NAIL: A Network Management Architecture for Deploying Intent into Programmable Switches

NAIL: A Network Management Architecture for Deploying Intent into Programmable Switches

Antonino Angi* ‡ Alessio Sacco * Flavio Esposito ‡ Guido Marchetto * Alexander Clemm †
* Department of Control and Computer Engineering, Politecnico di Torino, Italy
‡ Computer Science Department, Saint Louis University, USA
† Futurewei Inc., USA

***Abstract*—Programmable switches allow network operators to implement their customized network behavior. Despite its benefits, data-plane programmability has many practical challenges, including the ability to transfer intended behaviors on forwarding devices. Languages such as P4 represent a low level of abstraction, and corresponding programs are cumbersome to manage and configure for DevOps engineers, presenting a barrier to adoption. To facilitate such adoption, we propose NAIL, an architecture that allows network engineers to articulate desired network behaviors at a higher, more expressive layer and then translate such behaviors into executable code using a transpiler that acts as a network intent translator. NAIL can detect and properly instruct the network devices affected by the intent. Then, it offers continuous monitoring functionality to verify whether the intent is met. We demonstrate the effectiveness of our solutions with some use cases, showing the fast reaction to updates and the applicability of supplied intent.**

***Index Terms*—intent; network programmability; network architectures.**

I. Introduction

Recent advances in data-plane programmability have enabled users to inject rules into programmable switches, allowing network programmers to customize the network for their particular needs. These switches have distinct programmable architectures, among which the Programmable Independent Switch Architecture (PISA) stands out. PISA is a highly customizable pipeline architecture that is extensively employed in the telecommunications and network industries due to its ability to adapt to the specific requirements and needs of operators. The possibility of programming switches has launched networking into a new era of vendor-independent runnable software, providing network operators with greater flexibility, portability, and consistency. Researchers have shown that adopting PISA reduces costs and increases scalability when applied to highly demanding application layer systems, *e.g.*, Next Generation Firewall (NGFW) or Deep Packet Inspection (DPI) [1]. Different network programming approaches have been developed, protocol based - *e.g.*, using SRv6, where labels are interpreted as packet instructions or evolved MPLS, currently under standardization - but also data-plane based such as P4 and OpenFlow.

Data-plane programming languages such as P4 have caused quite a stir across the industry, as they allow network operators to program packet processing pipelines that used to be hard-coded, in the process allowing to implement custom packet processing behavior. However, it is also known that writing code in P4 is not easy as the language is limited by its internal resources, does not provide a rich abstraction, and debugging data plane programs is complicated, causing network programmers to engage in a series of "trial and errors" processes. Researchers have tried to make P4 code easier to write by defining data structures that allow a more elastic switch programming. As an example of this line of research P4All [2] stands out, where the authors presented an elastic data structure that automatically adapts to each switch's hardware resources, allowing more portable and modular modules. Another recent work, Lucid [3], presents a language that allows PISA switches to customize and control their data-planes with P4, thus allowing network management to users that are not familiar with network programming languages.

Despite such valuable improvements, data plane programming in general, and P4 programming in particular, is still considered challenging by many, and higher-level programming abstractions would alleviate the network programmability learning curve. To this aim, many authors have proposed systems that combine frameworks with low-level policy translators to automate the network as much as possible. At the same time, the concepts of Intent and Intent-Based Networking have been gaining traction, with *intent* being defined as a declarative set of goals and outcomes that a network should meet. An example of a possible intent could be: "Link utilisation in every link should be less than 70%" [4]. As a result, Intent-based networking does not only include proper rendering of intent as code on a per-switch basis, but also incorporates deployment aspects such as the "zero-touch networks" paradigm.

Following the intent paradigm, researchers have studied how to apply software methodologies to characterize and understand intent and have them applied to programmable switches. An example is P4I/O [5], where the authors implemented an Intent Definition Language with the goal of understanding and translating high-level intents to lower-level P4 code. While valuable, this study does not allow users to customize their network using an interactive API on-demand, but only through an Intent Definition Language. Researchers have also studied a way of translating high-level code into low-level rules using programming languages. This is the case of P4HLP, which uses a C-style high-level programming language, E-Domino, to generate P4 programs [6]. Another relevant work is Lumi [7], where the authors propose a system that allows users to ask questions about the network of a college in the

form of intents and have the specific intent translated and deployed in the data-plane programming language. While very interesting and valuable, the architecture behind it depends on a specific intent definition language (IDL) that, as the authors suggest, needs to be extended to work for different network environments. The emergence of new Large Language Models (LLMs) like ChatGPT [8] or Llama [9] has enabled the automation of translating high-level programming languages into low-level ones by prompting applications to generate code for a specific purpose. Nevertheless, these models invariably require human oversight to ensure the accuracy of the output and that it aligns with the input requests [10]. We experienced that ChatGPT generates code with functions that do not exist on original libraries and are only added by developers in a separate library branch. This has caused ChatGPT to propose programming codes that are not functioning if not with deep know-how and reverse-engineering integrating procedures.

Intersecting these efforts and leveraging such recent advances, in this paper, we present the design and prototype implementation of NAIL, a network management architecture that helps translate high-level intents to different lower-level data plane programming rules. Our solution is composed of a transpiler that converts human language-generated intents into data-plane programs, in general, and P4 (in this implementation) in particular, using natural language processing techniques, and a network management API, that contains the methods used by NAIL to manipulate the involved network elements. With NAIL, intents are translated into working programs, *e.g.*, P4 table rules, using our API, which, interacting with the NLP methods and databases, allows network programmers to customize their network according to some specific behavior (*e.g.*, load profiling, stateful firewall). To provide more flexibility, these rules can also be modified, and new ones can be added to the network or removed. Finally, NAIL allows the network programmer to collect statistics about demanded intents or a switch for troubleshooting purposes. We demonstrated NAIL's performance with some use cases, also focusing on the lines of code (LoC) and the updating rules reaction time. Our experimental results show that NAIL allows implementing any network intent in fewer LoCs than other state-of-art solutions, while fast reacting to any rule update.

II. Overall Architecture Design

This section describes the design of our network management architecture composed of a network management object model, a transpiler, and databases to help manipulate the intent.

A. Intent to data plane program workflow

When the network programmer specifies the intent, it can be in any natural language (*e.g.*, English, Italian, Chinese) since we developed our parser to be language-independent. However, the intent could also contain words that may be difficult to comprehend, bringing noise to our model and causing misunderstandings or wrong interpretations. For this reason, we developed our models using preprocessing techniques in combination with Natural Language Processing (NLP), a

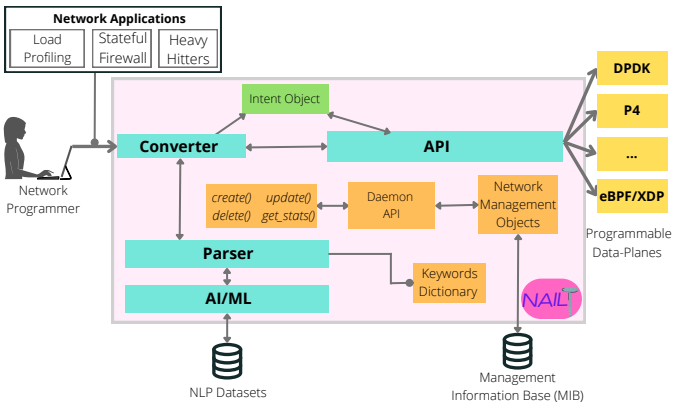


Fig. 1: Architecture Overview: intent is translated and injected into the programmable code of the network elements.

branch of Artificial Intelligence (AI) that deals with understanding natural language and extracting information needed for different cases. This involves a text-cleaning phase that removes all stop words and reduces them to their root form, effectively removing any noise and irrelevant information. The output of this NLP phase is a list of words, subsequently analyzed by our dictionary to determine the main objective of the inputted intent (*e.g.*, load profiling, firewall), and identifies all network components involved in such intent (*e.g.*, switches, links, and port numbers).

An overview of NAIL is shown in Fig. 1. Four main components form our architecture:

- Converter: this component includes NLP software libraries, connected with a parser and a dictionary to create intent objects;
- API: this includes all methods that allow NAIL intent object manipulation, i.e.,: (1) *create*, (2) *delete*, (3) *update*, (4) *get_stats()*;
- Management Information Base (MIB): this is a logically centralized database that maintains all NAIL states, including network elements, their ids, and subnets. This database is also used to store all the inputted intents, useful to perform modifications (*e.g.*, updates, deletion).
- NLP Datasets: these datasets are used by the NLP methods to perform the processing phases. They include data for stopword recognition and stemming techniques.

As shown in Fig. 1 (yellow blocks), our NAIL API can interact with different data-plane programming languages and frameworks, *e.g.*, P4, DPDK, or eBPF/XDP.

B. NAIL API: The User Perspective

NAIL allows network programmers to define an intent using the *create* method and manipulate it using three other operations: *delete*, *update* and *get_stats*. By using these four operations, network programmers can launch new instances of data-plane programs without knowing how to program in a specific data-plane programming language, empowering the *zero-touch* network paradigm.

Network programmers can define an intent using the *create()* API call:

$$\texttt{intent\_id} = \texttt{create}(\texttt{intent})$$

```
int_1 = "Block traffic from GroupA to GroupB"
intent_id = create(int_1)
#NLP techniques and keywords recognition
#Write into the switch's table for this action
int_2 = "Block traffic from GroupA to GroupC"
update(intent_id, int_2)
#To add a new intent update() calls create()
intent_id = create(int_1)
#To get statistics from an intent
get_stats(intent_id)
#To get statistics from a specific switch
get_stats(switch_id)
#To delete an intent
delete(intent_id)
```

Listing 1: A stateful firewall implementation with NAIL API (user perspective).

After being specified, the intent goes through a preprocessing phase using NLP methods and then an analysis with a keyword dictionary to identify the involved network elements. Finally, NAIL's interpreter selects the appropriate actions based on the intended network elements and primary objective. Such an objective could involve bypassing a particular switch, prioritizing traffic from a specific IP address, or blocking traffic between two specified groups. The return value of the *create()* method is a unique `intent_id`, corresponding to the intent object that contains the involved network elements and the network configuration states that have been added to satisfy the intent's goal.

Network programmers may retrieve statistics on a particular intent to troubleshoot the network by utilizing the *get_stats()* API, which accepts two optional parameters: $intent\_id$ and $switch\_id$.

$$\texttt{get\_stats}(\texttt{intent\_id}, \texttt{switch\_id})$$

The former parameter enables the collection of statistics related to a specific intent running on the network (*e.g.,* number of processed packets for each involved switch), while the latter is used to obtain information about a particular switch (*e.g.,* queue on ports, dropped packets).

To modify a previously added intent, it is possible to utilize the *update()* method. This function accepts a parameter that can be the $intent\_id$ of a previously added intent, or a new intent, which would invoke the *create()*, as before:

$$\texttt{update}(\texttt{intent\_id}, \texttt{new\_intent}).$$

If network programmers wish to delete a previously added intent, they can use the *delete()* API, which takes as a parameter the id of the created intent object:

$$\texttt{delete}(\texttt{intent\_id})$$

This function deletes all the network configurations that have been inserted with the intent and the corresponding intent object.

C. NAIL API: The Inner Workflow

To describe what happens in NAIL after the user inputs the desired intent, we start from the example described in Listing 1, where our API is used for requesting a stateful firewall.

After the network programmer has asked to deploy the intent for blocking traffic from a subnet group to another one via the *create()* method, NAIL runs a *pre-processing* phase. This step, utilizing NLP techniques alongside a pre-built dictionary containing regex and keyword recognition methods, separates the network elements associated with the intent and the intent's main goal. Such NLP preprocessing phase is composed of three main steps: *i)* The intent is converted into tokens resulting in "['block', 'traffic', 'from', 'groupa', 'to', 'groupb']". *ii)* We remove all the stopwords from the list of tokens, resulting in "['block', 'traffic', 'groupa', 'groupb']". *iii)* The remaining tokens are manipulated using the Porter stemmer, which brings each token to its canonical form. This third step does not change the content of our tokens as they already are in their canonical form. It is important to notice that these two last steps use the information contained in the NLP Dataset to retrieve the necessary words to remove canonical forms. These elements are then matched with the MIB database to identify them by finding relevant information (*e.g.*, id, number of ports, IP addresses). In the present scenario, the identified "group A" and "group B" correspond to two subnets of the topology with two diverse IP ranges. Then, to identify the intent goal, our API scans the tokens and matches them with our dictionary, identifying the keyword "block".

Concluded this preliminary phase, the *deploy* phase starts, and the program generates an intent object containing its unique incremental id (returned to the user), the involved network elements (subnets belonging to GroupA and GroupB), and the intent goal ("block traffic groupa groupb"). Subsequently, NAIL modifies a source template P4 code with minimal functionalities (*i.e.,* IPv4 forwarding), registers, and tables, with the blocking functions. This is done by injecting table entries into the switch connected to the subnet of groupB using two main methods from the P4Runtime library as described in Section III-A. While this approach allows reducing possible errors in compilation, we also deployed the *get_stats()* method to allow network programmers to verify the network behavior and perform two different actions: i) retrieve statistics from the customized network, ii) troubleshoot the network after the insertion of an intent to verify that the injected rules are correct.

III. Prototype Implementation and Evaluation

We evaluated NAIL by implementing several use-cases (described in Table I below) and choosing P4 as our data-plane programming language. To do so, we compiled the intent using the p4c compiler and adopted the behavioral model version 2 (bmv2) as the target for our software switch. Finally, we simulated and tested our P4-generated programs using a leaf-spine datacenter topology composed of 10 servers and deployed in Mininet, a network emulator that allows reproducing virtual networks.

A. NAIL Integration with P4Runtime

We chose P4 as our prototype implementation because of its compatibility with both software and hardware-based

platforms [1]. While P4 provides the potential to customize the network to suit diverse use cases, modifying the P4 program requires stopping the execution of the current program and restarting it with an updated one, which cannot be performed in real-time. For this reason, in NAIL, we adopted P4Runtime as a support to our API to improve the flexibility and the performance of the methods of whom NAIL is composed, *e.g.,* insert, delete and update intents. P4Runtime is an open-source API that enables the communication between a dataplane and a controller, allowing a single, standardized interface for controlling and monitoring the behavior of P4-programmable network devices [11]. Notably, although P4Runtime already provides a method to insert entries on P4 tables, it lacks methods for manipulating those entries after. For this reason, we added three more methods: to update and delete the added entries and to retrieve statistics. This has been shown to provide network programmers and administrators more flexibility and management of their network.

When NAIL finishes preprocessing a new intent and creates the intent object, it prepares the entry to be added using the *buildTableEntry* P4Runtime method which takes as parameters the table name, the action name, and the action parameters, taken from the intent object, in order to find the name of the table where the specified action and its parameters are. Finally, NAIL calls the *WriteTableEntry* P4Runtime method to add the entry to the table. The same procedure is visible when network programmers want to update or delete the rules of a previously added intent. In this cases, NAIL calls the *UpdateTableEntry* or the *DeleteTableEntry* function, respectively, and simply replaces the old entry with the new one or deletes the corresponding intent.

B. NAIL Prototype: Design and Evaluation Metrics

In evaluating our solution, we began by examining the lines of code (LoC) that comprise NAIL and comparing them to other relevant works. LoC serves as a simple yet effective software size validation metric, allowing us to more accurately assess the scale and complexity of our implementation [12]. For this reason, while writing the code for our use-cases, we report how many lines of code NAIL generated and compared it with the LoC generated by P4, P4All [13], and P4I/O [5].

As shown in Table I, in our implementation, we considered eight use-cases: an IPv4 forwarding, a stateful firewall, a load profiling, a heavy hitter detector, a DDoS attack detector, BeauCoup, PRECISION, and SketchLearn. BeauCoup is a system that monitors the network through queries, PRECISION is an algorithm that uses probabilistic recirculation to find top flows (*e.g.*, detect heavy hitters) on switches, SketchLearn uses multi-level sketches to identify flows that are statistically responsible for causing traffic conflicts. For these three network applications, the LoC is directly taken from [13]. An interesting observation is that for most applications, the internal code of which NAIL is composed is fewer lines of code than other intent-based architectures, such as P4I/O. Next to the internal LoC, we looked at how many LoC would take for network programmers to customize their network. This is shown in the "user perspective" column where we assumed that the network programmer uses at least all the methods *create(), delete(), get_stats(), update()* multiple times.

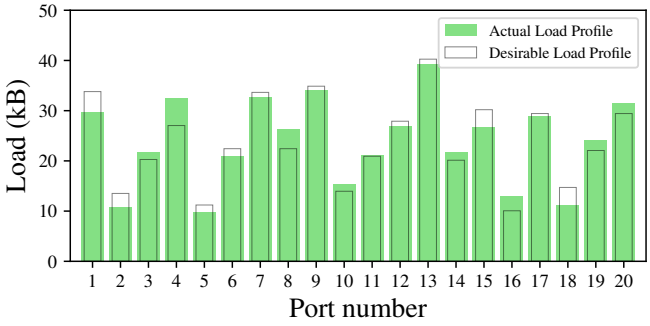


Fig. 2: Traffic follows the desired load profiling rule specified by the intent.

Table I also shows the "Installation time" column as the time needed for NAIL to customize the network after inputting an intent. In general, the process of translating an intent to code depends on two main factors: the intent needs to be translated into network policies (mapping the intent to specific network elements) and the code generation to implement the desired network policies. In the table, we can see that this value is relatively low for all network applications, showing how NAIL easily reacts to an intent and installs it into the network in only a few steps. Taking only a few seconds to parse and install network intents, we believe that NAIL can open up new opportunities for network programmers compared to the hours (or even days) that are needed to write P4 programs and then deploy them.

C. Load Profiling (LP)

Load profiling refers to the process of having different priorities for different traffic demands. Studies have shown how critical it can be to deploy an efficient load profiler, as the whole traffic optimization depends on it, especially when it is adopted in big topologies such as datacenters. By analyzing traffic patterns and associating a specific profile, the network administrator can identify bottlenecks, efficiently allocate resources and improve the network's overall performance. For this reason, researchers have studied different algorithms and ways of bringing load profiling into the data-plane of SDN programmable switches, using P4 as programming language [14].

In NAIL, we considered a use-case of a load profiler that sets different weights on each switch's port. To test our use-case and evaluate if our network respects the desired profile, we sent and received 1000 ICMP packets between two servers not belonging to the same leaf. Fig. 2 compared the generated load profile: the obtained one (in green) and the desired one (in white). It is noticeable that the actual load profile coherently respects the desired one throughout all ports of our switches.

D. Stateful Firewall (SFW)

We considered a stateful firewall (SFW) as a use-case scenario given the fact that nowadays companies and individuals widely use a firewall to protect their networks from unsolicited traffic, and researchers have introduced firewall applications in

	LoC in P4	LoC in P4All	LoC in P4I/O	LoC in NAIL (user perspective)	LoC in NAIL (internal code)	Installation time [s]
IPv4 Forwarding	197	217	416	5	241	0.558
Stateful Firewall	207	217	477	11	294	1.714
Load Profiling	294	286	N/A	8	305	1.784
Heavy hitters det.	316	N/A	477	6	298	1.762
DDoS Attack det.	233	N/A	477	9	298	1.883
BeauCoup	1500	541	N/A	10	320	1.707
PRECISION	283	266	N/A	9	297	2.487
SketchLearn	366	88	N/A	15	284	2.644

TABLE I: With NAIL several network applications can be implemented with a smaller number of lines of code (LoC) when compared to BeauCoup, PRECISION and SketchLearn in P4, P4All [2], P4I/O [5] and with NAIL both user perspective and internal code.

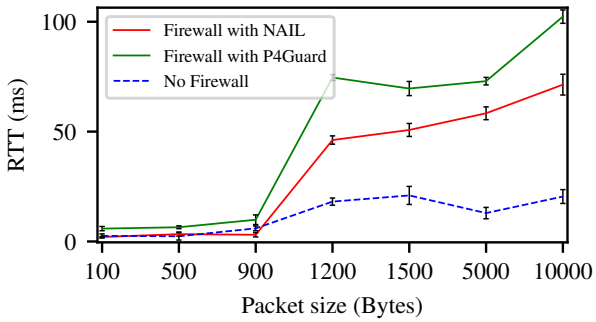


Fig. 3: RTT evolution for different packet size (in bytes) when a stateful firewall is installed.

their studies [1], [3]. As firewalls are used to categorize and filter traffic, we developed our solution in the data plane using a bloom filter: a probabilistic data structure mainly used for its fast computation time in small memory space. Although it may cause false positive filtering during the computation, the bloom filter is still widely adapted in various network security and privacy methods.

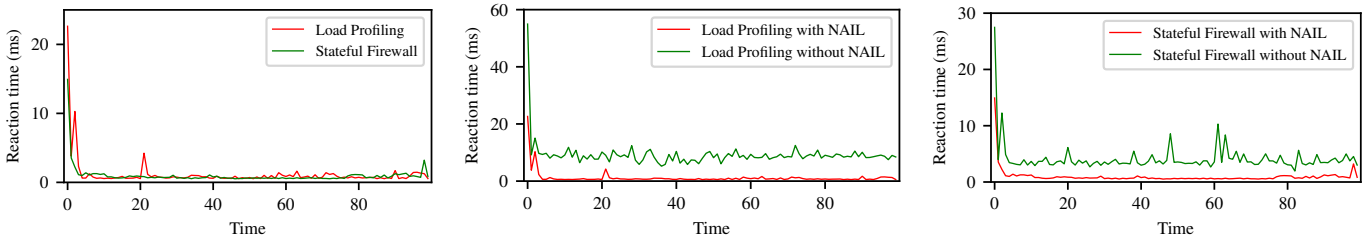
We implemented our SFW in the data-plane as it does not need constant interaction with the control-plane, which may cause delays and high response time. In this study, for each flow, our PISA switches compute a bloom filter hashing the 5-tuple composed of IP source & destination address, TCP source & destination port, and the protocol used. The results are stored in registers for a subsequent lookup function, which determines whether incoming packets are from the internal network or external and, in this latter case, discard them. Additionally, we track the number of packets passing through each switch, including those dropped, for statistical and troubleshooting purposes. One common problem in implementing a stateful firewall is the order in which rules are installed and evaluated in the device. If rules are not ordered correctly in a stateful firewall, it can lead to unintended consequences, *e.g.*, blocking desired traffic. In NAIL, we consider the intent specification time as a priority criterion and continuous monitoring as a way for the user to verify that rules are applied correctly and that desired security outcomes are achieved.

In Fig. 3, we evaluated how the round trip time (RTT) reacts when the packet size increases and compared it to another software firewall, P4Guard [15], and to a baseline case when there is no SFW installed. The figure shows that the three cases achieve almost the same performance when the packets are small (100-900 Bytes). However, when the packets are big (900-10000 Bytes), it is more visible that, despite using bloom filters, the firewall deployed with NAIL performs better than P4Guard. This shows that the implementation made by NAIL, despite a *zero-touch* data-plane coding from the network programmer, gives promising results. It is important to notice that P4Guard uses the $P4\_14$ version of P4 instead of the last one, $P4\_16$, which is known to perform better than the previous version. When our firewall is compared to a case with no firewall installed, we can notice a limited overhead that could be caused by the hash functions deployed in our firewall.

E. Updating Rules

Another important aspect of NAIL is the possibility of updating an existing table entry by just using the *update()* method. As mentioned earlier, NAIL can dynamically modify any forwarding table, adding new entries or even modifying existing ones, in response to changes in the network topology, traffic patterns, user needs, or other factors. This allows the network to adapt to changing conditions and optimize its performance in real-time without restarting the whole configuration. This dynamic modification of a table entry has to be performed as quickly as possible, since there might be situations in which a rerouting needs to be performed for a failed link or other reason, making the reaction time to an update a critical factor.

In NAIL we evaluated the reaction time of updating a table entry in Fig. 4, focusing on the applications that we deployed: load profiling (Fig. 4b), stateful firewall (Fig. 4c), and an overall reaction time when both applications are running (Fig. 4a). Looking at Fig. 4b, we evaluated the reaction time when the network administrator wants to update an existing rule on a table, modifying the weights that have been put on a specific switch's ports, and compared it to the traditional way of updating an existing rule: writing the new entry on a textual file and restarting the configuration of the network. From the figure, we can see that NAIL, thanks to the *update()* performed in real-time, always achieves lower reaction times (in ms) compared to a traditional entry update, allowing the network programmer to quickly modify the profile for its network according to its needs, failed links, traffic pattern or other factors. It is also visible that the highest reaction time is achieved when the function is called for the first time;



(a) (b) (c)

Fig. 4: Reaction time when an update() is requested for a load profiling and a stateful firewall application (a). Comparing the reaction time when an update() is performed with NAIL and without NAIL for a load profiling (b) and a stateful firewall (c).

meanwhile, the reaction time is even lower for all the next calls. The same behavior is noticeable in Fig. 4c, where a stateful firewall is installed in our network. Even in this case, we compared a stateful firewall rule updated with NAIL and a stateful firewall updated traditionally. It is visible from the figure that NAIL performs better and achieves up to 83.8% faster reaction time than the traditional update way when the *update()* is called for the first time, while still maintaining better performance when an update is requested other times. After collecting these results, we considered a situation when we have both applications running in our networks: load profiling and a stateful firewall; and the network programmer wants to update rules in both applications. The results are shown in Fig. 4a, where we computed the reaction time and, similarly to the case when only one of the applications is deployed, we can see that the reaction time is considerably low and achieves up to 22.6 ms in the case of update a rule for a load profiling, probably because of the different weights to install for each switch's port.

IV. Conclusion

In this paper, we presented NAIL, an architecture that translates network intents into programmable entities programs, with the aim of creating a management abstraction that is more flexible and simpler to manage even for users with limited technical expertise. While NAIL can be used for different programmable switches, here we focus on switches designed following the PISA architecture and on their default programming language, P4. We have evaluated our transpiler with known use-cases, demonstrating how different intent can be specified, ranging from prioritizing profiles to security issues detection.

That being said, we are far from being done, as considerable items for further work remain. For one, to simplify the task of network engineers further, NAIL could be complemented with automated deployment techniques, such as zero-touch Deployment (ZTD) to deliver continuous code updates. Secondly, the natural language interface can be improved by integrating NAIL and emerging chatbots to not only allow the user to input intent but to engage them in a dialog as needed to refine intent, inform of what intent the network can actually deliver, and negotiate issues such as the need to resolve potential conflicts between competing intent.

References

[1] F. Hauser, M. Häberle, D. Merling, S. Lindner, V. Gurevich, F. Zeiger, R. Frank, and M. Menth, "A survey on data plane programming with P4: Fundamentals, advances, and applied research," *Journal of Network and Computer Applications*, vol. 212, p. 103561, 2022.

[2] M. Hogan, S. Landau-Feibish, M. Tahmasbi Arashloo, J. Rexford, D. Walker, and R. Harrison, "Elastic Switch Programming with P4All," in *Proceedings of the 19th ACM Workshop on Hot Topics in Networks (HotNets)*, 2020, pp. 168–174.

[3] J. Sonchack, D. Loehr, J. Rexford, and D. Walker, "Lucid: A language for control in the data plane," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, 2021, pp. 731–747.

[4] A. Leivadeas and M. Falkner, "A Survey on Intent-Based Networking," *IEEE Communications Surveys & Tutorials*, vol. 25, no. 1, pp. 625–655, 2022.

[5] M. Riftadi and F. Kuipers, "P4I/O: Intent-based Networking with P4," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 438–443.

[6] Z. Hang, M. Wen, Y. Shi, and C. Zhang, "Programming Protocol-Independent Packet Processors High-Level Programming (P4HLP): Towards Unified High-Level Programming for a Commodity Programmable Switch," *Electronics*, vol. 8, no. 9, p. 958, 2019.

[7] A. S. Jacobs, R. J. Pfitscher, R. H. Ribeiro, R. A. Ferreira, L. Z. Granville, W. Willinger, and S. G. Rao, "Hey, Lumi! Using Natural Language for Intent-Based Network Management," in *USENIX Annual Technical Conference*, 2021, pp. 625–639.

[8] Introducing ChatGPT. Accessed: 2023-5-8. [Online]. Available: https://openai.com/blog/chatgpt

[9] Introducing Llama: A foundational, 65-billion-parameter language model. Accessed: 2023-5-8. [Online]. Available: https://ai.facebook.com/blog/large-language-model-llama-meta-ai/

[10] A. Borji, "A categorical archive of chatgpt failures," *arXiv preprint arXiv:2302.03494*, 2023.

[11] P4Runtime Spec. Accessed: 2023-5-8. [Online]. Available: https://p4.org/p4-spec/p4runtime/main/P4Runtime-Spec.html

[12] H. Zhang, "An Investigation of the Relationships Between Lines of Code and Defects (ICSM)," in *2009 IEEE International Conference on Software Maintenance*, 2009, pp. 274–283.

[13] M. Hogan, S. Landau-Feibish, M. T. Arashloo, J. Rexford, and D. Walker, "Modular Switch Programming Under Resource Constraints," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2022, pp. 1–15.

[14] A. Angi, A. Sacco, F. Esposito, G. Marchetto, and A. Clemm, "Howdah: Load Profiling via In-Band Flow Classification and P4," in *2022 18th International Conference on Network and Service Management (CNSM)*. IEEE, 2022, pp. 46–54.

[15] R. Datta, S. Choi, A. Chowdhary, and Y. Park, "P4Guard: Designing P4 Based Firewall," in *2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 2018, pp. 1–6.

Antonino Angi received his M.Sc. degree in Computer Engineering from Politecnico di Torino, Italy in 2020, and he is currently a Ph.D. student at the same university. His research interests include network architecture and management protocols, machine learning for computer networks, and data-plane programmability.

Alessio Sacco is an Assistant Professor at Politecnico di Torino, Italy. He received the Ph.D. degree in computer engineering from the same university in 2022. His research interests include architecture and protocols for network management; implementation and design of cloud computing applications; algorithms and protocols for service-based architecture, such as Software Defined Networks (SDN), used in conjunction with Machine Learning algorithms.

Flavio Esposito is an Associate Professor with the Department of Computer Science at Saint Louis University (SLU). He received an M.Sc. degree in Telecommunication Engineering from the University of Florence and a Ph.D. in computer science from Boston University. Flavio's main research interests include network management, network virtualization, and distributed systems.

Guido Marchetto received a Ph.D. degree in computer engineering from the Politecnico di Torino, in 2008, where he is currently an Ass. Prof. with the Department of Control and Computer Engineering. His research topics cover distributed systems and formal verification of systems and protocols. His interests also include network protocols and network architectures.

Alexander Clemm is a Distinguished Engineer in Futurewei's Future Networks and Innovation Group in Santa Clara, CA. He has been involved in networking software and management technology throughout his career. He has served on the Organizing Committees of many management and network softwarization conferences. He has around 50 publications, 50 issued patents, and several books and RFCs. He holds an M.S. in computer science from Stanford University and a Ph.D. from the University of Munich, Germany.