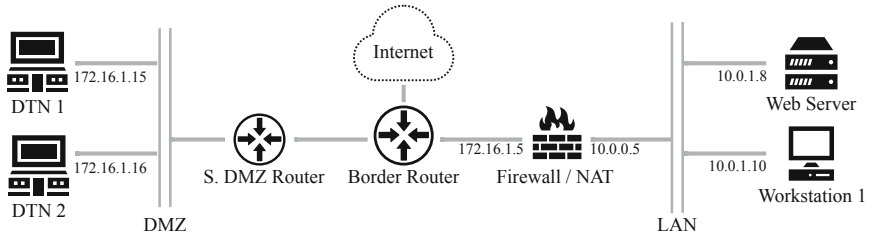


Fig. 2. Science DMZ network

We extract the network intents in a bottom-up approach. These intents hide the complexity involved to configure two separate devices to manage ACLs in a high-speed DMZ and a conventional stateful firewall managing internal connections. The ACL router begins with a default-blocked configuration, and it is possible to add network IP addresses and prefixes in a "whitelist", indicating what server is allowed to perform a high-speed connection with a specific DTN. NAT isolates the internal network, and the firewall device attends to translate external addresses to NAT addresses. The firewall rules deal with internal traffic, besides blocking suspected protocols, such as UDP or TCP for some specific internal hosts.

The output intents for this solution are shown in Listing 2. We observe a significant reduction in the number of lines of code. The original `iptables` code, available at our repository, has more than 40 lines of code, and the resulting Nile intents have 22 lines. Also, we have reduced the complexity by rising the abstraction level. From these generated intents listed in Listing 2, we can easily infer the base intents derived from this scenario. The two first intents (i) represents an allow intent between University A and the authorized DTN, labeled "DTN 1". The third intent of Listing 2 (ii) describes the behavior of allowing all types of traffic in the internal network. With the use of intuitive alias supported by SCRIBE, we can define in this example the alias "My Network", referring to the prefix 10.0.0.0/8 (prefix of the internal network). From intents `natIntent1` and

`natIntent2`, we can observe (*iii*) a NAT translation from all incoming traffic for the HTTP and HTTPS protocols, indicating that these types of traffic will be forwarded to the Web Server, which can only be accessed directly from the internal network.

```
define intent firewallIntent1:
    from endpoint ('University A')
    to endpoint ('DTN 1')
    allow traffic ('all')

define intent firewallIntent2:
    from endpoint ('University B')
    to endpoint ('DTN 2')
    allow traffic ('all')

define intent firewallIntent3:
    from endpoint ('My Network')
    to endpoint ('My Network')
    allow traffic ('all')

define intent natIntent1:
    from endpoint ('internet')
    to endpoint ('Firewall / NAT')
    for protocol ('HTTP')
    forward address ('Web Server')

define intent natIntent2:
    from endpoint ('internet')
    to endpoint ('Firewall / NAT')
    for protocol ('HTTPS')
    forward address ('Web Server')
```

Listing 2. Resulting Nile intents for the Science DMZ scenario

5 Evaluation

The evaluation of the bottom-up process requires a careful analysis of the intents, *i.e.*, we need to find ways to identify if the extracted intents represent the same configuration as the low-level configuration. In this section, we define and measure accuracy metrics collected from SCRIBE through several real-world network configurations.

To assert the feasibility of our bottom-up process, we evaluate two main aspects: (*i*) the accuracy of generated intents and (*ii*) SCRIBE support for the most present functionalities in real-world firewall dumps. To evaluate the accuracy of generated intents, we compare if the characteristics of the original configuration files are present in the extracted intents. We performed several

measurements using real-world security rules from a public collaborative repository [5], which contains dumps of firewall rules from various servers and institutions. For security concerns, some public IP addresses and MAC addresses are anonymized.

5.1 Translation Accuracy

We performed a static analysis to validate if each generated intent represents the configuration accurately. Each characteristic is a configuration element of a rule. For instance, source address, port number, and action (*e.g.*, allow) are characteristics of a traditional allow rule. We automatically to extract all firewall parameters of configurations. We use these extracted parameters to compare with parameters extracted from generated intents, in order to evaluate the accuracy.

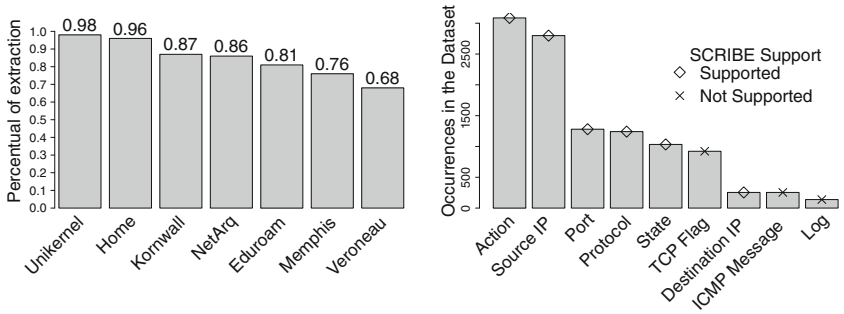
After, we dump the parameters of configurations present in generated intents. We then compare if the parameters dumped of low-level configurations are also present in generated intents. Thus, we calculate the accuracy by count how many parameters of the configuration files are also present in generated intents. Figure 3(a) presents the accuracy of the intents. The X-axis shows the firewall dump (with a compact name), and Y-axis represents the rate of correctly extracted configuration parameters, *i.e.*, the percentual of configuration parameters which are present in low-level configurations and also present in generated intents.

We can verify in Fig. 3(a) that the accuracy of generated intents is above 0.8 for most of the dumps, which represent that SCRIBE can extract intents from configuration files with high fidelity, missing very few configuration details. The worst case is Veroneau (a dump of *veroneau.net*) and contains several directives of ICMP messages and some directives of rate-limiting, both not supported for our solution yet. However, even for this dump, our solution can express most of the functionalities available in the firewall configuration. We can observe that we can represent high-level intents, closely resembling the natural language, and, nevertheless, maintain a high fidelity to the original configuration.

5.2 Functionality Support

We extract several properties from firewalls to recognize supported functionalities. For this, we extract all possible parameters from the firewall configuration files and gather all these parameters for posterior analysis. We carefully analyze the exported parameters to capture the meaning for the parameter, exploring for which functionality it belongs.

With the grasp of the functionalities, we extract these functionality parameters from several real-world configuration files. For this, we process all available configuration files and count the functionality parameters for each. After, we contrast all used functionality parameters in real-world configurations with security functionalities supported by SCRIBE. We display the functionalities results in



(a) Extraction percentage of correct intents for each firewall dump.

(b) Most used functionalities in security configurations and SCRIBE support for them.

Fig. 3. Evaluation of SCRIBE functionality support.

Fig. 3(b). We remove from the graphic all parameters with less than 30 utilizations, due to the little utilization in the dataset. Observing Fig. 3(b), it is possible to observe that SCRIBE supports the most used functionality parameters in real-world firewall configurations, and, due to this support, SCRIBE can represent the most used firewall functionalities at an intent-level.

Most of SCRIBE unsupported functionalities of firewalls refers to logging, TCP flags, and ICMP messages. SCRIBE also does not support rate-limiting functionalities yet. However, this functionality is rarely used directly in firewall configurations, according to our evaluated sources, and will be an object of study of future work.

6 Conclusion and Future Work

In this paper, we introduced a novel bottom-up approach to intent extraction, which synthesizes and represents existing network configurations in an intent-level. We developed a solution to validate the feasibility of our process, and we provide two case studies representing real-world scenarios where we can apply the process. Case studies evidence that we can extract intents from these scenarios, and represent the network behavior in intent-level, closely resembling the natural language. Additionally, we use real-world firewall configurations to evaluate the accuracy of our implemented solution. We find that it is possible to extract intents from real configurations using SCRIBE with high accuracy, preserving the majority of aspects of the original configurations in the translation process.

Although SCRIBE can extract intents with high accuracy, it still faces some limitations. In particular, we do not support firewall features of rate-limiting, logging, and ICMP messages. We suggest, as future work, including these functionalities into SCRIBE. For this, there is a need to add new features to FWS and also add new constructs in the Nile language. We also see as a perspective supporting routing algorithms, which also would require modifications into Nile and support from other synthesizers, besides FWS.

Acknowledgement. We thank CNPq for the financial support. This research has been supported by call Universal 01/2016 (CNPq), project NFV Mentor process 423275/2016-0.

References

1. Jacobs, A.S., Pfitscher, R.J., Ferreira, R.A., Granville, L.Z.: Refining network intents for self-driving networks. In: Proceedings of the Afternoon Workshop on Self-Driving Networks ser. SelfDN 2018, pp. 15–21. ACM, New York (2018). https://doi.org/10.1145/3229584.3229590
2. Scheid, E.J., Machado, C.C., Franco, M.F., dos Santos, R.L., Pfitscher, R.P., Schaeffer-Filho, A.E., Granville, L.Z.: INSpIRE: Integrated NFV-based Intent Refinement Environment. In: 2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM), pp. 186–194, May 2017
3. Tian, B., Zhang, X., Zhai, E., Liu, H.H., Ye, Q., Wang, C., Wu, X.: Safely and automatically updating in-network ACL configurations with intent language. In: SIGCOMM 2019 Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication (2019)
4. Bodei, C., Degano, P., Galletta, L., Focardi, R., Tempesta, M., Veronese, L.: Language-independent synthesis of firewall policies. In: 2018 IEEE European Symposium on Security and Privacy (EuroS P), pp. 92–106, April 2018
5. Diekmann, C., Michaelis, J., Haslbeck, M., Carle, G.: Verified iptables firewall analysis. In: 2016 IFIP Networking Conference (IFIP Networking) and Workshops, pp. 252–260, May 2016
6. Tongaonkar, A., Inamdar, N., Sekar, N.: Inferring higher level policies from firewall rules. In: Proceedings of the 21st Conference on Large Installation System Administration Conference ser. LISA 2007, Berkeley, CA, USA: USENIX Association, pp. 2:1–2:10 (2007) http://dl.acm.org/citation.cfm?id=1349426.1349428
7. Martínez, S., Cabot, J., Garcia-Alfaro, J., Cuppens, F., Cuppens-Boulahia, N.: A Model-driven approach for the extraction of network access-control policies. In: Proceedings of the Workshop on Model-Driven Security ser. MDsec 2012, pp. 5:1–5:6. ACM, New York (2012). https://doi.org/10.1145/2422498.2422503
8. Abhashkumar, A., Kang, J.-M., Banerjee, S., Akella, A., Zhang, Y., Wu, W.: Supporting diverse dynamic intent-based policies using janus. In: Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies ser. CoNEXT 2017, pp. 296–309. ACM, New York (2017). https://doi.org/10.1145/3143361.3143380
9. Birkner, R., Drachsler-Cohen, D., Vanbever, L., Vechev, M.: Net2Text: query-guided summarization of network forwarding behaviors. In: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18), Renton, WA: USENIX Association, pp. 609–623, April 2018. https://www.usenix.org/conference/nsdi18/presentation/birkner
10. Service Name and Transport Protocol Port Number Registry. https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml
11. Dart, E., Rotman, L., Tierney, B., Hester, M., Zurawski, J.: The science DMZ: a network design pattern for data-intensive science. Sci. Program. **22**(2), 173–185 (2014)

APPENDIX B — RESUMO

Intents são declarações abstratas em alto nível, escritas por operadores de rede para especificar o comportamento esperado da rede (BEHRINGER et al., 2015). O uso de *intents* ajuda os operadores a configurarem a rede usando linguagens de alto nível, dispensando o árduo aprendizado de diversas linguagens de configuração em baixo nível para cada equipamento. Em esforços recentes, o uso de linguagens de especificação de alto nível que remetem à lingua inglesa foi investigado na definição e implantação de *intents* de rede. Nile (JACOBS et al., 2018) and Jinjing (TIAN et al., 2019) são exemplos dessas linguagens. O emprego destas soluções na configuração de rede pode auxiliar no gerenciamento e possibilitar que novas funcionalidades sejam implantadas com facilidade. No entanto, os esforços atuais que utilizam *intents* focam em abordagens top-down, denotando que as configurações de rede devem ser primeiro especificadas como *intents* e então refinadas para diretivas de configuração de baixo nível, *i.e.*, assume-se que a rede já tem suporte a *intents*. Os operadores de rede ainda podem encontrar vários problemas ao implantar novos *intents*, tal como refletir sobre configurações complexas para entender as regras já implantadas antes de escrever um novo *intent*para atualizá-las.

Os operadores de rede podem se deparar com redes contendo pouca documentação das políticas de rede já implantadas, tornando difícil entender o comportamento da rede. Nesses caso, especialmente quando se trata de redes de larga escala, os operadores devem ler vários arquivos de configuração contendo diretivas de baixo nível para entender o comportamento esperado da rede. Redes de larga escala podem incluir vários dispositivos distribuídos em múltiplos níveis de sua topologia, cada um deles configurados através de linguagens especificas do fabricante. Desta forma, as abordagens *top-down* que usam *intents* não são simples de serem implementadas. Considera-se como exemplo a configuração de um firewall: um operador deve analisar manualmente vários arquivos que contêm regras de firewall em linguagens de baixo nível para extrair um intent simples, tal como “bloquear tráfego P2P (*Peer-to-peer*)”.

A abstração de *intents* oculta do operador os detalhes de baixo nível da infraestrutura subjacente da rede e permite que usuários e operadores configurem a rede usando *intents* de fácil compreensão, remetendo à linguagem natural. Há uma escassez de soluções que expressam o comportamento das configurações implantadas na rede em forma de *intents*. Alguns esforços, tais como Batfish (FOGEL et al., 2015) e ARC (GEMBER-JACOBSON et al., 2016) leem configurações de baixo nível de redes tradicionais e geram

modelos abstratos dos planos de dados e de controle. No entanto, essas soluções focam em técnicas de análise e verificação, objetivando verificar propriedades de protocolos de roteamento. Dessa forma, não proveem uma forma de expressar os modelos gerados em representações de mais alto-nível.

Dada a escassez de soluções para expressar o comportamento relativo às configurações já implantadas na rede, neste trabalho propõe-se um método *bottom-up* para expressar as configurações de rede existentes em nível de *intent*. A bordagem desenvolvida neste trabalho extrai *intents* a partir de arquivos de configuração exportados de dispositivos existentes na rede, realiza um processamento e no final expressa essas configurações em linguagem Nile, uma linguagem de definição de *intents* que remete à língua inglesa. Para provar o conceito, um protótipo chamado SCRIBE (*SeCuRity Intent-Based Extractor*) foi desenvolvido. O SCRIBE interage com ferramentas de terceiros que interpretam as configurações de rede específicas para cada dispositivo, combina a saída dessas soluções com informações complementares e, por fim, representa as regras atuais da rede em nível de *intent*.

A ferramenta desenvolvida recebe como entrada os arquivos de configuração de segurança que são importados de firewalls existentes. O SCRIBE então processa as regras de firewall e NAT e refina-as através do agrupamento por características similares (*e.g.*, pares de origem e destino, serviços de rede), gerando entidades intermediárias de configuração, chamadas de *meta-intents*. Por fim, os *meta-intents* são enriquecidos com informações complementares da rede e então traduzidos para a linguagem Nile. Como a linguagem Nile atualmente não tem suporte às diretivas de configuração de NAT, neste trabalho foram adicionados novas construções gramaticais visando a representação do comportamento de NAT em Nile.

Neste trabalho, a viabilidade da abordagem desenvolvida foi demonstrada através de dois estudos de caso e uma avaliação da acurácia da tradução. Os estudos de caso representam cenários realísticos de (1) uma infraestrutura típica de uma pequena companhia e (2) uma rede acadêmica que faz uso do padrão de projeto Science DMZ. Com estes estudos de caso, foi demonstrado que a solução desenvolvida pode descrever precisamente o comportamento da rede usando *intents* de alto nível, com a sintaxe próxima à língua inglesa. Além disso, a ferramenta SCRIBE foi avaliada usando *datasets* realísticos contendo regras de firewall de vários servidores e instituições. Os resultados mostram que o SCRIBE expressa as configurações de rede em nível de *intent* com alta acurácia, capturando a maioria dos detalhes de baixo nível presentes nos arquivos de configuração.