POLITECNICO DI TORINO
Repository ISTITUZIONALE

A Framework for Verification-Oriented User-Friendly Network Function Modeling

Original
A Framework for Verification-Oriented User-Friendly Network Function Modeling / Marchetto, G., Sisto, R., Valenza, F., Yusupov, J.. - In: IEEE ACCESS. - ISSN 2169-3536. - 7:(2019), pp. 99349-99359. [10.1109/ACCESS.2019.2929325]

Availability:
This version is available at: 11583/2749814 since: 2019-09-05T08:02:26Z

Publisher:
IEEE-INST ELECTRICAL ELECTRONICS ENGINEERS INC

Published
DOI:10.1109/ACCESS.2019.2929325

Terms of use:

This article is made available under terms and conditions as specified in the corresponding bibliographic description in the repository

Publisher copyright

(Article begins on next page)

22 November 2026

IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received June 19, 2019, accepted July 9, 2019, date of publication July 17, 2019, date of current version August 7, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2929325

A Framework for Verification-Oriented User-Friendly Network Function Modeling

GUIDO MARCHETTO, RICCARDO SISTO, FULVIO VALENZA, AND JALOLLIDDIN YUSUPOV
Dipartimento di Automatica e Informatica (DAUIN), Politecnico di Torino, 10129 Turin, Italy

Corresponding author: Fulvio Valenza (fulvio.valenza@polito.it)

This work was supported in part by the European Commission, under Grant Agreement no. 786922.

ABSTRACT Network virtualization and softwarization will serve as a new way to implement new services, increases network functionality and flexibility. However, the increasing complexity of the services and the management of very large scale environments drastically complicate detecting alerts and configuration errors of the network components. Nowadays, misconfigurations can be identified using formal analysis of network components for compliance with network requirements. Unfortunately, formal specification of network services requires familiarity with discrete mathematical modeling languages of verification tools, which requires extensive training for network engineers to have the essential knowledge. This paper addresses the above-mentioned problem by presenting a framework designed for automatically extracting verification models starting from an abstract representation of a given network function. Using guidelines provided in this paper, vendors can describe the forwarding behavior of their network function in developer-friendly, high-level languages, which can be then translated into formal verification models of different verification tools.

INDEX TERMS Network function modeling, model extraction, NFV.

I. INTRODUCTION

The development and deployment of software-centric, high-performance and automation networks, as well as processes and services, supported by breakthrough digital technologies, have become key to the future development of telecom operators worldwide. The innovative trends of Network Function Virtualization (NFV) [1] and Software Defined Networking (SDN) have opened up new business models, enabling telecom providers to lease or share their physical resources, increase the flexibility and controllability of the network infrastructure. SDN separates the network data and control planes to introduce (logically centralized) control plane programmability for novel networking virtualization (abstraction), simplified network (re)configuration, and policy enforcement. NFV, on the other hand, targets at virtualizing servers and integrates network hardware devices into a general-purpose x86-based server or other hardware platform through virtualization technology.

The ETSI (European Telecommunications Standards Institute) Industry Specification Group (ISG) [2] is the standardization initiative regarding the NFV domain, aiming at specifying a reference architecture. However, the way VNFs (Virtual Network Functions) are written is not standardized. As a result, the telecommunications world is filled with several different VNF implementations by different vendors. It is worth noting that there is an adopted standard specification based on TOSCA (Topology and Orchestration Specification for Cloud Applications), which is written in YAML (YAML Ain't Markup Language) language. However, it is only intended to describe topology, orchestration tasks, and services of Cloud Applications.

The problem is that there is no standard way to instantiate, configure, and operationalize these VNFs from multiple vendors, which increases the impact of possible network configuration errors. This in turn requires a substantial amount of effort to ensure networks' correctness, safety, and security. One solution for reducing errors and building robust infrastructures, is to check network software for bugs and verify its correctness prior to deployment. With this respect, great progress has been made recently in verifying network correctness both in data plane and control plane, with the help of formal methods and verification tools ([3]–[7]).

Formal methods require the creation of an abstract model of a system in a tool-specific modeling language, targeted

The associate editor coordinating the review of this manuscript and approving it for publication was Shagufta Henna.

 This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see http://creativecommons.org/licenses/by/4.0/

at a specific type of problem, and captures the semantics appropriate for the problem. It's a well known fact that formal methods don't prevent errors or eliminate bugs that go below the formal model of the system. In other words, incomplete or wrong functional coverage of specifications leaves the door open to errors resulting from not checking all properties mandated by the specifications. However, it is out of scope of our study to investigate these well known considerations. Instead, we address the main challenge providers of NFV software facing in order to enable formal verification of virtualized networks is the model construction: there is a large semantic gap between the artifacts produced by software developers and those accepted by current verification tools. For instance, the growing body of work on data-plane verification, such as NOD [3], SymNet [8] and VeriGraph [9] have evolved to a mature state in the last decade, but this gap might be a significant hurdle for their wide adoption in real production environments. Essentially, these tools are based on a complex modeling technique, tend to lock the user into a single kind of checking technology, require to accurately model network functionality, which requires an expert input, and usually oblige engineers to learn a whole new language (e.g., Alloy in [10]).

We recognize this gap and plan to address it by means of available languages that are easier to program with. In this paper we present a framework for a user-friendly VNF modeling that developers can use to provide a formal description of their network devices to be used in a verification process. Instead of modeling every detail of a NF, we focus only on the forwarding behavior, in order to enable the formal verification of typical reachability-oriented properties (e.g. isolation or absence of forwarding loops). This is also due to the fact that modeling every detail of a NF, by automatically synthesizing the source code, is practically unfeasible and does not scale for large networks. Our previous work in [11], concentrated solely on a specific verification environment. However, in this work, our goal is to define an abstraction model that is compliant with wide variety of verification tools.

As the main strength of our approach we see its simplicity and we realize it by addressing these goals:

i. To simplify the definition of a network function forwarding model by means of high-level languages.
ii. To offer a sufficient level of flexibility to developers in such a way that they could define the desired behavior for all their network functions.
iii. To provide an automatic translation from the function model definition into an abstract formal model for verification tools.

To address these challenges, we propose a typical set of high-level operations commonly used for describing the network function's forwarding behavior. With the introduction of a base class definition of a simple VNF, a network function developer can easily extend the provided artifacts to inherit basic properties, data types and methods and customize function behavior. Our framework also provides a parser that analyzes the source code and produces an abstract formal model of the VNF in a platform-independent XML specification. This intermediate XML-based model is fairly general to be translated to a multitude of different verification tools.

In this paper, VeriGraph [9] and SymNet [8] are adopted as a use case and Java as a programming language. However, we also show that the forwarding behavior can be written in most of the well-known programming languages and the output of the parser is generic, which can be translated into any verification tools of network forwarding behavior.

The remainder of this paper is organized as follows. We introduce the problems and summarize limitations of existing approaches in Section II. Section III presents an overview of the framework describing each component in detail. In Section IV we describe the implementation details and validation results. Finally, we present the conclusions and perspectives of future work in Section V.

II. MOTIVATION AND RELATED WORK

Currently there is no standard and widely known modeling language that can be used to accurately represent the forwarding behavior of network functions. Most of the research efforts related to network function models are focused on network verification and gained popularity in the verification community. In this section we list the open problems that we have encountered while looking at the proposed network function models in the verification context.

A. EASE OF USE

Modeling of network functions is effective for various uses, ranging from finding scalability issues in applications to finding network configuration bugs, especially with the use of formal verification tools. On the other hand, former modeling of network functionalities is challenging and requires detailed understanding of the specific verification tool internals, semantics, and modeling language.

With this problem in mind, introduction of an automated approach to generate models eliminates the necessity of having detailed knowledge in the formal verification domain and helps engineers to quickly determine the behavior of services comprising different types of network functions, starting from a more user-friendly description of the involved network functions. Notice that it is well known how formal methods do not prevent possible errors in constructing the formal model. In other words, incomplete or wrong models may lead to errors in the verification process. However, the investigation of these well known issues is out of scope of our study. Instead, given the fact that formal methods are a widely accepted methodology for property verification, we address the challenge of giving to NFV software providers the possibility to constuct formal models by meas of a programming paradigm they are familiar with. This would significantly lower entry barriers to these powerful verification approaches, somehow also reducing the probability of introducing errors in the model with respect to the utilization of complex formal languages.

Imperative languages such as Java, Python, C++ focus on describing *how* a program operates. A network function developer can write a code that describes in exact detail the forwarding decisions that the network function must make when a packet is received from one of its interfaces as a sequence of steps, without having the complexity of a function implementation. On the contrary, declarative languages used in logic-based formal verification tools do not specify a step or sequence of steps to execute, but rather predicates that must hold. The conceptual gap between these two paradigms is the vital challenge solved by our approach.

B. SUPPORT OF STATELESS AND STATEFUL FUNCTIONS

The existing verification methods can be divided into two groups according to their ability to model stateful functions. The former group focuses on modeling the forwarding behavior of stateless devices(e.g., switches and routers [12], ACL (Access Control List) Firewall [13], simple load-balancer [14]); by this we mean that the behavior is not modified until the control plane explicitly changes the configuration and there is no record of previous interactions. The latter group also considers the devices that are dynamic, in which every packet that the network device receives may alter the internal state, and the output is dependent on the sequence of previously encountered packets. Considering the fact that a significant portion of network devices are stateful, such as learning firewalls, load balancers, intrusion detection systems and the like, one cannot ignore these devices when verifying network configurations. Thus, we propose a general template to model, that covers a wide range of network functions, including both stateless and stateful ones.

C. RELATED WORK

There are two categories of related work to be considered. The first category relies on program analysis of available source code. In the past few years, there has been significant amount of work was done to provide a proper support for the translation of software system specifications to the input models of verification tools. Some of these tools including Bandera [15] and JavaPathFinder [16] have also developed tools to extract models. The two approaches are based on model checking, and the models they extract are models of Java software. We note these works solve different problems from our work, where they consider general-purpose Java programs and their main target is the identification of programming errors and bugs. In contrast to these works, we consider only the forwarding behavior of network functions and we are not interested in all the details of the network function's code execution. We also want formal verification of network configurations to be extremely fast, because it has to be performed in real-time when a network change occurs. For this reason, we need to extract customized, domain-specific models.

A proposal more similar to our target is NFactor [17], which provides a solution to automatically analyze the source code of a given network function to generate an abstract forwarding model, motivated by network verification applications. While relying on advanced tools([18]) and techniques([19], [20]) from the program analysis community, they do not require a specific structure of the source code of the function to be analyzed. This feature is considered as an advantage from a generality point of view. Unfortunately, creating a model that captures all code paths of a network function is challenging, because the state processing may be hidden deep in the code. This may cause the analysis to miss certain state changes. For example, implementations might use pointer arithmetic to access state variables, which is difficult to trace, and NFactor does not seem to deal with these language features appropriately. Another limitation of the approaches based on extraction of models from source code is that the code of many network functions is proprietary.

Another category of approaches and methods for static network analysis is based on hand-written function models ([3], [8], [9], [21]). For instance, Network Optimized Datalog [3] requires a Datalog both for network and function models and policy specifications. BUZZ [21] relies on manually written models of network functions defined in a domain specific language. As stated above, modeling network functionality for using these tools is challenging and requires detailed understanding of the verification tool semantics. Hence, our automated approach to generate models eliminates the necessity of having detailed domain knowledge and helps network engineers to quickly determine the behavior of a network function. On the contrary, SymNet [8] describes models using an imperative, modeling language, known as Symbolic Execution Friendly Language (SEFL). While the way this language has been designed has similarities with the modeling technique we propose, this approach lacks the idea of ease of modeling, by introducing a new language. Even though the authors provide parsers to automatically generate SEFL models from real network functions, this generation only covers routers and switches. Our approach, instead, relies on the well-known user-friendly programming languages and can be used to describe any kind of virtual network function.

III. MODELING TECHNIQUE

The proposed framework provides a library, a parser, and a translator. In the library we show the main design principles needed for modeling network functions by means of an imperative language. The parser then automatically generates abstract models from these descriptions. Basically, the parser takes as an input the definition written using our library and produces an abstract formal model describing the forwarding behavior of the network function. The translator finally translates this abstract model into a more high-level, domain specific constraint language, which would be difficult to deal with manually. The idea is that different translators can produce the input language of different verification tools.

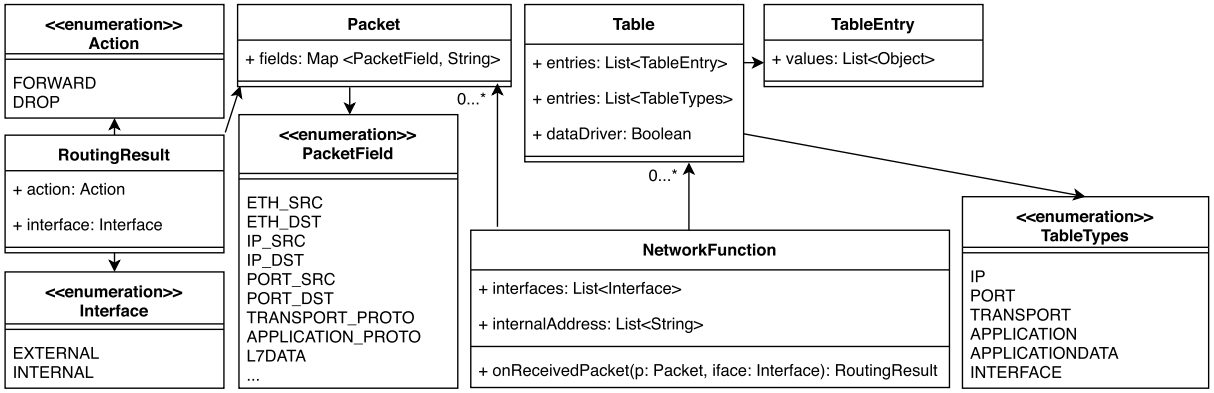


FIGURE 1. Class diagram of the library.

```
1  AclFirewall extends NetworkFunction
2      Table aclTable
3      aclTable = new Table(2, 0)
4      aclTable.setTypes(Table.TableTypes.Ip, Table.TableTypes.Ip)
5
6  RoutingResult onReceivedPacket(Packet packet, Interface iface)
7      if ( iface.isInternal() )
8          TableEntry entry = aclTable.matchEntry(packet.PCKField.IP_SRC,packet.PCKField.IP_DST)
9          if(entry!=null) return new RoutingResult(Action.DROP,null,null)
10         return new RoutingResult(Action.FORWARD, packet, externalInterface)
11     else
12         TableEntry entry = aclTable.matchEntry(packet.PCKField.IP_SRC,packet.PCKField.IP_DST)
13         if(entry!=null) return new RoutingResult(Action.DROP,null,null)
14         return new RoutingResult(Action.FORWARD, packet, internalInterface)
```

FIGURE 2. Pseudo code of the behavior of the ACL firewall in response to a received packet.

A. LIBRARY

The library provides the basic building blocks depicted in FIGURE 1 to write a network function. Following these design principles, a developer is able to easily characterize the forwarding decisions of the network function by instantiating objects of the library classes and by calling certain methods that correspond to typical operations performed inside network functions, and using the basic syntax of high-level programming languages.

The definition of a network function takes the form of a class that extends the library class `NetworkFunction`, which is the main class of the library. It is an abstract class whose abstract methods have to be implemented in the concrete extension provided by the user. The code snippet given in FIGURE 2 is an example for an ACL (Access Control List) enabled stateless firewall. The main method of the class (`onReceivedPacket()`) gets two parameters. The first one represents the incoming packet while the second one specifies the interface on which the function has received the packet. The actions that can be inserted inside `onReceivedPacket()` are divided into the following categories: instructions to get, set or check the contents of a packet field, instructions to store and retrieve a value into a lookup table defined by the user, instructions to define the forwarding action to be performed on a packet (through a `RoutingResult`). `RoutingResult` is the return value type of the `onReceivedPacket()` method. It represents the routing decision of the network function, after processing of the incoming packet. Its constructor receives following input parameters:

- A packet object that the network function produces.
- The action to perform on this packet (forward or drop).
- The forwarding direction (i.e. the interface the packet is forwarded to in case of forward action).

Taking into account that we are focusing on forwarding behavior of the network functions, we make an assumption that the forwarding action involves a single packet in response to a received packet. While this limitation could be removed by extending the `RoutingResult` definition, it does not prevent us from verifying reachability or isolation properties in any case. In fact, in the end, our model has to enable the evaluation of the possibility or impossibility for a function to send a certain type of packets on an interface, rather than representing exactly how many packets of a certain type it can send.

The `Interface` class models a logical interface on which a network function will receive or send packets. A typical need when writing network function models is to

distinguish different sets of interfaces. For example, a NAT (Network Address Translator) network function divides the network into two areas (an internal area and an external one) and applies different rules on the incoming packets if they are received from the internal or the external interface. Incoming packets from the external interface are forwarded to their destination only if a connection already exists while packets from the internal interface are always forwarded unless the NAT runs out of ports. Similarly, most of the network functions such as firewall, NAT and others differentiate the interface to which packets can be transferred, from the interface from which packets arrive. We will refer them as external and internal interface respectively in this paper.

The `Packet` class models an IP packet and gives access to the fields of the IP header that are relevant for the forwarding behavior of the network function. As can be seen from FIGURE 1 the `enum` type `PCKField` contains the constants representing the packet header fields that can be modeled. The string representation of these fields is chosen for convenience and arbitrary (raw) IP packets can always be represented as a string. Moreover our framework can support TCP and application protocols. For the moment, only HTTP and POP3 are supported, for demonstrative purposes. This list can be extended indefinitely to include more protocols when necessary. It is worth noting that the `clone()` method of the class is convenient in case of packet modifications. This in turn allows to modify the ''clone'' in the meantime keeping the old packet unchanged.

The `Table` class models a typical lookup table as a collection of `TableEntry` objects stored by the network function. For example, in the case of NAT-enabled network function, an object of this class stores the pairs of source/destination IP address/port of open connections. Alternatively, a table object can be used in an ACL firewall to store port numbers or IP addresses to be blocked. The `matchEntry()` method of the class on the other hand, serves to retrieve an entry from the table matching the value of the object. The `TableEntry` class itself contains a list of objects whose size is set according to the integer that the constructor receives. Moreover it is necessary to define the expected field type the table stores. This helps the parser to observe the type of the entry being retrieved from the table and extract the model accordingly. For instance, the element for IP source address and destination addresses is stored as an IP data type in the XML notation of the model. The list of currently supported types is given in `enum` type `TableTypes` (see FIGURE 1).

To grant the second objective of our work, to offer a greater flexibility, developers are provided with sufficient level of freedom to follow the guidelines and ''skeleton'' of the model mentioned above as building blocks of complex network functions. In particular there is no restriction on the number of tables, conditions and on the actions that follow these conditions, on the order of packet operations and the programming language used to describe these models. However, our framework supports only a subset of the object-oriented programming language features, which are supported by C++, Java, Objective-C and others. The list of supported features is shown in TABLE 1. While we believe this subset is enough for our purposes, it can be extended in order to further improve the user experience when specifying network functions.

TABLE 1. List of supported features of programming languages.

Data types: int, String, boolean, byte
Boolean operators
Comparison operators
Statements: return, if, if-else, variable assignment, method invocation

B. PARSER

The parser analyzes the class describing the virtual network function and generates a final formal model. The analysis part consists of the following activities:

- the identification of the instructions in the code that lead to a packet being sent through an interface;
- the identification of the instructions related to state insertion (write) and retrieval (read);
- the identification of the conditions (IF statements) that are traversed to reach the above mentioned instructions.

Finally, the analysis lets us identify (i) the possible sequences of instructions that can modify and finally send a packet, with all the conditions under which each one of them can be executed and (ii) the sequences of instructions that can lead to a modification of the state of the network function, and all the conditions under which each alteration can occur.

The implementation of the parser takes advantage of *abstract syntax tree (AST)* , which is a tree representation of the abstract syntactic structure of source code. AST is the syntactic analysis tool included almost in all well-known programming languages like C++, Java, JavaScript and Python. The formal model generated by the parser takes a form that is motivated by the vision of OpenFlow [22] forwarding abstraction of the form <*match, action*>. This abstraction model has been borrowed from the existing modeling techniques [17], [23] and most of the verification tools of forwarding behavior ([8], [9], [24]) rely on the models of this abstraction.

The parser generates the model in an XML format that expresses this abstraction, by building the <*match, action*> pairs from the parsed AST. We classify the boolean conditions in the code that lead to a return statement or to a state change statement as ''match'' and the forwarding and state changes as ''action'' respectively. The ''match'' conditions are further classified into two categories: those that refer to state variables and those that are state independent. Similarly, the ''action'' rules are categorized as: those that trigger a state transition and those that trigger a forwarding action. They are referred to as ''state'' and ''flow'' respectively, as shown in TABLE 2. This categorization is needed in the translation phase of the model, because the verification tool

specific model representation is formed depending on the type of <*match, action*> tuples. In the following subsections we walk through the steps the parser takes in building the abstraction model for stateless and stateful network functions and cover the classification of rules in detail.

1) STATELESS NETWORK FUNCTION

As an example, we use the NF definition for ACL firewall by means of our library, which is listed in Figure 2. `ACL` contains a `Table` object named `aclTable`. The method invocation in line 6 specifies that the table has two columns, both of type IP address (`Table.TableTypes.Ip`). The parser will store this information in the XML notation of the intermediate model. This table acts as a "blacklist": if the source and destination IP addresses of the received packet match an entry in the table, a drop action is performed. The illustration of such abstraction for the stateless ACL firewall network function, obtained automatically from the simple source code definition, is presented in TABLE 2, where only the first rule extracted from the code is presented. In fact, for each forwarding action in the `onReceivedPacket()` method, the parser constructs a separate rule. The forwarding actions are recognized by the parser by looking for the presence of `RoutingResult` class instance creation in the return statement, where the action argument is equal to `Action.FORWARD`. For instance the method in FIGURE 2 leads to the generation of two distinct rules due to the presence of two forwarding actions (lines 13 and 17, FIGURE 2). Due to the stateless nature of the NF, the final model does not contain any information regarding the state (i.e. lines 10 and 12 in TABLE 2 are empty).

TABLE 2. Rule abstraction for ACL firewall network function.

1	Match	Flow	recv(p_1,INTERNAL)
2			!matchEntry(p_1.IP_SRC, p_1.IP_DST)
3			p_0.IP_SRC == p_1.IP_SRC
4			p_0.IP_DST == p_1.IP_DST
5			p_0.PORT_SRC == p_1.PORT_SRC
6			p_0.PORT_DST == p_1.PORT_DST
7			p_0.TRANSPORT_P == p_1.TRANSPORT_P
8			p_0.APPLICATION_P == p_1.APPLICATION_P
9			p_0.L7DATA == p_1.L7DATA
10		State	*
11	Action	Flow	send(p_0,EXTERNAL)
12		State	*

The parser proceeds backwards in the control graph of the code, starting from the selected forwarding action, till it reaches the entry point of the method. In this way, it obtains the sequence of statements that have to be executed in order to reach the selected forwarding action. From this sequence, the parser extracts the conditions that must be satisfied in order for the sequence to be executed. They are essentially the conditions of the if statements found in the sequence, plus an additional predicate taking the form `recv(p,i)`, where p and i are the packet and interface passed to `onReceivedPacket()`. This last predicate expresses the condition that packet p is received on interface i. In all these conditions, variables are substituted with the values assigned to them in the sequence of statements, explicitly or implicitly.

2) STATEFUL NETWORK FUNCTION

Models of stateful NFs are more complex, because the match field may regard not only packet flows but also states, and the action to be performed not only forwards the packets but also triggers an update on state components. In order to indicate the difference between the two cases, we analyze the outcome of the parser for a NAT network function. From FIGURE 3 it is easy to realize that the model for the NAT network function consists of three rules due to the three forwarding actions. To extract these rules, the parser proceeds in an analogous manner to that shown in section III-B.1. At the beginning the parser classifies this as a stateful network function due to the explicit state change that occurs in the `natTable.storeEntry(e)` method invocation, in line 26, and identifies `natTable` as part of the function state. In contrast to the `matchEntry()` method, used to model a table lookup for the stateless network function, here the parser classifies the `matchEntry()` method invocation listed in line 7 as a state specific conditional statement, which corresponds to lines 8-10 in TABLE 3. Additionally, the packet modifications done by the NAT result in a form of constraints, where the source IP and port addresses of the

```
1  // pseudo code
2  RoutingResult onReceivedPacket(Packet packet, Interface iface) {
3
4    Packet packet_in = null
5    packet_in = packet.clone()
6    if(iface.isInternal()){
7      TableEntry entry = natTable.matchEntry(packet_in.getField(PCKField.IP_SRC),
8          packet_in.getField(PCKField.PORT_SRC))
9      if(entry != null){
10       packet_in.setField(PCKField.IP_SRC, (String)entry.getValue(4))
11       packet_in.setField(PCKField.PORT_SRC, (String)entry.getValue(5))
12
13       return new RoutingResult(Action.FORWARD, packet_in, externalInterface)
14     }else{
15
16       TableEntry e = new TableEntry(7)
17       e.setValue(0, packet_in.getField(PCKField.IP_SRC))
18       e.setValue(1, packet_in.getField(PCKField.PORT_SRC))
19       e.setValue(2, packet_in.getField(PCKField.IP_DST))
20       e.setValue(3, packet_in.getField(PCKField.PORT_DST))
21       e.setValue(4, natIp)
22       e.setValue(5, String.valueOf(new_port))
23       e.setValue(6, new Date())
24       packet_in.setField(PCKField.IP_SRC, natIp)
25       packet_in.setField(PCKField.PORT_SRC, String.valueOf(new_port))
26       natTable.storeEntry(e)
27       return new RoutingResult(Action.FORWARD, packet_in, externalInterface)
28     }
29   }else{
30     TableEntry entry = natTable.matchEntry(
31       packet_in.getField(PCKField.IP_SRC), packet_in.getField(PCKField.PORT_SRC),
32       packet_in.getField(PCKField.IP_DST), packet_in.getField(PCKField.PORT_DST))
33     if(entry == null)
34       return new RoutingResult(Action.DROP, null, null)
35     packet_in.setField(PCKField.IP_DST, (String) entry.getValue(0))
36     packet_in.setField(PCKField.PORT_DST, (String) entry.getValue(1))
37
38     return new RoutingResult(Action.FORWARD, packet_in, internalInterface)
39   }
40 }
```

FIGURE 3. Behavior (only the part) of the NAT network function model in response to received packet.

TABLE 3. Rule abstraction for NAT network function (part 1).

1	Match	Flow	recv(p_1,INTERNAL)
2			p_0.IP_DST == p_1.IP_DST
3			p_0.TRANSPORT_P == p_1.TRANSPORT_P
4			p_0.APPLICATION_P == p_1.APPLICATION_P
5			p_0.L7DATA == p_1.L7DATA
6		State	p_0.IP_SRC == p1.IP_SRC
7			p_0.PORT_SRC == p1.PORT_SRC
8			recv(p_2,INTERNAL)
9			p_1.IP_SRC == p_2.IP_SRC
10			p_1.PORT_SRC == p_2.PORT_SRC
11	Action	Flow	send(p_0,EXTERNAL)
12		State	*

TABLE 4. Rule abstraction for NAT network function (part 2).

1	Match	Flow	recv(p_1,INTERNAL)
2			p_0.IP_DST == p_1.IP_DST
3			p_0.TRANSPORT_P == p_1.TRANSPORT_P
4			p_0.APPLICATION_P == p_1.APPLICATION_P
5			p_0.L7DATA == p_1.L7DATA
6		State	p_0.IP_SRC == "natIP"
7			p_0.PORT_SRC == "new_port"
8			recv(p_2,INTERNAL)
9			!(p_1.IP_SRC == p_2.IP_SRC) ||
10			!(p_1.PORT_SRC == p_2.PORT_SRC)
11	Action	Flow	send(p_0,EXTERNAL)
12		State	store(p0,"new_port")

new packet, that is being sent, must correspond to the values retrieved from the entry, as shown in lines 6,7 of TABLE 3. Whereas the flow-related, state-independent conditions in lines 1-5 of TABLE 3 state that the new packet that is being forwarded, must keep the rest of the fields of the received packet unchanged. This is the first rule the parser builds following the if branches in lines 6 and 9 (FIGURE 3). When the parser visits the else branch of the code that starts at line 14, it extracts the set of conditions that lead to another forwarding action and builds the rule shown in TABLE 4. In contrast to the first rule, the second rule contains an action that triggers a state transition, by storing the new entry in the internal state of the network function (line 12 in TABLE 4). This implies that the NAT translation table does not contain an entry matching the IP and port source addresses as given in lines 9,10 (of TABLE 4) and the new entry is inserted in the translation table of the NAT network function.

The final rule to be extracted covers the behavior of the NAT network function when receiving a packet from the external interface, which is extracted in a similar manner.

C. TRANSLATOR

One of the strengths of our approach is the ability to serialize the final model abstraction across different languages, thus being able to target different verification programs. For instance, VeriGraph exploits network function models expressed as formulas in First Order Logic (FOL) [25], taking the form

```
send(NF, destination, packet) ->
CONDITIONS
```

send and *recv* are two predicates defined in the VeriGraph framework that receive as arguments two nodes representing the source and the destination of a packet that can be sent or received, and the packet itself. The right hand side of the formula expresses the conditions under which the packet is forwarded. These formulas are difficult to write. Hence, VeriGraph can greatly benefit from the automatic generation of models.

The conditions that are included in each rule are combined in conjunctive normal form (CNF) in order to obtain a single "match" and "action" rule respectively. For example, the rule in TABLE 4 results in the following FOL formula for VeriGraph:

```
((send(n_Nat,n_0,p_0) && !(isInternal
(p_0.IP_DST))) ->
  E(n_1, p_1 |(recv(n_1,n_Nat,p_1) &&
  isInternal(p_1.IP_SRC) &&
  !(E(n_2, p_2 |(recv(n_2,n_Nat,p_2) &&
        isInternal(p_2.IP_SRC) &&
       (p_1.IP_SRC == p_2.IP_SRC) &&
       (p_1.PORT_SRC == p_2.PORT_SRC))))&&
      (p_0.IP_SRC == natIp) &&(p_0.PORT_SRC
       == new_port) &&
      (p_0.IP_DST == p_1.IP_DST) && (p_0.PORT_DST
       == p_1.PORT_DST) &&
      (p_0.TRANSPORT_P == p_1.TRANSPORT_P) &&
      (p_0.APPLICATION_PROTOCOL == p_1.APPLICATION_
       PROTOCOL) &&
      (p_0.L7DATA == p_1.L7DATA))))
```

This FOL formula is interpreted as that a new packet p_0 is sent to a node n_0 through the external interface of the network function, which is translated as a negation of `isInternal(p_0.IP_DST)` VeriGraph specific predicate. This send action is valid, only if there isn't a packet p_2, received before the current packet p_1, having the same port and source IP addresses as packet p_1. The rest of the model is translated as is. However the state transition primitive in line 12 does not take place in the final translation, because VeriGraph does not support data structures to keep track of the internal state of the network function. Despite this, VeriGraph can model any stateful function that depends not just on static forwarding rules, but also on the sequence of previously encountered packets, introducing multiple implications in the function model.

On the other hand, SFC-Checker [23] supports predicates explicitly altering the state of the network function in the form of temporal forwarding behavior using Finite State Machines (FSM). TABLE 5 shows the output of the translator for the NAT network function model rule given in TABLE 4, by means of SFC-Checker supported primitives. As evident from the table, the state independent condition in line 1 of TABLE 4 is translated using the *pre-condition primitive* `IF` introduced in SFC-Checker. Whereas the state-relative conditions in lines 8-10 of TABLE 4 take the form of *state operations* primitive - `get()`. By means of the `set(f.p,"new_port")` operation, the information related to the state change in the NAT table is delivered.

TABLE 5. Translation of NAT network function model for SFC-Checker.

Match(f,s)	Action
IF(f.p.src,INTERNAL), get(f.p.ip_src)!=f.p.ip_src&& get(f.p.port_dst)!=f.p.port_dst	forward(f.p,out) set(f.p,"new_port") Modify(f.p.ip_src,"nat_ip") Modify(f.p.port_src,"new_port")

Similarly, the Symbolic Execution Friendly Language (SELF) proposed by SymNet [8] requires models in the *<match, action>* formalism and replaces unknown values in the conditions with symbolic values. This helps SymNet to explore different paths of the model. Eventually the output of the translation for the NAT models in TABLE 3 and TABLE 4 is identical to the model demonstrated in SymNet [8]:

```
InputPort(0):
Constrain(IPProto,==6) //only do TCP
    Allocate(``orig-ip'',32,local)
    Allocate(``orig-port'',16,local)
    Allocate(``new-ip'',32,local)
    Allocate(``new-port'',16,local)
    Assign(``orig-ip'',IpSrc) //save
    initial addr
    Assign(``orig-port'',TCPSrc) //
    save initial port
    Assign(IpSrc,``...'') //perform
    mapping
    Assign(TcpSrc, SymbolicValue())
    Assign(``new-ip'',IpSrc) //save
    assigned addr
    Assign(``new-port'',TcpSrc) //save
    assigned port
    Forward(OutputPort(0))
```

It is important to note that SymNet injects only one packet per execution when it performs verification [26]. Hence, we cannot translate the conditional statements on multiple packets of our data driven network functions such as NAT and Web Cache into SEFL language. As a result, models generated for SymNet cannot take state into account. In addition, in those network functions the idea of private or public network is described in terms of specific ports, as there is no such network division in SymNet. However, by defining a new abstraction, we can generate a model of a stateful function specifically designed for SymNet only. As our goal is to provide a generic modeling language that supports most of the verification tools, this approach is out of scope for this work.

IV. VERIFICATION

The resulting models, describing the forwarding behavior, are well suited for verification of the basic network invariants such as reachability and isolation. Among the existing logic-based verification tools we selected VeriGraph [9] and SymNet [8] as use cases to show how the model generated by the parser can be exploited, after proper translation, by a real verification tool. VeriGraph and SymNet are the formal verification tools that can automatically verify networks by checking certain policies before the service deployment. In this context, the term network is used to indicate a chain of network functions such as (load balancer, antispam, packet filter, DPI and so on) that starts from a source node and ends into a different destination node. In response to a verification request, a model of the network and the involved network functions, is checked against the provided policies, for instance isolation properties between multiple devices in the network.

To automatically validate network connectivity policies at scale, the verification engine of VeriGraph and SymNet exploits an off-the-shelf SAT solver (Z3), which verifies whether the considered policies are satisfied or not, thanks to the translation of these problems into SAT problems.

A. NETWORK FUNCTION CATALOGUE

We tested our framework with VeriGraph and SymNet using a set of network function models, written by means of our library during the development phase and used to evaluate the effectiveness of our method. The available network functions are listed in TABLE 6, together with the parsing time to generate each verification tool input model. The benchmarks were run on a machine with an Intel i5-4210M CPU and a memory limit of 8 GBs. It is worth noticing how these times are a satisfactory result, also considering that the parsing process is not a real-time task and is executed only once. In this subsection we discuss which types of network functions can be modeled by using our framework.

TABLE 6. Time spent to parse network function models.

Network function model	Time to parse (ms)
Mail Server	862
IDS	869
NAT	880
Web Server	920
Web Cache	952
Firewall	957
Antispam	963

Models of filtering functions, which includes DPI (Deep Packet Inspection), Antispam and ACL firewall are included in our catalog of network functions. However, any type of network functions can be modeled using the additional data structures and design rules. For instance, in our **DPI** network function, we introduce a table of type `Table.TableTypes.ApplicationData` with a single column. This type represents the packet data of Layers 5,6,7 in the OSI network model, which, in our model, corresponds to a special field of the packet

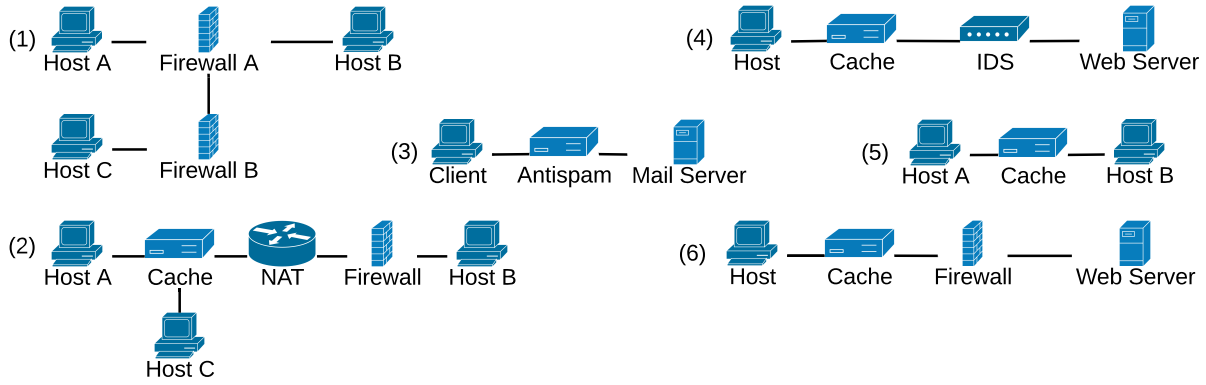


FIGURE 4. Set of network topologies created for the verification process.

named `L7DATA`. If the protocol of the received packet is equal to `HTTP_REQUEST` or `HTTP_RESPONSE`, the DPI performs a table lookup based on the `L7DATA` field of the packet. The presence of an entry corresponding to that data in the table results in a drop action of the packet. In addition, this is the structure of the code of DPI network function share in common with the model of **Antispam** network function, which includes a table with a single column and if the `L7DATA` field of the packet matches an entry in the table, the packet is dropped.

Instead, **WebCache** is a stateful network function, modeled by means of a table of two columns. The `onReceivedPacket()` method of the class includes four forwarding actions. The first two of them correspond to a packet arriving from an internal interface. If the `APPLICATION_PROTOCOL` of the packet is equal to `HTTP_REQUEST` and the table contains an entry matching the requested string in `L7DATA`, it can send back a packet, which is the (cached) response, containing the requested web page. If the requested web page is not available in the table of the network function, the original packet is forwarded through the external interface.

On the other hand, the next two forwarding actions do not alter the packet and acts as a packet forwarder. Only if the web content of the received packet is not found in the internal database, it is stored in the table of the network function. Additionally, we include in our catalog, models of VPN gateways, packet filters and traffic classifiers.

B. EXPERIMENTAL RESULTS

In order to check the correctness of the generated final models, we constructed a set of experiments with different network topologies containing the available network functions, and we performed a number of custom tests on the selected verification tools. VeriGraph and SymNet can perform different kinds of verification tests: reachability, which consists of checking if at least one packet can arrive at the destination from the source node, and isolation, namely, that packets sent from one host (or class of hosts) can never reach another host (or class of hosts).

FIGURE 4 illustrates the set of topologies adopted for our tests. By means of these tests, we show that generated abstract models are close to the actual behavior of network functions and can be used in various scenarios. For instance, topology (1) involves two firewalls and three end hosts. Firewalls are configured according to the following:

1A. Firewall A denies all traffic between host A and C and the default action of the firewall is allow.
1B. Firewall B denies all traffic between host B and C and the default action of the firewall is allow.

This test includes two isolation properties to be checked. In particular, we consider two packets, one flowing from host A to host C, and another one flowing from host A to host B. Taking into account the above firewall policies, we expect the isolation property is satisfied in case of A-to-C, indicating that no packet can reach the host C from A, while we expect it is not satisfied in the case of A-to-B. The other test cases are set up as follows (the numbers refer to the corresponding topology in FIGURE 4):

2. Rule: firewall denies traffic between NAT and host B. B is located in private network where NAT table contains an entry related to the previous connection B-to-A. Property: isolation between hosts A and B. Action: send a packet from host A to B.
3. Rule: antispam performs an application layer content filtering. A packet containing a string "discount" in its body is blocked. Property: isolation between mail server and client. Action: send a packet containing a string "discount" in its body from client to mail server.
4. Rule: DPI drops a packet containing specific string in the body of the packet. Property: isolation between host and web server. Action: send a packet containing the specific string in the body from host to web server.
5. Rule: local storage of web cache contains *"www.google.it"* URL address. Property: isolation between host A and B. Action: send a web page request containing *"www.google.com"* URL from host A to B.
6. Rule: firewall denies traffic between host and web server. Local storage of web cache is empty.

TABLE 7. Comparison of the verification results. Column N represents the number of the corresponding topology illustrated in FIGURE 4.

N	Tests	Verification results	Time to verify autogenerated NF model (ms)	
			VeriGraph	SymNet
(1)	DoubleFwTest A	SAT	214	530
	DoubleFwTest B	UNSAT		
(2)	CacheNatFwTest	SAT	318	787
(3)	AntispamTest	SAT	275	681
(4)	DPITest	SAT	192	475
(5)	CacheTest	UNSAT	260	644
(6)	CacheFwTest	SAT	200	495

Property: isolation between host and cloud web server. Action: send a packet from host to web server.

TABLE 7 delivers the results we obtained implementing these categories of tests in VeriGraph and SymNet. In the table, 'SAT' means the isolation property is satisfied, while 'UNSAT' means that the isolation property is not satisfied. Comparing the test results obtained by using a set of hand-written models and the ones obtained by means of the automatically generated ones (starting from the high-level description and then generated using the parser), we found that results are identical, as expected. This confirms the correctness of our modeling approach and also shows the efficiency of the developed framework.

V. CONCLUSION

This paper presents a "user-friendly" approach to network function modeling for formal verification of forwarding behavior. We focus on breaking the barrier between the two ways of representing a network function:
the imperative-centric function definition (proper of network function developers) and the more higher-level declarative representation (used by formal verification experts in order to instruct logic-based verification tools). The proposed approach provides a method to translate from the former to the latter automatically. The method relies on the modeling technique we presented in this paper which includes a modeling library, a parser and a translator. We validated the correctness of the models obtained using our framework by means of different verification tools.

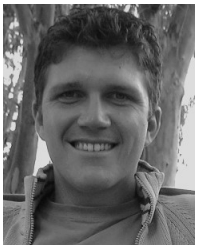
Considering what are the current requests of the market and looking at the possible future developments, this framework presents a further step towards the real implementation of these new concepts inside the networks. In fact, the framework and the available verification tools may be a basic structure to define Virtual Network Functions and test the overall network functionality before deployment.

As future work, the catalogue of network function models will be enriched with a wide variety of network devices. Additionally, we plan to add the possibility to automatically generate efficient implementations of the network functions from their high-level description. This will help developers to get implementations that are consistent with the models used for the analysis of the forwarding behavior of network configurations.

REFERENCES

[1] ETSI GS NFV, "Network Functions Virtualisation (NFV); Terminology," *IEEE Netw.*, vol. 1, no. 5, pp. 1–50, 2013.
[2] ETSI. (2017). *European Telecommunications Standards Institute*. [Online]. Available: http://www.etsi.org
[3] N. P. Lopes, N. Bjørner, P. Godefroid, K. Jayaraman, and G. Varghese, "Checking beliefs in dynamic networks," in *Proc. 12th USENIX Symp. Networked Syst. Design Implement.*, 2015, pp. 499–512.
[4] F. Valenza, S. Spinoso, C. Basile, R. Sisto, and A. Lioy, "A formal model of network policy analysis," in *Proc. IEEE 1st Int. Forum Res. Technol. Soc.*, Sep. 2015, pp. 516–522.
[5] S. Owre, J. M. Rushby, and N. Shankar, "PVS: A prototype verification system," in *Proc. 11th Int. Conf. Automated Deduction*, 1992, pp. 748–752.
[6] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King, "Debugging the data plane with anteater," in *Proc. ACM SIGCOMM Conf.*, Aug. 2011, pp. 290–301.
[7] C. Basile, F. Valenza, A. Lioy, D. R. Lopez, and A. P. Pastor, "Adding support for automatic enforcement of security policies in NFV networks," *IEEE-ACM Trans. Netw.*, vol. 27, no. 2, pp. 707–720, Apr. 2019.
[8] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, "SymNet: Scalable symbolic execution for modern networks," in *Proc. ACM SIGCOMM Conf.*, Aug. 2016, pp. 314–327.
[9] S. Spinoso, M. Virgilio, W. John, A. Manzalini, G. Marchetto, and R. Sisto, "Formal verification of virtual network function graphs in an SP-DevOps context," in *Proc. Eur. Conf. Service-Oriented Cloud Comput.*, Sep. 2015, pp. 253–262.
[10] S. Narain, "Network configuration management via model finding," in *Proc. 19th Conf. Large Installation Syst. Admin. Conf.*, vol. 19, 2005, p. 15. [Online]. Available: http://dl.acm.org/citation.cfm?id=1251150.1251165
[11] G. Marchetto, R. Sisto, M. Virgilio, and J. Yusupov, "A framework for user-friendly verification-oriented VNF modeling," in *Proc. IEEE 41st Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 1, Jul. 2017, pp. 517–522.
[12] G. G. Xie, J. Zhan, D. A. Maltz, H. Zhang, A. G. Greenberg, G. Hjalmtysson, and J. Rexford, "On static reachability analysis of IP networks," in *Proc. INFOCOM*, Mar. 2005, pp. 2170–2183.
[13] C. Basile, D. Canavese, C. Pitscheider, A. Lioy, and F. Valenza, "Assessing network authorization policies via reachability analysis," *Comput. Electr. Eng.*, vol. 64, pp. 110–131, Nov. 2017.
[14] P. Kazemian, G. Varghese, and N. McKeown, "Header space analysis: Static checking for networks," in *Proc. 9th USENIX Conf. Networked Syst. Design Implement.*, Dec. 2012, p. 9.
[15] J. C. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, A. Robby, and H. Zheng, "Bandera: Extracting finite-state models from Java source code," in *Proc. 22Nd Int. Conf. Softw. Eng.*, Jun. 2000, pp. 439–448.
[16] K. Havelund and T. Pressburger, "Model checking JAVA programs using JAVA PathFinder," *Int. J. Softw. Tools Technol. Transf.*, vol. 2, no. 4, pp. 366–381, Mar. 2000.
[17] W. Wu, Y. Zhang, and S. Banerjee, "Automatic synthesis of NF models by program analysis," in *Proc. 15th ACM Workshop Hot Topics Netw.*, Nov. 2016, pp. 29–35.
[18] J. Khalid, A. Gember-Jacobson, R. Michael, A. Abhashkumar, and A. Akella, "Paving the way for NFV: Simplifying middlebox modifications Using statealyzr," in *Proc. 13th Usenix Conf. Networked Syst. Design Implement.*, 2016, pp. 239–253.
[19] M. Weiser, "Program slicing," in *Proc. 5th Int. Conf. Softw. Eng.*, Jul. 1981, pp. 439–449.
[20] H. Agrawal and J. R. Horgan, "Dynamic program slicing," in *Proc. ACM SIGPLAN Conf. Program. Lang. Design Implement.*, Jun. 1990, pp. 246–256.
[21] S. K. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar, "BUZZ: Testing context-dependent policies in stateful networks," in *Proc. 13th USENIX Symp. Networked Syst. Design Implement.*, Jul. 2016, pp. 275–289.

[22] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling innovation in campus networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, Mar. 2008.
[23] B. Tschaen, T. B. Ying Zhang, J. L. Sujata Banerjee, and J.-M. Kang, "SFC-checker: Checking the correct forwarding behavior of service function chaining," in *Proc. IEEE Conf. Netw. Function Virtualization Softw. Defined Netw. (NFV-SDN)*, Nov. 2016, pp. 134–140.
[24] A. Panda, O. Lahav, K. J. Argyraki, M. Sagiv, and S. Shenker, "Verifying isolation properties in the presence of middleboxes," 2014, *arXiv:1409.7687*. [Online]. Available: https://arxiv.org/abs/1409.7687
[25] W. Hodges, "Elementary predicate logic," in *Handbook of Philosophical Logic*, D. M. Gabbay and F. Guenthner, Eds. Dordrecht, The Netherlands: Springer, 2001, pp. 1–129. doi: 10.1007/978-94-015-9833-0_1.
[26] A. Panda, O. Lahav, K. Argyraki, M. Sagiv, and S. Shenker, "Verifying reachability in networks with mutable datapaths," in *Proc. 14th USENIX Symp. Networked Syst. Design Implement.*, 2017, pp. 699–718.



GUIDO MARCHETTO received the Ph.D. degree in computer engineering from the Politecnico di Torino, in 2008, where he is currently an Associate Professor 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.



RICCARDO SISTO received the Ph.D. degree in computer engineering from the Politecnico di Torino, Italy, in 1992. Since 2004, he has been a Full Professor of computer engineering with the Politecnico di Torino. He has authored and coauthored more than 100 scientific papers. His main research interests include formal methods, applied to distributed software and communication protocol engineering, distributed systems, and computer security. He is a Senior Member of the ACM.



FULVIO VALENZA received the M.Sc. degree *(summa cum laude)* and the Ph.D. degree *(summa cum laude)* in computer engineering from the Politecnico di Torino, Torino, Italy, in 2013 and 2017, respectively, where he is currently a Researcher. His research activity focuses on network security policies, orchestration and management of network security functions in SDN/NFV-based networks, and threat modeling.



JALOLLIDDIN YUSUPOV received the M.S. degree in computer engineering from the Politecnico di Torino, Italy, in 2016, where he is currently pursuing the Ph.D. degree in control and computer engineering. His primary research interests include formal verification of security policies in automated network orchestration. His other research interests include modeling, cyber physical systems, and cloud computing systems.