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, 9500 – Porto Alegre, 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 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 Section 2, we review related work. In Section 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 Section 4. In Section 5, we conduct an evaluation of our system using real-world configurations, as well as a discussion of our findings. Finally, in Section 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 "chatbot", 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 Figure 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 3.4.

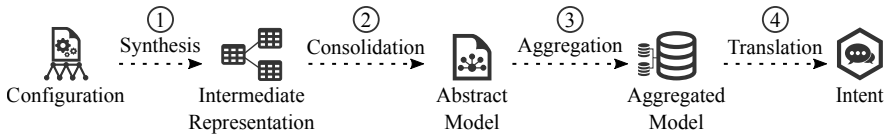


Fig. 1. Process of intent extraction

3.1 Configuration synthesis

In the first step of our process – indicated in Figure 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 Section 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 $Action = \{\texttt{forward}\}$ 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 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 Section 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 Section 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

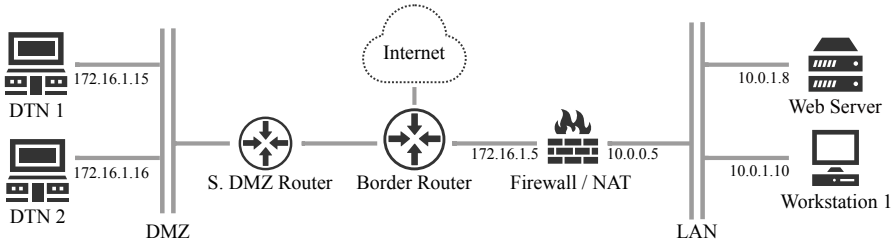


Fig. 2. Science DMZ network

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].

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.

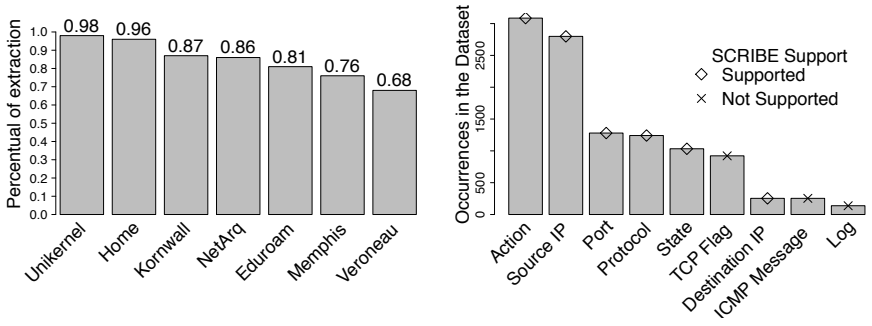
After, we dump the parameters of configurations present in generated intents. We then compare if the parameters dumped of low-level configurations are also present in generated intents. Thus, we calculate the accuracy by count how many parameters of the configuration files are also present in generated 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 Figure 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



(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.

by SCRIBE. We display the functionalities results in Figure 3(b). We remove from the graphic all parameters with less than 30 utilizations, due to the little utilization in the dataset. Observing Figure 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. A. S. Jacobs, R. J. Pfitscher, R. A. Ferreira, and L. Z. Granville, "Refining Network Intents for Self-Driving Networks," in *Proceedings of the Afternoon Workshop on Self-Driving Networks*, ser. SelfDN 2018. New York, NY, USA: ACM, 2018, pp. 15–21. [Online]. Available: http://doi.acm.org/10.1145/3229584.3229590
2. E. J. Scheid, C. C. Machado, M. F. Franco, R. L. dos Santos, R. P. Pfitscher, A. E. Schaeffer-Filho, and L. Z. Granville, "INSpIRE: Integrated NFV-based Intent Refinement Environment," in *2017 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, May 2017, pp. 186–194.
3. B. Tian, X. Zhang, E. Zhai, H. H. Liu, Q. Ye, C. Wang, and X. Wu, "Safely and Automatically Updating In-Network ACL Configurations with Intent Language," in *SIGCOMM '19 Proceedings of the 2019 Conference of the ACM Special Interest Group on Data Communication*, 2019.
4. C. Bodei, P. Degano, L. Galletta, R. Focardi, M. Tempesta, and L. Veronese, "Language-Independent Synthesis of Firewall Policies," in *2018 IEEE European Symposium on Security and Privacy (EuroS P)*, April 2018, pp. 92–106.
5. C. Diekmann, J. Michaelis, M. Haslbeck, and G. Carle, "Verified iptables firewall analysis," in *2016 IFIP Networking Conference (IFIP Networking) and Workshops*, May 2016, pp. 252–260.
6. A. Tongaonkar, N. Inamdar, and R. Sekar, "Inferring Higher Level Policies from Firewall Rules," in *Proceedings of the 21st Conference on Large Installation System Administration Conference*, ser. LISA'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 2:1–2:10. [Online]. Available: http://dl.acm.org/citation.cfm?id=1349426.1349428
7. S. Martínez, J. Cabot, J. Garcia-Alfaro, F. Cuppens, and N. Cuppens-Boulahia, "A Model-driven Approach for the Extraction of Network Access-control Policies," in *Proceedings of the Workshop on Model-Driven Security*, ser. MDsec '12. New York, NY, USA: ACM, 2012, pp. 5:1–5:6. [Online]. Available: http://doi.acm.org/10.1145/2422498.2422503
8. A. Abhashkumar, J.-M. Kang, S. Banerjee, A. Akella, Y. Zhang, and W. Wu, "Supporting Diverse Dynamic Intent-based Policies Using Janus," in *Proceedings of the 13th International Conference on Emerging Networking EXperiments and Technologies*, ser. CoNEXT '17. New York, NY, USA: ACM, 2017, pp. 296–309. [Online]. Available: http://doi.acm.org/10.1145/3143361.3143380
9. R. Birkner, D. Drachsler-Cohen, L. Vanbever, and M. Vechev, "Net2Text: Query-Guided Summarization of Network Forwarding Behaviors," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 609–623. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/birkner
10. Service Name and Transport Protocol Port Number Registry. [Online]. Available: https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml
11. E. Dart, L. Rotman, B. Tierney, M. Hester, and J. Zurawski, "The Science DMZ: A Network Design Pattern for Data-Intensive Science," *Scientific Programming*, vol. 22, no. 2, pp. 173–185, 2014.