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

ANDERSON SANTOS DA SILVA

Network Testing through Grammars: Towards the Analysis of Property Violation on Computer Networks

Thesis presented in partial fulfillment of the requirements for the degree of Doctor of Computer Science

Advisor: Prof. Dr. Alberto Egon Schaeffer-Filho

Porto Alegre
May 2023

CIP – CATALOGING-IN-PUBLICATION

Silva, Anderson Santos da

Network Testing through Grammars: Towards the Analysis of Property Violation on Computer Networks / Anderson Santos da Silva. – Porto Alegre: PPGC da UFRGS, 2023.

110 f.: il.

– Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2023. Advisor: Alberto Egon Schaeffer-Filho.

1. Networking Testing. 2. Property Violation. 3. Network Monitoring. 4. Formal Verification. I. Schaeffer-Filho, Alberto Egon. II. Título.

ABSTRACT

A clear trend within the context of computer networks is the use of software as an alternative to the use of specialized hardware. The benefit of this trend is an enhancement of flexibility, modularity, and maintainability of network components. Concurrently, it is challenging to determine if everything is happening correctly in a computer network where software, possibly with bugs, is very present. To deal with this challenge, network testing is frequently used to check if a network component respects a given property and consequently performs its actions correctly. As there is a variety of causes for the abnormal operation of the network, ranging from human mistakes (inserting misconfiguration) to malicious activities, there are many challenges to network testing to achieve satisfactory results. Research in this field frequently tries to improve network testing by the use of formal verification techniques and network monitoring to detect property violations, such as configuration errors and policy conflicts. However, formal verification by itself cannot detect a property violation that was not anticipated and included in the model. Similarly, network monitoring needs to wait for a property violation to occur to detect it. Consequently, both enhancement efforts fail to achieve a complete result. In this thesis, we investigate the problem of achieve the absence of property violations by combining the advantages of network monitoring for detecting property violations with the advantages of formal verification to model the network and prove the existence or absence of property violations. A highlight related to the success of such a combination is the use of a model based on grammars to capture the communication patterns existing on the network. Our analysis allows the evaluation of high-level properties such as "Can network component x send HTTP packets? " and the detection of property violations, such as conflicting forwarding rules, as soon they occurred in the network. As future research, we intend to further investigate how our grammar-based model can be extended to support temporal logic operators and how we can trace the effects of property violations in the network.

Keywords: Networking Testing. Property Violation. Network Monitoring. Formal Verification.

Teste de Rede Através de Gramáticas: Em Direção a Análise de Violação de Propriedades em Redes de Computadores

RESUMO

Uma tendência clara no contexto das redes de computadores é o uso de software como uma alternativa ao uso de hardware especializado. O benefício dessa tendência é um aprimoramento da flexibilidade, modularidade e capacidade de manutenção de componentes de rede. Ao mesmo tempo, é desafiador determinar se tudo está acontecendo corretamente em uma rede de computadores onde o software, possivelmente com erros (*bugs*), está muito presente. Para lidar com este desafio, teste de rede é freqüentemente usado para verificar se um componente de rede respeita uma determinada propriedade e conseqüentemente executa suas ações corretamente. Como há uma variedade de causas para o funcionamento anormal da rede, variando de erros humanos (inserir configuração incorreta) para atividades maliciosas, existem muitos desafios para o teste de rede alcançar resultados satisfatórios. Pesquisas neste campo freqüentemente tenta melhorar os testes de rede pelo uso de técnicas de verificação formal e monitoramento de rede para detectar violações de propriedade, como erros de configuração e conflitos de política. No entanto, verificação formal por si só não consegue detectar uma violação de propriedade que não foi antecipada e incluída no modelo. Da mesma forma, monitoramento de rede precisa aguardar a ocorrência de uma violação de propriedade para detectá-la. Consequentemente, ambos esforços de aprimoramento falham em alcançar um resultado completo. Nesta tese, nós investigamos o problema de garantir a ausência de violações de propriedade de rede combinando as vantagens do monitoramento de rede para detecção de violações de propriedade com as vantagens de verificação formal para modelar a rede e provar a existência ou ausência de violações de propriedade. Um destaque relacionado ao sucesso de tal combinação é o uso de um modelo baseado em gramáticas para capturar os padrões de comunicação existentes na rede. Nossa análise preliminar permite a avaliação de propriedades de rede de alto nível, como "É possível que o componente x envie pacotes HTTP?"e a detecção de violações de propriedade, como regras de encaminhamento conflitantes assim que ocorreram na rede. Como próximas etapas, pretendemos investigar de forma mais profunda como nosso modelo baseado em gramática pode ser estendido para suportar operadores lógicos temporais e como podemos rastrear o efeitos de violações de propriedade na rede.

Palavras-chave: Teste de Rede, Violação de Propriedades, Monitoramento de Rede, Verificação Formal.

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
ICMP	Internet Control Message Protocol
GUI	Graphical User Interfaces
FTP	File Transfer Protocol
IaaS	Infrastructure as a Service
ISP	Internet Service Providers
JSON	JavaScript Object Notation
NFS	Network File System
UDP	User Datagram Protocol
ONOS	Open Network Operating System
QoS	Quality of Service
REST	Representational State Transfer
TCP	Transmission Control Protocol
SDN	Software-Defined Networking
SLA	Service-Level Agreement
SNMP	Simple Network Management Protocol
VLAN	Virtual Local Area Network
HTTP	Hypertext Transfer Protocol
XML	eXtensible Markup Language

LIST OF FIGURES

LIST OF TABLES

CONTENTS

1 INTRODUCTION

Network property checking, a concept related to the use of techniques to determine if a network component (either simple forwarding devices like switches or routers, or middleboxes – both physical or virtualized) respects a set of predefined properties, has grown its importance in late years (REITBLATT et al., 2012; PRABHU et al., 2020; ZHANG et al., 2022). The main reason is the critical aspect of checking these predefined properties: a network component should perform correct actions and its correct functioning is only achievable if it respects its specification/properties. Additionally, the continuous complexity growth of network components increases the number of possible execution states, thus increasing the number of properties required to check if one of these states is correct or not. As consequence, when compared to the first network components that were simple and dedicated to performing a single action, such as forward packets or check errors, nowadays devices are responsible for several concurrent actions such as forward packets, collecting statistics, executing security routines, etc. Additionally, with software dissemination on computer networks and mainly the evolution of system requirements the number of properties to check if a device is functioning correctly increased as well (BIRKNER et al., 2020). As these new software components are frequently related to security, performance, and fault tolerance functions, a system to automatize and optimize the process of checking these properties is essential to the future of computer networks.

1.1 Contextualization

Unfortunately, the ecosystem of computer networks is dynamic enough to evolve so fast that any checking system soon becomes outdated (AVIZIENIS et al., 2004; DELMAS et al., 2020). When a network component cannot assure all its properties, we say that a property violation occurs. Property violation can have several causes but frequently occurs as a result of misconfiguration (LUCKCUCK et al., 2019). It is difficult to find misconfiguration causes because (i) the internal state of network components frequently is unknown; (ii) achieving the global state of the network is a challenging task because it needs to consider every network component and (iii) human mistakes are always present and insert misconceptions on the network, thus compromising even well-known properties.

Software-Defined Networking (SDN) (FEAMSTER; REXFORD; ZEGURA, 2013) and Network Functions Virtualization (NFV) (ZEGHLACHE, 2016) are examples of network paradigms strongly dependent on software components. By using software instead of specific hardware components, SDN and NFV promote more flexibility in the design of the network and facilitate the programmability of network equipment. On the one hand, the growing use of software components to perform network tasks brings flexibility to network administrators regarding the management and monitoring of network components (WICKBOLDT et al., 2015a). On the other hand, there is a price for this: all these network components should be configured to

perform their tasks and cooperate with other software components. It is challenging to guarantee an optimal configuration to exclude future property violations when using these software components jointly (SILVA et al., 2015a; BIRKNER et al., 2020). Typical property violations are created by the faulty management of forwarding entries, creating the possibility of packets behaving incorrectly (LIU et al., 2020). The source of these violations may be the implementation of routing or configuration tools that do not produce the desired output for the data plane. Additionally, the lack of interoperability between these applications can also produce erroneous configuration and conflicts (YANG et al., 2020).

To deal with these challenges, network testing (BARI et al., 2013) frequently is used to check if a component respects a given property and consequently performs its actions correctly. Network testing can be divided into three groups: (i) black-box testing when the internal state of the network component is not known and the test samples can determine which inputs and outputs the component recognizes; (ii) white-box testing when the internal state of the component is known and it is possible to see its implementation and test specific properties considering a high-level view of its functioning; and (iii) defect-based testing, a technique that generates test cases based on defect signatures instead of using the traditional coverage tests. There is a variety of challenges related to network testing, ranging from human mistakes (inserting misconfiguration) to malicious activities (AVIZIENIS et al., 2004). Human mistakes frequently increase the scope of the testing process because it needs to cover inconsistencies in configuration files and also malicious activities, including network anomalies related to human attacks, such as IP spoofing (ABDULQADDER et al., 2020).

Consequently, network testing techniques can be combined with other techniques to check more complex network properties. An example is the techniques of formal verification that can be used to detect configuration errors and policy conflicts, as well as to verify more traditional properties such as reachability and isolation. However, this research field poses many challenges, such as the ability to capture all network states. As consequence, the result is not optimal in general (AL-SHAER; AL-HAJ, 2010a), a reality that encourages alternative techniques to these limitations, such as the observation of device actions instead of modeling its possible states, using network monitoring techniques, such as flow sampling.

1.2 Motivation

A clear trend within the context of computer networks is the use of software as an alternative to the use of specialized hardware. The benefit of this trend is clear: network administrators can manage and control network aspects in a more flexible, modular, and reasonable way. Concurrently, a challenge in this context is also clear: how to determine if everything is happening correctly in a computer network where software, possibly with bugs, is very present? One way to answer this question relies on testing. However, to test a computer network we need to answer three basic questions before:

1) What components compose what we understand as a computer network?

2) What properties on this computer network are considered desirable?

3) What properties in this computer network are undesirable?

We answer question 1) with the following definition: We consider that a network consists of several *network components* and this is the most basic intuition of this study. A *network component* comprises all elements that directly interact with network packets. For example, switches, routers, middleboxes, communication links, and hosts are examples of *network components*. A network user sitting in front of his computer is not an example of a *network component* because even in the case that he interacts with the network as a whole, he does not directly manipulate packets.

In the case of question 2), it is important to define that properties are a set of the characteristics of a *network component*. Our interest in studying *network components* is focused on testing properties that they can assume or not during their life cycle. A subset of these properties is desirable, others not.

A *network component* can assume several execution states and each of them is the result of a set of influences capable of forcing them to change from one state to another. An internal influence changes the state of a *network component* considering only intrinsic aspects of the component's structure. For example, an implementation error on a switch arises due to an error in its initial design. An external influence changes the state of a *network component* considering external aspects to it and its interaction with the ecosystem in which it executes. For example, an error in a switch due to a power outage forces it from one state to another abruptly. We advocate that the study of properties in a *network component* needs to include external and internal influences on this component.

The set of properties that arises when only internal influences act on *network component* is called individual properties. These properties by themselves are difficult to check because we often do not know a precise view of the internal configuration of each *network component*. Concurrently, the set of properties that arise when external influences act on a *network component* is called global properties. Global properties are difficult to verify because they can include the verification of several individual properties of each *network component* involved in the process. Considering these definitions, a desirable property is the set of individual and global properties of a *network component* that does not violate a network administrator's requisites to the network.

To the final question 3) it is important to understand that the set of desirable and undesirable *network components* properties are dependent on each context. Desirable properties depend on the requirements of each network and what each administrator considers important. However, there is a consensus that the absence of any property that constitutes a threat to the network's correct functioning should be guaranteed. However, this discussion is long. We address it in the next chapter.

When a desirable property is not respected by a *network component* we say that a property violation occurs. Two principal strategies to test whether a property is being satisfied by a *network component* are represented by monitoring and formal verification. Monitoring techniques can check a property violation on *network components* because it observes the effects that this violation produces on the monitored data. A perfect monitoring strategy, in theory, would analyze all packets, and all states at all times. However, it could only catch a *property violation* after it occurred. In practice, it is not possible to save all packets and consider all states of each network component due to scalability restrictions. Thus, it is necessary to *reduce this search space* and consequently not obtain an optimal monitoring system. Concurrently, formal verification allows us to model properties that do not depend on observing the execution data of the *network components*. However, even a perfect formal verification checker in the context of a given *network component* could not test a property that depends on the real-time interaction with other components. These limitations show us that it is not possible to obtain a complete testing scheme using these strategies alone.

However, we aim to improve this test process to an enhanced design. A more effective way of testing properties on *network components* would combine monitoring and formal verification advantages and try to avoid their limitations. We can combine them sequentially or concurrently. The concurrent case executes a monitoring strategy on one side and formal verification on another side. It is similar to executing them individually. Consequently, reduced advantages can be obtained in this combination. The other alternative is to combine them sequentially into two layers where one can improve the result of the other. A monitoring layer could check property violations in *network components* considering an almost real-time perspective. A formal verification layer could check properties that are likely to occur - but have not yet occurred.

It is crucial to emphasize here that it could not be the inverse combination, *i.e*, first a formal verification layer, and then a monitoring layer. The main reason is that the monitoring layer being over a verification layer will always find something that is not modeled in the formal model. This will happen because a formal verification model cannot preview property violations resulting from accidents. Consequently, the monitoring layer in this case will be constantly not synchronized with the formal verification layer. For this reason, we understand that a monitoring layer coming first serves as a source to generate a verification layer in line with what happens in the network being able to extend this with more complex analysis.

1.3 Problem Statement

This thesis aims to tackle the following general problem:

"Devices, protocols, and system administrators change and evolve at any time. They should respect a predefined set of properties that define when they are functioning correctly. To guarantee the absence of property violations, it is possible to use network monitoring and formal

verification to jointly generate a system able to test, detect, classify, and treat property violations?"

1.4 Hypothesis & Research Questions

To overcome the limitations exposed in the context of network testing, particularly in terms of property checking, this thesis presents the following hypothesis.

Hypothesis: **computer networking systems considering their communication devices and their interactions can be verified using the composition of monitoring and formal verification techniques to test property violations using a verification process**

To guide the investigations conducted in this thesis, the following research questions (RQ) associated with the hypothesis are defined and presented.

RQ I. *By monitoring, is it possible to collect devices traffic features, such as packet statistics and communication patterns, to test network properties and detect misconfiguration?*

RQ II. *By modeling the communication pattern of network devices, such as received and sent packets, is it possible to check global and component-specific properties?*

RQ III. *How is it possible for a monitoring layer and a formal verification layer be used for testing computer networks?*

The methodology employed to show the feasibility of the proposed approach is based on the evaluation of two components. The first one represents a monitoring layer for detecting property violations. The second one represents a formal verification layer able to perform network testing. Together they compose a solution that combines network monitoring and formal verification for testing in a novel integrated design to search for property violations.

1.5 Goals and Contribution

Although the use of SDN and NFV promotes more flexibility in the design of the network and facilitates the programmability of network equipment, this also makes it more difficult to ensure that network configuration is free from property violations. We propose here a solution that explores the combination of two studies based on network monitoring and network testing based on formal verification.

The first contribution is represented by a monitoring layer and its prototype (named ARMOR) and intends to show a comprehensive architecture to manage property violation diagnosis on software-based networks. By using monitoring sampling tools, such as SNMP, REST API, and sFlow, we aim to reach flexibility, accuracy, and automatization of our solution. Our research relies on machine learning techniques, such as decision trees (AL-SHAER; AL-HAJ,

2010a). Our goal with this is to achieve high accuracy in data classification and automatization of the detection and remediation process.

The second contribution is represented by a formal verification layer able to perform network testing and its prototype (named NetWords). This thesis presents a novel approach to representing networks based on regular grammar that can be generic enough to represent the entire network and specific enough to model single devices. Our model uses as an information source the information of messages that enter and leave devices. In this way, it is not necessary to know the internal state of a device to model. Our model infers them by observation. By using the actions performed by network devices, it is possible to generate a description of the network very similar to grammar. This grammar represents the most used network communication paths and represents a model able to check the most well-known global network properties: reachability, isolation, loop freedom, a black hole, and specific issues related to devices, such as an internal configuration fault. Also, we analyze the necessary computational resources for the implementation of our grammar.

Our results suggest the following contributions:

1. A complete monitoring framework able to detect, classify, and remediate network violations;
2. A complete network testing engine able to model, analyze, and test network invariants and properties using grammars;

1.6 Organization

The remainder of this thesis is organized as follows:

In **Chapter 2**, a set of important background concepts and studies related to this thesis are reviewed. Initially, a brief overview of the evolution of property violation context is presented followed by an overview of techniques to verify, understand and detect property violations.

In **Chapter 3**, we present the solution proposal covering aspects such as limitations on network verification and the conceptual description of our solution.

In **Chapter 4**, we present our proposed monitoring layer for detecting property violations including the description and the evaluation of its prototype named ARMOR.

In **Chapter 5**, we present our proposed formal verification layer to perform network testing including the description and the evaluation of its prototype named NetWords.

In **Chapter 6**, we present the final remarks, future work, and conclusions. Also, answers to the fundamental questions proposed are discussed and justified.

2 BACKGROUND & RELATED WORK

In this chapter, we provide a brief overview of the main research efforts to support network testing and verification for Software-defined Networking. The organization of this chapter is as follows: Section 2.1 presents the most used design to perform network verification and testing. Section 2.2 discusses the most representative techniques to perform network verification and testing. Finally, Section 2.3 presents a related work with highlights in the research related to property checking to illustrate how this research field is evolving.

2.1 Network Verification and Testing

Network verification and network testing are terms that can refer to different activities related to configuring and checking a network (MATTEIS; SECCI; ROSSI, 2021). In general, network verification is a process that verifies that the network is functioning correctly and satisfies high-level requirements defined by the network administrator. Network testing is an orchestrated process of running multiple tests to assess network functionality (LI et al., 2018). Frequently, network verification can be done before or after network testing and is more comprehensive and theoretically addresses aspects such as topology, protocols, policies, and architecture. Network testing, on the other hand, tends to be more specific and practical, focusing on aspects such as connectivity and quality of service (GARG; TOTLA; KHANDELWAL, 2021).

As computer networks were built using several heterogeneous devices, it is difficult to maintain a dynamic and reliable view of the current state of each network component in a given time interval (LI et al., 2018). To deal with this complexity, verification and testing were proposed by several studies concentrating on defining strategies to tackle this reality. Consequently, this studies addressed three main strategies to collect network information: static, incremental, and dynamic verification and a set of representative techniques to help them (Model Checking, Symbolic Simulation, SAT Solver, and Theorem Proving). Next, we review the most important ones in detail.

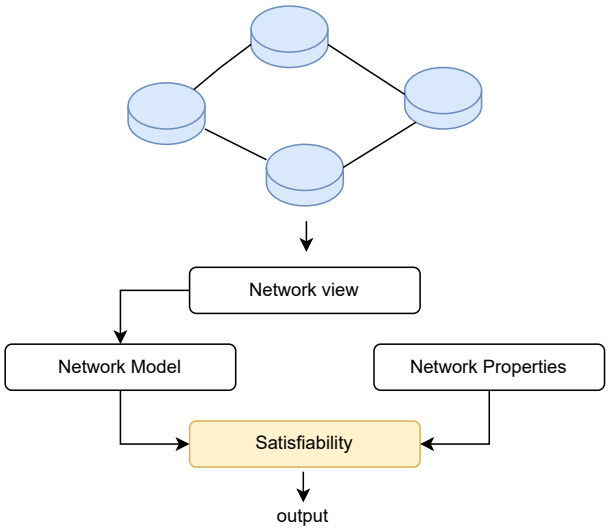
2.1.1 Static Verification

Also known as offline verification (JARRAYA; MADI; DEBBABI, 2014), static verification is used to check fixed network configurations that do not change frequently, such as IP addresses of network devices. Static verification is made by assessing the source code, but not executing it. Generally, a copy of network devices and their states in a given moment, named *snapshot*, is used (TESTA et al., 2012). Using snapshots it is possible to verify network properties, such as reachability and isolation. As a disadvantage, often the verification process can be compromised since every snapshot represents an older network state that can be outdated. An example of a

tool for static verification is Minesweeper (BECKETT et al., 2017), which can check properties such as reachability, router equivalence, and fault-tolerance.

Static verification can be combined with formal methods (TESTA et al., 2012). For example, static verification combined with symbolic simulation avoids the full representation of the network states and simulates possible network inputs instead of the actions of individual components. Using this technique, it is not necessary to model all the elements that compose the problem domain. Static verification can help to ensure that a network design is secure, reliable, and efficient. By identifying and fixing potential errors and vulnerabilities in the network design, this verification can identify and prevent potential errors or vulnerabilities. In general, the design of static verification is represented by Fig 2.1.

Figure 2.1 – Static Verification General Design



Source: by author

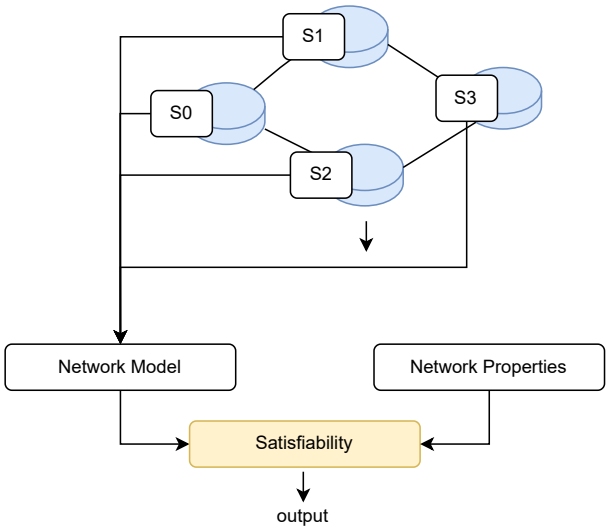
The main steps in static verification typically involve the following: a step of obtaining a static *Network View* used to construct a *Network Model*. This step involves defining the structure and behavior of the network using mathematical or logical language. A set of *Network Properties* specifies the properties or requirements that the network or protocol should satisfy. This involves defining the desired behavior of the network and identifying any constraints or limitations. Once the model has been created, it is analyzed using formal methods or other techniques. The goal of the analysis is to identify any errors or vulnerabilities in the network design or protocol and produce the results (output). It is checked using the *Network Model* to check the satisfiability (if the property is respected or not).

Due to the fundamental limits imposed by the theory of computation, such as, Turing's halting problem (LUCAS, 2021), an undecidable problem, static verification cannot extract the behavior of all monitored elements perfectly. Thus, static verification address this problem not focusing on the completeness, but focusing on the approximation of the real states of monitored programs (QADIR; HASAN, 2014).

2.1.2 Dynamic Verification

Sometimes, static verification is not able to prevent the misconfiguration fast enough until the next snapshot is collected (GUPTA; IRANI; SHUKLA, 2003). In this case, it is necessary to monitor the network in real-time to determine the source of a potential misconfiguration (HAYDEN et al., 2012). Frequently, the dynamic verification scope allows assessing and running the source code of network components, thus being capable of performing network verification analyzing implementation bugs (*e.g.,* a bug in the network controller code or a policy misconfiguration).

Figure 2.2 – Example of Dynamic Verification Process



Source: by author

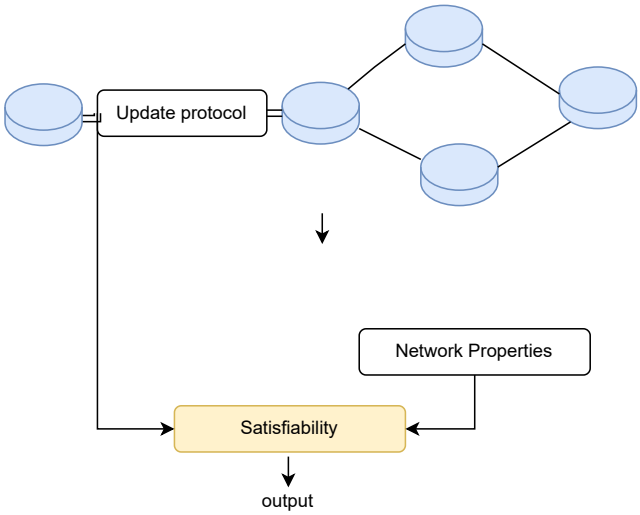
Dynamic Verification follows the design represented by Fig 2.2. Instead of a snapshot representing the information of all network components, a scheme identifies and verifies each network component. This scheme can be a program that can collect runtime information within the network components (S0, S1, S2...) and export this information. All this information is organized in *Network Model*. This *Network Model* is combined with *Network Properties* to determine if a set of *Network Properties* is satisfiable.

Dynamic verification is frequently used to check vulnerabilities that compromise the network after its components have been deployed. For example, a well-defined network router with a well-constructed formal model will pass through a static verification process regarding network properties when checked (OSMAN; ROBERTSON, 2007). However, an arbitrary error, such as an internal and unpredictable fault, can cause a network threat. Concurrently, it is arguable that the cost involved in dynamic verification can be relatively high, however, the benefits are crucial for network correctness.

2.1.3 Incremental Verification

Incremental verification is used to check network configurations when, for example, an update is necessary or a parameter is modified (SETHI; NARAYANA; MALIK, 2013a). This approach is generally used in the modular development of software, in which small parts are verified incrementally, and then deployed. Note that in this case there is an update protocol in which various steps guarantee that a change is well defined and executed in a way to not compromise the correctness of the network. For example, incremental verification fits perfectly in situations where a network outage causes some equipment failure, requiring a set of devices to be removed from the network and substituted by others.

Figure 2.3 – Example Incremental Verification Process



Source: by author

Incremental Verification frequently follows the design represented by Fig 2.3. There is an *update protocol* that defines which actions are allowed and which ones are not. The idea is to check every action while updating is performed. All this information is organized and combined with *Network Properties* to determine if a set of *Network Properties* is satisfiable.

For example, network calculus is a formal representation related to incremental verification. Network calculus generates insights about execution performance results that the network system will have even in the case that an optimal result is impractical (CIUCU; SCHMITT, 2012). Rothenberg et al (ROTHENBERG; DIETSCH; HEIZMANN, 2018) state that there are challenges involving incremental verification, such as, 'you should run the tests that produce the highest coverage first?' OR should 'the coverage produced by a test in the past be the same after the design changed?". While it is easy to concentrate on functional aspects of verification, incremental verification is frequently used as a step in the middle of static and dynamic verififcation (BIRKNER et al., 2020).

2.1.4 Network Testing

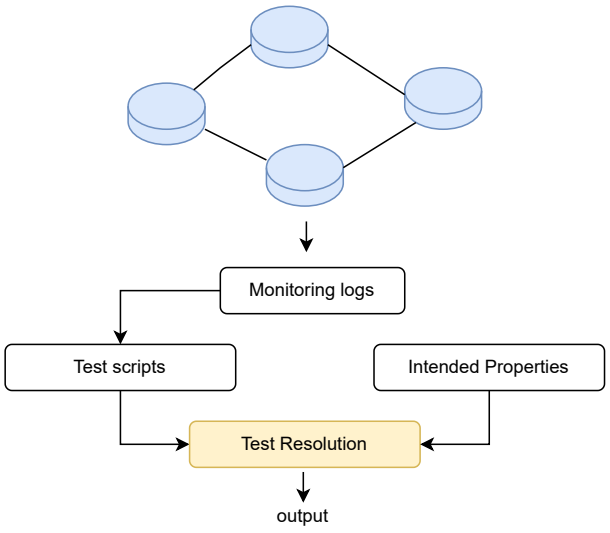
The majority of resources available in the literature about network testing are concerned with test performance and frequently the testing concept is confused with the verification concept. Testing is the process of searching for an implementation bug. It can use a verification strategy, for example, static, dynamic, or incremental. The verification strategy can use a formal method, for example, model checking, symbolic simulation, or SAT (LI et al., 2018). A few works are centered on the pure definition of testing and frequently they are more concerned with checking property violations (maybe because authors prefer to use formal methods for this).

An important aspect of network testing is the state associated with the dynamic components of the network. Considering the state of network components, two designs can be used:

- *Stateless testing*: historical communication patterns of devices and packets involved are not used as information in the testing process. The only information used is the current state of packets and devices. In general, it is the most used strategy for testing (CARDOZA et al., 1997).
- *Stateful testing*: historical communication patterns, context-dependent policies, and inter-device interactions of devices and packets are considered in the testing process. There is a high cost to store this information and it is a high processing task. For this reason, it is challenging to perform this approach optimally in general (HANNEL et al., 2007).

Note that the testing process is done over the network, that is, the administrator needs to know what he wants to test and then test it specifically using *Monitoring logs* and *Test scripts* to represent what should be tested (Fig 2.4). It may even have a moment of saving historical tests, but it never escapes having to think of new ones.

Figure 2.4 – Example of Network Testing Proces



Source: by author

When we consider the internal implementation of network devices, there are three types of testing approaches to follow (LI et al., 2018):

- *White-box testing*: when the internal state of the component is known and it is possible to see its implementation and test specific properties considering a fine-grain view of its internal functioning(CHAN et al., 2022);
- *Black-box testing*: when the internal state of network components is not known, thus resulting in test samples that represent the inputs and outputs that the component recognizes (THOMBARE; SONI, 2022);
- *Defect-based testing*: a technique that generates test cases based on defects observed instead of using the traditional coverage tests(FAQIH; ZAYED, 2021).

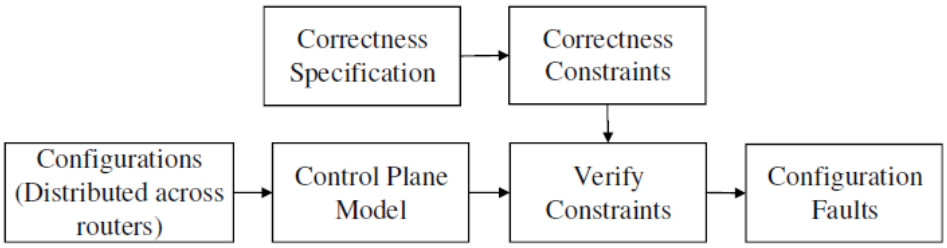
Several combinations are possible considering network testing, network verification, and formal methods. In the context of computer networks, it is difficult to maintain a dynamic and reliable test scheme considering the internal state of each network component in a given time interval.

2.2 Representative Techniques on Network Verification and Testing

Formal verification is the act of (i) proving the correctness of the design/implementation of a system or (ii) finding a counter-example presenting a violation of its predetermined properties (ALUR, 2011). In general, formal verification research is concerned with four major techniques: model checking, symbolic simulation, SAT, and Theorem Proving (LI et al., 2018).

The old design to verify computer networks was depicted in Fig 2.5.

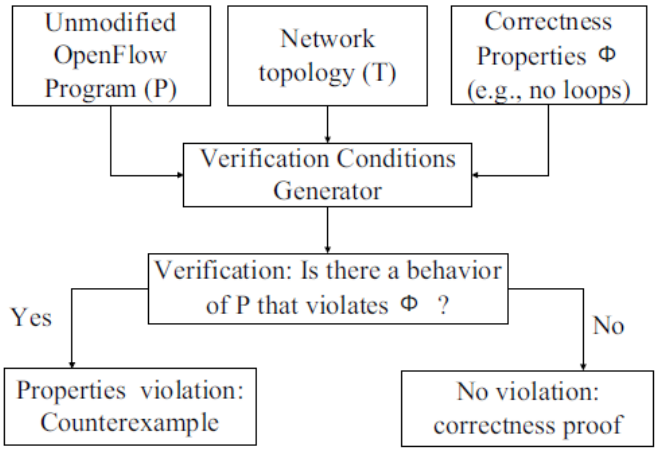
Figure 2.5 – Traditional Verification Design



Source: (LI et al., 2018)

The old verification scheme was represented by two streams running in parallel (LI et al., 2018). The first of them was represented by a *Correctness specification* of what is correct and the second by a specification of which *Configurations* the network should respect. Note that there was a lot of human intervention because we need to obtain a *Control Plane Model*, a define *Correcteness Constraints* to finally execute the *Verify Constraints* and obtain the *Configuration Faults*.

Figure 2.6 – Advanced Verification Design



Source: (LI et al., 2018)

Nowadays, the process has evolved to now have software support on the network where more complex solutions can be combined or executed without human effort (Fig 2.6). There is still a considerable human contribution to think, define, and program what is correct or not in the network. However, after that, the automation is greater. There is an *input program specification*, *network topology*, and a *description of correctness*. The next step is a processing phase where the checking process checks if *verification conditions* hold. Finally, the output produces a *counterexample* of the answer if the property is correct or not. Several techniques can compose the "Verification" step. Each one has pros and cons. Next, we comment on the most important aspects of each technique.

2.2.1 Model Checking

By definition, model checking is a verification technique[1] able to represent a problem domain as a set of finite states and the conditions that trigger a transition from one state to another. It is frequently used to model finite-state concurrent systems and in general, it is an exhaustive verification procedure (EMERSON, 1990). Model checking was in the first set of techniques designed to be an alternative to exhaustive search techniques common in the early days of circuit verification (BURCH et al., 1994). It was developed independently by Clarke and Emerson (CLARKE; EMERSON; SISTLA, 1986) and by Queille and Sifakis (QUEILLE; SIFAKIS, 1982) in the early 1980s.

Model-checking implementations include (i) a model and (ii) a set of properties. Generally, the model is represented as a finite-state machine (BAIER; KATOEN, 2008). This finite-state

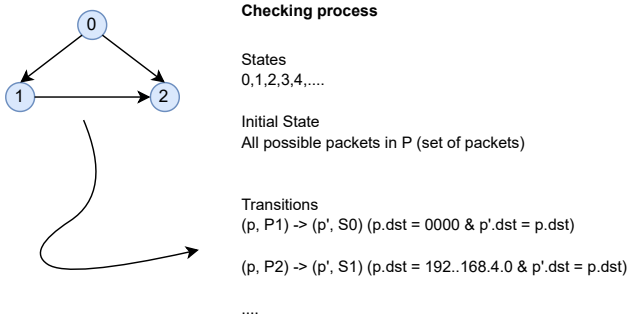
[1]Research in this field frequently uses "model checking techniques" instead of "model checking methods". For clarity, even knowing that model checking is a formal method, we will use the word technique for model checking to maintain consistency with the literature.

machine and the set of properties are compared to determine if the system behavior conforms to its specification. It is an important technique in the history of formal methods because all previous attempts to validate systems were based entirely on informal techniques. It was only in 1995 that Clarke and his students at Carnegie Mellon University (CMU) used Symbolic Model Checking (SMV) to verify the IEEE Future+ cache coherence protocol (STERN; DILL, 1995). They found several previously undetected errors in the design of the protocol thus confirming the utility of such a technique. This was the first time that formal methods were used to find errors in an IEEE standard.

As principal advantages, model checking is fast, does not need formal proofs, can produce counter-examples, and is useful to represent many concurrency properties (BAIER; KATOEN, 2008). In the context of computer networks, properties such as reachability, loop freedom, and isolation can be verified using Conditional Temporal Logic (CTL) over the finite-state machine representing elements from the network (BRIM; CRHOVA; YORAV, 2002). Model checking has been extensively used to perform static verification (a network verification technique presented in the previous section) and it is very flexible. This flexibility is observed when model checking is presented in research studies combined with symbolic approaches, logical formulas, statistical transitions, and Markov diagrams (BAIER; KATOEN; HERMANNS, 1999).

As principal challenges, there are scalability issues related to state explosion and data paths: the comparison between the finite-state machine representing the model and its specification can be a very time-consuming task, thus becoming a performance bottleneck. Binary Decision Diagrams (YANG et al., 1998) can be used to represent state transition systems more efficiently. Additionally, partial order reduction can be used to reduce the number of states that must be enumerated. A large number of other techniques for alleviating state explosion include (i) abstraction (CLARKE et al., 2001); (ii) compositional reasoning (BEREZIN; CAMPOS; CLARKE, 1997); (iii) symmetry (CLARKE et al., 1998); (iv) cone of influence reduction (PARK; KWON, 2006); and (v) semantic minimization (ALUR et al., 1992). Further discussion about this can be found in Clarke *et. al* (CLARKE; WING, 1996).

Figure 2.7 – Example of Model Checking Process



Source: by author

The basic flow within model checking follows the notion of network states (Fig 2.7). A set of states that a network component can assume. This can be infinite, but here we try to imagine the most representative ones. There is a differentiated state called *initial state* and then rules that define the transition from one state to another, *state transitions*. A human should think about and define these transitions by observing the system modeled.

In the specific case of computer networks, the high-level view of the SDN control plane facilitates the model-checking process even with the presence of an arbitrary number of packets. In general, applying model checking to large networks is challenging. Network components can generate packets in an unbounded manner, and these packets can be processed in arbitrary interleaved orders. Thus, the state space remains unbounded, and the topology is fixed with several network components.

Directions for future research indicate that the use of abstraction techniques should be combined with others to enhance performance (PRABHU et al., 2020). Computer networks are composed of several components that interact with each other. Many systems are composed of several components that interact with each other, and new model-checking techniques will be needed to verify composite systems. Consequently, there is space to research the combination of model checking and deductive verification, for example, to produce interfaces more suitable for system designers (LI et al., 2018). Additionally, there is a need for the development of methods for verifying parameterized designs and practical tools for real-time and hybrid systems. Scalability is one of the biggest challenges in model checking due to the combinatorial explosion of states as the system size increases. Real-time applications can produce so much information that a model-checking scheme can verify.

2.2.2 Theorem Proving

Theorem proving is a set of procedures able to determine if a formula P is a logical consequence of other formulas (BACHMAIR; GANZINGER; WALDMANN, 1994). The first step frequently is represented by a set of assumptions, which are formulas whose logic value is considered true by default. The second step is the definition of the formulas that describe the problem domain. The proving process can be done when we consider the following rules to reason about formulas representing the domain:

- *Refutational theorem provers* can analyze if the negation of P triggers an inconsistency in the formulas determined as axioms and the formulas that represent the problem domain. The checking process of these formulas is performed by the use of deductive inferences predefined by a set of inference rules (BACHMAIR; GANZINGER; WALDMANN, 1994).
- *Deductive provers* can directly apply resolution techniques using the axioms and problem domain formulas to produce a logical consequence. This logical consequence frequently

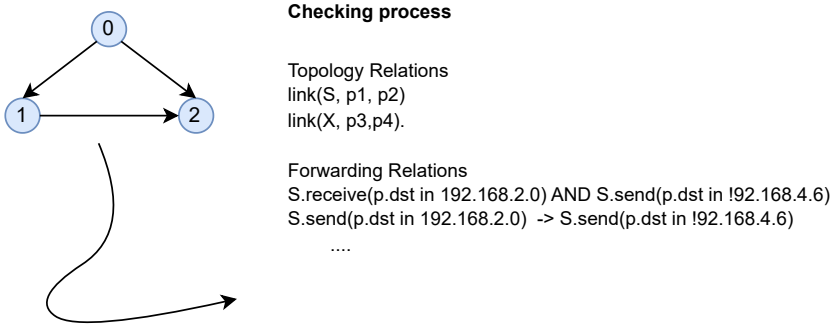
represents a property of the domain modeled. The main advantage of this proving process is the simplicity of the data structures involved in representing clause sets and the operations performed at each iteration - the resolution process. As a disadvantage, the prover can produce a relevant or irrelevant result if there are no additional constraints to bound the process (REITER, 1976).

- *Inductive provers* is a set of techniques that consider a base case as proved, generally defined a priori without using any knowledge of derivative cases. The second step consists of the resolution that the statement holds for any given derivative case of the initial base case (AVENHAUS et al., 2003).

Some of the earliest work in formal hardware verification dates from 1986 (CLARKE; EMERSON; SISTLA, 1986) and was characterized mainly by the heavy focus on rigor and strong abstraction capabilities. Nowadays research still evolves proposing solutions related to selectively applying different levels of abstraction, increasing the degree of automation, and automating the generation of simulation relations. The research on automatic theorem provers is intense too mainly in the context of pipelined microprocessors (NAWAZ et al., 2019).

Here we are not trying to model the states but the relationships between the device by generating axioms. Axioms are statements that are assumed to be true without proof. By using the axioms, such as *topology relations*, we generate a set of predicative that can be TRUE or FALSE (for example, *forwarding relations*). There is a human effort to determine axioms that a network component must accept, such as "network component A must communicate with network component B" and thereby extract theorems from it. The theorem proving general process includes a set of axioms and predicatives connected by logical operators (Fig 2.8).

Figure 2.8 – Example of Theorem Proving Process



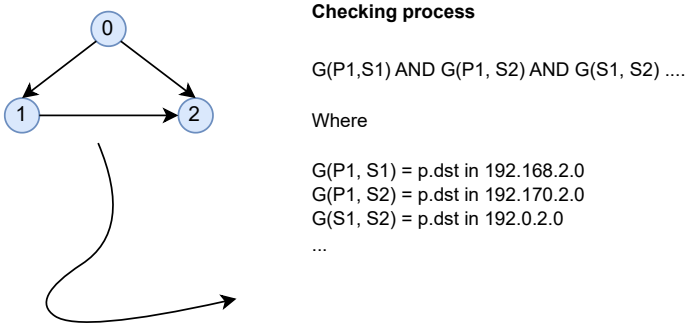
Source: by author

2.2.3 SAT Solvers

The satisfiability problem (SAT) is central in Computer Science, with both theoretical and practical interests (PRASAD; BIERE; GUPTA, 2005). Its formal definition is "Given a circuit with N inputs which configuration of these N inputs will produce 1?". SAT was the 1st NP-complete problem because the most reliable way to produce an answer to its main question is by testing all possible inputs, a total of 2^N cases. Due to the several applications on the logical synthesis of electronic circuits, SAT received a lot of attention [1960-now] and for this reason, there are very efficient implementations of applications. SAT has become the “assembly language” of hard problems because so many problems can be modeled as a conjunction of AND and OR clauses (even problems related to computer networks).

The basic idea is to model the network component as a logical formula and thus try to determine where it produces TRUE and where it produces FALSE (see Fig 2.9). Consequently, we need to describe all network relations and components as logic formulas.

Figure 2.9 – Example of SAT Solver Process



Source: by author

SAT-based techniques suffer from the limitations related to the satisfiability of boolean functions, which is an NP-Complete problem (RAMAN; RAVIKUMAR; RAO, 1998). Consequently, using this always involves hard optimizations and demands high processing. However, SAT-based techniques also suffer from scalability issues, since the complexity of model evaluation grows exponentially.

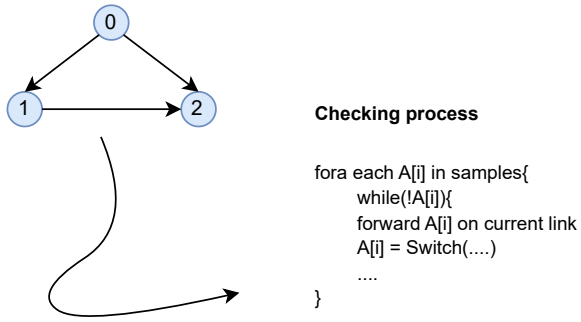
In the specific case of computer networks, techniques based on SAT provide a means of expressing network properties using logical formulas and solving the satisfiability of specific predicates, consequently verifying if the properties hold in the network. Research in this field is intense, encompassing techniques to model the network as a logic formula that enables the use of logic reduction techniques, such as Karnaugh maps and reduced binary decision diagrams (SOOS; MEEL, 2019).

2.2.4 Symbolic Simulation

Symbolic simulation (sometimes called symbolic execution) was proposed in 1990 for Bryant and Seger (BRYANT, 1990). The idea here is related to the limitations faced by SAT solvers: given a circuit with N inputs, instead of checking its 2^N inputs, we extend this logic to support an additional logical value, the 'don't care' value, represented by the X symbol. Consequently, the logical possible values to a sentence become (0,1, X) instead of only (0,1). This simple modification on the core of logical thinking enables powerful simplifications: the state 000 and 001 can be represented by the symbol $00X$, a direct implication that reduces drastically the number of states to represent the problem (for example, think about the state $0XXX$). Consequently, a practical way of generating an execution tree was created, where nodes are program states and edges are executions enabling the simulation of many possible executions simultaneously (DOBRESCU; ARGYRAKI, 2014).

Due to its scalability advances, symbolic simulation represents an important research field responsible for (i) accelerating simulation; (ii) generating new forms of formal verification to circuits with large memories; and (iii) increasing generalizations as formal verifiers to be used in more sophisticated circuit models (GIELEN; WALSCHARTS; SANSEN, 1990). In the symbolic simulation, the verification process is performed using a set of equations or formulas that describe its behavior. These equations can be manipulated mathematically to derive new information without actually executing it. It can be much faster than verifying a network component by executing it, however for complex systems with many variables and dependencies, we will need human intervention and knowledge to create the sets of equations or formulas. (see Fig 2.10)

Figure 2.10 – Example of General Symbolic Simulation Process



Source: by author

In the context of computer networks, verification issues about packet-processing pipelines can benefit from symbolic simulation because two elements generally do not read or write in the same state concurrently, thus reducing the chances of path explosion. Properties such as crash freedom, bounded execution, and filtering are frequently studied. Despite the great flexibility in modeling network packets, symbolic simulation suffers from the limitation of expressing or

checking network properties that depend on device state, such as isolation(QADIR; HASAN, 2014). Additionally, scalability issues exist and the frequent simulation of packet traversal through the network does not include the modeling of the internal components, e.g. the source code of a network component (BRYANT, 1990).

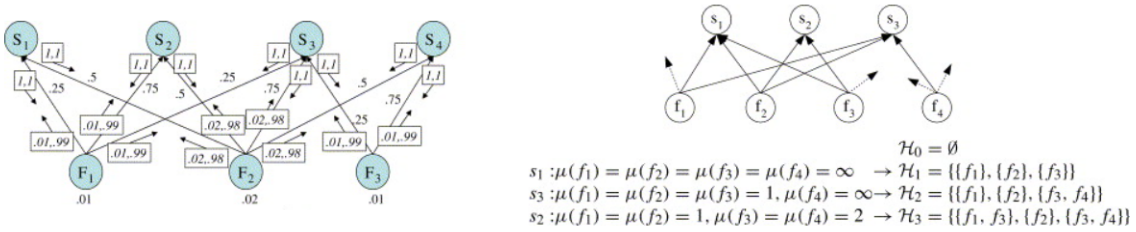
Open issues in this research field include debugging aspects since the set of simplified states can introduce errors. Another interesting investigation is the search for new useful ways to compare coverage of using symbolic simulation with coverage of conventional techniques. Finally, an interesting question is how many bugs can symbolic simulation find in comparison with others (DILL; TASIRAN, 1999).

2.2.5 Grammars

Network models and associated optimization algorithms are important to ensure the correctness and efficient operation of a network component. Such models are designed to formulate problems occurring in the design of a new component or checking older ones. One example is the Pyretic language and system itself(REICH et al., 2013). Pyretic is a language and system that enables programmers to specify network policies at a high level of abstraction, compose them together in a variety of ways, and execute them on abstract network topologies (REICH et al., 2013). The authors have introduced an abstract packet model, an algebra of high-level policies, and network objects.

Fault localization problems are another example of the use of network models. By using context-free grammar as a fault propagation model the authors address the problem of locating the source of a failure (STEINDER; SETHI, 2004). The problem is a limited class of context-free grammar models, semantically equivalent to dependency graphs, and can be transformed into a zero/one integer linear programming problem. Using solutions to integer linear programming problems, optimal (but complex) and suboptimal solutions can be found.

Figure 2.11 – Grammar use for Fault Propagation



Source: (STEINDER; SETHI, 2004)

Context-free grammars (CHEN; FU, 2022) are expressive enough to capture the recursive syntactic structure of represented languages. Context-free grammar is composed of a set of rules where a rule is typically defined as a name and an expansion for that name. The Backus-Naur

form (BNF) is a formal notation used for encoding the grammar of a language in a human-friendly. A rule of the BNF notation has the following structure (QADIR; HASAN, 2014):

"name -> expansion" where the symbol -> means 'expands to' or 'may be replaced with'.

According to Figure 2.11, grammars can be used for modeling fault propagation in complex systems. Firstly, we define a grammar that captures the structure of the system being modeled (left). The grammar should include a set of production rules that specify how the system's components are combined to form larger structures. Once the grammar has been defined, assign fault models to the elements of the grammar and propagate faults through the grammar. This process can identify the points in the grammar where faults can occur and determine how those faults propagate through the system.

2.2.6 Network Monitoring

Network monitoring is the practice of collecting network information, including device state and protocol execution statistics to obtain a fine-grained view of the network (ENGEL et al., 2000). This network view can be used to detect malicious activities, misconfigurations, performance bottlenecks, and general faults. As computer networks produce a considerable amount of data, the process of monitoring all network components can be a difficult and time-consuming task (CHOWDHURY et al., 2014). For this reason, there is a variety of network monitoring schemes.

The most fundamental strategy to perform network monitoring is polling. Polling-based strategies can be divided into three subsets:

- *Direct monitoring* (LIMA et al., 2010) is represented by strategies that actively query network devices for their execution information with a clear relationship between the monitor and the monitored component. An example of this is the SNMP protocol[2].
- *Indirect monitoring* (KIM; FEAMSTER, 2013) is represented by strategies that query the information known by a component that monitors the network. An indirect monitor may be used to query the network controller to obtain information instead of querying directly individual switches (LEE; LEVANTI; KIM, 2014). For example, the SDN controller knows a considerable amount of information about switches and traffic flows and exports this information to other devices.
- *Distributed monitoring* (FEAMSTER; REXFORD; ZEGURA, 2013) is the case of direct or indirect monitoring occurring in several devices at the same time. There is a need for a centralized component to store and analyze all the information collected by the distributed monitors. An example of this is the sFlow[3] protocol present in network switches.

[2]https://tools.ietf.org/html/rfc1157
[3]http://www.inmon.com/technology/sflowTools.php

Due to the vast amount of data produced with monitoring tools, it is prohibitive to store information about every packet. For this reason, there is a tradeoff between storing a high amount of monitoring information and using a lot of resources to process it, which typically may lead to non-optimal solutions. However, it is not possible to rely solely on packet information to build a network profile. The existence of mutable datapaths when third-party middleboxes exist can change the routing schemes based on their internal policies (KAZEMIAN et al., 2013a). Thus, it is necessary to check if the insertion/removal of a new policy inside a specific middlebox can affect packet counters.

Monitoring can be performed on a single device or in a distributed design (ACETO et al., 2013). Distributed monitoring designs are often used in tasks that require a fine-grained view of the network, such as network management, visualization, and performance evaluation. In general, monitoring can encompass several layers, such as middleware, and middleboxes; should respect properties such as scalability, extensibility, accuracy, elasticity, adaptability, resilience, etc...

2.3 Related Work

Formal methods are powerful mathematical techniques for the verification and validation of software and hardware systems. They provide formal guarantees that a system works correctly and meets security and reliability requirements. This section discusses the main research efforts in the context of SDN related to formal verification and topics that have concerned computer network researchers in past years. The operation of network devices is often subject to several types of configurations, operational constraints, and security policies (WICKBOLDT et al., 2015b). This reality in addition to the increasing number of devices and their inherent heterogeneity brings network update difficulties to network operators. We will discuss the main challenges and limitations associated with the adoption of formal methods.

2.3.1 Model Checking

The main concern about verification using model checking in SDN research is guaranteeing that the implementation of SDN applications and their communication standards conform to the network's correctness and safety properties. The centralization of the control plane enhances the use of software abstractions to compose the network, encouraging the verification of these elements to enforce network correctness. The main algorithmic and modeling challenges in model checking include:

Network State Explosion: The number of states in the network can become too large to store or process, causing the model checker to run out of memory or time (ROUMANE; KECHAR, 2022). As the information about network topology, devices, and communication paths is visible to the control plane, it is easy to elaborate a detailed finite state machine (FSM)

to model the network in a given moment. Unfortunately, even in this environment the state space of verification still can be huge because there are several network components, communication paths, and packets processed to be considered by the control plane (PADON et al., 2015), thus presenting a **scalability** issue to verification. According to Majumdar et al. (MAJUMDAR; TETALI; WANG, 2014a), a solution for this limitation appears with reduced FSM to model the network using partial-order reductions and abstractions techniques to reduce the state space generated. For example, forcing a model representing a network switch, to generate a state for the processing of all packets matched by some rule instead of generating intermediate states by successive match actions for each packet.

Scalability: The state space of the system being checked can grow exponentially with the number of variables, making model checking infeasible for large systems. Scalability in model checking not only encapsulates issues related to the state space of verification but encapsulates the processing time of the verification process (SABUR et al., 2022). According to Sethi et al. (SETHI; NARAYANA; MALIK, 2013b), the time between updates in the network state and the network controller is larger than the lifetime of the packet in the network, thus it is important to achieve fast verification schemes even in the SDN environments. Considering an FSM representing a set of network states, there is a subset of paths in this FSM that do not need to be explored since they are dependent, i.e, two or more paths that are similar but differ only in order of the events modeled. (YAKUWA; TOMIZAWA; TONOUCHI, 2014).

Incompleteness: Model checking can only check for properties that are explicitly specified (LI et al., 2022), and it may not detect errors that arise from unspecified or unexpected behavior. The ability to combine model checking with other techniques is studied in the literature frequently to improve this reality. For example, Verificare (SKOWYRA et al., 2014) enables the verification of not only correctness requirements but verifies too SLA, safety, and security requirements considering SDN components as a larger domain-specific system. Still, Verificare enables the checking of network requirements using different formalisms and verification tools freeing the network administrator to remodel the same model in different languages. Also, aspects such as the safety of network controllers are treated by (MAJUMDAR; TETALI; WANG, 2014b) that concludes partial order reduction techniques significantly reduce the state space when exploiting large-scale environments. Additionally, Matthews et al. (MATTHEWS; BINGHAM; SORIN, 2016) present the framework Neo, a framework that is designed for a protocol component that is instantiated and connected in an arbitrary hierarchy ensuring the correctness of the network regarding some network property. They use a parametric model Checker, Cubicle, to allow the verification of different dimensions of configurations such as cache coherence protocol using Input-output Automata formalism.

Difficulty in modeling real-world systems: The modeling process can be difficult and error-prone (KWIATKOWSKA; NORMAN; PARKER, 2022), and it may be challenging to accurately capture the behavior of complex systems with many interacting components. Model checking can be used too in the context of verification regarding the implementation of real-

world network components. An example is the framework NICE (CANINI et al., 2012) which can find bugs in the implementation of control plane and SDN applications. The authors emphasize again that the verification of SDN applications is a challenging task because of the large environment, thus generating a large state space for switches, input packets, and event orderings. To simplify the problem, often domain-specific solutions appear, but it is well known that the models generated are difficult to update and time-consuming. To overcome these limitations, the authors combine symbolic simulation to model checking to create Openflow-specific strategies such as reordering events to reduce the state space in flow tables.

Specifically, in the data plane context, FlowChecker (AL-SHAER; AL-HAJ, 2010b) uses model checking to identify intra-switch misconfigurations when analyzing a single flow table. They perform this by encoding the flow table into a binary decision diagram and then being able to find conflicting rules. Bifulco et al. (BIFULCO; SCHNEIDER, 2013a) present formal definitions for the interaction of flow rules entries in the network. They conclude that runtime is dependent on the number of rules in the switch and the challenge of its reduction as well as its consistency(HUSSEIN I. H. ELHAJJ, 2016). Kozat et al. (KOZAT; LIANG; KöKTEN, 2014) install optimal or near-optimal forwarding rules to verify topology connectivity and link failures. They guarantee the location of single link failure and probabilistically find multiple link failures considering latency performance however with a slight increase in bandwidth usage and forwarding rules. Nguyen et al. (NGUYEN et al., 2014) treat the problem of OpenFlow rule selection and placement problem ignoring the routing policy to use the verification of packets are delivered correctly. They propose an integer linear programming variant and argue that the placement of the network controller has an impact on system performance. PrintQueue (LEI et al., 2022) is concerned with the P4 context and studies a practical data-plane monitoring scheme for tracking the packets to understand the source of delays.

Finally, model checking can assist in the checking of network invariants, a real-world need. Anteater (MAI et al., 2011) and the authors conclude that it is feasible to check the switches in the data plane, however with overhead in the processing time of network snapshots. Another example is the Kinect language proposed by Kim et al. (KIM et al., 2015) that automatically verifies the correctness of control programs regarding temporal-specific properties, such as reachability. Finally, Vericon (BALL et al., 2014) can check infinite network states to guarantee network-wide invariants and does not use model checking for this agreeing with the limitations of model checking. The authors claim that model checking might identify errors but cannot guarantee that errors do not exist in the network. The NFV context faces the same difficulties of research about physical middleboxes, but potentially in a more challenging manner. One example is the dependency of network services (SHIN et al., 2015a) and how their communication can be protected, as well as the verification of network properties, such as loop-freedom (SHIN et al., 2015c). Again, these issues have a fundamental role in NFV research, policy management, and state consistency when searching for misconfiguration in the network. Additionally, in the middleboxes/NFV context, the challenge of dynamic datapaths is the main

concern. Research is concentrated on centralizing the information about mutable datapaths that reside in-network middleboxes/NFV.

2.3.2 Symbolic Simulation

Often, formal verification problems can be modeled as a set of states, actions, and properties. This set of states tends to be large enough if all admissible states are considered and inaccurate if a subset of possible states is ignored. Therefore, there are cases in which the use of finite-state machines is not suitable to model the verification problem. Symbolic simulation tries to overcome this reality but still depends on the expressive power of the underlying formalism used to model the network. If the formalism is too restrictive or limited, it may be difficult to accurately model the network and capture its behavior. As computer networks are dynamic systems that can change rapidly over time, a symbolic simulation may have difficulty keeping up with these changes and accurately modeling the network behavior (due to a combinatorial explosion of possible combinations of events and interactions). However, concise research and challenges exist in this field.

Scalability: Symbolic simulation can suffer from the same scalability issues as model checking, with the number of paths through the program or system growing exponentially with the number of variables (KUSHWAHA et al., 2022). According to Velner et al. (VELNER et al., 2016), it is difficult to extract the network state (considering all of its packets) due to the existence of stateful middleboxes. The authors show that safety checking (that includes properties such as isolation) is exponential considering the space used. In this way, other strategies need to be considered to verify the network. Existing research applied to middlebox verification can be applied in the NFV context too. For example, Aurojit et al. (PANDA et al., 2016) propose simplified middlebox models (ignoring irrelevant aspects regarding network properties, such as reachability, and focusing on the middlebox path that a given packet traverses). Regarding the verification of isolation and connectivity property, Panda et al. (PANDA et al., 2014) treat each dynamic datapath as a subroutine of a complex system represented by the network connected by switches and routers. Each invariant changes directly this model.

Network Path Explosion: The number of paths through the network can become too large to store or process (SÁNCHEZ et al., 2019), causing the symbolic simulator to run out of memory or time. This is a problem related to property evaluation since the process of checking this should consider all execution paths. Xie et al. (XIE et al., 2005) present a precise definition of reachability property to model the influence of packet transformation in the network. They do not specify SDN or NFV concepts to propose their solution, however, their analysis is useful to incorporate in these environments. The reachability property is rigorously defined and the paper provides a unified way for jointly reasoning about the effects the three very different mechanisms of packet filters, routing policy, and packet transformations have on reachability.

Note that it is important to check not only the network elements but also the input and output of each component. Wundsam et al. (WUNDSAM et al., 2011) present an approach to record and replay network events using the framework OFrewind. OFrewind enables the re-execution of network traces and repeats experiments to isolate network problems and assist network operators in the remediation phase of formal verification. Kuzniar et al. (KUZNIAR et al., 2012) present SOFT, which is a framework capable of checking interoperability between OpenFlow switches and identifying the key inputs that can generate inconsistencies in the network. They symbolically execute each network under test to define these test inputs. Fayazbakhs et al. (FAYAZBAKHSH et al., 2014) present the FlowTags approach that inserts the casual context of the packets in the network, helping the preservation of packet datapath context, which is essential to verify when mutable datapaths occur. The Flowtags approach can be used to verify middleboxes (or VNF) since it can operate jointly with header-space annotations.

In Kazemian et al. (KAZEMIAN et al., 2013b) SDN networks are checked using a tool named *NetPlumber*. NetPlumber operates jointly with the control plane to observe network changes and monitors every event in the network (such as a flow rule installation) using SNMP traps or polling the switches. In the case of policy violation detection, the violation can be checked by the network user. The authors advocate that NetPlumber can detect a simple invariant violations, such as loops and reachability failures. By the use of symbolic rules, Vera (STOENESCU et al., 2018) can catch bugs using a fair execution time - around 15s in the worst case. The execution flow is defined by a set of arguments, a P4 source file, and a set of commands that insert table rules by translating the P4 program/table rules into the SEFL language.

Difficulty in handling complex code: Symbolic simulation can struggle with complex control structures or loops (GIANNARAKIS; SILVA; WALKER, 2021), making it difficult to generate accurate or complete path conditions. Choi et al. (SHIN et al., 2015b) present a framework to check complex code related to network services and their components. They adopt STG symbolic verification for pACSR processes helping to verify and debug NFV-enabled network services. According to Velner et al., (VELNER et al., 2016), it is difficult to extract the network state (considering all of its packets) due to the existence of stateful middleboxes. The authors show that safety checking (that includes properties such as isolation) is exponential considering the space used. In this way, other strategies need to be considered to verify the network. Kuzniar et al. (KUZNIAR et al., 2012) present SOFT, which is a framework capable of checking interoperability between OpenFlow switches and identifying the key inputs that can generate inconsistencies in the network. They symbolically execute each network under test to define the test input.

2.3.3 Theorem Proving and SAT Solver

Theorem proving involves the use of formal methods to prove that a network protocol or system meets its specifications and behaves as intended. However, while theorem proving has been used for many years in computer networks, there are still challenges in this research. For example, the need for specialized human intervention and knowledge, the difficulty of scaling theorem-proving techniques to large and complex networks, and the cost and time required to perform theorem-proving analysis. Research in this area is focused to overcome these challenges and make theorem proving a more practical and widely used technique for ensuring the correctness and security of computer networks. Similarly, SAT solvers can be a powerful tool for analyzing and verifying network protocols and systems, they also face challenges in scaling to large and complex networks. Research in this field still faces challenges related to the following aspects.

Lack of guidance: SAT solvers can only answer "yes" or "no" to a given query (JABAL et al., 2019), providing little insight into the structure of the problem. Zhang et al. (ZHANG; MALIK, 2013) focus on the verification of configurations in the data plane using a static view of it (thus trying to be more explainable). The authors model network middleboxes using propositional logic models of a generic component and model a rich set of network properties, such as reachability and forwarding loop, reductions as Boolean formulas. In the context of BGP configurations, the Propane framework can check for misconfiguration using SAT (BECKETT et al., 2016). Finally, Vericon (BALL et al., 2014) can check infinite network states to guarantee network-wide invariants and does not use model checking for this agreeing with the limitations of model checking. The authors claim that model checking might identify errors but cannot guarantee that errors do not exist in the network.

Limited expressiveness: SAT solvers are designed to work with propositional logic formulas (SHUKLA; PANDEY; SRIVASTAVA, 2019), and cannot handle more complex forms of reasoning or higher-order logic. ConfigChecker (AL-SHAER; ALSALEH, 2011) models the network as a state machine and packet headers and location determines the full set of states. The semantics of access control policies was encoded as Boolean functions using Binary Decisions Diagrams (BDD). The authors can verify reachability and security properties using a complex formalism constructed over BDD. Lee et al.(LEE et al., 2015) propose a graph abstraction system (PGA) to express network policies and service chains with diagrams. In the context of automation of network policy instantiation, the framework MAPLE(VOELLMY et al., 2013) can define algorithms policies using abstractions that records reusable policy decisions. A similar problem is tacked by Kang et al. (KANG et al., 2012) that proposes the automatic transformation of policies by moving, merging, or splitting rules across a set of switches. The framework Merlin(SOULÉ et al., 2014) does a similar job and offers a language that can be used to verify if modifications in the network do not violate any global constraint.

Aurojit et al. (PANDA et al., 2016) focused on middleboxes (complex environment by itself), but their research findings can be inserted in the context of NFV. Their focus is on modeling mutable data paths to ensure correctness even in the presence of failures. They model the network as a logical formula representing middleboxes as static devices, such as switches. Using this formulation, they check invariants, however, they use limited invariants and simple high-level models due to scalability issues. In the context of P4 programs, p4V (LIU et al., 2018) is a tool to verify data planes described using the P4 programming language. There are several challenges in the verification of P4 programs, and the most concerning one is that the program is incomplete until packets traverse the network.

Difficulty in encoding specific problems: Certain problems may be difficult to encode in propositional logic, making them challenging to solve using an SAT solver. Vericon (BALL et al., 2014) presents the union between controller programs expressed as event-driven programs and invariants expressed as first-order logic formulas that are suitable/easy to prove the correctness of controller programs. Arashloo et al. (ARASHLOO et al., 2016) propose the SNAP compiler to optimize the placement and how to distribute network programs discovering its dependencies. They use a mix of linear programming and binary decision diagrams to provide its optimization. Yang et al.(YANG; LAM, 2013) present the ATomic Predicates(AP) Verifier able to speed the verification time when the properties such as reachability are verified in the network. McClurg et al. (MCCLURG et al., 2015) tackle the problem of misconfiguration in the network avoiding loops, black holes, and access control violations. They formalize the network as a program and use synthesis algorithms to find counterexamples in its execution. In (KANG et al., 2013), the authors present the experience of the use of formal techniques for modeling and analyzing SDN networks using VERSA and UPPAAL. (BIFULCO; SCHNEIDER, 2013b) verifies forwarding rules in a single OpenFlow switch showing the performance and a real example of use.

Kozat et al. (KOZAT; LIANG; KöKTEN, 2014) install optimal or near-optimal forwarding rules to verify topology connectivity and link failures. They guarantee the location of single link failure and probabilistically find multiple link failures considering latency performance with a slight increase in bandwidth usage and forwarding rules. Nguyen et al. (NGUYEN et al., 2014) treat the problem of OpenFlow rule selection and placement problem ignoring the routing policy to use the verification of packets are delivered correctly. They propose an integer linear programming variant and argue that the placement of the network controller has an impact on system performance. The checking of invariants is performed by Anteater (MAI et al., 2011) and the authors conclude that it is feasible to check the switches in the data plane, however with overhead in the processing time of network snapshots.

Table 2.1 is summarizing the main research in formal methods was created to understanding the current state of research in this area. This table may include information such as the year of publication, author, problem addressed, and techniques used. By examining such a table,

it is possible to identify trends in formal verification research, as well as areas that need more attention and development.

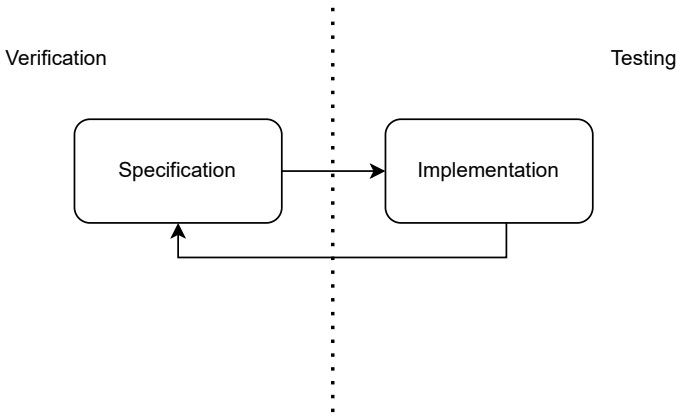
Table 2.1 – Summary of Main Topics in Formal Methods

Technique	Challenges	Research
Model	configuration checking	(SKOWYRA et al., 2014)
Checking	protocol checking	(MATTHEWS; BINGHAM; SORIN, 2016)
	SLA correctness	(SKOWYRA et al., 2014)
	misconfiguration detection	(ROTSOS et al., 2012),
	verification language	(SOULÉ et al., 2014)
	reachability	(KIM et al., 2015)
	intra-switch misconfiguration	(AL-SHAER; AL-HAJ, 2010b)
	checking incoming packtes	(LOPES et al., 2015)
	property checking	(ROUMANE; KECHAR, 2022), (KAZEMIAN et al., 2013b)
	invariant checking	(BALL et al., 2014), (LI et al., 2022)
	flow consistency	(HUSSEIN I. H. ELHAJJ, 2016)
	component update	(SETHI; NARAYANA; MALIK, 2013b)
	configuration checking	(MATTHEWS; BINGHAM; SORIN, 2016)
	intra-switch verification	(AL-SHAER; AL-HAJ, 2010b)
	flow verification	(BIFULCO; SCHNEIDER, 2013a)
	controller placement	(NGUYEN et al., 2014)
	network debug	(MAI et al., 2011)
	scalability	(SABUR et al., 2022)
Symbolic	interoperability	(KUZNIAR et al., 2012)
Simulation	reachability	(XIE et al., 2005), (PANDA et al., 2016)
	network debug	Wundsam et al. (WUNDSAM et al., 2011)
	path explosion	(SÁNCHEZ et al., 2019)
	mutable datapaths	(FAYAZBAKHSH et al., 2014)
	safety checking	(VELNER et al., 2016)
	invariant checking	(PANDA et al., 2014) , (KUSHWAHA et al., 2022)
	network debug	Choi et al. (SHIN et al., 2015b)
	stateful middleboxes	Velner et al., (VELNER et al., 2016)
	network inconsistence	Kuzniar et al. (KUZNIAR et al., 2012)
Theorem	device synchronization	(LIU et al., 2013)
Proof	device update	(GIANNARAKIS; SILVA; WALKER, 2021) ,(JIN et al., 2014)
	network consistency	(HANDIGOL et al., 2012)
	policy instantiation	(LEE et al., 2015)
	OpenFlow formalisation	(GUHA; REITBLATT; FOSTER, 2013)
	policy configuration	(GEMBER-JACOBSON et al., 2016)
	configuration checking	(MCCLURG et al., 2015)
	optimal forwarding rules	(KOZAT; LIANG; KöKTEN, 2014)
	rule selection	(NGUYEN et al., 2014)
	performance of forwarding rules	(BIFULCO; SCHNEIDER, 2013a)
	reusable policy decisions	(VOELLMY et al., 2013)
SAT	input/output checking	. (WUNDSAM et al., 2011)
	stateful checking	(VELNER et al., 2016)
	policy configuration	(PRAKASH et al., 2015)
	static verification	(JABAL et al., 2019),(ZHANG; MALIK, 2013)
	BGP configuration	(BECKETT et al., 2016)

2.3.4 Overall Discussion

Testing and verification are similar concepts but different when we think about their main goal. Testing is the process of executing testing samples over software or system to find errors. Testing should detect errors using as baseline the implementation. Verification, on the other hand, is the process of evaluating a specification error using as baseline the requirements that it should satisfy, *i.e*, its specification. Sometimes we can use monitoring schemes to update the specification and perform better verification schemes. Fig 2.12 summarizes this relation.

Figure 2.12 – Verification and testing Relation



Source: by author

We can use several techniques as an instrument to verify or test networks. However, challenges to obtaining an optimal result in the use of these techniques still exist. In the previous section, we summarize the most frequent challenges in the following items: it is impossible to store all information of packets that traverse in a network, all network components that they interact with, and all states that they can assume. Consequently, it is essential to limit the scope of verification/testing tools. There is a trade-off between the quality of these tools and the financial resources available to expend to acquire them. Even in the case that plenty of information could be obtained to execute these tools, it is difficult to understand and filter useful insights from this.

We can summarize the challenges of related work as:

- **Set 1** - related to state explosion, network path explosion;
- **Set 2** - related to scalability;
- **Set 3** - related to incompleteness;
- **Set 4** - related to expressibility, modeling real world, complex code, and effective test cases;
- **Set 5** - related to lack of guidance, reproducing errors, effective test cases;

Table 2.2 – Related Work and Criteria

Technique	Set 1	Set 2	Set 3	Set 4	Set 5
Model Checking	Yes	Yes	Yes	Yes	No
Symbolic Simulation	Yes	Yes	Yes	Yes	No
Theorem Proving /SAT Solver	No	No	Yes	Yes	Yes

Tabel 2.2 summarizes the key points discussed. Alternatively, sometimes the best solution is to perform a network test combined with the assistance of formal verification techniques. However, all challenges in this table should be avoided. To achieve this goal we explain our research in the next chapters.

3 SOLUTION PROPOSAL

In this thesis, we investigate the problem of guaranteeing the absence of network property violations by combining the advantages of network monitoring for detecting property violations with the advantages of formal verification to model the network. A highlight related to the success of such a combination is the use of a model based on grammar to capture the communication patterns existing on the network. The use of grammar means that we do not need to collect all the information on the network to obtain a high-level view of what happens on it. We desire that we can study properties such as “Can network component x send HTTP packets?". Next, we will discuss how we achieved this goal.

3.1 Network Testing: A Discussion About Properties

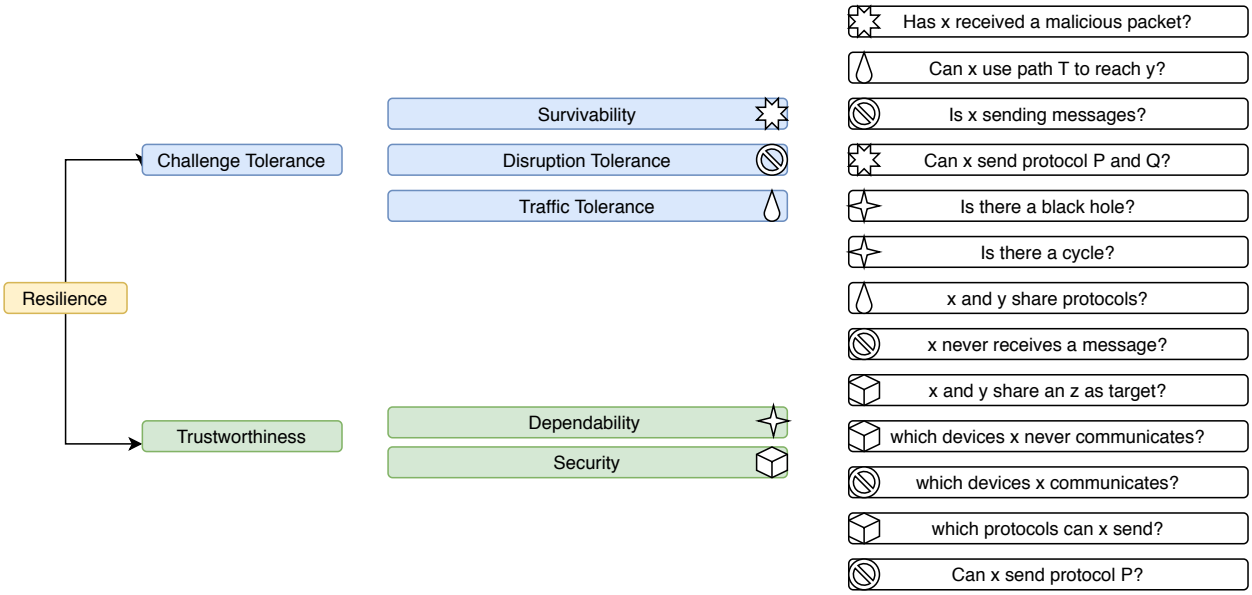
We used the resilience disciplines presented in the taxonomy proposed by Sterbenz et al. (STERBENZ et al., 2010) to group the set of individual and global properties that our solution should be able to analyze. Performance properties will not be addressed for a while because they need accurate devices to meter the throughput and execution times of network devices. The remaining properties are the focus of this work and are represented by disruption tolerance, traffic tolerance, dependability, survivability, and security properties.

We chose these properties mainly because their testing routine includes the analysis of the network communication patterns, *i.e*, the questions "which type of packets component x sent?" or "can component x send packets of type y to component z?" are important in the testing process. Considering these simple questions we can formulate the general questions presented in Fig. 3.1 that composes an example of properties that our solution should be able to test on *network components*. This taxonomy can be used as a source to study the set of properties to consider when testing networks.

It is important to emphasize that there is a subset of properties that hides challenging issues. The question of "Is there a cycle in this network?" is an NP-Complete problem in general. The question "Can x communicate with y" when x and y are two network hosts encompasses the expensive processing task of determining all packets that leave x and are sent to y. There are so many protocols and network paths that it is prohibitive to store or analyze all this information. We argue that it is only possible if we reduce this search space to a limited but consistent subset representing the problem domain. Additionally, we need to consider the internal and external aspects of *network component* and communication patterns: information that sometimes we do not have.

Thus, the scope of this solution is limited to testing what is observable by the communication pattern between the network components. We infer whether the property is respected or not using these observations. Here the pure notion of testing (submitting a system to test samples) can be confused with the notion of conformance of properties. The latter is the test notion used

Figure 3.1 – Example of Network Properties



Source: by author

in this research and tries to verify if a property is satisfied in the network and this can be done by testing a question (precisely the one that checks the property).

3.2 Top-Down and Bottom-up Verification

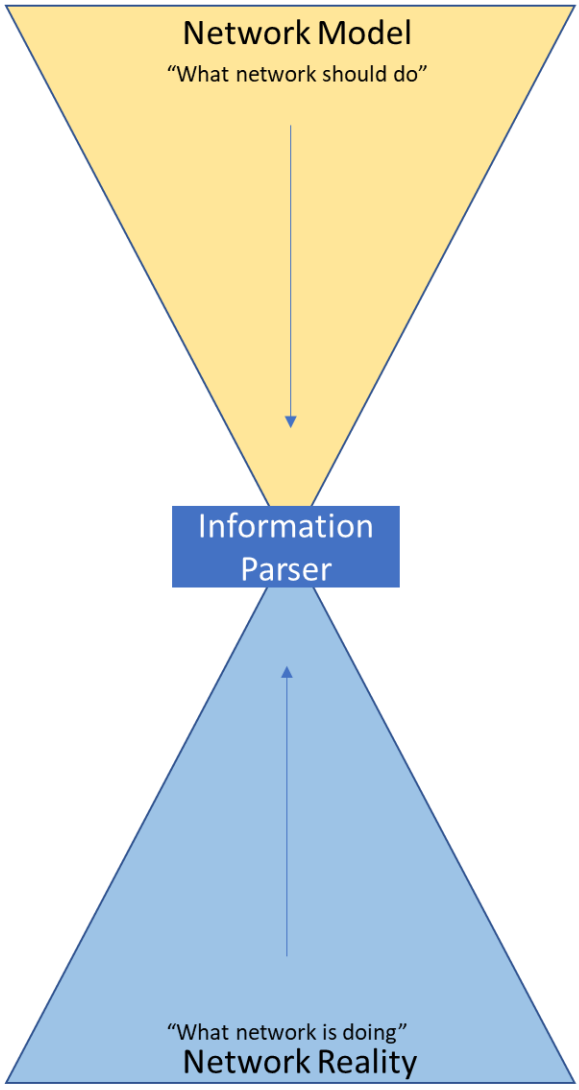
Our proposed verification schema relies on two components. Firstly, the top-down view of the network is responsible for proving and verifying all the properties that the network should respect all the time. For example, the top-down view is responsible for checking whether a network component **can** communicates with an undesirable network component. Note that here we have the concern of enforcing the network model or *"whats the network should do"*. We named this first step top-down because the top of the network is where the network administrator operates and manages the network.

The bottom-up view must be able to monitor and use each piece of information monitored to check if an undesirable behavior occurs in the network. For example, this bottom-up view is responsible to verify whether a link failure is generating a delay in network communications. Achieving this objective can be very difficult because there are several pieces of information to capture and analyze, thus we need a strategy able to store and manipulate this information to extract only the useful part of the data. Here we have as a goal the understanding of *"what the network is doing"*. We named this network view as bottom-up because it relies on monitoring and monitoring schemes that act directly with the hardware/software considering its executions. Fig 3.2 summarizes these ideas graphically.

Using these two network views we can analyze properties in a very different design. For example. the property *Some packets from host x cannot reach host y* can determine if this

communication can occur in the future, *i.e*, if the network allows this communication. The bottom-up view can check if this communication has occurred in the past by monitoring network devices. Using only one or another view alone we cannot analyze this property using this level of detail. For this reason, we advocate that our research is complete by combining these two opposite notions of verification into a single view named *Information Parser*.

Figure 3.2 – Top-Down and Bottom-up Verification



Source: by author

3.2.1 Bottom-up: Understanding Network Reality

We propose that the design of the monitoring layer should consider aspects of different network portions. There are requirements for testing *network components* properties involving (i) data that travels on the network, (ii) control logic of network devices, and (iii) configurations that they have. Thus we need at least the following conceptual steps:

- **Information Gathering Step:** Considering a view based on Software-defined networking, there are at least three central sources of network information: (i) data plane, (ii) control plane, and (iii) and management plane. Our monitoring layer must certainly include information on at least these three aspects of the network.
- **Data Classification Step:** It is necessary to keep the monitored data considering an abstraction that can reason about the data. We decided to use *snapshots* and machine learning for it. A *snapshot* represents a photo of the state of each of the data, control, and management planes. This snapshot should be used to (i) reduce the monitored data and (ii) periodically identify what is happening on the network. Additionally, it is useful to maintain a database with examples of property violations to help the identification of if something happened when a *snapshot* is analyzed. Whenever a property violation is identified, the monitoring layer must identify it by classifying its nature. We understand that similar property violations can repeat on the network. For this reason, we propose that our monitoring layer should support a classifier based on machine learning to determine if two network events A and B are similar. Machine learning classifiers can perform these classifications in a very fast and accurate design using the information that repeats in the network. The scope of machine learning here is only to classify data.
- **Diagnosis and Remediation Step:** Finally, a third step of summarizing (i) the normal behavior observed on the network and (ii) the violations found is necessary because we need to create a formal model based on the monitored data. Thus, the generated model takes into account *what is really happening on the network* and not just what can happen. Obviously, this initial model must include the possibility of modeling future possibilities, otherwise, it would only be a description of what happens and not a tool to infer properties.

3.2.2 Top-Down: Understanding Network Model

The design of the formal verification layer should consider the existence of the monitoring layer before it. Thus, the purpose of this layer is an extension to the monitoring layer with **proving capabilities**. As formal verification suffers from scalability, it is desirable that this layer could reduce the search space in the process of analyzing properties.

We understand that to achieve this, three steps are needed:

- **Information Collection Step:** a step able to obtain all information monitored by the monitoring layer including the data, control, and managing information. It is useful to construct a model that knows what is happening in the network.
- **Model Synthesis Step:** a step able to generate a scalable model to describe the network including *network components* and their characterization and properties.

- **Model Reasoning Step:** a step able to search for inconsistencies in the network to signalize possibilities of future errors before it happens. The generated model should be able to prove properties.

Instead of using pure well-known abstractions to model the network, such as model checking, theorem proving, and symbolic simulation, we propose a new abstraction based on the property evaluation process of model checking and the generalization provided by grammars. As *network components* exchange messages with each other, we advocate that *it is possible to model the language* in which a *network component* uses to communicate with another. There is an extensive set of messages that are allowed to be exchanged between two *network components*. Additionally, there is an extensive set of messages that are not allowed because they violate some property defined by the network administrator. Consequently, this generates a notion of allowed communications. The rules of these communications can be represented by grammar since it is a simple view of objects sending messages to others - like humans talking with others.

3.3 Overall Discussion

We aim to propose an enhanced solution to test if a property violation can occur in the network. Here the idea is similar to an antivirus (a monitoring tool) enhanced by a solver (a proving tool), thus proving that the monitored property violation can/cannot occur. We advocate that if we combine the two principal techniques to assist network testing: monitoring and formal verification, we could achieve a more optimized design. The following scope should be considered in the study of this research:

In the case of the monitoring layer:

- **Scope of Verification:** We cannot check all the properties in all contexts. Thus, we first focus on properties that can be verified when we observe the communication standards between the network devices. We use static verification by the use of snapshots to obtain network information;
- **Scope of Applicability:** We wish our solution to be applicable in various contexts without a high cost to be used. Thus, we chose to use only information such as data, control, and management information. This information can be freely obtained in the majority of popular network controllers, such as ONOS.
- **Scope of Machine Learning:** We know that within the machine learning context, there are several methods and databases. We just want to use Machine Learning here to provide the best network traffic classification for the type of modeled traffic. We aim to use the best-known techniques for traffic classification and we do not want to perform an optimal study about which one is the best one. Consequently, all monitored data is analyzed

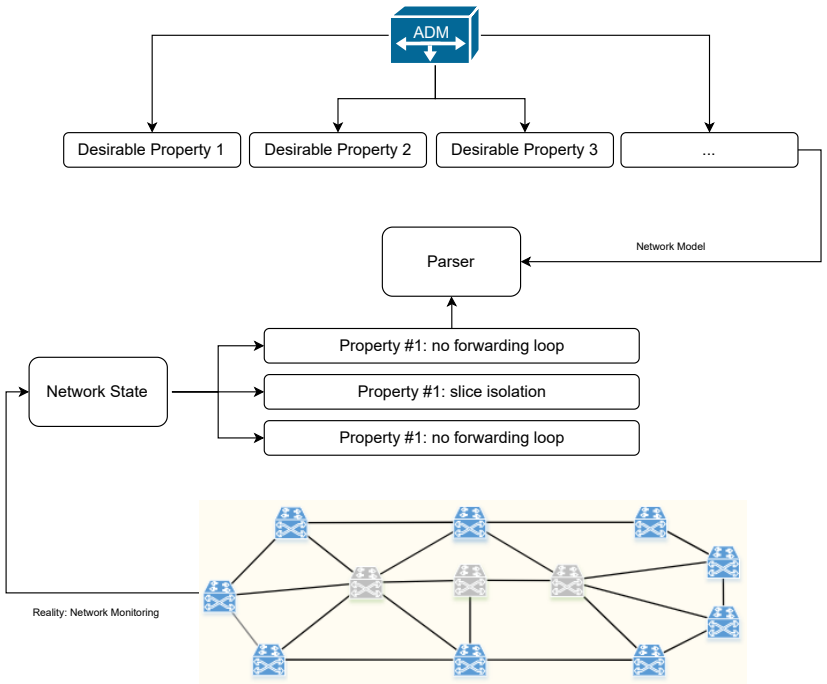
by machine learning classifiers to detect property violations and learn the set of most appropriate strategies for remediating them.

In the case of the formal verification layer:

- **Scope of Scalability :** We know that a network component can assume infinite states within its execution cycle. The generation of a grammar to represent network communications has the potential to decrease the amount of data that should be stored to perform verification since only possible communications are stored and not all packets. By using grammar, network communications can be analyzed considering all network components.
- **Scope of Proving:** We are interested here in studying the language generated by grammar produced and understanding which property violations it can prove. This grammar should be able to detect the possibility of property violation before it occurs using the capabilities of grammar to check if communication is possible.

In the rest of this thesis, we study these ideas through the evaluation of the proposed solution (Fig 3.3). In special, we instantiate our conceptual Fig 3.2 with (i) a monitoring layer using data, control, and management information to detect, diagnose, and suggest remediation for property violations. Here we use a view of the network named *Network state*; (ii) a formal verification layer relying on a monitoring layer to be constructed. We use a parser to combine these two views into a final network view that understands the set of *desirable/undesirable properties*. In the next chapters, we discuss the conceptual architecture represented by Fig 3.3 and share our conclusions.

Figure 3.3 – Top-Down and Bottom-up Verification Scheme



Source: by author

4 BOTTOM-UP DETECTION: A NETWORK MONITORING LAYER

In the first part of this thesis, we present a detailed description of the monitoring layer proposed in the previous chapter. This monitoring layer can detect, diagnose, and remediate property violations by observing the effect that they produce on monitored data from the *network components*. Next, we explain the main design decisions behind this layer and the evaluation of its research questions.

4.1 ARMOR Overview

Here we present the design of a monitoring layer to test property violations in *network components*, named ARMOR. ARMOR is composed of three conceptual steps. The first is the *Information Gathering Step*, which is able to collect information from the network. The second is the *Data Processing and Classification Step*, which is able to detect and classify network violations. Finally, the third is the *Diagnosis and Remediation Step*, which is able to learn from past violations and remediations to suggest actions when a similar violation occurs.

The *Information Gathering Step* monitors the network in three levels: (i) a portion of the messages exchanged between devices; (ii) the internal state of these devices (when available); and (iii) the network decisions over devices and links. By analyzing the messages on the network, ARMOR can extract patterns of communication and update traffic counters. The analysis of the observed information of devices enables this layer to extract a notion of the internal configuration installed on each device, an useful information to find property violations. Additionally, the analysis of network decisions enables this layer to capture property violations triggered by the interaction of network devices.

The *Data Processing and Classification Step* is where all collected data is analyzed by a set of traffic classifiers that can determine if a pattern of a property violation can be found in the data collected. New types of data are classified as unknown and need further investigation. ARMOR in this step labels all data as *normal* data, *violation* data, and *unknown* data.

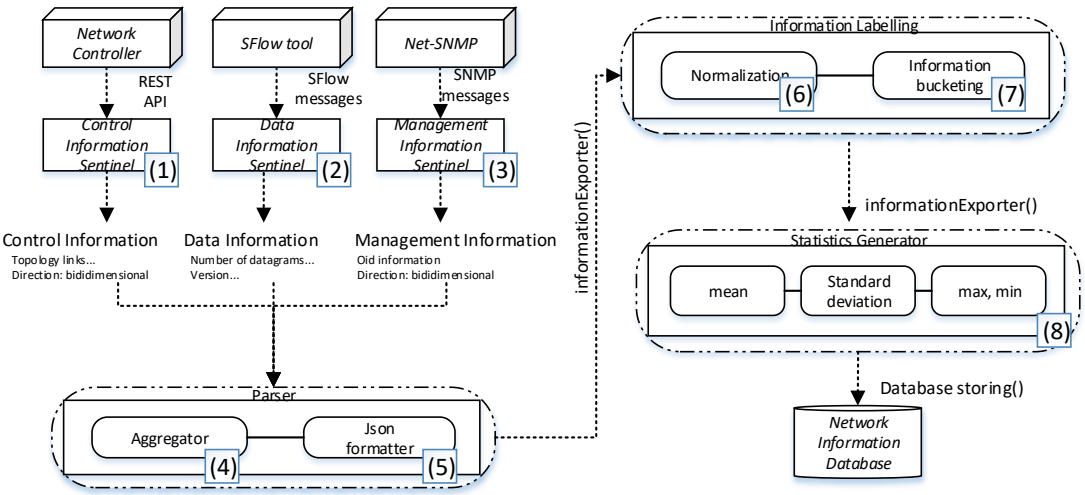
The *Diagnosis and Remediation Step* can store past violation profiles for future analyses and be able to store remediation actions to learn tips for future remediations. Next, we describe the operational details and the internal architecture of each step.

4.1.1 Information Gathering Step

The Information Gathering Step is presented in Figure 4.1. Next, we explain its details. ARMOR monitors three aspects of the network: control (REST API), data (sFlow), and management information (SNMP). Since this information is not static, components are needed to collect network data, *i.e*, components capable of periodically requesting network information to keep ARMOR up-to-date on the state of the network. We name these components as sentinels,

respectively, *control information sentinel (1)*, *data information sentinel (2)* and *management information sentinel (3)*.

Figure 4.1 – ARMOR: Information Gathering



Source: by author

Table 4.1 – Protocols to Obtain Network Information

SNMP	**REST API**	**Sflow**
trap dropped	flow packet count	packet drop events
subnet mask	byte count	packet errors
packet sent	flow duration	maximum packet size
packet discards	inter-arrival-time	TCP flags
illegal operation	IP address	bad values
next hop	flow path	packet speed

As depicted in Figure 4.1, we understand that the following components are crucial to obtain reliable information from the network.

- *Information Parsing:* Since the set of information collected by the sentinels is distinct and numerous, it is necessary to preprocess the obtained raw data. ARMOR uses this layer for the parsing of ”useful“ information and for filtering out non-essential network information (excluding for example specific MAC address of devices). A partial view of the information that can be collected is summarized in Table 4.1 and it is the result of the *aggregator (4)* and *json formatter (5)* routines.
- *Information Labelling:* The Information Labeling process is responsible for qualitatively grouping the data collected by the network sentinels. The determination of each group is configured manually, *i.e*, any new information should be classified manually by the network administrator. For example, the determination if *packet count* is a *general counter*

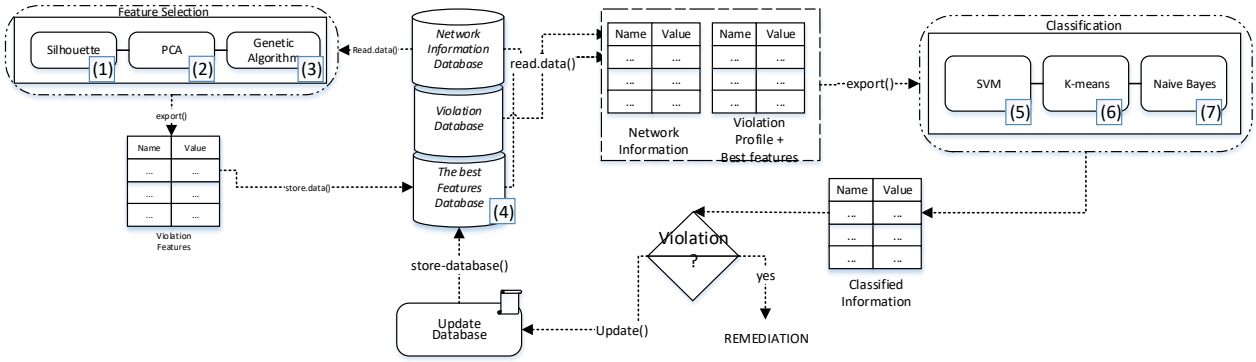
information. However, the administrator can change or insert more classes anytime (6). Since the data are from different sources it is needed to divide them into the four groups of data found in the monitoring process: *temporal*, data related to time measurement (example: inter-arrival-time between packets); *device description*, for data related to the state and operation of a device (example: IP address); *general counters*, for data related to packet transmission (example: number of transmitted packets); and *other,* that groups data that does not apply to the previous categories. This grouping process is performed by the Information bucketing routine (7) and generates a high-level description of what our monitoring layer can monitor.

- *Statistical Analysis:* Little information can be obtained with this data without applying statistical analyses over it. The statistics generator (8) is responsible for computing quantities such as *mean*, *standard deviation* and *variance* over all collected data. This information is stored in the Network Information Database.

4.1.2 Data Processing and Classification Step

The Data Processing and Classification Step is responsible for using feature selection over the data collected and for detecting each network violation and later classifying them in violation samples. Figure 4.2 shows its main components.

Figure 4.2 – ARMOR: Data Processing and Classification



Source: by author

This step needs several components to classify data into property violations or not. Next, we comment on them in detail.

- *Feature Selection:* The feature selection process occurs a priori by analyzing two data sources: a database of common network violations and the Network Information Database. Probably not all flow information is suitable to detect a given type of violation. In this case, we combine a genetic algorithm (1), principal component analyses (PCA) (2), and silhouette of k-means (3) to detect which subset of network information is the best for

detecting a certain type of violation. Firstly, the genetic algorithm generates subsets of features taking them randomly in groups of equal size, including a special group with all features. By using 0.1 of probability to generate a subset with one less feature we guarantee that new combinations are inserted in the study. The crossover process takes the five first features and combines with the five last features of another solution. The PCA and silhouette are used jointly to decide the best initial set of features. This process of feature selection is done initially in the system and used later for classification generating the Best Features Database (4).

- *Data Classification:* The classification module uses SVM (5), Naive Bayes (6), and Decision Trees (7) as traffic classifiers combined with a voting strategy (the most voted class will be chosen). These classifiers use the Network Information Database, the Best Features Database, and the Violation Database. Given the Network Information Database, these classifiers will indicate network violations using the Best Features Database and the Violation Database. For example, imagine that switch A has packet-count equals to zero in the flow associated with a given IP representing the controller. This behavior can indicate a communication fault between switch A and the controller. Several machine learning algorithms can run independently over the Network Information Database using the Best Features Database to find information similar to the Violation Database. We apply Support Vector Machine (SVM), Naive Bayes, and Decision Trees for classification because these techniques can operate jointly and presented the best classification accuracy in our tests. If no violation is detected in the network, ARMOR uses the information collected to update the Violation Database with a more accurate model of network information. After the classification, ARMOR updates the Network Information Database with the class of each entry: normal, violation, or unknown.

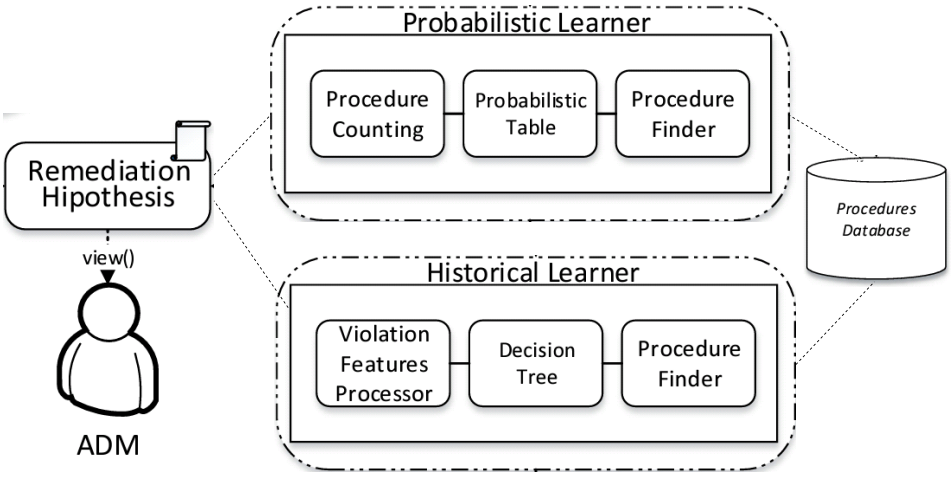
In the classification step, we chose machine learning classifiers due to their composability properties (one can easily be combined with the other) and their wide spectrum of base techniques enhancing the variability of ARMOR analysis over the collected data (**Information Reasoning**). Support Vector Machine was our first option due to its high accuracy presented in the literature (ALPAYDIN, 2004) and in our tests. Naive Bayes was our second choice due to its knowledge table, a useful data structure to infer patterns. The Decision Tree was our third choice due to its ability to create a tree representing the concepts learned. Considering several repetitions until an experimental error is under 0.1, we observed the following: (i) SVM uses less memory than Naive Bayes and Decision Tree because this algorithm does not need a large database; (ii) Decision Tree uses a high amount of memory to store a tree representing network information; (iii) Naive Bayes uses a reasonable amount of memory considering its probability table.

4.1.3 Diagnosis and Remediation Step

After a property violation is found by the *Data Processing and Classification Step*, an event is triggered to inform that at least one occurrence of a violation was found. This event identifies the set of network violations classified according to their nature. It is useful to annotate "what is happening" in order to remember past property violations found in the network and determine the probability of the future occurrence of this property violation again.

The main components that belong to this step are discussed below:

Figure 4.3 – ARMOR: Diagnosis and Remediation



Source: by author

- *Historical Learner:* If a violation is identified in the classification process, this step itself can inform the error occurrence to the network administrator and suggest a set of remediation actions automatically. This step must also be able to perform approximations if a similar error has occurred in the past and now is occurring again using the Historical Learner component. For this reason, the Historical Learner component executes a Naive Bayes algorithm to produce a score of similarities with past network errors by saving part of the information classified in the Network Information Database. To produce a detailed description of past violations found, several strategies can be used. However, we advocate that the use of decision trees can facilitate this process by storing the information discovered until now, thus facilitating the determination if a new violation is critical and never seen before. Another advantage is the capacity of data analysis offered by decision trees enabling the search for potential property violations as soon as possible. In the specific case of an unknown error, its occurrence can be signalized to the network administrator.
- *Probabilistic Learner:* Concurrently, the Probabilistic Learner component analyses the frequency that a specific remediation strategy is used by the network administrator (named as Procedures) to solve that violation in the past. For this task, this component uses the Naive Bayes algorithm again, but now with the purpose of remembering the frequency

of use of remediation strategies. In the end, a precise profile of network violation is produced including its historical profile and the Procedures that should be performed. Finally, this component updates ARMOR to reflect the current network state. For example, if the violation encountered could break some fundamental network configurations, such as erroneous forwarding paths, ARMOR executes the remediation actions determined by this layer. If the actions do not eliminate the network violation, the network administrator should be informed.

4.2 Case Study: A Data Center Network

The performance analysis of ARMOR was divided into four aspects:

- **Vast Amount of Data**: does the use of snapshots reduce the amount of data that needs to be stored and does the use of this snapshot maintain the quality of property violation detection?
- **Financial Limitations**: is it possible to extract network information without the use of third-party expensive products?
- **Information Reasoning**: are Machine learning algorithms a suitable alternative to detect property violations and learn the set of most appropriate strategies for remediating them?
- **Configuration Checking**: are Machine learning algorithms a suitable alternative to detect property violations and learn the set of most appropriate strategies for remediating them?

The financial limitation is an aspect already evaluated by definition because all the description of ARMOR shows how to obtain network information without the use of third-party expensive products. Thus, we only need to evaluate the other aspects remaining. Next, we comment on the simulation profile used to study them.

4.2.1 Using Monitoring for Detecting Property Violation

We highlight that it is not possible to theoretically study every question related to a monitoring layer to detect property violations without considering a concrete implementation. For example, we cannot define a priori the classification accuracy of a machine learning algorithm without using it in a dataset. For this reason, we need to discuss some key points.

4.2.1.1 On collecting Reliable network data

Several research studies are involved in the task of collecting network information. It is desirable that it is as close to the real as possible, without errors, and updated. A flow sampling scheme that samples network packets at regular intervals could provide information such

as source and destination IP addresses, source and destination ports, protocols used, packet lifetime (TTL), and the number of bytes and packets transferred. Strategies such as polling are often used to obtain these samples, however, it is a challenge to determine when the information collected represents the reality of the network and not just a seasonal behavior. Since a *polling time interval* equal to zero is a real-time system we need to define a number because we do not aim to propose a real-time system that stores every packet information (UJJAN et al., 2020). Consider the *inter-arrival-time* of packets in a network component, *i.e*, after receiving a packet from a network component *x* how long it takes to receive another? If the *polling time interval* corresponds exactly to the *inter-arrival time* thus we will take only 1 packet. To obtain a sample size we need to study a number between [1 * inter-arrival time, x * inter-arrival time]. Where *x* is the amount of time in seconds to wait until query information from the network again.

Using the *inter-arrival-time* we calculated the time interval to pooling network information and capture a higher number of flows. We obtained a polling time of around [2.5, 5] seconds to reach the peak of collecting all flow existing in the network. An example of collected data is shown in Fig 4.2.

Table 4.2 – ARMOR Traffic Raw Information

Flow	**Switch**
priority	0
duration nanoseconds	484000000
hard timeout	0
idle timeout	5
actions	drop or send to controller or ...
duration seconds	4
byte count	196
table id	0
packet count	0
cookie	900071..
match	src ip, dst ip,...
...	...

In total, 30 features were collected considering the concept of flow, a 5-tuple using the following format:*src_ip, dst_ip, src_port, dst_port, protocol*. Fig 4.4 shows the format in which the raw information is collected from the network.[1]. The information stored in switches about packets passing through the switch includes duration in nanoseconds (the duration time of this entry in nanoseconds), byte count (the number of bytes corresponding to this entry), packet count (the number of packets matching this entry) and others. The quality of these data depends on how well they capture the relevant patterns and structures in the data for the specific classification task. The collected data should be discriminative, meaning that they can distinguish between different flows with high accuracy, and they should also be robust, meaning that they can generalize to new data that was not seen during the first observation. Evaluating the quality of data can be done through techniques such as cross-validation, where a model is trained and tested on multiple subsets of the data, where the contribution of different data to the classification accuracy is measured.

[1] remember data the data is scaled so its value is centered in mean equal to 0 em standard deviation of 1

Table 4.3 – ARMOR Traffic Features

Feature	Description
0 - packet count	how many packets were sent
1 - inter-arrival-time	time between two packets
2 - byte count	how many bytes were sent
4 - source device	source of communication
5 - destination device	destination of the communication
6 - destination count	total of different destinations
7 - payload length	length of payload
8 - source count	number of communication sources
9 - internal	if interacts with hosts
10 - anomaly	if presented anomaly in the past
11 - frequency	frequency that appears in snapshots
12 - max protocol	max number of protocols used
13 - max destination	max number of different destinations
14 - min protocol	destination of the communication
15 - min destination	max number of different destinations
16 - packet length	sum of packet lengths
17 - mean packet length	mean of packet lengths
18 - std packet length	standard deviation of packet lengths
19 - diff packet length	max of packet lengths - min of packet lengths
20 - max inter-arrival-time	max value of inter-arrival-time
21 - min inter-arrival-time	min value of inter-arrival-time
22 - mean inter-arrival-time	mean value of inter-arrival-time
23 - std inter-arrival-time	std value of inter-arrival-time
24 - diff inter-arrival-time	max - min of value of inter-arrival-time
25 - min dst count	minimum number of different destinations
26 - max dst count	maximum number of different destinations
27 - num diff anomaly	number of different anomalies observed
28 - max packet length	min of packet lengths
29 - min packet length	max of packet lengths
30 - flow duration	duration of communication

4.2.1.2 Understanding Network Data

The next step of transforming raw information into information that summarizes the network communication pattern. We used transformations such as mean, and standard deviation to describe the collected data. These transformations were used to create traffic features and help to better understand the characteristics of the data set, such as its centrality, dispersion, and presence of extreme values. Fig 4.5 exemplifies how these transformations separate data.

The main goal here is to find a data signature (HAMOLIA et al., 2020). A data signature is a unique pattern or characteristic of the collected data that can be used to identify and detect property violations. The first step is to establish what is considered normal behavior for the dataset. This can be done through statistical analysis or machine learning algorithms that learn from historical data. There is a need to search for data signatures that represent a property violation and classify it. The following modeling schema was considered:

- **Collection 1:** Collection of traffic considered normal during a day representing examples such as streaming video and web pages.
- **Collection 2:** Insertion of a property violation into the network and collection of the network communication pattern.
- **Collection 3:** Collection of normal traffic and traffic representing a property infringement on the network. The idea here is to note what the network communication pattern looks like when we have both types of traffic on the network.

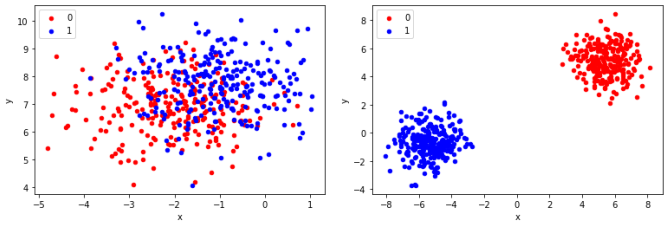
Figure 4.4 – Data Collected by ARMOR

	0	1	2	3	4	5	6	7	anomaly
0	1.848537	-3.894399	-7.530395	1.212674	-6.481316	0.795049	-9.097769	0.614420	0
1	3.489179	-2.077115	-8.141778	1.779347	-7.939098	0.195662	-12.106933	0.542379	0
5	2.374547	-2.898021	-10.760561	1.166085	-8.250874	-0.597140	-9.671687	0.485904	0
9	-0.228091	-2.895965	-9.362182	1.374464	-7.564464	1.043802	-11.175796	0.932673	0
10	2.872205	-2.165822	-7.894395	2.029273	-7.668138	2.211445	-10.265465	2.021194	0
...	...	...	...	...	...	...	...	...	...
481	-2.825494	2.731091	8.423813	2.360656	7.169661	-7.791545	-2.106101	-1.978942	1
307	-6.804185	-0.795451	0.129827	-6.985034	NaN	NaN	NaN	NaN	1
465	-1.695460	6.267030	-9.832392	6.123063	NaN	NaN	NaN	NaN	1
484	-8.058864	-1.610187	1.147352	-5.298122	NaN	NaN	NaN	NaN	1
428	-2.858091	2.293286	8.018991	4.532515	7.561789	-10.180920	2.208923	-4.999700	1

Source: by author

Once the normal behavior is established, you can create a data signature that represents the expected pattern of the dataset. We need to compare the actual collected data with the signature. If the actual collected data deviates significantly from the signature, a property violation has likely occurred. The degree of deviation needed to trigger an alert or action depends on the specific use case and the level of risk associated with property violation.

Figure 4.5 – ARMOR Separation between Normal and Violation



Source: by author

Note that here we are interested in collecting data to be our reference, that is, our training data. Concurrently we want data to check later if we can classify new data.

4.2.2 Data Processing and Classification

We used machine learning techniques to classify data because (i) we can use examples of well-known property violations to search their pattern in network-collected data, and (ii) we can use these algorithms to find similar but not equal property violations. To accomplish this, we need to study which are the best algorithms to classify data. It is an important choice dependable on the nature of the problem to solve. Here we comment on the possibilities:

- **Supervised Learning (HOSSEINI; AZIZI, 2019):** type of machine learning where an algorithm is trained using labeled data (examples of signatures). It is suitable for classifying data samples into a range of known attacks because these techniques use a data model that describes the known classes to classify data using a similarity measure. Example: SVM, KNN.
- **Unsupervised Learning (BOWMAN et al., 2020):** these techniques do not use historical information or a data model to produce the data clustering, but only the similarities observed in the data. Clustering methods are typical examples of unsupervised learning and are organized by the modeling approaches aiming to group data, including sometimes centroid-based and hierarchal methods. Example: K-means, KNN.

We typically need to take care of three steps. The first one is to split our data into two subsets: a training set and a test set. The training set is used to train the machine learning model, while the test set is used to evaluate its performance on unseen data. The second step is to evaluate these sets using metrics, such as precision and recall. Precision and recall are important metrics to understand machine learning classifiers (DINA; MANIVANNAN, 2021). Formally they are defined as:

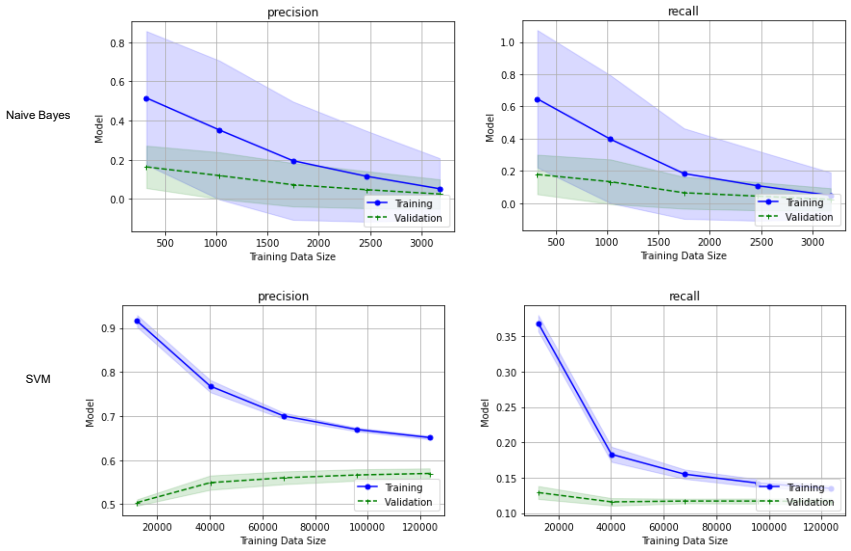
$$Precision = \frac{TP}{TP+FP}$$
$$Recall = \frac{TP}{TP+FN}$$

Where *TP* represents the true positive, that is cases in which it classifies as an anomaly what is an anomaly. *FP* is the case where the classifier points out as an anomaly what is not an anomaly and finally, *FN* is what the classifier says is not an anomaly but unfortunately is an anomaly. These metrics measure different aspects of model performance. High precision is important in situations where FP is costly, such as a malicious attack, where a false positive could lead to unnecessary and potentially harmful treatment. On the other hand, high recall is important in situations where FN is costly, such as in property violation, where a false negative could result in a change of equipment when it is not needed.

We monitor *precision* and *recall* with an increasing number of training samples. The learning curve (see Fig 4.6) is created by plotting the training and test samples performance of the model as a function of the number of training samples. As we increase the number of training samples, we expect both the training and validation accuracy to increase initially, but eventually, the validation accuracy may start to plateau or even decrease, while the training accuracy continues to increase. This indicates that the model is overfitting and needs regularization to improve its generalization performance.

The third step is to describe the nature of a property violation selecting the best features (KIRA; RENDELL, 1992). In the context of machine learning, the problem of discovering the optimal set of features to describe input data is named feature selection and often represents a challenge because (i) there is a lack of a priori information about the relevance of features collected to

Figure 4.6 – ARMOR Learning Curves



Source: by author

describe some data, thus leading to inaccurate classifications; (ii) the excessive number of collected features can lead to classification with a high computational cost; (iii) there is a lack of information on the combination between different data features, thus encouraging the study of their joint influence.

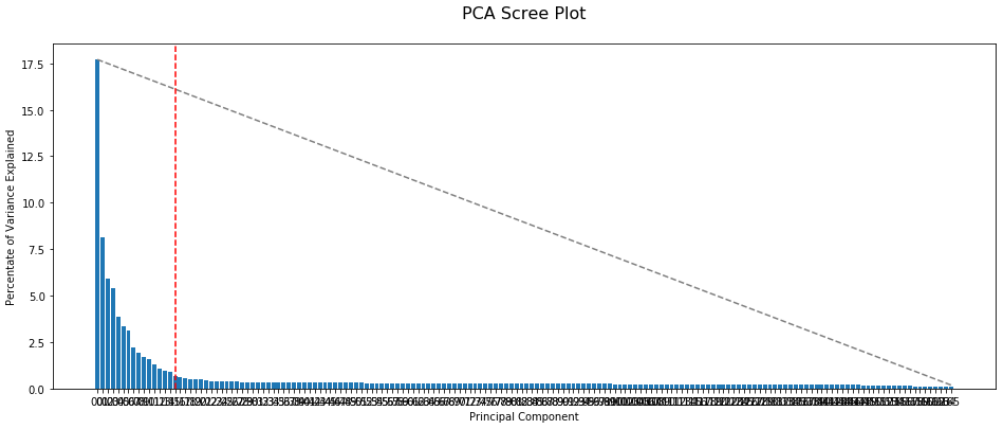
The techniques for feature selection can be broken down into the following classes (ALTASHI et al., 2020):

- **Filter Methods:** use variable ranking techniques such as Principal Component Analysis (PCA) to determine a feature ordering. In this way, these methods can determine the set of less relevant features that can be ignored in the feature selection process.
- **Wrapper Methods:** use a predictor, such as a machine learning classifier, to evaluate each subset of features generated. The task of evaluating all possible subsets of features is impracticable, thus this type of method, in general, generates a suboptimal feature set. Examples are Sequential Selection algorithms and Heuristic Search methods.
- **Embedded Methods:** aim to reduce the computation time used in wrapper methods. The idea behind this is to incorporate feature selection in the training process of Wrapper methods. The strategies for this can use greedy algorithms or the assignment of weights for the classifier and its subset of features.

We choose Filter methods with PCA because they could eliminate redundant data since we decide to collect all information available in the network. The basic idea of PCA is to find a new set of variables, called principal components, that are linear combinations of the original variables and explain most of the variability in the data. The first principal component is the direction in which the data varies the most. The second principal component is the direction that explains the most variability in the data, given that it is orthogonal to the first principal

component. And so on for the third, fourth, and so on principal components. Fig. 4.7 shows a histogram of feature importance using PCA.

Figure 4.7 – ARMOR PCA analysis



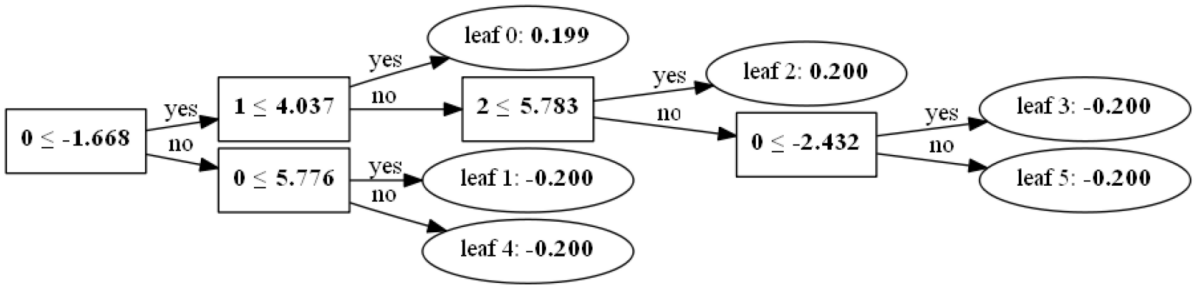
Source: by author

4.2.3 Diagnosis and Remediation

Two basic execution paths are defined to remediate property violations:

- **New Property Violation:** in this case we need to save the profile - features that were used to detect it - in a database (the generated tree). We need to use a Decision Tree to search past violations to determine if this new one is similar to an old one and in this case suggest remediation similarly. Decision Trees typically achieve high classification accuracy (NGUYEN; ARMITAGE, 2008) and thus are commonly chosen to compose anomaly detection systems.
- **Repeated Property Violation:** we need to search which of the saved property violations is the most similar to this one, thus we need to define a probabilistic learner based on Naive Bayes able to determine a score of similarity between property violation samples.

Figure 4.8 – ARMOR property violation tree

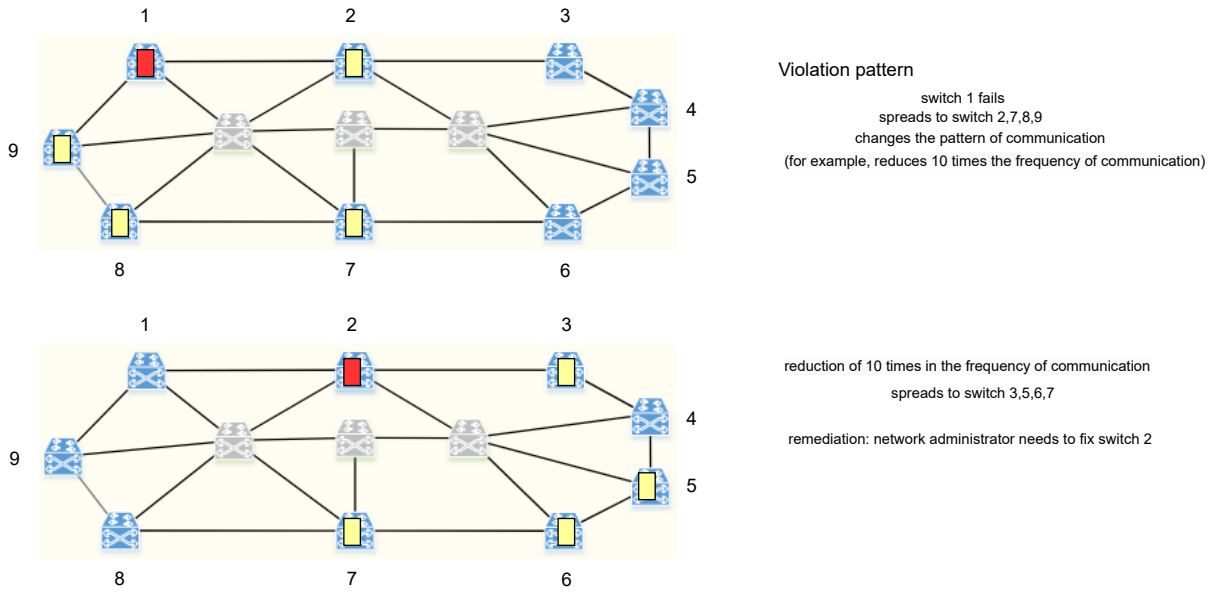


Source: by author

Note that here we use the tree generated in the classification step to understand a process property violation (see Fig 4.8). For example: if feature 0 <= - 1.6 then we can check feature 1 if its value is <= 4.03 until all features are checked. At the end of this process, the path traversed by the sample in this tree evaluation will determine if we have a property violation or not.

Thus, according to Fig. 4.9. A network anomaly being monitored can by default have the following signature: switch 1 fails, and this spreads to switch 2,7,8,9. This will certainly change the communication pattern of packets on the network, as a path will be compromised, which could, for example, modify the communication frequency. If by chance this same communication pattern occurs in future monitoring: communication frequency reduction and pattern change in other switches in the network in a very similar way to what was seen previously, then ARMOR will already know that it is probably a switch failure and will warn the network administrator. This will answer what he did to fix it, such as changing the switch. In the future ARMOR manages to save this and shows it as a suggestion. Metrics such as *precision* and *recall* take into account false positives, that is, how many times ARMOR reports that it has a property violation and it is not. A false alarm. Thus, by keeping these metrics we can be sure if everything is going well or not.

Figure 4.9 – ARMOR Remediation Process



Source: by author

4.2.4 Simulation Profile

ARMOR was implemented using the ONOS controller[2]. The network itself was emulated in the Mininet emulator. The traffic profile used follows research studies, such as (ISOLANI

[2]https://opennetworking.org/onos/

et al., 2015). Table 5.3 summarizes the traffic profile simulated. Table 5.2 summarizes the execution environment used in the experiments. This traffic profile simulates statistically the behavior of common web users who watch video streaming and make requests for web pages with an average size of 4kB. For each HTTP request, there is a request for a video stream that can last for a period between 3 and 5 seconds. The experiments were performed 35 times each lasting around 10 to 40 minutes until the experimental error was less than 0,01 with a confidence interval of 95%. We used a topology of a campus network[3]. It consists of a partial mesh topology comprising 100 hosts and 11 switches. We chose this scenario because of recent vulnerabilities occurrences due to configuration errors in similar environments. Experiments were run using actual topologies/traffic in Mininet (information was obtained via REST) in a similar manner to what was done in (ISOLANI et al., 2015; SILVA et al., 2016).

Table 4.4 – Background traffic profile used in the experiments

Parameter	**Value**
Number of hosts	200
Number of switches	12
Number of VNF servers	1 (Firewall)
Protocols	HTTP, TCP
Host behavior	Randomly distributed
Violations	Fake Ip and Conflicting forwarding rules

Table 4.5 – Execution environment used in the experiments

Parameter	**Value**
Operational System	Microsoft Windows 10
Processor	Intel(R) Core i-5-825OU, 1.60Ghz,1800
Number of Cores	4 cores, 8 processors
RAM memory	8 GB
Virtual Memory	10GB

In this evaluation process, we chose two types of property violations to exemplify the functioning of our layer. The first one corresponds to a fault represented by the interaction of several components inserting different entries in a flow-table, thus generating conflicting forwarding rules. Consequently, we are evaluating, in this case, the occurrence of *shadowing rules* (SANGER; LUCKIE; NELSON, 2020), a rule that conflict with others matching the same packet space but with different action and higher priority. The second property violation is represented by malicious interactions that change a flow-table rule to modify the network controller IP to a fake IP.

4.2.5 Effectiveness of Property Violation Remediation

Firstly, we evaluated the **configuration checking** capacity of ARMOR to detect and remediate network violations. The experiment is as follows:

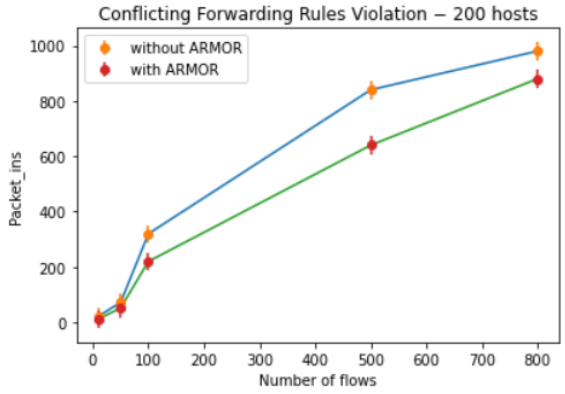
- in one case, there is a conflict on forwarding rules. On the other, there is not

[3]https://www.cisco.com/c/en/us/td/docs/solutions/Enterprise/Campus/HA_campus_DG/hacampusdg.html

- we will trigger two times the same simulation transmitting files from a host A to host B
- in the case with conflicting rules, the number of packet_ins generated is *x* because the conflict will trigger many messages to the network controller
- the use of ARMOR in the same scenario should reduce these *x* packet_ins generated

Figure 4.10 represents the number of packet_ins produced. In the case of conflicting forwarding rules without using ARMOR, the number of packet_ins generated is higher than the case of when ARMOR is executing and protecting the network.

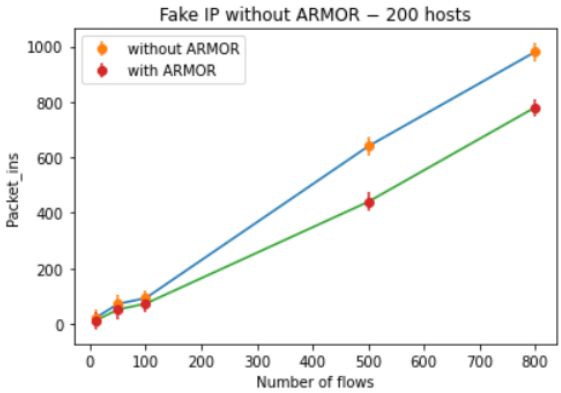
Figure 4.10 – ARMOR results with Policy Violation



Source: by author

The case of fake/wrong IP is simple to check if every flow that arrives generates a packet_in to the controller and not to another network device. By monitoring the number of packet_ins sent to the network controller, ARMOR can detect this type of violation too. Figure 4.11 shows that ARMOR keeps the number of packet_ins consistent with the number of flows after discovering the violation on the network.

Figure 4.11 – ARMOR Results with Fake IP Network Violation



Source: by author

After all this, the Historical Learner saves the violation profile, and the Probability Learner searches which remediation was used historically to solve that violation. Two courses will be

Table 4.6 – Network Violations and best remediation schemes

Network Violation	**Remediation**	
Fake IP controller	80% was used	Procedure 1: Reset_rule
Conflicting forwarding rules	60% was used	Procedure 2: Call ADM

triggered:

- **Remediation Analysis:** A Naive Bayes and Decision Tree algorithm will be triggered to search from past violations to determine if this new one is similar to an old one and in this case suggest remediation similar. The input of this classifier is (i) the property violation name, in this case, *Fake ip controller* and *Conflicting forwarding rules*; and (ii) the set of features collected to describe this property violation. The output is the name of remediation frequently used.
- **Remediation Annotation:** the results of the Remediation Analysis will be stored for future use.

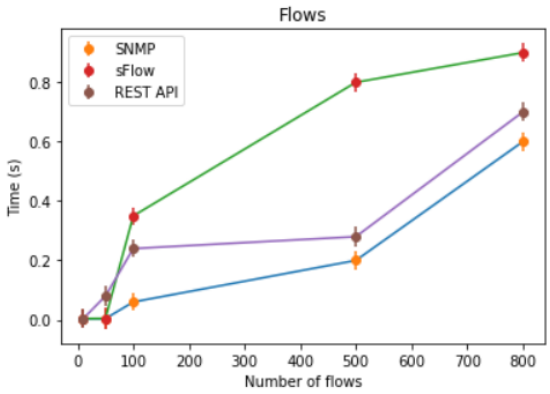
Table 4.6 summarizes the frequency that each remediation was suggested. The *Reset_rule* remediation will suggest the reset of table rule with a property violation associated. The remediation *Call ADM* will trigger the network administrator to think and update the model with new remediation for future use.

4.2.6 Property Violation Detection & Classification

The **vast amount of data** research question can be analyzed here concluding that ARMOR can obtain enough network information to understand what is happening without being prohibitive. The execution time of the sFlow, SNMP, and REST API sentinels are analyzed in Figure 4.12. The polling time used is around 2 seconds, but it depends on the polling frequency configured by the network administrator. Notice that the sFlow protocol is responsible for the majority of the time used in this layer. A reason for this is the need of preprocessing the raw information obtained for network packets. As SNMP and REST API brings organized information, their execution time is reduced in this step.

For the SNMP protocol, the processing time is quite low and almost constant for different numbers of flows, varying only slightly between 0.004 and 0.6 seconds. For the sFlow protocol, processing time increases fastly as the number of flows increases, with values ranging from 0.004 to 0.9 seconds. For the REST API protocol, processing time also increases with the number of flows, but less sharply than sFlow, with values ranging from 0.004 to 0.7 seconds.

Figure 4.12 – Runtime of information collectors



Source: by author

4.2.7 Accuracy of Property Violation Classification

For the feature selection process, we used a genetic algorithm with Principal Component Analysis and Silhouette from K-means to find the best subset of features to represent network traffic (SILVA et al., 2015b). The Genetic Algorithm (GA) was set with a population size of 500 individuals randomly generated and the crossover percentage to 15%. The mutation probability did not have an impact on classification accuracy, and we use 0.01 as the standard mutation probability. Because of the amount of time taken to execute this algorithm (nearly 90 minutes) we also applied PCA. PCA determines the most important features by creating one principal component to match each variable (feature), i.e., in our experiments, it creates 7 principal components.

Table 4.7 – Minimal Features to Classify Fake IP and Conflicting Forwarding Rules

Feature	**Description**
0 - packet count	how many packets were sent
1 - inter-arrival-time	time between two packets
2 - byte count	how many bytes were sent
3 - source count	number of communication sources
4 - min destination	max number of different destinations
5 - internal	if interacts with hosts
6 - frequency	frequenct that appears in snapshots

The classification process was performed using the violation database obtained a priori by ARMOR. According to Table 4.8, the Naive Bayes algorithm shows the worst result in accuracy. This happens because its execution takes in consideration data by similarity and it is difficult to find exactly the pattern of violation. SVM and Decision Tree show optimal classification result but Decision tree is faster because it has an eficient tree data structure and entropy based criteria of insertion. All algorithms were trained using cross validation to validate our results. The database was divided in k=4 groups of equal size and the all combinations were tested

considering that the k=1 group was the validation group and the others the training group. Table 4.7 summarizes the minimal number of features to classify these examples.

Table 4.8 – Classification performance

Algorithm	Recall	Precision	Training Time (s)
Naive Bayes	60.3%	56%	0.03
SVM	99.3%	98.7%	7.03
Decision Tree	99.1%	98.2%	0.05

4.2.8 Additional Examples

Here we present two more examples of property violation. The first one introduces how ARMOR can discover a device that has stopped working on the network. This example is important because when a device stops working it can have a communication pattern changed in a way similar to conflicting forwarding rules (previous example). At every moment, the information of packets that are exchanged over the network is exchanged and this information is saved by ARMOR. ARMOR creates a tree like the one shown in Figure 4.13. Every time this communication standard is not maintained, an alert is generated and ARMOR will check if its communication tree that represents a device failure is similar to the communication tree observed at that moment. If yes, an alert is triggered and the last action taken when this occurred in the past is suggested to the network administrator. Each internal node represents a "test" on an attribute, each branch represents the result of the test, and each leaf node represents the classification obtained that represents a decision after considering the evaluation of the attributes. The paths from the root to the leaf represent classification rules. In decision analysis, a decision tree can be alternatives are calculated (see Fig 4.13)

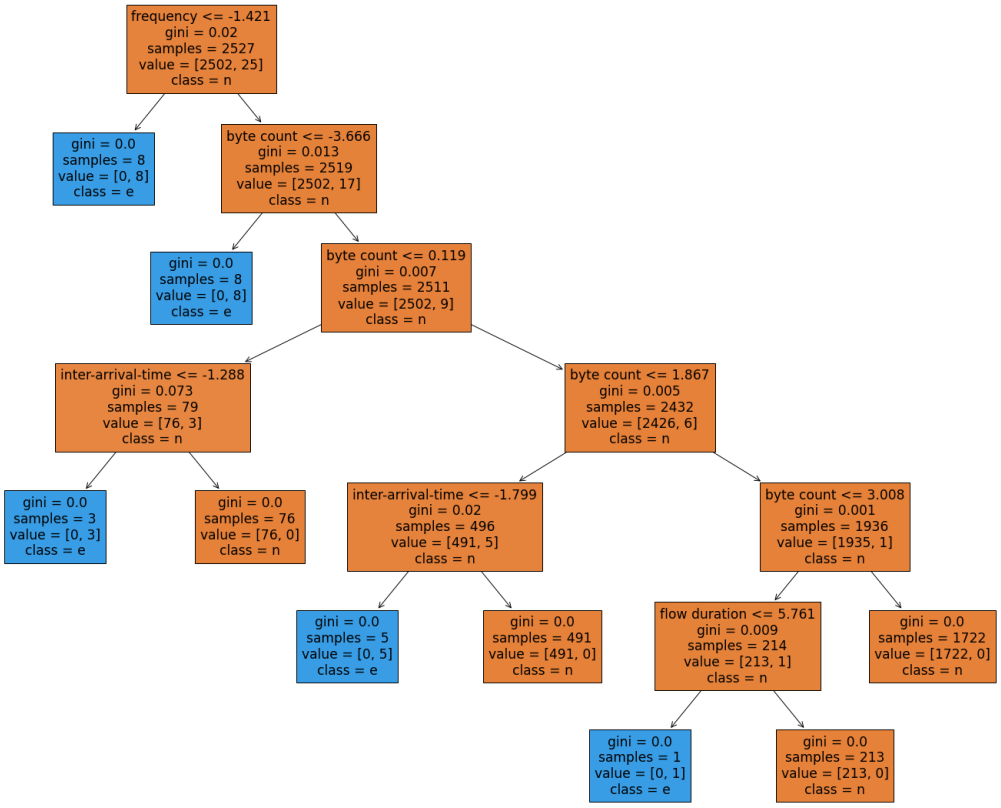
The set of features used in case a device stops working differs from the set presented for the example of conflicting forwarding rules. We note here that the features used are not the same.

Table 4.9 – ARMOR Traffic features

Feature	**Description**
0 - packet count	how many packets were sent
1 - inter-arrival-time	time between two packets
2 - byte count	how many bytes were sent
8 - source count	number of communication sources
10 - anomaly	if presented anomaly in the past
11 - frequency	frequency that appears in snapshots
12 - max protocol	max number of protocols used
20 - max inter-arrival-time	max value of inter-arrival-time
28 - max packet length	min of packet lengths
30 - flow duration	duration of communication

In the second case (Fig 4.14) we want to understand what types of communication the devices communicate with each other. ARMOR allows the use of the K-means algorithm to group the traffic samples and describe their shape using the original features. Each of the

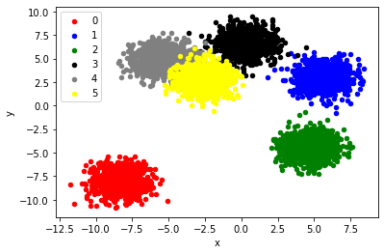
Figure 4.13 – ARMOR classification tree example



Source: by author

clusters is represented by a type of traffic and this allows understanding and differentiation. We illustrate only the 10 most meaningful features (in order of importance) selected through PCA in each scenario, depicted in Figure 5.15.

Figure 4.14 – ARMOR traffic patterns



🔴: byte count <= 7.8511903285980225 and source count <= 1.1883927583694458 and source count > -1.8968608975410461 and anoamly <= 4.676044225692749 and anoamly > 2.102351665496826 and frequency <= 9.454567909240723 and max protocol <= -1.8509194254875183 and max protocol > -4.853481769561768 and max inter-arrival-time > 7.912356853485107 and max packet length > -5.676023006439209
Precision: 0.71 Recall : 0.82

🟡: source count > 1.2458760738372803 and max inter-arrival-time > 10.2496657371521 and flow duration <= -3.550855875015259
Precision: 1.00 Recall : 1.00

...

Source: by author

As a result, we have seven different subsets of traffic. Figure 4.14 shows the classification result for each type of traffic. For example, the traffic related to simple HTTP requests (yellow) can be described by the following rule: *byte count <= 7.8511903285980225 and source count <= 1.1883927583694458 and source count > -1.8968608975410461* . We can draw several conclusions observing this: (i) not all subsets lead to the same description compared to the others; (ii) a precision of 0.71 seems to lead to the best solutions to avoid false positives (classify this traffic as HTTP in the case that it is not); and (iii) a recall of 0.82 indicates that the classification does not miss the opportunity the detect HTTP.

Table 4.10 – ARMOR Traffic features

Feature	**Decision Rule**
0 - packet count	how many packets were sent
1 - inter-arrival-time	time between two packets
2 - byte count	how many bytes were sent
8 - source count	number of communication sources
10 - anomaly	if presented anomaly in the past
11 - frequency	frequency that appears in snapshots
12 - max protocol	max number of protocols used
15 - min destination	max number of different destinations
19 - diff packet length	max of packet lengths - min of packet
20 - max inter-arrival-time	max value of inter-arrival-time
22 - mean inter-arrival-time	mean value of inter-arrival-time
23 - std inter-arrival-time	std value of inter-arrival-time
24 - diff inter-arrival-time	max - min of value of inter-arrival
28 - max packet length	min of packet lengths
30 - flow duration	duration of communication

4.2.9 Network Monitoring Limitations

ARMOR was implemented using tools such as SNMP, sFlow, and REST API. The detection of network violations was achieved with the use of Decision Trees, SVM, and Naive Bayes algorithms. The remediation process was performed with decision trees that analyze network violations and suggest remediation strategies based on historical actions. Our results show that we proposed a comprehensive approach to detect and remediate network violations.

Note that these traffic profiles and their explanation features do not allow for a difference between, for example, the absence of traffic and blocked traffic if this does not change any of their features. If in the first case, the absence of traffic produces the same signature as blocking traffic, then ARMOR will not be able to distinguish between these two types of anomalies. Here we hope that the behavior observed throughout the day will allow the differentiation of these two types of anomaly.

We make it clear that the generated features are only produced with *header* and *counters* so encryption or NAT, for example, does not interfere with the proposed solution.

However, there are crucial limitations on the monitoring layer that should be addressed in a more complex design:

- **Anticipation:** it is not possible to anticipate property violations before they occur. The monitoring schemes for property violation detection perform the observation of the symptoms of a modification in the network being incapable of analyzing data to define a future occurrence of this. To do this task we need a strategy able to prove the absence of this property in any scenario, a different task from monitoring.
- **Inspection:** there is not a human-friendly way to describe property violations to be checked. This description is needed because sometimes the understanding of "what is a property violation" is a human-centric concept, such as the "absence of a cycle in the network".
- **Proofing:** it is not possible to prove the absence of property violation. A desirable feature is to define the set of property violations that cannot occur in a given network.

Despite all benefits achieved with ARMOR, we need to encapsulate this layer with a new one able to alleviate these limitations. The next chapter will present Networds, a complementary layer focused on these limitations.

5 TOP-DOWN DETECTION: A FORMAL VERIFICATION LAYER

In the previous chapter, we presented ARMOR, a framework able to use the programmability offered by SDN/OpenFlow to monitor network aspects such as topology, communication patterns, and traffic statistics. ARMOR can detect, diagnose, and remediate network property violations. However, it cannot determine if the network is prone to a particular property violation before it occurs.

We argue that network administrators can benefit from higher-level abstractions over a monitoring scheme. In particular, we advocate that the use of formal verification using grammars for modeling network communication patterns can enhance the expressiveness of the resulting network model. The prototype of the formal verification step evaluated in this chapter is named Networds, a testing engine able to model network events and reason about the possibility of future violations. Networds enhance ARMOR with the following concrete aspects: (i) the possibility of testing complex properties using a language based on first-order logic; and (ii) a model based on grammars to check global and internal network properties in a scalable way before property violations occurs.

Consequently, in this chapter we are concerned to study and determine the answer to the following research questions:

- **State Space Explosion**: Can the use of grammars to decrease the number of states to model network communication patterns?
- **Modelling Choices**: How good is the use of grammars to model the entire network and search for property violations?
- **Mitigation Strategies**: Can grammars detect the possibility of property violation before it occurs?

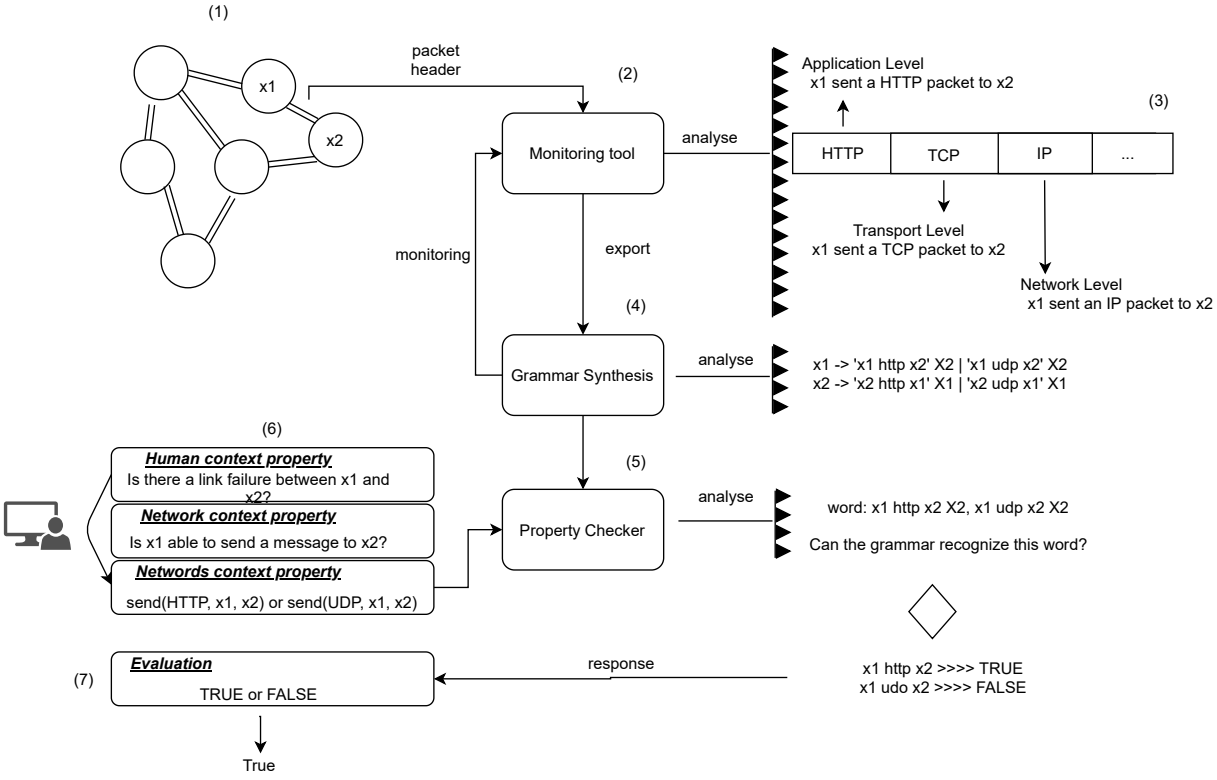
Next, we comment in detail on this extension to ARMOR.

5.1 NetWords Overview

Consider that a network administrator suspects that there is a link failure between a network component $x1$ and another $x2$. A natural question s/he can ask is *"Can component $x1$ send TCP messages to component $x2$?"* and, additionally, *"Is there a property violation?"*. NetWords is a system that uses grammar as a network model to test questions in the network. Its main purpose is to offer the possibility of testing global (related to all components) and individual (component-specific) properties in a scalable way. We argue the following three hypotheses as to the main principles of NetWords: (i) The external actions of network components partially depend on their internal configurations; (ii) An internal property violation always will reflect on some unusual external action; (iii) By observing what a component is doing it is possible to

determine its most frequent actions. Next, we explain in detail this idea using Figure 5.1 as an example.

Figure 5.1 – Detailed workflow of NetWords considering its execution possibilities



Source: by author

A more effective way of testing properties on network components would be one that combines monitoring and formal verification advantages and try to avoid their limitations. NetWords combines them sequentially into two layers where one can improve the result of the other. A monitoring layer can check property violations in network components considering an almost real-time perspective. Complementary, a formal verification layer can check properties that are likely to occur – but that have not yet occurred. It is crucial to emphasize here that it could not be the inverse combination, *i.e.,* first a formal verification layer, and then a monitoring layer. The main reason is that the monitoring layer being over a verification layer would always find something that was not modeled in the formal model. This would happen because a formal verification model cannot preview property violations resulting from accidents. Consequently, the monitoring layer, in this hypothetical case, would be constantly not synchronized with the formal verification layer. For this reason, we understand that a monitoring layer coming first is the right design choice, as it serves as a source to generate a verification layer in line with what happens in the network, and is able to extend this with more complex analysis.

Using these two layers, we model the connectivity of network components using the source and destination information from traversing packets. For example (see Fig. 5.1, steps 1, 2, 3), if component $x1$ sends an ICMP message to component $x2$, we know nothing about these partic-

ular components, but we know that $x1$ can send an ICMP message. To store this information, we use a grammar. Grammars can (i) compress several packet header information into a single rule, (ii) infer new patterns of communication by combining the rules learned, (iii) answer questions such as "Is this communication valid?" considering the *words* (packet header information) exchanged between network components, (iv) be rapidly updated by inserting/removing rules, and (v) perform a black box (all the words saved do not consider the implementation of network component) and stateful (the historical behavior is annotated) testing.

5.1.1 Grammar-based Network Modeling

Grammars are a generative description of a language including a set of rules that structure a language, including syntax - the arrangement of words - and morphology - how these words are formed. Grammars are defined precisely in (GROSS; LENTIN, 2012). Formally, a grammar is a 4-tuple $G = (V, E, R, X)$, where V is an alphabet, $E \in V$ is the set terminal symbols ($V - E$ is the set of non-terminal symbols), $R \in (V^{+}xV^{*})$ is a finite set of production rules (also called simply rules or productions), $X \in V - E$ is the start symbol.

A grammar generates a set of rules that describe a language (see Fig. 5.1, step 4). We hypothesize that if the language is built on network packets in the same context as network components, then it will observe any pattern of wrong communication, that is, any wrong message sent. We could understand which packets are correct on the network by observing the normal communication pattern of the network where a violation has not yet occurred. Recall that we have defined the combination of monitoring and formal verification in which the monitoring scheme should identify property violations before the formal verification layer. Thus, if the monitoring layer does not report any violation, the model can use the messages exchanged between network devices to create its grammar with normal communication patterns. Using a grammar representation, the basic idea is: write down the header information – source, destination, and protocol – of packets sent from one device to another to have a trace of what is communicated. NetWords performs the conversion of header information into words to compose its grammar.

Grammar Inference:

$C = \{c_1, c_2, c_3, ...\}$ - set of network components - anything that receives and sends packets, such as switches, routers, and hosts.

$P = \{p_1, p_2, p_3, ...\}$ - set of protocols in the network.

$N = \{(c_1, p_1, c_2), (c_1, p_2, c_3), ...\}$ - set of tuples representing the communications in the network - source, protocol, and destination.

$G = (V, E, R, X)$ - a grammar, where:

V is P $\cup$ C - network components, protocols
E is C$\times$P$\times$C - set of terminal symbols
R is a finite set of rules (R $\simeq$ N)
$X \in V - E$ is the start symbol, a dummy symbol S

Fundamental properties:

- All elements of N are in G:
  $\forall a, b, c$ such that $a, c \in C$ and $b \in P$
  $(a, b, c) \in N \rightarrow (a, b, c) \in G$
- Communication induced by G is possible in N
  $\exists a, b, c$ such that $a, c \in C$ and $b \in P$
  $(a, b, c) \in G \rightarrow (a, b, c) \in N$

Considering Figure 5.1, a grammar will be defined as:

- V - protocols and network components:
  $\{x1, x2, ..., TCP, UDP, ...\}$
- E - terminal symbols: *{(x1,p,x2),....}*
- R - rules: element of E + next state
  (x1,p,x2) becomes x1: "x1 p x2" x2
- X - the start symbol: $\{S\}$

If component $x1$ sends a packet to $x2$ with protocol p, we generate the rule *"x1 p x2" x2* in the grammar and all encapsulated protocols in p (Fig. 1, step 4). In order to provide a high-level language for the network administrator to use, we defined the following three predicates:

$send(p, x1, x2)$: component $x1$ sent packet p to $x2$
$receive(p, x1, x2)$: component $x2$ received packet p from $x1$

$communicate(p, x1, x2) : send(p, x1, x2)$ and $receive(p, x1, x2)$

The high-level predicates defined above always return a logical value, representing True or False (see Figure 5.1, step 6). These predicates allow writing expressions that support logical connectives such as AND, OR, and NOT operators to expand the expressiveness of the questions that can be asked in the model. Consequently, the questions: *send(p, x1, x2)* AND *send(p, x2, x3)* will produce a logical value. With these logic operators, one can express any logic function, consequently generating a large set of possibilities. Table 5.1 defines these logical operators an their syntax in NetWords.

Table 5.1 – Logic operators in NetWords

Logic Function	**Description**	**Example Expression**
AND	Logical and	send(p,x1,x2) AND send(p,x1,x2)
OR	Logical or	send(p,x1,x2) OR send(p,2,x3)
!	Negation	!send(p,x1,x2)
@	Universal quantifier “for all”	@x send(p,x1,x2)
&	Exist quantifier “there is/are”	&x send(p,x1,x2)

Our grammar is used within the formal verification layer to check property violations considering the existence of a monitoring layer that can collect network information to produce the N, P and C sets. In particular, our monitoring is ARMOR, which uses techniques such as machine learning to detect property violations and warn their occurrences. This monitoring layer gathers and consolidates information from different data sources, and provide means of querying possible violations in the network through a $query(p, x1, x2)$ predicate, which will return True if a property violation related to network component $x1$ or $x2$ was found when protocol p is used.

Using NetWords, a network administrator will specify properties of interest that s/he wants to be checked by using logical expressions based on the predicates defined above. The grammar that represents the network will be used to check the validity of these properties. To do so, the logical expressions written by the administrator must be translated by NetWords into grammar productions which can be used for automatic property checking. The productions have the form defined in E, *i.e.,* a cartesian product $C \times P \times C$. For simplicity, NetWords takes any element (a, b, c) produced by the cartesian product and represents it as the *“a b c”* string. This conversion is necessary to generate a rule to be inserted in R. Finally, the next state c is inserted at the end, producing *“a b c” c*.

Grammar Recognition:

- *Step 1:* The network administrator describe properties using *send*, *receive* and *communicate* predicates.
- *Step 2:* NetWords will convert *send(p, x1, x2)* to an element of E represented by *(x1, p, x2*) and later simplified to *"x1 p x2" x2*.
- *Step 3:* NetWords will convert *receive(p, x1, x2)* to an element of E represented by *(x1, p, x2)* and later simplified to *"x1 p x2" x2*.
- *Step 4:* NetWords will recursively, repeat this task. It will return TRUE if the word is possible in the language, or FALSE, otherwise.

This process of recognition is represented in Figure 5.1, step 5. If we want to test if $x1$ can communicate with $x2$, we only need to create the string *send(p, x1, x2)*, where p is a protocol, such as TCP. NetWords will convert *send(TCP, x1, x2)* to *"x1 tcp x2" x2*.

5.1.2 Working Example

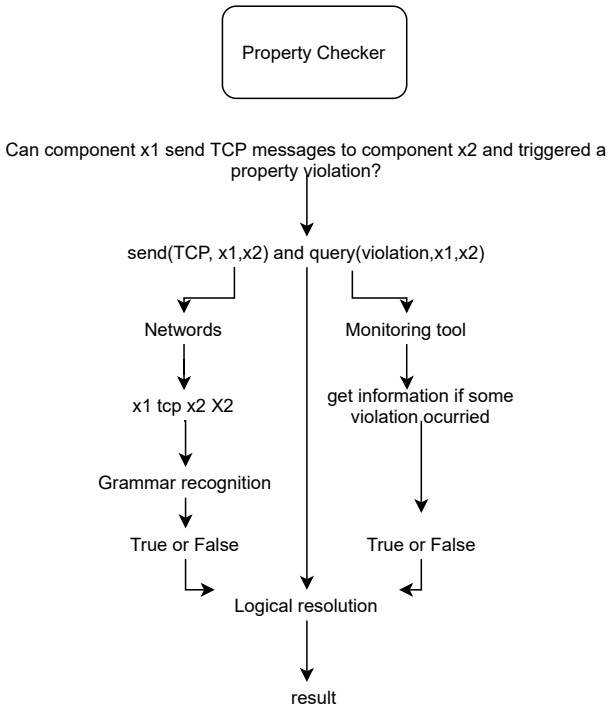
An illustrative example of how to evaluate a network property using the grammar produced by NetWords is depicted in Figure 5.2. There are two paths to follow. The first one is responsible for evaluating a single property. In this case, the network administrator will convert a property into *send* and *receive* primitives and evaluate it in the grammar. In the second, the network administrator can ask the monitoring layer if there was a property violation identified previously. To explain both of them, consider the general question: *"Is there a link failure in the network?"*.

It is desirable that packets can traverse the network and the absence of this condition is an example of a property violation. First of all, we need to describe this as a more specific question. An alternative question to check this property would be *"Can component x1 send TCP messages to component x2?"* (see Figure 5.2, left-side). Also, recall that there is a monitoring layer operating jointly with the verification layer, and as such we can ask additionally if some property violation was found during the monitoring process: *"Is there a property violation involving x1 or x2?"* (see Figure 5.2, right-side).

The network administrator can express these questions using the *send*, *receive*, and *query* primitives, in particular: *"send(TCP, x1, x2) and query(TCP, x1, x2)"*. Next, NetWords will convert this property into a grammar input. For example, as explained earlier, *send(TCP, x1, x2)* is converted into *"x1 tcp x2" x2*.

In NetWords, recognition is performed using the *Cocke-Younger-Kasami (CYK)* algorithm where a word can be recognized in grammar (Figure 5.3). This process considers *"x1 tcp x2" x2* as a valid word and tries to define which rule could be the one that generated this in a bottom-up process. In Figure 5.3, for example, the rule generated by $x1$ was selected and $x2$ is not because

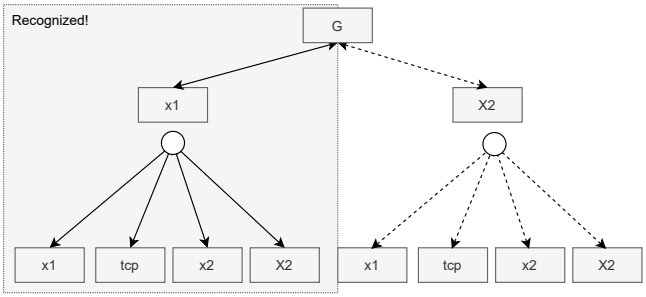
Figure 5.2 – Working Example of Property Evaluation



Source: by author

the algorithm could not generate the word using $x2$. The process is repeated until it reaches the top of the tree. In this case, it returns TRUE, otherwise returns FALSE.

Figure 5.3 – Cocke-Younger-Kasami Algorithm to Recognize *"x1 tcp x2" x2*



Source: by author

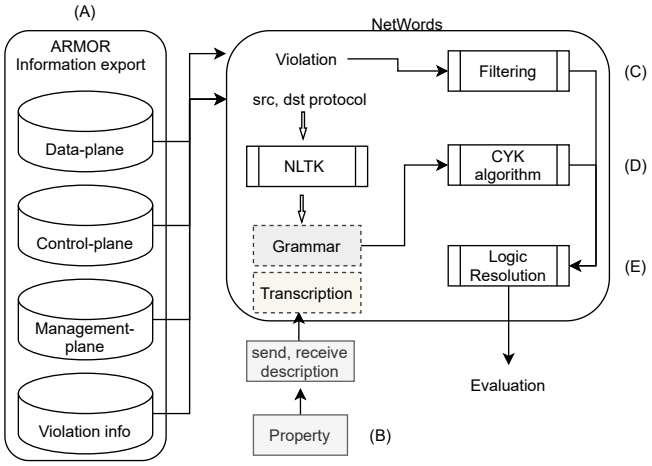
Furthermore, the monitoring layer returns the information if a property violation occurred in response to *query(TCP, x1, x2)*, resulting in a logical value TRUE or FALSE. The logical resolution step simply determines the final logical result considering the operators in the property query. Finally, the result is returned.

5.1.3 System Components

As described in Figure 5.4, the NetWords workflow starts with step A. This comprises the network monitoring layer. As mentioned earlier, NetWords monitoring capabilities build upon

our previous layer, called ARMOR, which is based on three conceptual principles. The first one is the use of *heterogeneous network information sources*, which can collect information considering different views of the network (management, control, and data plane). Here we decided to use an *indirect monitoring* schema since we can obtain information directly from the S-flow protocol, SNMP protocol, and from the SDN controller. The second one is *learning from past properties violations*, which can detect and classify network violations using machine learning algorithms. Finally, the third one is that *property violations can repeat*, thus we need to suggest actions when a similar violation occurs. ARMOR can export information about protocols and property violations found in the network through a predicate named $query$.

Figure 5.4 – Detailed architecture of NetWords considering its internal components and execution flow



Source: by author

In step B, the network administrator should convert properties of interest to NetWords context using the primitives *send* and *receive*. We defined a transcription routine able to convert the primitives into a word to be recognized by the grammar (see Section 5.1.1). Next, NetWords receives tuples of (source, destination, protocol) from the monitoring layer and constructs the grammar to describe the relationship between network components in the monitored topology. We used here the Natural Language Toolkit from the Python language[1] to generate a parser to represent the grammar.

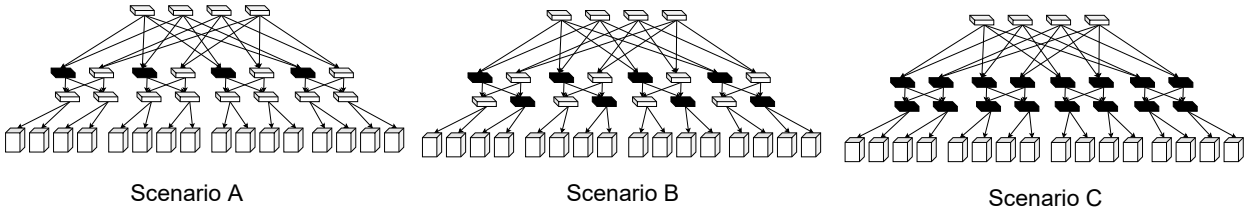
Step C represents the conversion of ARMOR violation information into a logical value, evaluating if the network component informed as a parameter is related to a property violation caught by the monitoring process. Step D executes the CYK algorithm to check if the transcripted word is recognized by the grammar. Here, it is performed a syntax check (a routine able to check if only functions that NetWords can understand are present in the property query) and retrieves any information required to process the query. Finally, in step E the complete

[1] https://www.nltk.org/

execution of this flow enables the understanding of the set of desirable and undesirable network properties.

5.2 Case study: A Data Center Network

Figure 5.5 – Alternatives for positioning NetWords observation points in a fat-tree topology



Source: by author

We present an experimental evaluation of the NetWords testing system considering the performance of the checking process and resource usage. In particular, we study the best configuration to place observation spots and demonstrate which properties the implemented system can check, as well as issues related to scalability, such as the runtime needed to check network properties.

NetWords was implemented using the ONOS controller[2]. The network itself was emulated in the Mininet emulator. Table 5.2 summarizes the execution environment used in the experiments described next.

Table 5.2 – Execution environment used in the experiments

Parameter	**Value**
Operational System	Microsoft Windows 10
Processor	Intel(R) Core i-5-825OU, 1.60Ghz,1800
Number of Cores	4 cores, 8 processors
RAM memory	8 GB
Virtual Memory	10GB

In our evaluation scenario, we used a topology of a data-center network. It consists of a fat-tree topology comprising 48 hosts and 20 switches. We chose this scenario because of recent vulnerabilities occurrences due to configuration errors in similar environments (BURNETT et al., 2020). The traffic profile simulates statistically the behavior of common web users who watch video streaming and make requests for a data center. For each HTTP request, there is a request for a video stream that can last for a period between 3 and 5 seconds. The experiments were performed 35 times each lasting around 10 to 40 minutes until the experimental error was less than 0,01 with a confidence interval of 95%. Table 5.3 summarizes the traffic profile simulated. Experiments were run using actual topologies/traffic in Mininet (information was obtained via REST) in a similar manner to what was done in (ISOLANI et al., 2015; SILVA

[2]https://opennetworking.org/onos/

et al., 2016). Considering the analysis of the best observation spots to place NetWords, three scenarios were elaborated (see Figure 5.5):

- scenario A: few observation spots
- scenario B: medium number of observation spots
- scenario C: large number of observation spots

After identifying the best scenario (as presented next in Section 5.2.1), we defined four metrics to evaluate NetWords: (i) CPU runtime, (ii) number of observation spots required, (iii) number of network properties successfully checked.

Table 5.3 – Traffic profiles used in scenarios A, B, C

Parameter	**Value**
Number of hosts	16
Number of switches	20
Protocols	HTTP, TCP, UDP
Host behavior	Randomly distributed

5.2.1 Best Observation Spot Evaluation

We understand that the first criterion for determining where to place NetWords is maximizing the number of monitored packets. To find the best scenario to collect network samples that will allow generating the grammar, we used graph theory to study the best position to collect network information. We defined three scenarios representing in black color the switches that NetWords is observing (see Figure 5.5). Note that we only chose switches in the aggregation layer because they present a higher number of connections, thus maximizing the communication relations in the observation spots.

We ran our experiments for 1-hour collecting network communications patterns considering scenarios A, B, and C. In each scenario, we annotated the percentage of traffic collected over the total (around 170,000 flows). We understand as collected traffic the information necessary to generate the grammar by NetWords. We observed that using scenario A only 47% of the flows could be collected and thus used for building the grammar. In scenario B, 86% of flows were collected and in scenario C, 97% of flows were collected. We repeated this experiment 29 times until the experimental error was less than 0.1. We highlight that even Scenario C rarely could obtain 100% of flows because polling-based traffic monitoring can miss communications happening between two polling intervals.

The second criterion to define the best scenario to place NetWords is the number of properties it can check with the information obtained. Fig 5.6 summarizes the information obtained from the previous monitoring layer. Protocols exchanged and network components (represented by the *g0, g1,...*). Later, a parser uses this information to understand the actions performed by network components and construct a matrix representing the communication pattern of the net-

Figure 5.6 – NetWords Communications Patterns

	g0	g1	g2	g3	g4	g5	g6	g7	g8	g9	g10	g11	g12	g13	g14	g15	g16	g17	g18	g19
0	[ICMP, TCP, HTTP, UDP]	[ICMP, UDP, RTP]	[IP, UDP, HTTP, ICMP, RTP]	[HTTP, ICMP, UDP]	[RTP, TCP]	[ICMP, RTP, HTTP, IP, TCP, UDP]	[TCP, IP, RTP, ICMP, HTTP]	[UDP, TCP, HTTP]	[RTP, IP]	[RTP, HTTP]	[UDP, TCP, ICMP]	[RTP]	[UDP, ICMP, HTTP, RTP, TCP]	[RTP, IP]	[HTTP, UDP, IP, TCP, RTP]	[IP, ICMP, HTTP, TCP, UDP, RTP]	[UDP, RTP, HTTP]	[RTP]	[UDP, ICMP, HTTP, IP, RTP, TCP]	[TCP, UDP, RTP, IP, ICMP, HTTP]
1	[HTTP]	[]	[TCP, IP, UDP, RTP, HTTP, ICMP]	[UDP, ICMP, RTP, HTTP]	[RTP, ICMP, TCP, HTTP, IP, UDP]	[RTP, HTTP, TCP, IP, ICMP, UDP]	[UDP]	[IP, ICMP, TCP]	[]	[HTTP, UDP, IP, TCP, RTP]	[ICMP, UDP]	[ICMP, UDP]	[]	[HTTP, IP, UDP]	[TCP, HTTP, ICMP, UDP, IP, RTP]	[HTTP, IP, UDP]	[IP]	[TCP, IP]	[RTP, IP]	[UDP, ICMP, TCP, IP, HTTP]
2	[ICMP, UDP, RTP]	[HTTP, ICMP, UDP, RTP, TCP]	[UDP, RTP, HTTP, ICMP, TCP]	[HTTP, IP, UDP, ICMP, TCP]	[TCP, IP, UDP]	[]	[IP, UDP, ICMP, HTTP, TCP]	[RTP, HTTP, TCP, UDP, ICMP]	[IP, HTTP, RTP, ICMP, UDP, TCP]	[HTTP, IP, UDP]	[ICMP, HTTP, RTP]	[UDP, IP, HTTP]	[HTTP, TCP, RTP, UDP, ICMP, IP]	[TCP, IP]	[UDP, TCP, RTP]	[HTTP, UDP, RTP]	[RTP, IP, UDP, HTTP, ICMP, TCP]	[ICMP]	[UDP, HTTP, ICMP]	[TCP, RTP, ICMP]
3	[]	[RTP, HTTP, IP, ICMP, TCP, UDP]	[HTTP, TCP, RTP, ICMP, IP]	[]	[ICMP, RTP, UDP, HTTP, TCP]	[TCP, IP, ICMP]	[ICMP]	[HTTP, TCP, ICMP, UDP]	[HTTP]	[]	[IP]	[IP, HTTP, TCP]	[IP, UDP, TCP]	[UDP, HTTP, IP, ICMP, TCP]	[IP, HTTP, RTP, TCP]	[]	[IP, TCP, UDP, RTP, ICMP, HTTP]	[UDP, TCP, ICMP, RTP]	[HTTP, IP, TCP, UDP, RTP, ICMP]	[]
4	[ICMP]	[IP, TCP, ICMP, HTTP]	[RTP, ICMP, UDP, HTTP]	[]	[HTTP, UDP, IP, ICMP, TCP, RTP]	[TCP, IP, ICMP]	[UDP, HTTP]	[]	[IP]	[UDP, TCP]	[UDP, TCP, IP, HTTP, ICMP, RTP]	[HTTP, UDP, ICMP, IP, TCP]	[HTTP, TCP, IP, UDP, RTP, ICMP]	[TCP, UDP, RTP]	[IP, UDP]	[ICMP, TCP, RTP, UDP, HTTP]	[TCP]	[UDP, RTP]	[ICMP, UDP, HTTP, IP, TCP, RTP]	[RTP, ICMP]

Source: by author

work. This matrix represents all the information obtained from ARMOR. NetWords obtain information from the ARMOR without imposing any changes on this. The following information about the network is presented in this matrix: (i) network topology provided with identifiers for hosts, switches, and communication links; and (ii) flows that are active and inactive on the network including packet header information.

Figure 5.7 – NetWords Actions Interpretation

```
[2] Action [3] Protocol [4] ==> [2] SOURCE [4]
[5] Action [6] Protocol [7] ==> [5] SOURCE [7]
[1] TEST [2] SOURCE [4] ==> [1] END [4]
[4] dest [5] SOURCE [7] ==> [4]  PP [7]
[0] SOURCE [1] END [4] ==> [0]   S [4]
[1] END [4]  PP [7] ==> [1] END [7]
[0] SOURCE [1] END [7] ==> [0]   S [7]
```

```
WFST 1    2    3    4    5    6    7
0    SOURCE .    .    .    .    .    .
1    .    TEST .    .    .    .    .
2    .    .    Action .    .    .    .
3    .    .    .    Protocol .    .    .
4    .    .    .    .    dest .    .
5    .    .    .    .    .    Action .
6    .    .    .    .    .    .    Protocol
```

Source: by author

After Networds creates the extended matrix of actions in the network (see Fig. 5.7). This step performs the organization of ARMOR information that should be used to produce the grammar. Note that this component accesses the raw information collected previously and transforms it into a set of traffic actions to understand which types of communications there are in the network. The final step generates a grammar as depicted in Fig 5.8.

Finally, Table 5.4 summarizes 7 examples of other properties studied in this evaluation. We show the runtime of the checking process with an error rate less than 0.01 for each scenario that

Figure 5.8 – NetWords Grammar Generated

```
S -> SOURCE END
PP -> dest SOURCE
SOURCE -> Action Protocol | Action Protocol PP | 'x1'
END -> TEST SOURCE | END PP
Action -> 'send' | 'receive'
Protocol -> 1?
TEST -> 'can'
dest -> 'x2'
```

Source: by author

could evaluate the respective property using the information available[3]. After 35 executions with an interval ranging from 800 to 170,000 flows we determined that (i) scenarios B and C could check all the properties, however, scenario B used fewer observation spots than C; (ii) the most time-consuming property was the third one, which took 2m3s to be checked in scenario B. This happened because this property required NetWords to explore all the topologies searching for all TCP communications.

Table 5.4 – Properties evaluation

Prop.	NetWords Expression	A	B	C
A	&x communicate(x, 1,2)	1.3s	1.2s	1.2s
B	@x@y communicate(y, 1, x)	1.5s	1.3s	1.3s
C	&x communicate(x, 1,1) AND query(TCP,x,1)	N/A	2m3s	2m2s
D	communicate(x,3,4)	39s	35s	33s
E	&x communicate(x,1,2) AND query(IP, 1,2)	1m3s	1m2s	1m
F	&x &y communicate(x,2,y)	N/A	1m3s	1m1s
G	&x communicate(HTTP, 2, x)	N/A	1m35s	1m2s

Considering that scenario B could detect all properties and cover 86% of the flows, we chose to proceed with this scenario for the remaining evaluation described in this section. As for the other scenarios, scenario A could not check 3 out of 7 properties, and scenario C used more observation spots than scenario B without a considerable gain in performance. For example, a simple property can be tested like in Fig 5.11.

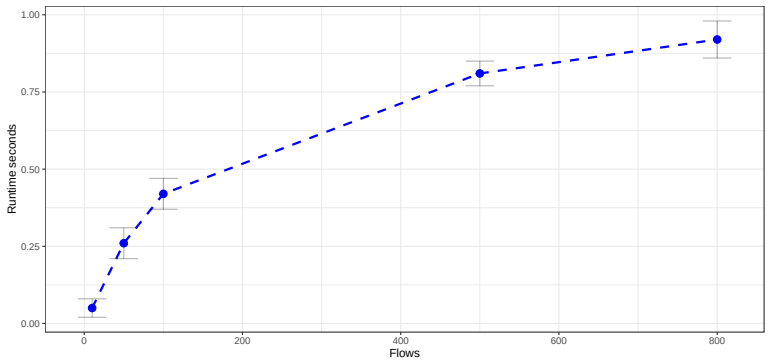
5.2.2 Resources Usage Evaluation

Considering scenario B, we evaluate the runtime of our system. The runtime of the integration with ARMOR when the number of flows grows is less than 1s. We ran 38 times until the experimental error was less than 0.01 and with a confidence interval of 95%. After an initial growth in the traffic rate, the number of flows stops growing and maintains its distribution. Note yet, the initial process of taking information about our grammar does not need to collect

[3]If the property could not be evaluated because the produced grammar did not capture the required information, this is indicated with N/A.

this directly from the network because ARMOR still has this information and maintains this updated. After this initial step, the grammar learns the network communication patterns and stops monitoring it frequently (Figure 5.9).

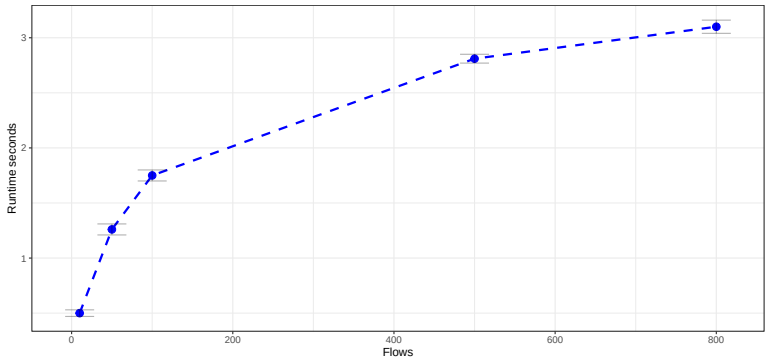
Figure 5.9 – Runtime Integration with ARMOR



Source: by author

The runtime of the grammar generation when we analyze a growing number of flows is around 3s (Figure 5.10). This was stressed and tested over a simulation during one day where we studied how long NetWords needs to create grammars. We concluded that the grammar generation performance takes around 3s for a growing number of flows. Further, after the grammar is generated, there is no need to generate it again. Only simple updates – insert one or more productions – are needed periodically to update communication patterns monitored in the network.

Figure 5.10 – Runtime of Grammar Generation



Source: by author

Based on these results, we understand that Networds is not prohibitive in terms of CPU usage.

5.2.3 Complex Properties Evaluation

NetWords is not intended to be a model that checks all possible properties on a computer network as this would be impractical. Only a network model identical to reality could achieve this, but then it would no longer be a model that is a generally lossy representation of reality. We understand that NetWords can model any property that can be described as a combination of sending and receiving specific messages. Note that this covers an extensive set of properties. Below, we discuss four examples.

1 - Black hole: Every packet should reach its intended destination without being dropped along the way. An intermediate device that incorrectly discards packets results in what is called a *black hole*. Black holes can arise due to several reasons, such as misconfigurations;

2 - Reachability: At least one packet from x can reach y. This invariant guarantees that the network works correctly concerning packet routing;

3 - Isolation: Some packets from x should not reach y. Due to security issues, a given host may be prevented from communicating with other hosts;

4 - Loop-free network: property related to reachability ensures that a given packet cannot be forwarded back to the source without reaching the target;

Invariants are properties that should be always satisfied in the network. Table 5.5 shows the results for their evaluation in scenario B. Their execution time may be higher since we need to evaluate not only a restricted subset of network components.

To simulate the grammar use, we show a simple example and a set of more complex ones. For each word in the grammar question *'x1', 'can', 'send', 'TCP', 'x2', 'receive', 'TCP'*. The next step is recognizing the S step, the initial step of grammar. The action step, protocol step, source, and destination step represent the grammar check of the matrix generated using the traffic information collected by ARMOR. Next, we explain the behavior of our simulated properties.

Figure 5.11 – NeWords Evaluation Example

```
['x1', 'can', 'send', 'TCP', 'x2', 'receive', 'TCP']
(S
  (SOURCE x1)
  (END
    (END (TEST can) (SOURCE (Action send) (Protocol TCP)))
    (PP (dest x2) (SOURCE (Action receive) (Protocol TCP)))))
(S
  (SOURCE x1)
  (END
    (TEST can)
    (SOURCE
      (Action send)
      (Protocol TCP)
      (PP (dest x2) (SOURCE (Action receive) (Protocol TCP))))))
```

Source: by author

Table 5.5 compares the execution time of examples of other properties checked. Note that the execution time is not prohibitive considering the formal verification context. Their exe-

cution time may be higher since we need to evaluate not only a restricted subset of network components.

Table 5.5 – Invariant evaluation

Inv.	NetWords Expression	Runtime	Error
1	!send(x, y,z) OR receive(x,y,z)	6min	0.4
2	&x !send(x, y,z) OR receive(x,y,z)	4min	0.2
3	!send(x, y,z) OR !receive(x,y,z)	5.7min	0.8
4	!receive(x,y,y)	10.4min	1.2

Considering the combination of ARMOR and NeWords, some properties will combine them in different ways. Table 5.6 compares 3 complex properties and shows the combination of the solutions. Considering the properties that use the query function a combination with ARMOR is used. Properties related to only simple communication do not use NetWords. Still, some properties will use only ARMOR. The ones related to query function only.

Table 5.6 – Logic operators in NetWords

Logic Function	Primitives	ARMOR	NetWords
Are there any devices on the network that are not receiving packets from the B portion due to property violations?	&x p x,y send(p, x,y) -> !receive(p, x1, 3) AND !receive(p, x1, 4) AND query (violation, x, B)	YES	YES
Are there any devices on the network that are not receiving packets from the B portion	&x p x,y send(p, x,y) -> !receive(p, x1, 3) AND !receive(p, x1, 4)	NO	YES
Are there any property violations in the B portion of network?	query (violation, x, B)	YES	NO

The first case is an ownership check where both views (monitoring and formal) need to be used together. The question "Are there any devices on the network that are not receiving packets from the B portion due to property violations?" needs to check if it is possible for some devices not to receive packets from B when they should, and also if any property violation has already occurred in B (and therefore it is no longer able to receive packets). Evaluating this property includes asking NetWords whether &x p x,y send(p, x,y) -> !receive(p, x1, 3) AND !receive(p, x1, 4) and to ARMOR if query (violation, x, B) occurred. Only if both answers are yes will this property be satisfied.

The second case is a property check where only the formal view needs to be used. The question "Are there any devices on the network that are not receiving packets from the B portion" needs to check if it is possible for some devices not to receive packets from B when they should be receiving packets and also if there is any property violation. Evaluating this property includes asking NetWords whether &x p x,y send(p, x,y) -> !receive(p, x1, 3) AND !receive(p, x1, 4) . Here we only need to use one of the views to determine whether this property will be satisfied.

The last case is a property check where only the monitoring view needs to be used. The question "Are there any property violations in the B portion" needs to check if there has been a

property violation in the past. Evaluating this property includes asking ARMOR if query (violation, x, B). Here we only need to use one of the views to determine whether this property will be satisfied or not using the data models saved by ARMOR over the network communication. Table 5.6 summarizes the execution of these more complex properties

6 FINAL CONSIDERATIONS

In this chapter, we present the conclusions and contributions obtained from the work developed in the context of this Ph.D. research. We also present the publications already obtained.

6.1 Conclusions

Although the use of SDN promotes more flexibility in the design of the network and facilitates the programmability of network equipment, this also makes it more difficult to ensure that network configuration is free from property violations. We propose here a combination of two studies based on network monitoring and formal verification. In this thesis, we report our research to achieve a combination of network monitoring and formal verification to test computer networks. Our research concludes with the following concrete considerations about the research questions.

The first contribution is represented by a monitoring layer for detecting property violations and intends to show a comprehensive architecture to manage property violation diagnosis by using monitoring sampling tools, such as SNMP, REST API, and sFlow. We guarantee the flexibility, accuracy, and automatization of our solution by using machine learning techniques, such as decision trees to perform our property violation classification.

The second contribution is represented by a formal verification layer. This thesis presents a novel approach to representing networks based on regular grammar that can be generic enough to represent the entire network and specific enough to model single devices. Our model uses as an information source the information of messages that enter and leave the devices. In this way, it is not necessary to know the internal state of a device to model its behavior.

Our results suggest the following contributions: (i) **Vast Amount of Data -** the use of snapshots reduces the amount of data that needs to be stored and the use of these snapshots maintains the quality of property violation detection. (ii) **Financial Limitations-** it is possible to extract network information without the use of third-party expensive products by using SNMP, s-flow, and REST API in SDN. (iii) **Information Reasoning -** Machine learning algorithms are a suitable alternative to detect property violations and learn the set of most appropriate strategies for remediating them. (iv) **Configuration Checking:** Networds can check property described by the network administrator using a combination of monitoring and formal verification techniques; (v) **State Space Explosion:** The use of grammars can decrease the number of states necessary to model network communication patterns. (vi) **Mitigation Strategies:** Grammars can detect the possibility of property violation before it occurs when the network administrator asks.

6.2 Future Work

This thesis has mainly focused on designing a complete strategy to perform network testing. To achieve this, we studied how network testing can benefit from the joint operation of (i) property violation monitoring and (ii) formal verification to search for property violations. We advocate that property violation monitoring by itself cannot be complete because there are no capabilities of proving the absence of property violations. Concurrently, formal verification to search property violations by itself cannot detect a property violation that was not anticipated and included in the model. A special feature related to the formal verification context is the use of a model based on grammar to capture the communication patterns existing on the network. This model enabled the study of property violations that are global on a network or specific from a single component.

However, there are still opportunities to enhance the research done so far. Although a property violation is often related only to the context of a single network component, it can happen to affect neighboring network components. In this case, it is necessary to understand the concept of the *property violation propagation*.

In the fault tolerance area (AVIZIENIS et al., 2004), we say that when a *failure propagation* occurs there is a chain of effects related to the original failure in the neighbors of the faulty device (PI et al., 2018). We extend this definition to encompass the property violation vision that we use in this thesis. Consequently, property violation propagation is a natural extension of the monitoring layer for property violation detection.

The formal verification layer needs this enhancement to increase the expressibility of questions that the network administrator can evaluate on the network. For example, with this extension, we can check for properties like "Until network component x1 does not send an HTTP message, x1 does not send an ICMP message".

6.3 Achievements

This section provides an overview of the main research activities carried out during the period of the Ph.D.

- **NetWords: Enabling the Understanding of Network Property Violation Occurrences** AS da Silva, A Schaeffer-Filho NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium
- **ARMOR: An Architecture for Diagnosis and Remediation of Network Misconfigurations** AS da Silva, A Schaeffer-Filho 2019 IEEE Symposium on Computers and Communications (ISCC), 1-6

- **Using NFV and reinforcement learning for anomalies detection and mitigation in SDN** LSR Sampaio, PHA Faustini, AS Silva, LZ Granville, A Schaeffer-Filho 2018 IEEE Symposium on Computers and Communications (ISCC), 00432-00437
- **Improved network traffic classification using ensemble learning** IP Possebon, AS Silva, LZ Granville, A Schaeffer-Filho, A Marnerides 2019 IEEE Symposium on Computers and Communications (ISCC), 1-6
- **PRIME: Programming In-Network Modular Extensions** R Parizotto, L Castanheira, F Bonetti, A Santos, A Schaeffer-Filho NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium, 1-9

REFERENCES

ABDULQADDER, I. H.; ZHOU, S.; ZOU, D.; AZIZ, I. T.; AKBER, S. M. A. Multi-layered intrusion detection and prevention in the sdn/nfv enabled cloud of 5g networks using ai-based defense mechanisms. **Computer Networks**, Elsevier, p. 107364, 2020.

ACETO, G.; BOTTA, A.; DONATO, W. D.; PESCAPÈ, A. Cloud monitoring: A survey. **Computer Networks**, Elsevier, v. 57, n. 9, p. 2093–2115, 2013.

AL-SHAER, E.; AL-HAJ, S. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In: **Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration**. New York, NY, USA: ACM, 2010. (SafeConfig '10), p. 37–44. ISBN 978-1-4503-0093-3.

AL-SHAER, E.; AL-HAJ, S. Flowchecker: Configuration analysis and verification of federated openflow infrastructures. In: **Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration**. New York, NY, USA: ACM, 2010. (SafeConfig '10), p. 37–44. ISBN 978-1-4503-0093-3. Available from Internet: <http://doi.acm.org/10.1145/1866898.1866905>.

AL-SHAER, E.; ALSALEH, M. N. Configchecker: A tool for comprehensive security configuration analytics. In: **2011 4th Symposium on Configuration Analytics and Automation (SAFECONFIG)**. [S.l.: s.n.], 2011. p. 1–2.

AL-TASHI, Q.; ABDULKADIR, S. J.; RAIS, H. M.; MIRJALILI, S.; ALHUSSIAN, H. Approaches to multi-objective feature selection: A systematic literature review. **IEEE Access**, IEEE, v. 8, p. 125076–125096, 2020.

ALPAYDIN, E. **Introduction to Machine Learning (Adaptive Computation and Machine Learning)**. [S.l.]: The MIT Press, 2004. ISBN 0262012111.

ALUR, R. Formal verification of hybrid systems. In: **Proceedings of the ninth ACM international conference on Embedded software**. [S.l.: s.n.], 2011. p. 273–278.

ALUR, R.; COURCOUBETIS, C.; HALBWACHS, N.; DILL, D.; WONG-TOI, H. Minimization of timed transition systems. In: SPRINGER. **International Conference on Concurrency Theory**. [S.l.], 1992. p. 340–354.

ARASHLOO, M. T.; KORAL, Y.; GREENBERG, M.; REXFORD, J.; WALKER, D. Snap: Stateful network-wide abstractions for packet processing. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. New York, NY, USA: ACM, 2016. (SIGCOMM '16), p. 29–43. ISBN 978-1-4503-4193-6. Available from Internet: <http://doi.acm.org/10.1145/2934872.2934892>.

AVENHAUS, J.; KÜHLER, U.; SCHMIDT-SAMOA, T.; WIRTH, C.-P. How to prove inductive theorems? quodlibet! In: SPRINGER. **International Conference on Automated Deduction**. [S.l.], 2003. p. 328–333.

AVIZIENIS, A.; LAPRIE, J.-C.; RANDELL, B.; LANDWEHR, C. Basic concepts and taxonomy of dependable and secure computing. **IEEE Trans. Dependable Secur. Comput.**,

IEEE Computer Society Press, Los Alamitos, CA, USA, v. 1, n. 1, p. 11–33, jan. 2004. ISSN 1545-5971. Available from Internet: <http://dx.doi.org/10.1109/TDSC.2004.2>.

BACHMAIR, L.; GANZINGER, H.; WALDMANN, U. Refutational theorem proving for hierarchic first-order theories. **Applicable Algebra in Engineering, Communication and Computing**, Springer, v. 5, n. 3-4, p. 193–212, 1994.

BAIER, C.; KATOEN, J.-P. **Principles of model checking**. [S.l.]: MIT press, 2008.

BAIER, C.; KATOEN, J.-P.; HERMANNS, H. Approximative symbolic model checking of continuous-time markov chains. In: SPRINGER. **International Conference on Concurrency Theory**. [S.l.], 1999. p. 146–161.

BALL, T.; BJØRNER, N.; GEMBER, A.; ITZHAKY, S.; KARBYSHEV, A.; SAGIV, M.; SCHAPIRA, M.; VALADARSKY, A. Vericon: Towards verifying controller programs in software-defined networks. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 49, n. 6, p. 282–293, jun. 2014. ISSN 0362-1340. Available from Internet: <http://doi.acm.org/10.1145/2666356.2594317>.

BARI, M.; BOUTABA, R.; ESTEVES, R.; GRANVILLE, L.; PODLESNY, M.; RABBANI, M.; ZHANG, Q.; ZHANI, M. Data Center Network Virtualization: A Survey. **IEEE Communications Surveys Tutorials**, Piscataway, NJ, USA, v. 15, n. 2, p. 909–928, Second Quarter 2013. ISSN 1553-877X.

BECKETT, R.; GUPTA, A.; MAHAJAN, R.; WALKER, D. A general approach to network configuration verification. In: **Proceedings of the Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2017. (SIGCOMM '17), p. 155–168. ISBN 978-1-4503-4653-5. Available from Internet: <http://doi.acm.org/10.1145/3098822.3098834>.

BECKETT, R.; MAHAJAN, R.; MILLSTEIN, T.; PADHYE, J.; WALKER, D. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. New York, NY, USA: ACM, 2016. (SIGCOMM '16), p. 328–341. ISBN 978-1-4503-4193-6. Available from Internet: <http://doi.acm.org/10.1145/2934872.2934909>.

BEREZIN, S.; CAMPOS, S.; CLARKE, E. M. Compositional reasoning in model checking. In: SPRINGER. **International Symposium on Compositionality**. [S.l.], 1997. p. 81–102.

BIFULCO, R.; SCHNEIDER, F. Openflow rules interactions: Definition and detection. In: **2013 IEEE SDN for Future Networks and Services (SDN4FNS)**. [S.l.: s.n.], 2013. p. 1–6.

BIFULCO, R.; SCHNEIDER, F. Openflow rules interactions: definition and detection. In: IEEE. **2013 IEEE SDN for Future Networks and Services (SDN4FNS)**. [S.l.], 2013. p. 1–6.

BIRKNER, R.; DRACHSLER-COHEN, D.; VANBEVER, L.; VECHEV, M. Config2spec: Mining network specifications from network configurations. In: **17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)**. [S.l.: s.n.], 2020. p. 969–984.

BOWMAN, B.; LAPRADE, C.; JI, Y.; HUANG, H. H. Detecting lateral movement in enterprise computer networks with unsupervised graph ai. In: **RAID**. [S.l.: s.n.], 2020. p. 257–268.

BRIM, L.; CRHOVA, J.; YORAV, K. Using assumptions to distribute ctl model checking. **Electronic notes in theoretical computer science**, Elsevier, v. 68, n. 4, p. 559–574, 2002.

BRYANT, R. E. Symbolic simulation-techniques and applications. In: IEEE. **27th ACM/IEEE Design Automation Conference**. [S.l.], 1990. p. 517–521.

BURCH, J. R.; CLARKE, E. M.; LONG, D. E.; MCMILLAN, K. L.; DILL, D. L. Symbolic model checking for sequential circuit verification. **IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems**, IEEE, v. 13, n. 4, p. 401–424, 1994.

BURNETT, S.; CHEN, L.; CREAGER, D. A.; EFIMOV, M.; GRIGORIK, I.; JONES, B.; MADHYASTHA, H. V.; PAPAGEORGE, P.; ROGAN, B.; STAHL, C. et al. Network error logging: Client-side measurement of end-to-end web service reliability. In: **17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)**. [S.l.: s.n.], 2020. p. 985–998.

CANINI, M.; VENZANO, D.; PEREsÍNI, P.; KOSTIć, D.; REXFORD, J. A nice way to test openflow applications. In: **Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2012. (NSDI'12), p. 10–10. Available from Internet: <http://dl.acm.org/citation.cfm?id=2228298.2228312>.

CARDOZA, W. M.; DIEWALD, J. M.; NELSON, J. E.; DIPIRRO, S. D.; GODDARD, J. R.; JR, W. B. F.; MCELEARNEY, A. E.; SAYDE, R. **Method and apparatus for testing software on a computer network**. [S.l.]: Google Patents, 1997. US Patent 5,630,049.

CHAN, K. H. R.; YU, Y.; YOU, C.; QI, H.; WRIGHT, J.; MA, Y. Redunet: A white-box deep network from the principle of maximizing rate reduction. **J Mach Learn Res**, v. 23, n. 114, p. 1–103, 2022.

CHEN, W. Y.; FU, A. M. A context-free grammar for the e-positivity of the trivariate second-order eulerian polynomials. **Discrete Mathematics**, Elsevier, v. 345, n. 1, p. 112661, 2022.

CHOWDHURY, S. R.; BARI, M. F.; AHMED, R.; BOUTABA, R. Payless: A low cost network monitoring framework for software defined networks. In: IEEE. **2014 IEEE Network Operations and Management Symposium (NOMS)**. [S.l.], 2014. p. 1–9.

CIUCU, F.; SCHMITT, J. Perspectives on network calculus: No free lunch, but still good value. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 42, n. 4, p. 311–322, aug. 2012. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/2377677.2377747>.

CLARKE, E.; GRUMBERG, O.; JHA, S.; LU, Y.; VEITH, H. Progress on the state explosion problem in model checking. In: SPRINGER. **Informatics**. [S.l.], 2001. p. 176–194.

CLARKE, E. M.; EMERSON, E. A.; JHA, S.; SISTLA, A. P. Symmetry reductions in model checking. In: SPRINGER. **International Conference on Computer Aided Verification**. [S.l.], 1998. p. 147–158.

CLARKE, E. M.; EMERSON, E. A.; SISTLA, A. P. Automatic verification of finite-state concurrent systems using temporal logic specifications. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM New York, NY, USA, v. 8, n. 2, p. 244–263, 1986.

CLARKE, E. M.; WING, J. M. Formal methods: State of the art and future directions. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 28, n. 4, p. 626–643, 1996.

DELMAS, T.; IANNONE, L.; GARCIA, J.-P.; MONSUEZ, B. A new network configuration management architecture for future aircraft systems. In: **10th European Congress on Embedded Real Time Software and Systems (ERTS 2020)**. [S.l.: s.n.], 2020.

DILL, D.; TASIRAN, S. Simulation meets formal verification. **slides from a presentation at ICCAD**, 1999.

DINA, A. S.; MANIVANNAN, D. Intrusion detection based on machine learning techniques in computer networks. **Internet of Things**, Elsevier, v. 16, p. 100462, 2021.

DOBRESCU, M.; ARGYRAKI, K. Software dataplane verification. In: **Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2014. (NSDI'14), p. 101–114. ISBN 978-1-931971-09-6. Available from Internet: <http://dl.acm.org/citation.cfm?id=2616448.2616459>.

EMERSON, E. A. Temporal and modal logic. In: **Formal Models and Semantics**. [S.l.]: Elsevier, 1990. p. 995–1072.

ENGEL, F.; JONES, K. S.; ROBERTSON, K.; THOMPSON, D. M.; WHITE, G. **Network monitoring**. [S.l.]: Google Patents, 2000. US Patent 6,115,393.

FAQIH, F.; ZAYED, T. Defect-based building condition assessment. **Building and Environment**, Elsevier, v. 191, p. 107575, 2021.

FAYAZBAKHSH, S. K.; CHIANG, L.; SEKAR, V.; YU, M.; MOGUL, J. C. Enforcing network-wide policies in the presence of dynamic middlebox actions using flowtags. In: **Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2014. (NSDI'14), p. 533–546. ISBN 978-1-931971-09-6. Available from Internet: <http://dl.acm.org/citation.cfm?id=2616448.2616497>.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The road to sdn. **Queue**, ACM, New York, NY, USA, v. 11, n. 12, p. 20:20–20:40, dec. 2013. ISSN 1542-7730. Available from Internet: <http://doi.acm.org/10.1145/2559899.2560327>.

GARG, V.; TOTLA, N.; KHANDELWAL, A. Recent advances in network verification and synthesis: A tutorial. **Proceedings of the IEEE**, IEEE, v. 109, n. 2, p. 240–263, 2021.

GEMBER-JACOBSON, A.; VISWANATHAN, R.; AKELLA, A.; MAHAJAN, R. Fast control plane analysis using an abstract representation. In: **Proceedings of the 2016 ACM SIGCOMM Conference**. New York, NY, USA: ACM, 2016. (SIGCOMM '16), p. 300–313. ISBN 978-1-4503-4193-6. Available from Internet: <http://doi.acm.org/10.1145/2934872.2934876>.

GIANNARAKIS, N.; SILVA, A.; WALKER, D. Probnv: probabilistic verification of network control planes. **Proceedings of the ACM on Programming Languages**, ACM New York, NY, USA, v. 5, n. ICFP, p. 1–30, 2021.

GIELEN, G. G.; WALSCHARTS, H. C.; SANSEN, W. M. Analog circuit design optimization based on symbolic simulation and simulated annealing. **IEEE Journal of Solid-State Circuits**, IEEE, v. 25, n. 3, p. 707–713, 1990.

GROSS, M.; LENTIN, A. **Introduction to formal grammars**. [S.l.]: Springer Science & Business Media, 2012.

GUHA, A.; REITBLATT, M.; FOSTER, N. Machine-verified network controllers. In: **Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation**. New York, NY, USA: ACM, 2013. (PLDI '13), p. 483–494. ISBN 978-1-4503-2014-6. Available from Internet: <http://doi.acm.org/10.1145/2491956.2462178>.

GUPTA, R. K.; IRANI, S.; SHUKLA, S. K. Formal methods for dynamic power management. In: IEEE. **ICCAD-2003. International Conference on Computer Aided Design (IEEE Cat. No. 03CH37486)**. [S.l.], 2003. p. 874–881.

HAMOLIA, V.; MELNYK, V.; ZHEZHNYCH, P.; SHILINH, A. Intrusion detection in computer networks using latent space representation and machine learning. **International Journal of Computing**, Research Institute of Intelligent Computer Systems, v. 19, n. 3, p. 442–448, 2020.

HANDIGOL, N.; HELLER, B.; JEYAKUMAR, V.; MAZIÉRES, D.; MCKEOWN, N. Where is the debugger for my software-defined network? In: **Proceedings of the First Workshop on Hot Topics in Software Defined Networks**. New York, NY, USA: ACM, 2012. (HotSDN '12), p. 55–60. ISBN 978-1-4503-1477-0. Available from Internet: <http://doi.acm.org/10.1145/2342441.2342453>.

HANNEL, C. L.; SCHAFER, D. E.; GINSBERG, E.; PEPPER, G. R. **Methods and systems for testing stateful network communications devices**. [S.l.]: Google Patents, 2007. US Patent 7,194,535.

HAYDEN, C. M.; MAGILL, S.; HICKS, M.; FOSTER, N.; FOSTER, J. S. Specifying and verifying the correctness of dynamic software updates. In: SPRINGER. **International Conference on Verified Software: Tools, Theories, Experiments**. [S.l.], 2012. p. 278–293.

HOSSEINI, S.; AZIZI, M. The hybrid technique for ddos detection with supervised learning algorithms. **Computer Networks**, Elsevier, v. 158, p. 35–45, 2019.

HUSSEIN I. H. ELHAJJ, A. C. A. K. A. Sdn verification plane for consistency establishment. In: **2016 IEEE Symposium on Computers and Communication (ISCC)**. [S.l.: s.n.], 2016. p. 519–524.

ISOLANI, P. H.; WICKBOLDT, J. A.; BOTH, C. B.; ROCHOL, J.; GRANVILLE, L. Z. Interactive monitoring, visualization, and configuration of openflow-based sdn. In: **2015 IFIP/IEEE International Symposium on Integrated Network Management (IM)**. [S.l.: s.n.], 2015. p. 207–215. ISSN 1573-0077.

JABAL, A. A.; DAVARI, M.; BERTINO, E.; MAKAYA, C.; CALO, S.; VERMA, D.; RUSSO, A.; WILLIAMS, C. Methods and tools for policy analysis. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 51, n. 6, p. 1–35, 2019.

JARRAYA, Y.; MADI, T.; DEBBABI, M. A survey and a layered taxonomy of software-defined networking. **IEEE communications surveys & tutorials**, IEEE, v. 16, n. 4, p. 1955–1980, 2014.

JIN, X.; LIU, H. H.; GANDHI, R.; KANDULA, S.; MAHAJAN, R.; ZHANG, M.; REXFORD, J.; WATTENHOFER, R. Dynamic scheduling of network updates. In: **Proceedings of the 2014 ACM Conference on SIGCOMM**. New York, NY, USA: ACM, 2014. (SIGCOMM '14), p. 539–550. ISBN 978-1-4503-2836-4. Available from Internet: <http://doi.acm.org/10.1145/2619239.2626307>.

KANG, M.; KANG, E.-Y.; HWANG, D.-Y.; KIM, B.-J.; NAM, K.-H.; SHIN, M.-K.; CHOI, J.-Y. Formal modeling and verification of sdn-openflow. In: IEEE. **2013 IEEE Sixth International Conference on Software Testing, Verification and Validation**. [S.l.], 2013. p. 481–482.

KANG, N.; REICH, J.; REXFORD, J.; WALKER, D. Policy transformation in software defined networks. In: ACM. **Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication**. [S.l.], 2012. p. 309–310.

KAZEMIAN, P.; CHANG, M.; ZENG, H.; VARGHESE, G.; MCKEOWN, N.; WHYTE, S. Real time network policy checking using header space analysis. In: **Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2013. (nsdi'13), p. 99–112. Available from Internet: <http://dl.acm.org/citation.cfm?id=2482626.2482638>.

KAZEMIAN, P.; CHANG, M.; ZENG, H.; VARGHESE, G.; MCKEOWN, N.; WHYTE, S. Real time network policy checking using header space analysis. In: **Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2013. (nsdi'13), p. 99–112. Available from Internet: <http://dl.acm.org/citation.cfm?id=2482626.2482638>.

KIM, H.; FEAMSTER, N. Improving network management with software defined networking. **Communications Magazine, IEEE**, v. 51, n. 2, p. 114–119, February 2013. ISSN 0163-6804.

KIM, H.; REICH, J.; GUPTA, A.; SHAHBAZ, M.; FEAMSTER, N.; CLARK, R. Kinetic: Verifiable dynamic network control. In: **Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation**. Berkeley, CA, USA: USENIX Association, 2015. (NSDI'15), p. 59–72. ISBN 978-1-931971-218. Available from Internet: <http://dl.acm.org/citation.cfm?id=2789770.2789775>.

KIRA, K.; RENDELL, L. A. A practical approach to feature selection. In: **Machine Learning Proceedings 1992**. [S.l.]: Elsevier, 1992. p. 249–256.

KOZAT, U. C.; LIANG, G.; KöKTEN, K. On diagnosis of forwarding plane via static forwarding rules in software defined networks. In: **IEEE INFOCOM 2014 - IEEE Conference on Computer Communications**. [S.l.: s.n.], 2014. p. 1716–1724. ISSN 0743-166X.

KUSHWAHA, S. S.; JOSHI, S.; SINGH, D.; KAUR, M.; LEE, H.-N. Ethereum smart contract analysis tools: A systematic review. **IEEE Access**, IEEE, 2022.

KUZNIAR, M.; PERESINI, P.; CANINI, M.; VENZANO, D.; KOSTIC, D. A soft way for openflow switch interoperability testing. In: **Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies**. New York, NY,

USA: ACM, 2012. (CoNEXT '12), p. 265–276. ISBN 978-1-4503-1775-7. Available from Internet: <http://doi.acm.org/10.1145/2413176.2413207>.

KWIATKOWSKA, M.; NORMAN, G.; PARKER, D. Probabilistic model checking and autonomy. **Annual Review of Control, Robotics, and Autonomous Systems**, Annual Reviews, v. 5, p. 385–410, 2022.

LEE, J.; KANG, J.-M.; PRAKASH, C.; BANERJEE, S.; TURNER, Y.; AKELLA, A.; CLARK, C.; MA, Y.; SHARMA, P.; ZHANG, Y. Network policy whiteboarding and composition. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 45, n. 4, p. 373–374, aug. 2015. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/2829988.2790039>.

LEE, S.; LEVANTI, K.; KIM, H. S. Network monitoring: Present and future. **Computer Networks**, Elsevier, v. 65, p. 84–98, 2014.

LEI, Y.; YU, L.; LIU, V.; XU, M. Printqueue: performance diagnosis via queue measurement in the data plane. In: **Proceedings of the ACM SIGCOMM 2022 Conference**. [S.l.: s.n.], 2022. p. 516–529.

LI, W.; ZHANG, X.-Y.; BAO, H.; WANG, Q.; LI, Z. Robust network traffic identification with graph matching. **Computer Networks**, Elsevier, v. 218, p. 109368, 2022.

LI, Y.; YIN, X.; WANG, Z.; YAO, J.; SHI, X.; WU, J.; ZHANG, H.; WANG, Q. A survey on network verification and testing with formal methods: Approaches and challenges. **IEEE Communications Surveys & Tutorials**, IEEE, v. 21, n. 1, p. 940–969, 2018.

LIMA, M.; ZARPELAO, B.; SAMPAIO, L.; RODRIGUES, J.; ABRAO, T.; PROENçA, M. Anomaly detection using baseline and k-means clustering. In: **Software, Telecommunications and Computer Networks (SoftCOM), 2010 International Conference on**. [S.l.: s.n.], 2010. p. 305–309.

LIU, H. H.; WU, X.; ZHANG, M.; YUAN, L.; WATTENHOFER, R.; MALTZ, D. zupdate: Updating data center networks with zero loss. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 43, n. 4, p. 411–422, aug. 2013. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/2534169.2486005>.

LIU, J.; HALLAHAN, W.; SCHLESINGER, C.; SHARIF, M.; LEE, J.; SOULÉ, R.; WANG, H.; CAŞCAVAL, C.; MCKEOWN, N.; FOSTER, N. p4v: Practical verification for programmable data planes. 2018.

LIU, Y.; GRIGORYAN, G.; LI, J.; SUN, G.; TAUBER, T. Veritable: Fast equivalence verification of multiple large forwarding tables. **Computer Networks**, Elsevier, v. 168, p. 106981, 2020.

LOPES, N. P.; BJØRNER, N.; GODEFROID, P.; JAYARAMAN, K.; VARGHESE, G. Checking beliefs in dynamic networks. In: **12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)**. Oakland, CA: USENIX Association, 2015. p. 499–512. ISBN 978-1-931971-218. Available from Internet: <https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/lopes>.

LUCAS, S. The origins of the halting problem. **Journal of Logical and Algebraic Methods in Programming**, Elsevier, v. 121, p. 100687, 2021.

LUCKCUCK, M.; FARRELL, M.; DENNIS, L. A.; DIXON, C.; FISHER, M. Formal specification and verification of autonomous robotic systems: A survey. **ACM Computing Surveys (CSUR)**, ACM New York, NY, USA, v. 52, n. 5, p. 1–41, 2019.

MAI, H.; KHURSHID, A.; AGARWAL, R.; CAESAR, M.; GODFREY, P. B.; KING, S. T. Debugging the data plane with anteater. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 41, n. 4, p. 290–301, aug. 2011. ISSN 0146-4833.

MAJUMDAR, R.; TETALI, S. D.; WANG, Z. Kuai: A model checker for software-defined networks. In: **Formal Methods in Computer-Aided Design (FMCAD), 2014**. [S.l.: s.n.], 2014. p. 163–170.

MAJUMDAR, R.; TETALI, S. D.; WANG, Z. Kuai: A model checker for software-defined networks. In: **Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design**. Austin, TX: FMCAD Inc, 2014. (FMCAD '14), p. 27:163–27:170. ISBN 978-0-9835678-4-4. Available from Internet: <http://dl.acm.org/citation.cfm?id=2682923.2682953>.

MATTEIS, T. D.; SECCI, S.; ROSSI, D. Automated network verification and testing: A survey. **IEEE Communications Surveys & Tutorials**, IEEE, v. 23, n. 1, p. 57–89, 2021.

MATTHEWS, O.; BINGHAM, J.; SORIN, D. J. Verifiable hierarchical protocols with network invariants on parametric systems. In: **Proceedings of the 16th Conference on Formal Methods in Computer-Aided Design**. Austin, TX: FMCAD Inc, 2016. (FMCAD '16), p. 101–108. ISBN 978-0-9835678-6-8. Available from Internet: <http://dl.acm.org/citation.cfm?id=3077629.3077650>.

MCCLURG, J.; HOJJAT, H.; CERNý, P.; FOSTER, N. Efficient synthesis of network updates. **SIGPLAN Not.**, ACM, New York, NY, USA, v. 50, n. 6, p. 196–207, jun. 2015. ISSN 0362-1340. Available from Internet: <http://doi.acm.org/10.1145/2813885.2737980>.

NAWAZ, M. S.; MALIK, M.; LI, Y.; SUN, M.; LALI, M. A survey on theorem provers in formal methods. **arXiv preprint arXiv:1912.03028**, 2019.

NGUYEN, X.-N.; SAUCEZ, D.; BARAKAT, C.; TURLETTI, T. Optimizing rules placement in openflow networks: Trading routing for better efficiency. In: **Proceedings of the Third Workshop on Hot Topics in Software Defined Networking**. New York, NY, USA: ACM, 2014. (HotSDN '14), p. 127–132. ISBN 978-1-4503-2989-7. Available from Internet: <http://doi.acm.org/10.1145/2620728.2620753>.

OSMAN, N.; ROBERTSON, D. Dynamic verification of trust in distributed open systems. In: **IJCAI**. [S.l.: s.n.], 2007. p. 1440–1445.

PADON, O.; IMMERMAN, N.; KARBYSHEV, A.; LAHAV, O.; SAGIV, M.; SHOHAM, S. Decentralizing sdn policies. In: **Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages**. New York, NY, USA: ACM, 2015. (POPL '15), p. 663–676. ISBN 978-1-4503-3300-9. Available from Internet: <http://doi.acm.org/10.1145/2676726.2676990>.

PANDA, A.; LAHAV, O.; ARGYRAKI, K. J.; SAGIV, M.; SHENKER, S. Verifying isolation properties in the presence of middleboxes. **CoRR**, abs/1409.7687, 2014. Available from Internet: <http://arxiv.org/abs/1409.7687>.

PANDA, A.; LAHAV, O.; ARGYRAKI, K. J.; SAGIV, M.; SHENKER, S. Verifying reachability in networks with mutable datapaths. **CoRR**, abs/1607.00991, 2016. Available from Internet: <http://arxiv.org/abs/1607.00991>.

PARK, S.; KWON, G. Avoidance of state explosion using dependency analysis in model checking control flow model. In: SPRINGER. **International Conference on Computational Science and Its Applications**. [S.l.], 2006. p. 905–911.

PI, R.; CAI, Y.; LI, Y.; CAO, Y. Machine learning based on bayes networks to predict the cascading failure propagation. **IEEE Access**, IEEE, v. 6, p. 44815–44823, 2018.

PRABHU, S.; CHOU, K. Y.; KHERADMAND, A.; GODFREY, B.; CAESAR, M. Plankton: Scalable network configuration verification through model checking. In: **17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)**. [S.l.: s.n.], 2020. p. 953–967.

PRAKASH, C.; LEE, J.; TURNER, Y.; KANG, J.-M.; AKELLA, A.; BANERJEE, S.; CLARK, C.; MA, Y.; SHARMA, P.; ZHANG, Y. Pga: Using graphs to express and automatically reconcile network policies. In: **Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication**. New York, NY, USA: ACM, 2015. (SIGCOMM '15), p. 29–42. ISBN 978-1-4503-3542-3. Available from Internet: <http://doi.acm.org/10.1145/2785956.2787506>.

PRASAD, M. R.; BIERE, A.; GUPTA, A. A survey of recent advances in sat-based formal verification. **International Journal on Software Tools for Technology Transfer**, Springer, v. 7, n. 2, p. 156–173, 2005.

QADIR, J.; HASAN, O. Applying formal methods to networking: theory, techniques, and applications. **IEEE Communications Surveys & Tutorials**, IEEE, v. 17, n. 1, p. 256–291, 2014.

QUEILLE, J.-P.; SIFAKIS, J. Specification and verification of concurrent systems in cesar. In: SPRINGER. **International Symposium on programming**. [S.l.], 1982. p. 337–351.

RAMAN, V.; RAVIKUMAR, B.; RAO, S. S. A simplified np-complete maxsat problem. **Information Processing Letters**, Elsevier, v. 65, n. 1, p. 1–6, 1998.

REICH, J.; MONSANTO, C.; FOSTER, N.; REXFORD, J.; WALKER, D. Modular sdn programming with pyretic. **Technical Reprot of USENIX**, v. 30, 2013.

REITBLATT, M.; FOSTER, N.; REXFORD, J.; SCHLESINGER, C.; WALKER, D. Abstractions for network update. **ACM SIGCOMM Computer Communication Review**, ACM New York, NY, USA, v. 42, n. 4, p. 323–334, 2012.

REITER, R. A semantically guided deductive system for automatic theorem proving. **IEEE Transactions on Computers**, IEEE, n. 4, p. 328–334, 1976.

ROTHENBERG, B.-C.; DIETSCH, D.; HEIZMANN, M. Incremental verification using trace abstraction. In: SPRINGER. **International Static Analysis Symposium**. [S.l.], 2018. p. 364–382.

ROTSOS, C.; SARRAR, N.; UHLIG, S.; SHERWOOD, R.; MOORE, A. W. Oflops: An open framework for openflow switch evaluation. In: **Proceedings of the 13th International Conference on Passive and Active Measurement**. Berlin, Heidelberg: Springer-Verlag, 2012. (PAM'12), p. 85–95. ISBN 978-3-642-28536-3.

ROUMANE, A.; KECHAR, B. A statistical model checking approach to analyse the random access protocol. **International Journal of Wireless and Mobile Computing**, Inderscience Publishers (IEL), v. 23, n. 3-4, p. 338–349, 2022.

SABUR, A.; CHOWDHARY, A.; HUANG, D.; ALSHAMRANI, A. Toward scalable graph-based security analysis for cloud networks. **Computer Networks**, Elsevier, v. 206, p. 108795, 2022.

SÁNCHEZ, C.; SCHNEIDER, G.; AHRENDT, W.; BARTOCCI, E.; BIANCULLI, D.; COLOMBO, C.; FALCONE, Y.; FRANCALANZA, A.; KRSTIĆ, S.; LOURENÇO, J. M. et al. A survey of challenges for runtime verification from advanced application domains (beyond software). **Formal Methods in System Design**, Springer, v. 54, p. 279–335, 2019.

SANGER, R.; LUCKIE, M.; NELSON, R. Towards transforming openflow rulesets to fit fixed-function pipelines. In: **Proceedings of the Symposium on SDN Research**. [S.l.: s.n.], 2020. p. 123–134.

SETHI, D.; NARAYANA, S.; MALIK, S. Abstractions for model checking sdn controllers. In: **Formal Methods in Computer-Aided Design (FMCAD), 2013**. [S.l.: s.n.], 2013. p. 145–148.

SETHI, D.; NARAYANA, S.; MALIK, S. Abstractions for model checking sdn controllers. In: CITESEER. **FMCAD**. [S.l.], 2013. p. 145–148.

SHIN, M. K.; CHOI, Y.; KWAK, H. H.; PACK, S.; KANG, M.; CHOI, J. Y. Verification for nfv-enabled network services. In: **Information and Communication Technology Convergence (ICTC), 2015 International Conference on**. [S.l.: s.n.], 2015. p. 810–815.

SHIN, M. K.; CHOI, Y.; KWAK, H. H.; PACK, S.; KANG, M.; CHOI, J. Y. Verification for nfv-enabled network services. In: **2015 International Conference on Information and Communication Technology Convergence (ICTC)**. [S.l.: s.n.], 2015. p. 810–815.

SHIN, M.-k.; NAM, K.; PACK, S.; LEE, S.; KRISHNAN, R.; KIM, T. **Internet-Draft - Verification of NFV Services : Problem Statement and Challenges** . [S.l.], 2015.

SHUKLA, N.; PANDEY, M.; SRIVASTAVA, S. Formal modeling and verification of software-defined networks: A survey. **International Journal of Network Management**, Wiley Online Library, v. 29, n. 5, p. e2082, 2019.

SILVA, A. d.; SMITH, P.; MAUTHE, A.; SCHAEFFER-FILHO, A. Resilience support in software-defined networking: a survey. In: **Computer Networks**. [s.n.], 2015. Available from Internet: <http://dx.doi.org/10.1016/j.comnet.2015.09.12>.

SILVA, A. S. d.; MACHADO, C. C.; BISOL, R. V.; GRANVILLE, L. Z.; SCHAEFFER-FILHO, A. Identification and selection of flow features for accurate traffic classification in sdn. In: **2015 IEEE 14th International Symposium on Network Computing and Applications**. [S.l.: s.n.], 2015. p. 134–141.

SILVA, A. S. da; WICKBOLDT, J. A.; GRANVILLE, L. Z.; SCHAEFFER-FILHO, A. Atlantic: A framework for anomaly traffic detection, classification, and mitigation in sdn. In: **NOMS 2016 - 2016 IEEE/IFIP Network Operations and Management Symposium**. [S.l.: s.n.], 2016. p. 27–35. ISSN 2374-9709.

SKOWYRA, R.; LAPETS, A.; BESTAVROS, A.; KFOURY, A. A verification platform for sdn-enabled applications. In: **2014 IEEE International Conference on Cloud Engineering**. [S.l.: s.n.], 2014. p. 337–342.

SOOS, M.; MEEL, K. S. Bird: Engineering an efficient cnf-xor sat solver and its applications to approximate model counting. In: **Proceedings of the AAAI Conference on Artificial Intelligence**. [S.l.: s.n.], 2019. v. 33, p. 1592–1599.

SOULÉ, R.; BASU, S.; MARANDI, P. J.; PEDONE, F.; KLEINBERG, R.; SIRER, E. G.; FOSTER, N. Merlin: A language for provisioning network resources. In: **Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies**. New York, NY, USA: ACM, 2014. (CoNEXT '14), p. 213–226. ISBN 978-1-4503-3279-8. Available from Internet: <http://doi.acm.org/10.1145/2674005.2674989>.

STEINDER, M. łgorzata; SETHI, A. S. A survey of fault localization techniques in computer networks. **Science of computer programming**, Elsevier, v. 53, n. 2, p. 165–194, 2004.

STERBENZ, J. P. G.; HUTCHISON, D.; CETINKAYA, E. K.; JABBAR, A.; ROHRER, J. P.; SCHöLLER, M.; SMITH, P. Resilience and survivability in communication networks: Strategies, principles, and survey of disciplines. **Comput. Netw.**, Elsevier North-Holland, Inc., New York, NY, USA, v. 54, n. 8, p. 1245–1265, jun. 2010. ISSN 1389-1286.

STERN, U.; DILL, D. L. Automatic verification of the sci cache coherence protocol. In: SPRINGER. **Advanced Research Working Conference on Correct Hardware Design and Verification Methods**. [S.l.], 1995. p. 21–34.

STOENESCU, R.; DUMITRESCU, D.; POPOVICI, M.; NEGREANU, L.; RAICIU, C. Debugging p4 programs with vera. In: **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. [S.l.: s.n.], 2018. p. 518–532.

TESTA, A.; CORONATO, A.; CINQUE, M.; AUGUSTO, J. C. Static verification of wireless sensor networks with formal methods. In: IEEE. **2012 Eighth International Conference on Signal Image Technology and Internet Based Systems**. [S.l.], 2012. p. 587–594.

THOMBARE, B. M.; SONI, D. R. Prevention of sql injection attack by using black box testing. In: **23rd International Conference on Distributed Computing and Networking**. [S.l.: s.n.], 2022. p. 266–272.

UJJAN, R. M. A.; PERVEZ, Z.; DAHAL, K.; BASHIR, A. K.; MUMTAZ, R.; GONZÁLEZ, J. Towards sflow and adaptive polling sampling for deep learning based ddos detection in sdn. **Future Generation Computer Systems**, Elsevier, v. 111, p. 763–779, 2020.

VELNER, Y.; ALPERNAS, K.; PANDA, A.; RABINOVICH, A.; SAGIV, M.; SHENKER, S.; SHOHAM, S. Some complexity results for stateful network verification. In: **Proceedings of the 22Nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636**. New York, NY, USA: Springer-Verlag New York, Inc., 2016. p. 811–830. ISBN 978-3-662-49673-2.

VOELLMY, A.; WANG, J.; YANG, Y. R.; FORD, B.; HUDAK, P. Maple: Simplifying sdn programming using algorithmic policies. **SIGCOMM Comput. Commun. Rev.**, ACM, New York, NY, USA, v. 43, n. 4, p. 87–98, aug. 2013. ISSN 0146-4833. Available from Internet: <http://doi.acm.org/10.1145/2534169.2486030>.

WICKBOLDT, J. A.; JESUS, W. P. D.; ISOLANI, P. H.; BOTH, C. B.; ROCHOL, J.; GRANVILLE, L. Z. Software-defined networking: management requirements and challenges. **IEEE Communications Magazine**, v. 53, n. 1, p. 278–285, January 2015. ISSN 0163-6804.

WICKBOLDT, J. A.; JESUS, W. P. D.; ISOLANI, P. H.; BOTH, C. B.; ROCHOL, J.; GRANVILLE, L. Z. Software-defined networking: management requirements and challenges. **IEEE Communications Magazine**, v. 53, n. 1, p. 278–285, January 2015. ISSN 0163-6804.

WUNDSAM, A.; LEVIN, D.; SEETHARAMAN, S.; FELDMANN, A. Ofrewind: Enabling record and replay troubleshooting for networks. In: **Proceedings of the 2011 USENIX Conference on USENIX Annual Technical Conference**. Berkeley, CA, USA: USENIX Association, 2011. (USENIXATC'11), p. 29–29. Available from Internet: <http://dl.acm.org/citation.cfm?id=2002181.2002210>.

XIE, G. G.; ZHAN, J.; MALTZ, D. A.; ZHANG, H.; GREENBERG, A.; HJALMTYSSON, G.; REXFORD, J. On static reachability analysis of ip networks. In: **Proceedings IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies.** [S.l.: s.n.], 2005. v. 3, p. 2170–2183 vol. 3. ISSN 0743-166X.

YAKUWA, Y.; TOMIZAWA, N.; TONOUCHI, T. Efficient model checking of openflow networks using sdpor-ds. In: **The 16th Asia-Pacific Network Operations and Management Symposium**. [S.l.: s.n.], 2014. p. 1–6.

YANG, B.; BRYANT, R. E.; O'HALLARON, D. R.; BIERE, A.; COUDERT, O.; JANSSEN, G.; RANJAN, R. K.; SOMENZI, F. A performance study of bdd-based model checking. In: SPRINGER. **International Conference on Formal Methods in Computer-Aided Design**. [S.l.], 1998. p. 255–289.

YANG, H.; LAM, S. S. Real-time verification of network properties using atomic predicates. In: **2013 21st IEEE International Conference on Network Protocols (ICNP)**. [S.l.: s.n.], 2013. p. 1–11. ISSN 1092-1648.

YANG, L.; NG, B.; SEAH, W. K.; GROVES, L.; SINGH, D. A survey on network forwarding in software-defined networking. **Journal of Network and Computer Applications**, Elsevier, p. 102947, 2020.

ZEGHLACHE, D. **NFV Service Chaining Challenges**. 2016. IEEE Softwarization eNewsletter.

ZHANG, P.; GEMBER-JACOBSON, A.; ZUO, Y.; HUANG, Y.; LIU, X.; LI, H. Differential network analysis. In: **USENIX NSDI**. [S.l.: s.n.], 2022.

ZHANG, S.; MALIK, S. Sat based verification of network data planes. In: ____. **Automated Technology for Verification and Analysis: 11th International Symposium, ATVA 2013, Hanoi, Vietnam, October 15-18, 2013. Proceedings**. Cham: Springer International Publishing, 2013. p. 496–505.

7 APPENDIX - RESUMO EXPANDIDO EM PORTUGUÊS

Verificação de propriedades de rede, um conceito relacionado ao uso de técnicas para determinar se um componente de rede (seja dispositivos de encaminhamento simples, como switches ou roteadores, ou caixas intermediárias - físicas ou virtualizadas) respeita um conjunto de propriedades predefinidas, ganhou importância nos ultimos anos. A principal razão é o aspecto crítico da verificação dessas propriedades predefinidas: um componente de rede deve executar ações corretas e seu funcionamento correto só é alcançável se respeitar sua especificação/propriedades. Além disso, o crescimento contínuo da complexidade dos componentes da rede aumenta o número de possíveis estados de execução, aumentando assim o número de propriedades necessárias para verificar se um desses estados está correto ou não. Como consequência, quando comparados aos primeiros componentes de rede que eram simples e dedicados a realizar uma única ação, como encaminhar pacotes ou verificar erros, hoje em dia os dispositivos são responsáveis por várias ações simultâneas como encaminhar pacotes, coletar estatísticas, executar rotinas de segurança, etc. Além disso, com a disseminação dos softwares nas redes de computadores e principalmente com a evolução dos requisitos do sistema, aumentou também o número de propriedades para verificar se um dispositivo está funcionando corretamente (BIRKNER et al., 2020). Como esses novos componentes de software estão frequentemente relacionados a funções de segurança, desempenho e tolerância a falhas, um sistema que automatize e otimize o processo de verificação dessas propriedades é essencial para o futuro das redes de computadores.

7.1 Contexto

Infelizmente, o ecossistema de redes de computadores é dinâmico o suficiente para evoluir tão rápido que qualquer sistema de verificação logo se torna obsoleto (AVIZIENIS et al., 2004; DELMAS et al., 2020). Quando um componente de rede não pode garantir todas as suas propriedades, dizemos que ocorre uma violação de propriedade. A violação de propriedade pode ter várias causas, mas frequentemente ocorre como resultado de configuração incorreta (LUCKCUCK et al., 2019). É difícil encontrar causas de configuração incorreta porque (i) o estado interno dos componentes da rede é freqüentemente desconhecido; (ii) alcançar o estado global da rede é uma tarefa desafiadora porque precisa considerar todos os componentes da rede e (iii) erros humanos estão sempre presentes e inserem equívocos na rede, comprometendo até mesmo propriedades bem conhecidas.

Software-Defined Networking (SDN) (FEAMSTER; REXFORD; ZEGURA, 2013) e Network Functions Virtualization (NFV) (ZEGHLACHE, 2016) são exemplos de paradigmas de rede fortemente dependentes de componentes de software. Ao utilizar software em vez de componentes específicos de hardware, SDN e NFV promovem maior flexibilidade no projeto da rede e facilitam a programabilidade dos equipamentos de rede. Por um lado, o uso crescente de

componentes de software para executar tarefas de rede traz flexibilidade aos administradores de rede no que diz respeito ao gerenciamento e monitoramento de componentes de rede (WICKBOLDT et al., 2015a). Por outro lado, há um preço para isso: todos esses componentes de rede devem ser configurados para realizar suas tarefas e cooperar com outros componentes de software. É um desafio garantir uma configuração ideal para excluir futuras violações de propriedade ao usar esses componentes de software em conjunto (SILVA et al., 2015a; BIRKNER et al., 2020). Violações de propriedade típicas são criadas pelo gerenciamento defeituoso das entradas de encaminhamento, criando a possibilidade de os pacotes se comportarem incorretamente (LIU et al., 2020). A origem dessas violações pode ser a implementação de roteamento ou ferramentas de configuração que não produzem a saída desejada para o plano de dados. Além disso, a falta de interoperabilidade entre esses aplicativos também pode produzir configuração errônea e conflitos (YANG et al., 2020).

Para lidar com esses desafios, o teste de rede (BARI et al., 2013) é freqüentemente usado para verificar se um componente respeita uma determinada propriedade e, consequentemente, executa suas ações corretamente. O teste de rede pode ser dividido em três grupos: (i) teste de caixa preta quando o estado interno do componente de rede não é conhecido e as amostras de teste podem determinar quais entradas e saídas o componente reconhece; (ii) teste de caixa branca quando o estado interno do componente é conhecido e é possível ver sua implementação e testar propriedades específicas considerando uma visão de alto nível de seu funcionamento; e (iii) teste baseado em defeitos, uma técnica que gera casos de teste com base em assinaturas de defeitos em vez de usar os testes de cobertura tradicionais. Há uma variedade de desafios relacionados ao teste de rede, variando de erros humanos (inserção de configuração incorreta) a atividades maliciosas (AVIZIENIS et al., 2004). Erros humanos freqüentemente aumentam o escopo do processo de teste porque ele precisa cobrir inconsistências em arquivos de configuração e também atividades maliciosas, incluindo anomalias de rede relacionadas a ataques humanos, como IP spoofing (ABDULQADDER et al., 2020).

Consequentemente, as técnicas de teste de rede podem ser combinadas com outras técnicas para verificar propriedades de rede mais complexas. Um exemplo são as técnicas de verificação formal que podem ser utilizadas para detectar erros de configuração e conflitos de política, bem como para verificar propriedades mais tradicionais como acessibilidade e isolamento. No entanto, este campo de pesquisa apresenta muitos desafios, como a capacidade de capturar todos os estados da rede. Como consequência, o resultado não é ótimo em geral (AL-SHAER; AL-HAJ, 2010a), uma realidade que incentiva técnicas alternativas a essas limitações, como a observação de ações do dispositivo em vez de modelar seus possíveis estados, usando técnicas de monitoramento de rede, como amostragem de fluxo .

7.2 Motivação

Uma tendência clara no contexto das redes de computadores é o uso de software como alternativa ao uso de hardware especializado. O benefício dessa tendência é claro: os administradores de rede podem gerenciar e controlar os aspectos da rede de maneira mais flexível, modular e razoável. Concomitantemente, um desafio neste contexto também é claro: como determinar se tudo está acontecendo corretamente em uma rede de computadores onde o software, possivelmente com bugs, está muito presente? Uma maneira de responder a essa pergunta depende de testes. No entanto, para testar uma rede de computadores, precisamos responder a três perguntas básicas antes:

1) Quais componentes compõem o que entendemos por rede de computadores?

2) Quais propriedades nesta rede de computadores são consideradas desejáveis?

3) Quais propriedades nesta rede de computadores são indesejáveis?

Respondemos à questão 1) com a seguinte definição: Consideramos que uma rede é constituída por vários *componentes de rede* e esta é a intuição mais básica deste estudo. Um *componente de rede* compreende todos os elementos que interagem diretamente com os pacotes de rede. Por exemplo, switches, roteadores, middleboxes, links de comunicação e hosts são exemplos de *componentes de rede*. Um usuário de rede sentado na frente de seu computador não é um exemplo de *componente de rede* porque mesmo no caso de ele interagir com a rede como um todo, ele não manipula pacotes diretamente.

No caso da questão 2), é importante definir que propriedades são um conjunto de características de um *componente de rede*. Nosso interesse em estudar *componentes de rede* está focado em testar propriedades que eles podem assumir ou não durante seu ciclo de vida. Um subconjunto dessas propriedades é desejável, outros não.

Um *componente de rede* pode assumir vários estados de execução e cada um deles é resultado de um conjunto de influências capazes de obrigá-lo a passar de um estado para outro. Uma influência interna altera o estado de um *componente de rede* considerando apenas aspectos intrínsecos da estrutura do componente. Por exemplo, um erro de implementação em um switch surge devido a um erro em seu projeto inicial. Uma influência externa altera o estado de um *componente de rede* considerando aspectos externos a ele e sua interação com o ecossistema no qual ele executa. Por exemplo, um erro em um switch devido a uma queda de energia o força de um estado para outro abruptamente. Defendemos que o estudo das propriedades em um *componente de rede* precisa incluir influências externas e internas sobre este componente.

O conjunto de propriedades que surge quando apenas influências internas atuam sobre *componente de rede* é chamado de propriedades individuais. Essas propriedades por si só são difíceis de verificar porque muitas vezes não temos uma visão precisa da configuração interna de cada *componente de rede*. Concomitantemente, o conjunto de propriedades que surgem quando influências externas atuam sobre um *componente de rede* é chamado de propriedades globais.

Propriedades globais são difíceis de verificar porque podem incluir a verificação de várias propriedades individuais de cada *componente de rede* envolvido no processo.

Considerando essas definições, uma propriedade desejável é o conjunto de propriedades individuais e globais de um *componente de rede* que não viole os requisitos de um administrador de rede para a rede.

Para a questão final 3) é importante entender que o conjunto de propriedades desejáveis e indesejáveis *componentes de rede* são dependentes de cada contexto. As propriedades desejáveis dependem dos requisitos de cada rede e do que cada administrador considera importante. No entanto, é consenso que deve ser garantida a ausência de qualquer propriedade que constitua uma ameaça ao correto funcionamento da rede. No entanto, essa discussão é longa. Abordamo-lo na próxima subseção.

Em suma, entendemos que determinamos se tudo está acontecendo corretamente em uma rede de computadores com soluções de software quando entendemos seus *componentes de rede* e podemos verificar suas propriedades individuais e globais desejáveis.

Quando uma propriedade desejável não é respeitada por um *componente de rede* dizemos que ocorre uma violação de propriedade. Duas estratégias principais para testar se uma propriedade está sendo satisfeita por um *componente de rede* são representadas por monitoramento e verificação formal. Técnicas de monitoramento podem verificar uma violação de propriedade em *componentes de rede* porque observam os efeitos que esta violação produz nos dados monitorados. Uma estratégia de monitoramento perfeita, em teoria, analisaria todos os pacotes e todos os estados o tempo todo. No entanto, ele só conseguiu capturar uma *violação de propriedade* depois que ela ocorreu. Na prática, não é possível salvar todos os pacotes e considerar todos os estados de cada componente da rede devido a restrições de escalabilidade. Assim, é necessário *reduzir este espaço de busca* e conseqüentemente não obter um sistema de monitoramento ótimo. Concomitantemente, a verificação formal nos permite modelar propriedades que não dependem da observação dos dados de execução dos *componentes de rede*. No entanto, mesmo um verificador de verificação formal perfeito no contexto de um dado *componente de rede* não poderia testar uma propriedade que dependa da interação em tempo real com outros componentes. Essas limitações nos mostram que não é possível obter um motor de teste completo usando apenas essas estratégias.

No entanto, pretendemos melhorar esse processo de teste para um design aprimorado. Uma maneira mais eficaz de testar propriedades em *componentes de rede* combinaria monitoramento e vantagens de verificação formal e tentaria evitar suas limitações. Podemos combiná-los sequencialmente ou simultaneamente. O caso concorrente executa uma estratégia de monitoramento de um lado e verificação formal do outro lado. É semelhante a executá-los individualmente. Consequentemente, vantagens reduzidas podem ser obtidas nesta combinação. A outra alternativa é combiná-los sequencialmente em duas camadas onde uma pode melhorar o resultado da outra. Uma camada de monitoramento pode verificar violações de propriedade em *componentes de rede* considerando uma perspectiva quase em tempo real. Uma camada de ver-

ificação formal pode verificar as propriedades que provavelmente ocorrerão - mas ainda não ocorreram.

É crucial enfatizar aqui que não poderia ser a combinação inversa, *ou seja*, primeiro uma camada de verificação formal e depois uma camada de monitoramento. A principal razão é que a camada de monitoramento estando sobre uma camada de verificação sempre encontrará algo que não está modelado no modelo formal. Isso acontecerá porque um modelo de verificação formal não pode prever violações de propriedade resultantes de acidentes. Consequentemente, a camada de monitoramento neste caso será constantemente não sincronizada com a camada de verificação formal. Por isso, entendemos que uma camada de monitoramento vindo primeiro serve como fonte para gerar uma camada de verificação de acordo com o que acontece na rede podendo estender isso com análises mais complexas.

7.3 Contribuição

Embora o uso de SDN e NFV promova mais flexibilidade no projeto da rede e facilita a programabilidade dos equipamentos de rede, isso também dificulta a certifique-se de que a configuração de rede esteja livre de violações de propriedade. Propomos aqui uma solução que explora a combinação de dois estudos baseados em monitoramento de rede e teste de rede com base na verificação formal. A primeira contribuição é representada por uma camada de monitoramento e seu protótipo (denominado ARMOR) e pretende mostrar uma arquitetura abrangente para gerenciar o diagnóstico de violação de propriedade nosis em redes baseadas em software. Usando ferramentas de amostragem de monitoramento, como SNMP, REST API e sFlow, garantimos a flexibilidade, precisão e automatização da nossa solução. Nosso a pesquisa se baseia em técnicas de aprendizado de máquina, como árvores de decisão (AL-SHAER; AL-HAJ, 2010a). Nosso objetivo com isso é alcançar alta precisão na classificação e automatização de dados do processo de detecção e correção.

A segunda contribuição é representada por uma camada de verificação formal capaz de realizar testes de trabalho e seu protótipo (denominado NetWords). Esta proposta de tese apresenta uma nova abordagem abordagem para representar redes com base na gramática regular que pode ser genérica o suficiente para representar reenviar toda a rede e específico o suficiente para modelar dispositivos únicos. Nosso modelo usa como fonte de informação as informações das mensagens que entram e saem dos dispositivos. Desta forma, é não é necessário conhecer o estado interno de um dispositivo para modelar. Nosso modelo os infere observando ção. Utilizando as ações realizadas pelos dispositivos de rede, é possível gerar uma descrição da rede muito semelhante à gramática. Esta gramática representa os comandos de rede mais usados caminhos de comunicação e representa um modelo capaz de verificar a rede global mais conhecida propriedades: acessibilidade, isolamento, liberdade de loop, um buraco negro e questões específicas relacionadas vícios, como uma falha de configuração interna. Além disso,

analisamos os cálculos computacionais necessários recursos para a implementação da nossa gramática.

Nossos resultados sugerem as seguintes contribuições: 1. uma estrutura de monitoramento completa capaz de detectar, classificar e remediar violações de rede ções; 2. um mecanismo de teste de rede completo capaz de modelar, analisar e testar invariantes de rede e propriedades usando gramáticas;