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

GABRIEL VASSOLER

VERMONT: An In-band Telemetry-Based Approach for Live Network Property Verification

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

Advisor: Prof. Dr. Luciano Paschoal Gaspary

Porto Alegre
April 2023

CIP — CATALOGING-IN-PUBLICATION

Vassoler, Gabriel

VERMONT: An In-band Telemetry-Based Approach for Live Network Property Verification / Gabriel Vassoler. – Porto Alegre: PPGC da UFRGS, 2023.

62 f.: il.

Thesis (Master) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR–RS, 2023. Advisor: Luciano Paschoal Gaspary.

1. Network Verification. 2. In-Band Network Telemetry. 3. On-Demand Verification. 4. Production Traffic. I. Paschoal Gaspary, Luciano. II. Título.

You are protected, in short, by your ability to love!

— ALBUS DUMBLEDORE

AGRADECIMENTOS

Destoando de maior parte dessa dissertação, os agradecimentos que realizo estão em português. Na minha mais humilde opinião, não existe forma mais verdadeira de expressar gratidão do que o fazer em sua língua materna.

Primeiramente agradeço ao Professor Dr. Luciano Paschoal Gaspary, por sua paciência, conselhos, amizade, e suporte incondicional. Durante todo o processo de desenvolvimento e escrita dessa dissertação diversas dificuldades cruzaram meu caminho, e foi graças a ele que hoje estou aqui. Foi uma honra estudar sob sua supervisão, e me considero uma pessoa de muita sorte por tê-lo feito. És uma pessoa muito gentil, e, mais uma vez, agradeço ao senhor por tudo.

Também agradeço **profundamente** ao Dr. Jonatas Adilson Marques, que apesar de não ser formalmente co-orientador dessa dissertação, o considero como tal. Agradeço sua paciência, disponibilidade, atenciosidade, gentileza e suporte. Se essa dissertação existe hoje, foi em grande parte por seus conselhos, orientação e esforço. Muito obrigado.

Agradeço a todos os meus amigos, que estiveram ao meu lado durante todo o processo de desenvolvimento deste trabalho e jamais deixaram de expressar seu amor por mim. Muito obrigado pela calma, pelas risadas, pelas conversas, pela implicância e pelas brincadeiras durante todo esse tempo. Eu amo muito todos vocês, e agradeço todo dia por encontrar pessoas tão maravilhosas em meu caminho.

Finalmente (e mais importante), agradeço e dedico esse trabalho aos meus pais, Celso e Elení, minha irmã e minha sobrinha/afilhada, Melize e Raíssa, e aos integrantes de minha família do coração, Amanda, Edivar, Eliseu, Eliane, Emanuelli, **Fernanda** e Viviana. Eu não seria nada sem vocês, e só tenho a agradecer por absolutamente tudo. Eu amo muito todos vocês.

ABSTRACT

The verification of network properties is often an exhaustive and time-consuming effort. The number of configurations needed to be analyzed by static verification increases as the networks grow larger, and the processing time consumed becomes prohibitive. Equally important, existing approaches fall short of detecting violations in dynamic environments. While the field of static verification has received significant attention in the last few years, few research efforts have been made to verify networks in production time. Capitalizing on the emergence of programmable data planes, in this thesis, we propose VERMONT, an In-Band Network Telemetry verification approach that continuously verifies network properties as the state of the networks changes. The key contribution of our work is an in-network system capable of continuously collecting the metadata from the network to verify properties in real-time. By efficiently retrieving only the necessary information from the network, VERMONT can accurately and quickly reason whether a set of properties is being held or not at a given time within the network. We implemented VERMONT, evaluated its performance using realistic settings, and compared it with a state-of-the-art approach. The results show that the proposed solution is technically feasible and performs at least one order of magnitude faster than a static verification counterpart. We also provide evidence that VERMONT incurs a very low resource usage footprint considering its application in several real-world networks.

Keywords: Network Verification. In-Band Network Telemetry. On-Demand Verification. Production Traffic.

VERMONT: An In-band Telemetry-Based Approach for Live Network Property Verification

RESUMO

A verificação de propriedades de rede geralmente representa um esforço exaustivo e demorado. O número de configurações que precisam ser analisadas pela verificação estática aumenta à medida que as redes crescem, e o tempo de processamento consumido torna-se proibitivo. Igualmente importante, as abordagens existentes não conseguem detectar violações em ambientes dinâmicos. Embora o campo de verificação estática tenha recebido atenção significativa nos últimos anos, poucos esforços de pesquisa foram feitos para verificar redes em tempo de "execução". Aproveitando o surgimento de planos de dados programáveis, nesta dissertação propomos VERMONT, uma abordagem de verificação baseada em telemetria de rede *in-band* que verifica continuamente propriedades à medida que o estado da rede muda. A principal contribuição do trabalho é um sistema capaz de coletar, continuamente, metadados da rede para verificar as propriedades em tempo real. Ao recuperar com eficiência apenas as informações necessárias, VERMONT pode "raciocinar" com precisão e rapidez se um conjunto de propriedades está sendo satisfeito ou não em um determinado momento. Implementamos VERMONT, avaliamos seu desempenho usando configurações realistas e a comparamos com uma abordagem de última geração. Os resultados mostram que a solução proposta é tecnicamente viável e executa pelo menos uma ordem de grandeza mais rápido do que uma contraparte de verificação estática. Também fornecemos evidências de que VERMONT incorre em uma utilização de recursos muito baixa, considerando sua aplicação em várias redes do mundo real.

Palavras-chave: Verificação de redes. Telemetria em rede. Verificação sob demanda. Tráfego de produção.

LIST OF ABBREVIATIONS AND ACRONYMS

API	Application Programming Interface
BMv2	Behavioral Model version 2
CPU	Central Processing Unit
DSL	Domain Specific Language
FSM	Finite State Machine
GHz	Giga Hertz
GUI	Graphical User Interface
ID	Identification
INT	In-band Network Telemetry
IP	Internet Protocol
MTU	Maximum Transmission Unit
OAM	Operations, Administration and Maintenance
P4	Programming Protocol-independent Packet Processors
PDP	Programmable Data Planes
PISA	Protocol Independent Switch Architecture
RAM	Random Access Memory
RMT	Reconfigurable Match-Action Table
SAT	Satisfiability
SDN	Software Defined Network (SDN)
SMT	Satisfiability Modulo Theories
SRAM	Static Random Access Memory
SSD	Solid State Drive
TCAM	Ternary Content Addressable Memory
TLV	Type-Length-Value

UDP User Datagram Protocol

WAN Wide-Area Network

LIST OF FIGURES

Figure 2.1 Forwarding device implementing the PISA architecture. Adapted from (KFOURY; CRICHIGNO; BOU-HARB, 2021) 17
Figure 2.2 INT workflow. Adapted from (KIM et al., 2015) 18
Figure 2.3 Reachability property. From the author. 21
Figure 2.4 Waypoint property. From the author. 22
Figure 2.5 Restricted Path Length property. From the author. 22
Figure 2.6 Disjoint Paths property. From the author. 23
Figure 2.7 Loop Freedom property. From the author. 24

Figure 3.1 VERMONT's example scenario. From the author. 27
Figure 3.2 VERMONT's device configuration. From the author. 28
Figure 3.3 VERMONT's network flow. From the author. 29
Figure 3.4 VERMONT's data reporting. From the author. 30
Figure 3.5 VERMONT's report analysis. From the author. 31
Figure 3.6 VERMONT's path tracing mechanism. From the author. 33
Figure 3.7 Complete telemetry header embedded into an Ethernet/IPv4 packet. From the author. 36

Figure 4.1 VERMONT enabled switch pipelines. From the author. 41
Figure 4.2 Flows and topology used for the evaluation. From the author. 50
Figure 4.3 Time to verify properties. From the author. 52

LIST OF TABLES

Table 2.1 Comparison between the type of verification and objectives of the discussed approaches. 25

Table 3.1 Path tracing mechanism example. 34
Table 3.2 Description of telemetry fields used by network devices and in reports sent to the control plane application. 34
Table 3.3 Telemetry header fields required to verify each property. 35
Table 3.4 Usage of telemetry fields by network devices and in reports sent to the control plane application. 35

Table 4.1 Received reports by VERMONT's control plane. 51
Table 4.2 Metadata for the network topologies used on the performance evaluation. 52
Table 4.3 VERMONT's memory (Mbits and %RMT) and header space (Bytes and %MTU) usage. 53

CONTENTS

1 INTRODUCTION 12
2 BACKGROUND AND STATE-OF-THE-ART 15
2.1 Programmable Data Planes (PDPs) 15
2.1.1 P4 16
2.1.2 In-Band Network Telemetry (INT) 17
2.2 Network Verification 19
2.3 Network Properties 21
2.4 Related Work 24
3 VERMONT 27
3.1 Overview 27
3.1.1 Data Plane Configuration 28
3.1.2 Data Plane Operation 29
3.1.3 Control Plane Verification 31
3.2 Design and Architecture 31
3.2.1 Time Framing Events Using Epochs 32
3.2.2 Path Tracing Mechanism 32
3.2.3 Employing Different Telemetry Headers and Storing Data in Network Devices 34
3.2.4 Finding Violations and Analyzing Reports 35
4 IMPLEMENTATION AND EVALUATION 40
4.1 Implementation 40
4.2 Evaluation 49
4.2.1 Verifying Properties in a WAN: The Case of Abilene 49
4.2.2 Scalability Analysis: VERMONT vs. Minesweeper 51
4.2.3 Resource Usage: Network and Physical Devices 51
4.3 Discussion and Limitations 53
5 CONCLUSION 55
REFERENCES 57
APPENDIX A — RESUMO EXPANDIDO 60

1 INTRODUCTION

Communication networks must be resilient. They are expected to overcome operational challenges while maintaining proper functioning. These expectations for behavior can be expressed as network properties. Examples of such properties that are important to network operation are end-to-end reachability, path waypointing, path length limitation, and loop freedom (BECKETT et al., 2017). A myriad of problems may cause the violation of these properties. For example, a bug in a routing application may induce the creation of incorrect paths that do not provide end-to-end reachability. Likewise, a single misconfigured data-plane device may lead to packets being forwarded in a loop. Considering that modern services and applications are usually composed of many components distributed across multiple end-points and even networks, leaving these sorts of problems unnoticed and untracked can quickly result in violations to these network properties, extended service downtime and, consequently, high financial loss.

An important building block for addressing these problems is network verification. This discipline evolved significantly in the last ten years, with the proposal of several pieces of work. These works aim to analyze the correctness of the control plane applications and switch code as well as verify network properties (BECKETT et al., 2017; LIU et al., 2018; DUMITRESCU et al., 2020; STEFFEN et al., 2020; FANTOM et al., 2022; BASAT et al., 2020). Most of the existing approaches work on top of static data only, typically topology descriptions, configuration files, and switch code. These approaches do not take into consideration, however, the actual network traffic or the content of match-action tables in the data plane at runtime. As a result, they cannot detect property violations that may arise only during the network operation. This limitation is exacerbated in the context of software-defined networks (SDNs), since code and configuration are subject to more frequent changes to handle the dynamics of traffic and service demands. These static verification approaches leave a considerable research gap, falling short of determining whether or not networks operate correctly at runtime, considering a highly dynamic environment.

While static verification of network properties has received considerable attention, dynamic verification based on real traffic has been addressed by few studies. Approaches like the one proposed by P4Consist (SHUKLA et al., 2020) analyze artificially generated traffic in the network to verify properties specified on-demand by a human operator. This type of analysis captures a complete view of the inner functioning of the network

from the point of view of a packet but imposes a significant burden on the network. Recent work based on In-band Network Telemetry (INT) for network monitoring (KIM et al., 2015; MARQUES; LEVCHENKO; GASPARY, 2020; TIAN et al., 2021) hints at a more efficient and accurate way for dynamic property verification. These monitoring approaches allow for fine-grained visibility on what is happening inside a network while imposing little overhead on the production packets and without the need for generating probe traffic.

Aiming at capitalizing on the mentioned opportunity to advance the field of property verification, in this thesis, we propose VERMONT, an approach for continually monitoring network properties and quickly detecting violations at runtime. Using INT-based efficient metadata collection, VERMONT aggregates and correlates information to detect violations at their source and report them to an external control-plane management application. More specifically, traffic of interest is monitored on an epoch basis, and each monitored packet carries only essential information for the verification process, gathered as the packet traverses the network. At egress devices, VERMONT selectively decides, based on the information contained in the packets, whether report packets should be sent to verification servers. The generated reports are analyzed in the control plane to verify properties such as reachability, waypointing, and loop freedom.

The *main research contributions* of this thesis are:

- A distributed, in-network approach capable of collecting and analyzing metadata from the data plane to monitor network properties in real time. By dividing time in slices, this approach is able to accurately verify whether defined properties are being satisfied or not for each slice in a network.
- Design and implementation of a proof-of-concept system that allows operators to express network properties, which are translated to INT-based monitoring campaigns.
- Evaluation of VERMONT and comparison with a state-of-the-art approach, considering performance and costs.

The remainder of this thesis is organized as follows. In Chapter 2, we introduce the background for this research, motivate VERMONT's design by discussing the related work, and formalize relevant network properties. In Chapter 3, we introduce VERMONT, the proposed approach for violation detection in real-time. In Chapter 4, we describe design and implementation details and present the experimental evaluation and results

while including a discussion on different aspects of VERMONT's operation. In Chapter 5, we conclude our work with our final considerations.

2 BACKGROUND AND STATE-OF-THE-ART

This chapter covers the fundamental concepts and discusses previous work carried out in the area of this thesis. Specifically, in Section 2.1, we briefly explain the evolution of programmable data planes and their need to be verified. In Section 2.2, we bring forth the main concepts and techniques developed in the recent past for the verification of networks. In Section 2.3, we introduce the motivation and definition of some crucial network properties, and finally, in Section 2.4, we briefly describe and discuss the state-of-the-art approaches to network verification.

2.1 Programmable Data Planes (PDPs)

Traditional network devices are usually tied to vendor specifications. A network operator can configure network devices solely using vendor-provided interfaces. These interfaces are typically focused on a set of protocols or mechanisms but do not give enough independence to the network operator. This slows innovation and does not give the network owners/operators complete control over their devices (FEAMSTER; REXFORD; ZEGURA, 2014). The Software Defined Network (SDN) concept emerged, intending to provide control of the network devices to their owners. The concept elaborates on separating the control and data planes. The control plane is responsible for configuring how the devices will forward packets, while the data plane is composed of the devices that forward these packets (FEAMSTER; REXFORD; ZEGURA, 2014). These components are connected through an Application Programming Interface (API), and OpenFlow was one of the first to emerge. A network device that supports Openflow (commonly referred to as a switch) is characterized by a set of tables with rules that match pieces of information (fields) on the packets. After the matching, the switch will perform actions based on the rule activated. For this, a switch can behave beyond a simple switch, like a firewall, router, or other application-specific network devices (FEAMSTER; REXFORD; ZEGURA, 2014; MCKEOWN et al., 2008).

Initially, OpenFlow explored the fact that many network devices, such as switches, nowadays possess tables for flow control. So, in a way, implementing the OpenFlow protocol into a commodity device would be “fairly easy”. The idea was well received by the vendors, who could enable innovation without exposing the inner functioning of their devices (FEAMSTER; REXFORD; ZEGURA, 2014; MCKEOWN et al., 2008).

Openflow started with a simple specification but became rather complex in a few years. For example, it started with the capacity to match 12 header fields, and each new version extended this list, whereas, on version 1.4, Openflow could match 41 different fields. Unfortunately, many more fields would need to be checked, and it became clear that extending Openflow's specification was not the way to go (BOSSHART et al., 2014).

2.1.1 P4

The Domain Specific Language (DSL), Programming Protocol-independent Packet Processors, or simply P4, was proposed to solve the previously mentioned issue, along with programmable architectures, as is the case of the Reconfigurable Match-action Table (RMT) (BOSSHART et al., 2013) architecture. It is defined as a language to allow the programmer to change how a switch processes each packet. Unlike what is observed and proposed in Openflow, P4 is not tied to any protocol. The switch can view the packet as a set of bits, and through the P4 program installed, it knows how to process them (BOSSHART et al., 2014). One of the main concerns of the P4 language is to allow reconfigurability. This means the control plane should be able to change packet processing behavior at any given time. Another concern was to guarantee that one P4 program could be installed on any P4-enabled switch. In this manner, the P4 programmer would only need to write the code once and run it on any switch (BOSSHART et al., 2014).

To process the large number of protocols that could emerge, P4 implements the abstraction of headers. Headers are a description of fields that could exist inside a packet. For example, a P4 programmer can define a header specifically for IPv4 with all the protocol's fields and sizes. Thus, when the packet enters the switch, the header will be fitted in this structure for further processing (BOSSHART et al., 2014).

P4-enabled forwarding devices implement the Protocol Independent Switch Architecture (PISA). In Figure 2.1, we illustrate the basic "anatomy" of such a device and the interaction between its components. The switch processes ingress packets with three parts: the programmable parser, the programmable match-action pipeline, and the programmable deparser. The programmable parser is a state-driven program that extracts the headers from packets. After the headers are extracted, they are submitted to the match-action pipeline. In this stage, the header fields are compared to table entries (one or more tables) inserted by the control program. After the matching, action(s) will be executed according to what was determined in the "matched" table entry(ies). These actions can

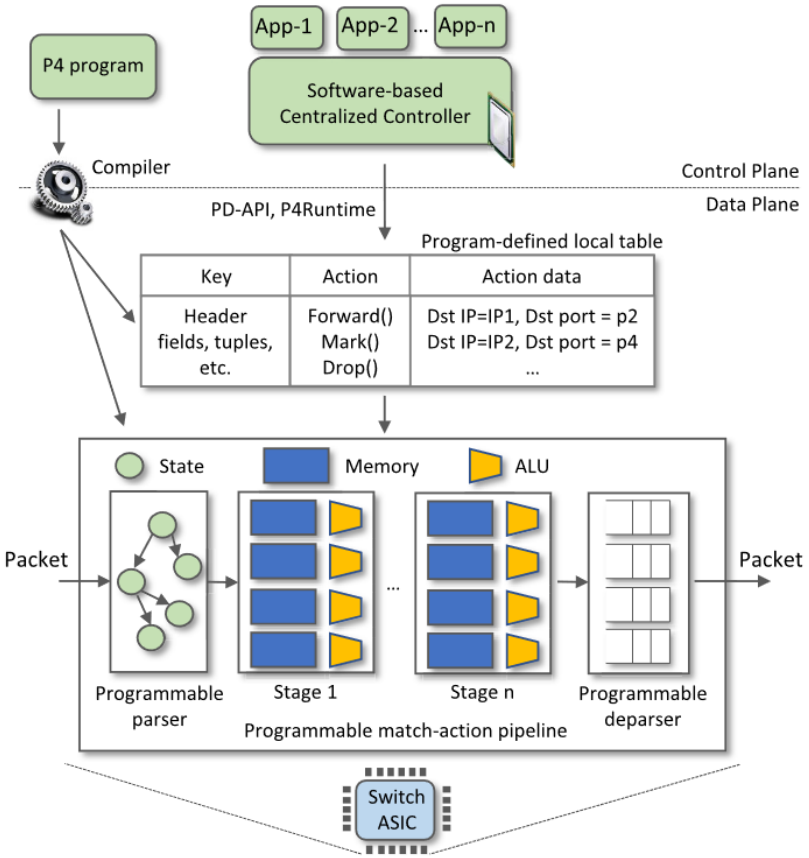


Figure 2.1 – Forwarding device implementing the PISA architecture. Adapted from (KFOURY; CRICHIGNO; BOU-HARB, 2021)

define how packets are to be forwarded, set header fields, or even add new headers to be inserted into the packets (BOSSHART et al., 2014). The final step of the workflow is executed by the deparser, which inserts the headers that will be part of packets as they leave the switch.

With this amount of programmability inside a network, one problem continuously grew in size: bugs. Since the programmer now holds the power to modify the inner functioning of a network device, any mistake in a program could lead to catastrophic failure. Fortunately, to try to solve this issue, the field of network verification was created and developed (FREIRE et al., 2018a). Later in this chapter, we will briefly discuss how this area is evolving.

2.1.2 In-Band Network Telemetry (INT)

Data plane programmability allowed for a novel way of collecting metrics from within the networks, called *in-situ* Operations, Administration and Maintenance (OAM)

(BROCKNERS et al., 2017), or INT (KIM et al., 2015). This method consists of the collection of data regarding network OAM information with data packets as they traverse through the network. INT (KIM et al., 2015) is the most prominent framework nowadays that allows for in-band network telemetry. One example of such collected data are device-internal state metrics, such as queue size. These metrics will then be stored within telemetry-specific headers carried by packets.

The INT framework is composed of a controller, source, destination (or sink), and transit nodes. Each node is responsible for specific actions regarding the packets flowing through the network. Source nodes are responsible for enabling packets to carry information about the network. They insert a telemetry-specific header in packets (of interest) that will be continuously updated as they traverse the network. Transit nodes, among regular operations, update this header as needed. Destination nodes are responsible for stripping the telemetry headers from the packets and forwarding them to the controller (for further inspection). The "original" packets are then forwarded to the recipients. This behavior is exemplified in Figure 2.2.

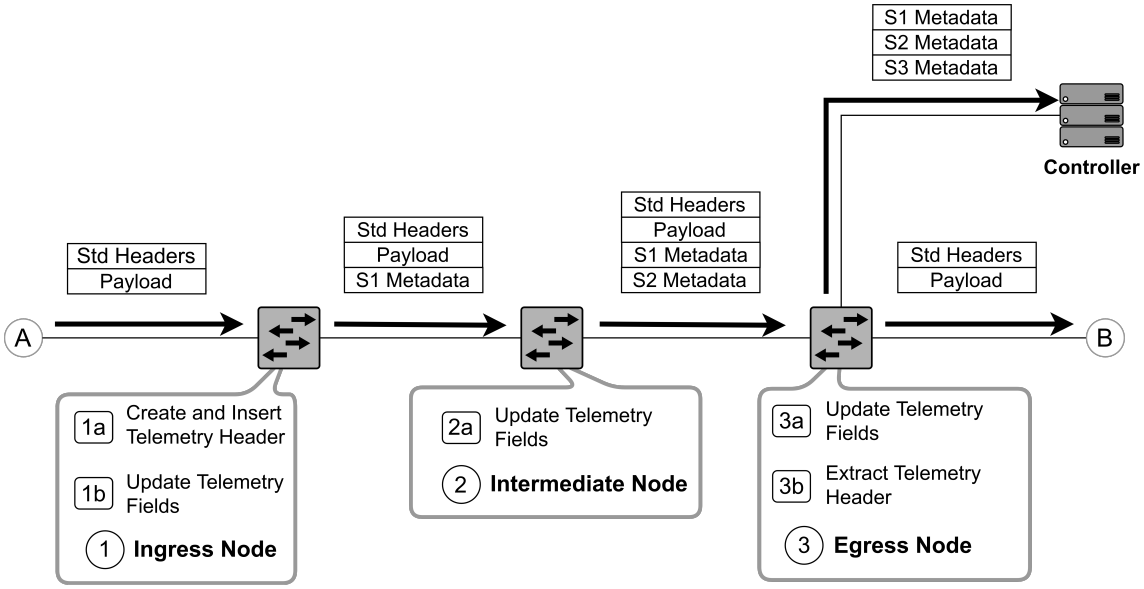


Figure 2.2 – INT workflow. Adapted from (KIM et al., 2015)

According to (P4ORG, 2020), INT has some header types that may be applied to any of its implementations. These header types are MD (type 1), Destination (type 2), and MX (type 3). Types 1 and 3 refer to headers that must be processed by intermediate nodes. Type 3, specifically, indicates that this header may generate reports while the packet has not yet reached its egress device. Type 2 indicates headers that are solely processed by

destination (egress) devices, and may or may not generate reports. These specifications apply to all headers used in this work, but there is an overlap between types 2 and 3, since our headers are both verified and may generate reports in intermediates and egress devices.

2.2 Network Verification

As networks grew more complex, it is natural that their problems became harder to solve. When certain network expectations are violated (e.g., policies and properties), the lack of availability or reliability becomes a problem. As network operators can modify the devices' inner functioning of packet processing and forwarding, the number of bugs and configuration problems has risen to an unprecedented level in the last few years (ZENG et al., 2012). Aiming to solve this problem, the field of network verification, through multiple methods, tries to find and evaluate errors inside the networks. This field is capable of employing diverse disciplines as means to find and correct bugs and misconfigurations. The first methods to appear were the formal methods. Formal methods use mathematically proven techniques to model and solve theorems to find and identify problems. The most prominent formal methods are model checking, theorem proving, symbolic execution, and Satisfiability Modulo Theories/Satisfiability (SMT/SAT) solvers (LI et al., 2019; SOURI et al., 2020). The following paragraphs shed light on the functioning of each of these methods.

According to (LI et al., 2019), model checking relies on verifying whether a model satisfies certain specifications. Model checkers usually employ Finite State Machines (FSMs) to define and check the models. Model checkers are composed of three parts: the model of the network (represented as a state-machine-readable language), the specification of the expected system, and the checking procedure. As the FSM calculates the satisfiability of the model compared to the expected outcome, one or more properties may not be satisfied. If that is the case, the FSM will produce a counterexample, showing exactly where the problem lies. Model checkers usually verify the entire system space state, increasing the verification time exponentially as the system space grows.

Theorem proving is one of the most common verification techniques. The network is represented as a set of formulas that describe the system implementation and properties. The properties desired to be checked are defined as axioms. The verification takes place as the formulas are evaluated against the axioms. If the theorem is proved, the network

satisfies all properties. Theorem proving is a technique that does not require checking the entire system space but relies heavily on the knowledge of the applicator. The applicator, or network operator, must know precisely the network's functioning and how to describe the network to the theorem prover. The process is still slow and very fallible due to heavy user interaction (LI et al., 2019).

Symbolic execution is the most straightforward formal verification method. The network is represented as a program, and all its configurations are inputs to the program. The network is then simulated, and the desired properties are observed in the execution. If a failure happens, the symbolic execution program can show why and where the problem happened. Symbolic execution approaches can not evaluate large networks by default due to the path explosion problem, and several research efforts tried to solve this issue, as stated later in Section 2.4. Still, symbolic execution is a slow technique but very reliable in scenarios where few changes are made within a significant amount of time (LI et al., 2019).

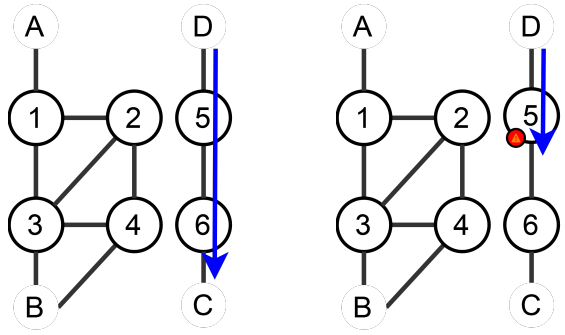
SMT/SAT solvers work by reducing the network to an SAT problem. As the network is reduced, any typical SAT solver can verify whether the desired properties of the network are being achieved. The problem of these solvers lies in the translation of the network to the desired format. Translating a network with all its configurations and specifications to a Boolean formula is no easy task and requires great effort. SMT/SAT solvers cannot define where the problem lies if any is encountered; they only return that the network does not satisfy all properties queried (LI et al., 2019).

All the aforementioned methods have one characteristic in common: they execute outside the network. We call this type of analysis static analysis, as the verifying system does not consider dynamic network behavior (due to fluctuations or modifications done in real-time). This type of analysis also usually takes an unaffordable amount of time to execute. In the last few years, some research efforts have been employed to develop techniques that use actual network execution information to check policies and properties. This type of analysis is called dynamic analysis. A dynamic analyzer has, by definition, a way of verifying network-related expectations with pieces of information derived or extracted from the network during execution time. This type of analysis is capable of, beyond identifying problems during execution, consuming much less time (FREIRE et al., 2018a).

2.3 Network Properties

Before moving to the design of the property violation detection approach, we introduce a non-exhaustive list of network properties whose verification is relevant as recognized by other investigations in the area, such as Minesweeper (BECKETT et al., 2017) and NetDice (STEFFEN et al., 2020). In this thesis, we assume that traffic flows are identified by the usual 3-tuple, i.e., source IP address, destination IP address, and transport protocol or application type (e.g., video streaming, web). Formally, we define a traffic flow as a collection of packets F such that $\forall p_i \in F; \forall p_j \in F$; the identification tuple for p_i and p_j have identical values. Furthermore, we formalize the path of a packet p as a multiset $P(p)$ of network switches. We also denote the number of elements in a set S by $|S|$.

Reachability. This property expresses the expectation that traffic from a particular flow of interest can reach its destination starting from its source in the network during a window in time. As an example, in Figure 2.3a, flow Blue originates in D and is expected to reach C. In the example, packets are correctly forwarded, and the property is satisfied. As a counterexample, in Figure 2.3b, due to an unexpected problem, node 5 can not forward any packets; thus, the property is not satisfied.



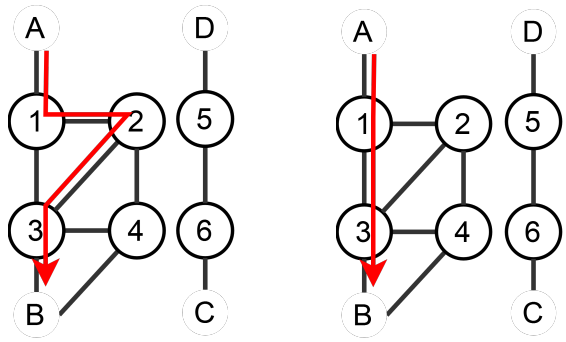
(a) Satisfied property (b) Violated property

Figure 2.3 – Reachability property. From the author.

Formally, given s and t the expected source and destination (respectively) of a flow of interest F, $F(t_{begin}, t_{end})$ the subset of packets of F that were forwarded along the network between instants t_{begin} and t_{end}, the Reachability property is satisfied during this time window when $\exists p \in F(t_{begin}, t_{end})$ such that $p.ingress = s$ and $p.egress = t$.

Waypointing. The Waypointing property states that there are one or more switches considered $waypoints$, which all packets of a flow must visit. There are two variations to this property. Either each packet must visit (i) *all* of the waypoints or (ii) *at least one* of

the waypoints. As an example, in Figure 2.4a, flow Red from A to B is expected to visit node 2 at any moment during its traversal through the network. In this example, packets are forwarded to node 2, and the property is satisfied. In Figure 2.4b, due, e.g., to the network employing the shortest path to the packets, node 1 forwards packets directly to node 3, and as such, the property is not satisfied.

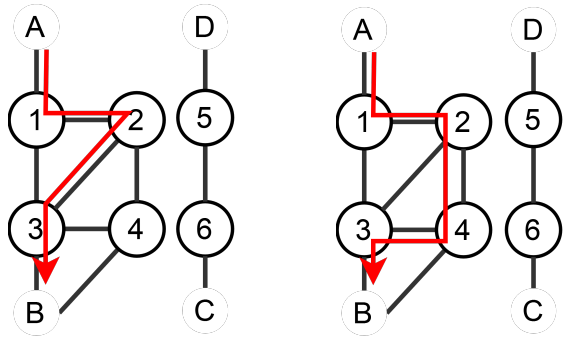


(a) Satisfied property (b) Violated property

Figure 2.4 – Waypoint property. From the author.

We provide formal definitions for both previously mentioned variations as follows. Given a set of waypoint switches W: (i) the first variation is satisfied for a flow F iff $\forall p \in F; \forall s \in W; s \in P(p)$ and (ii) the second variation is satisfied for F iff $\forall p \in F; \exists s \in W$ such that $s \in P(p)$.

Restricted Path Length. This property imposes restrictions on the length of the path taken by all packets of a flow of interest. There are two variations to this property. Either the path cannot have a length (i) that exceeds a limit or (ii) that is different from a fixed value. In Figure 2.5a, we provide an example of this property being satisfied for flow Red, from A to B, considering a path with a maximum length of 3. In Figure 2.5b, the property would not be satisfied, given that the path exceeds the imposed limit.



(a) Satisfied property (b) Violated property

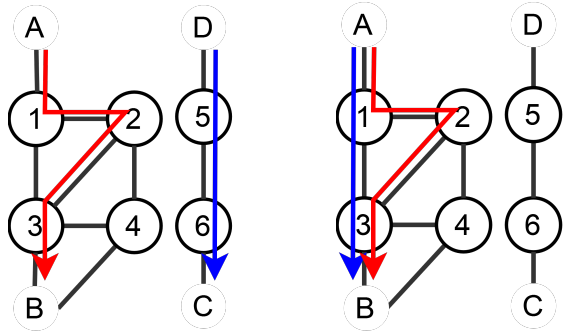
Figure 2.5 – Restricted Path Length property. From the author.

For the limited path length case and considering a threshold k_l that indicates the

maximum number of switches that may be visited, the property is satisfied for a flow F iff $\forall p \in F; |P(p)| \leq k_l$. For the fixed path length case and considering a value k_f that indicates the exact number of switches that must be visited, the property is satisfied for a flow F iff $\forall p \in F; |P(p)| = k_f$.

Path Preference. The Path Preference property states that all packets from a flow must follow a specific path inside a network. Looking back to the same figures used to illustrate the Restricted Path Length property, consider that flow Red is expected to be forwarded using the path [1-2-3]. The property is satisfied for Figure 2.5a but is violated in Figure 2.5b. As a formal definition, given Q the set of switches the packets from F are required to visit, the Path Preference property is satisfied for F iff $\forall p \in F; P(p) = Q$.

Disjoint Paths. The Disjoint Paths property states that two flows of packets inside a network *must not* share any device within their paths. In Figure 2.6a, flows Red (from A to B) and Blue (from D to C) do not share any devices in their path through the network, and, as such, the property is satisfied. The property is violated in Figure 2.6b, as both flows share devices 1 and 3.



(a) Satisfied property (b) Violated property

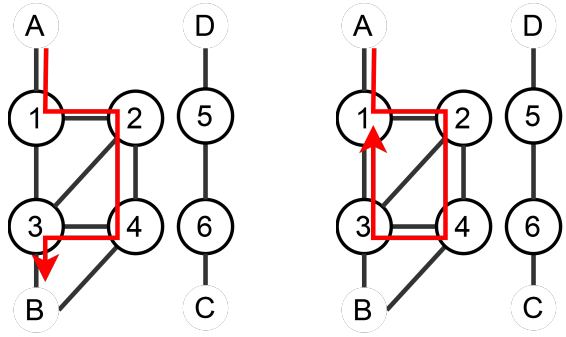
Figure 2.6 – Disjoint Paths property. From the author.

Formally, given two flows of interest F_a and F_b, the Disjoint Paths property is satisfied for these flows iff $\forall p_a \in F_a; \forall p_b \in F_b; \forall s \in P(p_a); s \notin P(p_b)$.

Loop Freedom. The Loop Freedom property states that the packets from a flow *must not* visit any device from the network more than once. As an example, Figure 2.7a demonstrates how this property would be satisfied for flow Red, while Figure 2.7b demonstrates how it would be violated.

Specifically, the Loop Freedom property is satisfied for a flow of interest F iff $\forall p \in F; \forall s \in P(p)$; after removing one occurrence of s from $P(p), s \notin P(p)$.

We call the reader's attention to the fact that the properties above are fundamental to the operation of modern networks. As an example, the Restricted Path Length may



(a) Satisfied property (b) Violated property

Figure 2.7 – Loop Freedom property. From the author.

be crucial to ensure flows of time-sensitive applications meet maximum tolerable end-to-end delays (BüLBüL; ERGENç; FISCHER, 2022). It is also worth mentioning that the proposed solution introduced in the next Chapter is not restricted to the properties above. While they are good representatives, other properties can be readily derived.

2.4 Related Work

Several expectations regarding the treatment of flows, i.e., properties, exist in a network. From basic reachability to loop freedom, the network is expected to maintain these properties, ensuring correct functioning. While it is possible to verify such properties prior to execution, modifications done exclusively in the data plane or even the physical failure of devices may lead to violations to these properties. In this section, we motivate VERMONT and discuss how some state-of-the-art approaches try to solve this problem.

Minesweeper (BECKETT et al., 2017) is one of the most prominent efforts on network-wide property verification. It carries out verification offline and relies on a static snapshot of the network and its configuration. As part of the verification, Minesweeper provides a detailed description of how forwarding rules impact the properties associated to flows of interest. Unfortunately, this approach is neither able to keep up the frequency of change (e.g., rule insertion/deletion) in modern software-defined networks nor supports configuration files of state-of-the-art programmable switches.

In addition to Minesweeper, multiple approaches (LIU et al., 2018; FREIRE et al., 2018b; NEVES et al., 2018; STOENESCU et al., 2018; SHUKLA et al., 2019; STEFFEN et al., 2020; ZHENG et al., 2022; ALBAB et al., 2022) focus on the verification of data-plane programs. A common challenge is that of exploring all of the possible execution

paths in these programs without exhausting computing resources and taking prohibitive time. In this context, Meissa (ZHENG et al., 2022) proposes a technique to summarize switch code that simplifies symbolic execution, allowing for shorter program verification run times. SwitchV (ALBAB et al., 2022) introduces fuzz testing into the symbolic analysis, identifying possibly unwanted behaviors in switches (also called device invariants). The fuzz testing generates possible irregular packets and processes them via symbolic execution using the devices configuration files. P4RL (SHUKLA et al., 2019) goes a step further by proposing a reinforcement learning approach to guide the fuzzing process while testing if the packets, even with irregularities, are correctly handled by the devices. Compared to our work on VERMONT, all these efforts work on top of static elements, for example, configuration files. Thus, these approaches are not able to catch failures or fluctuations that occur during the execution of the network. In addition to that, (ALBAB et al., 2022) and (SHUKLA et al., 2019) focus on bugs on a single program or device and, thus, cannot evaluate network-wide properties.

Table 2.1 – Comparison between the type of verification and objectives of the discussed approaches.

Name	Type of Verification	Types of analyzed properties	Type of analysis
Minesweeper	Static	Network-wide invariants	Symbolic execution
VERA	Static	Network-wide invariants	Symbolic execution
Netdice (PVNC)	Static	Network-wide invariants	Symbolic execution
Meissa	Static	Network-wide and device invariants	Symbolic execution
SwitchV	Static	Device invariants	Symbolic execution and synthetic traffic analysis
P4RL	Static	Device invariants	Symbolic execution and synthetic traffic analysis
BF4	Dynamic	Device invariants	Symbolic execution
Veriflow	Dynamic	Network-wide invariants	Symbolic execution
P4Consist	Dynamic	Network-wide invariants	Synthetic traffic analysis
Vermont	Dynamic	Network-wide invariants	Real traffic analysis

Another line of work seeks to catch bugs in software and inconsistencies between data and control plane configuration as they arise at runtime (KHURSHID et al., 2013; DUMITRESCU et al., 2020; SHUKLA et al., 2020; SILVA; SCHAEFFER-FILHO, 2022). This type of approach is called dynamic verification. Veriflow (KHURSHID et al., 2013) and Bf4 (DUMITRESCU et al., 2020) place themselves between both SDN planes.

VeriFlow analyzes table rules as they are inserted into OpenFlow-enabled devices. For every forwarding rule that is inserted, VeriFlow analytically evaluates if it could lead to network-wide property violations via symbolic execution. Similar to the works described in the previous paragraph, Bf4 is focused on P4 program verification, identifying bugs in the switch code using assertions via symbolic execution, preventing disruptive rule insertions and identifying possible device invariants. In contrast to those, it considers table rules inserted at runtime to guide the verification. P4Consist (SHUKLA et al., 2020) goes in a different direction by generating probes (synthetic traffic) to test the consistency of the rules that should be applied to specific packets in the network, identifying whether packets flowing through a network are being correctly matched through all the switches.

In Table 2.1, we present a comparison between the approaches used by the discussed works. We classify them using the most important features of each approach to create an easily understandable comparison. It becomes clear that only one approach, in fact, uses the network to verify the data plane, while the others focus on static verification or preventing unwanted device behaviors. Compared to VERMONT, none of these approaches monitor the actual traffic flows (whose properties are expected to be verified).

Given the recent advances in network programmability, marked especially by in-band network telemetry (KIM et al., 2015), we argue that the building blocks for efficient dynamic property verification are in place and represent a unique opportunity to promote verification upon production network packets instead of using synthetic traffic or probe packets. This venue allows for a new method of network verification that is both more representative as well as more efficient in terms of time and resources.

3 VERMONT

In this chapter, we present VERMONT's approach to monitoring and verifying network properties. In the data plane, VERMONT employs the concept of in-band telemetry to collect and embed important path metadata for packets belonging to flows of interest. This metadata is aggregated and stored temporarily at the last hop of each flow of interest. This partially processed telemetry data is periodically exported to the control plane, which is responsible for verifying network properties and detecting violations.

3.1 Overview

This section illustrates VERMONT's approach to network verification considering two example properties. For clarity, in this example, we focus on the waypointing and loop freedom properties as well as consider only two traffic flows, as seen in Figure 3.1. The first flow (represented in blue in the figure) comes from End-point A and should be routed via waypoint Node 5 on the way to End-point B. For this example, we consider the case where this flow is routed as expected. The second flow (in red) enters the network through End-point C, is expected to be routed using path [6, 3, 2, 1], and exits the network using End-point A. For this example, we assume that Nodes 3 and 4 have inconsistent forwarding rules that lead packets to a routing loop. Next, we describe VERMONT's operation step-by-step considering this example scenario.

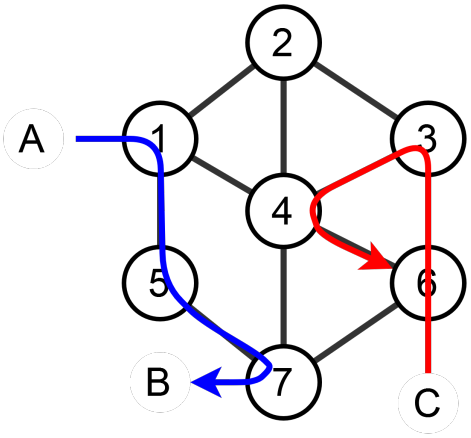


Figure 3.1 – VERMONT's example scenario. From the author.

3.1.1 Data Plane Configuration

The first task executed by VERMONT is the configuration of data plane devices. In our example, Figure 3.2 represents the configuration steps, preparing the network for the incoming flows of packets. VERMONT initiates the configuration by setting the same epoch length to every device in the network. Epochs allow network devices to indicate when events happened and the control plane to verify properties during discrete windows of time. Verifying network properties also requires tracing the paths taken by the packets of each flow of interest. VERMONT dynamically installs all the configurations needed for this task in the data plane devices. Further details on the implementation of epochs and path tracing are introduced later in Sections 3.2.1 and 3.2.2.

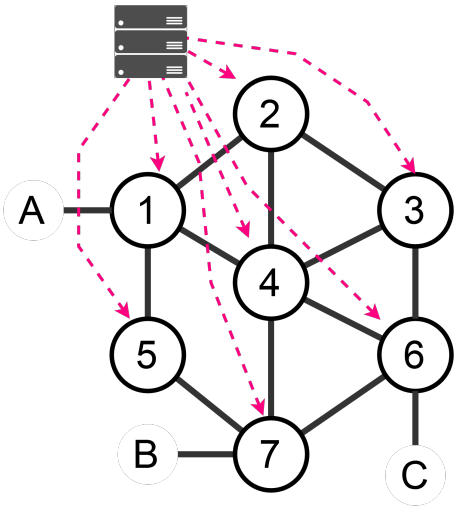


Figure 3.2 – VERMONT's device configuration. From the author.

VERMONT is based on in-band network telemetry. Depending on the properties being verified, packets of a flow under analysis may carry different sets of telemetry fields. For example, to verify a waypointing property, it is of utmost importance that the telemetry fields are updated with the identification of every visited device. This differs from the reachability property, where it is only necessary to carry the specific identification of the flow and the ingress and egress devices used. To avoid unnecessarily burdening packets with superfluous telemetry data, VERMONT configures forwarding devices to annotate the packets with only the fields strictly needed for the specific properties of interest. These fields are dynamically inserted in the packets as needed. Further details of how VERMONT implements this mechanism are described in Section 3.2.3.

3.1.2 Data Plane Operation

As soon as the devices are configured, we consider that VERMONT is ready to verify the properties of the flows of interest. Each device that forwards the packets executes procedures according to its role respective to the flow. The roles can be *ingress*, *intermediate*, or *egress* node. First, the packets enter the network through their ingress nodes. The ingress node is responsible for embedding VERMONT's telemetry header into packets. The first action performed by the ingress node upon receiving a packet, is determining the current epoch. The current epoch indicates at what point in time the packet arrived at the network. Next, it creates and inserts the telemetry header into the packet. This header contains the current epoch as well as all the fields necessary to verify the properties associated to the flow of the packet. These fields are initialized to base values. Coming back to the example, packets from flow Blue arrive at Node 1, while packets from flow Red arrive at Node 6. Both have VERMONT's telemetry header inserted and initialized.

As soon as the telemetry header is initialized, the ingress device updates all the fields related to the properties being verified. In the case of flow Blue, the fields that trace the path of its packets are updated using rules installed previously during the data plane configuration phase. The fields that check for loops are marked for flow Red, indicating the packet visited Node 6. Once the update of the telemetry header fields is finished, the packet is then sent to an intermediate node (Node 5 for flow Blue, Node 3 for flow Red, in Figure 3.3).

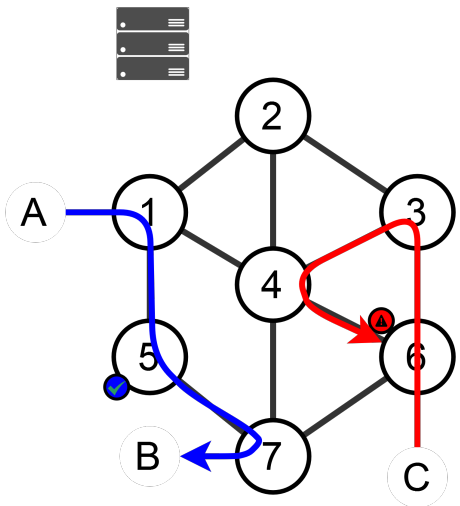


Figure 3.3 – VERMONT's network flow. From the author.

Intermediate nodes only update the data of the telemetry headers as the packets

traverse the networks for most of the properties being verified. The egress is the last node to process production packets (of the flows of interest). It extracts the header after updating the telemetry fields. Thus, the packets from production flows return to their original form before leaving the network. The egress node then updates the metadata regarding the flow using the values of the telemetry header. This information is sent to the control plane at the end of an epoch for verification. In our example, this is the case of Node 7, which sends its information to the data plane at the end of an epoch (Figure 3.4).

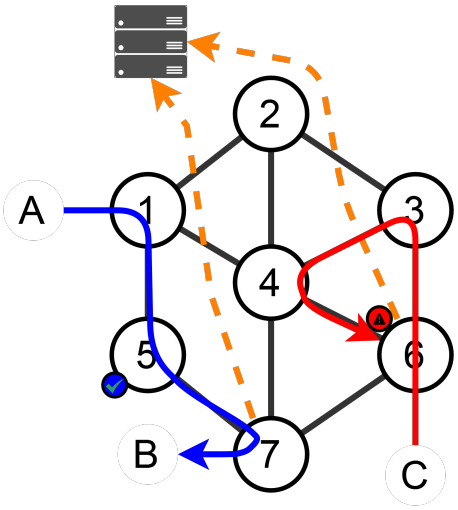


Figure 3.4 – VERMONT's data reporting. From the author.

In addition to the processing carried out by each node respective to its specific role, they are also responsible for verifying the loop freedom property. More specifically, nodes mark and check the appropriate fields in the telemetry header that enables them to detect the occurrence of loops. Whenever a loop is detected, a special report is immediately generated. This is because loops may prevent normal reports from being generated, so it is crucial to report them as soon as they are observed. This report contains information about the flow and its path, allowing for further diagnosis in the control plane. Further details on how loops are detected and reported are presented in Section 3.2.4.

As seen in Figure 3.4, packets from flow Red are incorrectly forwarded to Node 4. At Node 4, the packets follow the intermediate node processing and are forwarded to Node 6. Node 6 immediately notices that the packets have already passed through it and sends a notification to the control plane. Packets from flow Red are then marked to not send other loop notifications to the control plane in this epoch.

3.1.3 Control Plane Verification

VERMONT's control plane continuously listens for reports arriving from the data plane, verifying the associated properties. The reports are of two types: *End-Of-Epoch* reports and *Loop Detection* reports. End-Of-Epoch reports are used for the verification of all properties with the exception of loop freedom. For example, in the case of the report generated in Node 7 for flow Blue, the control plane is able to determine all devices visited by packets of this flow. It can also verify that Node 5 was visited and, thus, the property was satisfied during the last epoch. Loop Detection reports contain the information needed to verify where the error occurred. In the case of the loop report generated by Node 6, the control plane is able to verify that Nodes 3 and 4 forwarded the packets incorrectly, as seen in Figure 3.5. To perform the verification, VERMONT employs multiple mechanisms, as we detail next.

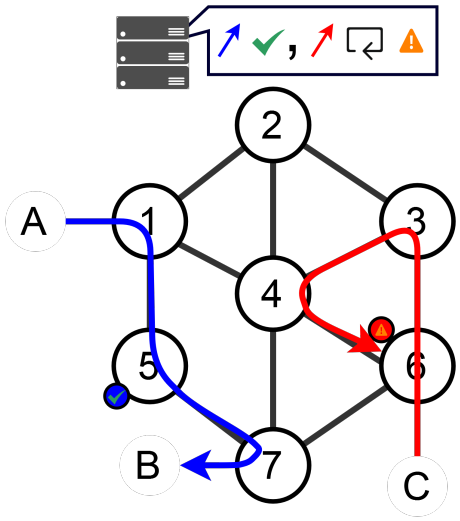


Figure 3.5 – VERMONT's report analysis. From the author.

3.2 Design and Architecture

VERMONT is composed of a set of mechanisms that allow for the verification of network properties. We previously presented an overview of how our approach is materialized in the data and control planes. In this section, we will provide further details on these mechanisms and how they interact.

3.2.1 Time Framing Events Using Epochs

In order to execute verification tasks in a time scale appropriate to present-day networks (where minimal downtime periods are expected), it is necessary to work with a suitable event framing mechanism. Discretizing time into windows or epochs is a natural course of action that enables VERMONT to monitor properties in short intervals and summarise multiple events of interest from a period with a single report.

The devised mechanism is based on the clocks of forwarding devices, which use their local timestamps to determine the epoch number. The timestamp is divided up by a predefined epoch length to determine the current epoch. Every packet from a flow of interest arriving at the network is embedded with the current epoch value of its ingress switch. This makes it possible to verify at which point in time the packet traversed the network and pinpoint events of interest in the corresponding time window. Devices recalculate their local epoch field every time a packet arrives with a value greater than the one stored. An *End-Of-Epoch* report is triggered when the epoch value update happens. In VERMONT's analysis, the effects of clock drifting on correlating events impacting multiple flows are mitigated by considering reports for a few of the earlier and later epochs.

Determining the appropriate length for epochs in a single network is dependent on multiple factors, such as the capacity of both data and control plane devices. The higher the epoch time, the lower the resources used (reports generated). Conversely, the lower the epoch time, the higher the usage of resources. The verification of properties usually does not require fine-grained precision in time, so we suggest epoch lengths in the order of hundreds of milliseconds or even a few seconds. This allows for an accurate verification time while using limited resources in the network.

3.2.2 Path Tracing Mechanism

VERMONT depends on the tracing of packet paths to be able to verify properties. This is traditionally a high-cost mechanism when dealing with INT-based solutions. Most approaches append the identification of every visited device on packets in order to be able to trace their path within the network. Inspired by previous successful experiences of our research group (MARQUES; LEVCHENKO; GASPARY, 2020), VERMONT uses a different mechanism.

The mechanism used allows for the identification of different paths using a single

fixed-size tuple. This tuple has the form <*source, destination, length, code*>. Every packet of interest in the network carries this tuple, which is gradually updated by every device the packet visits. The control plane computes the possible paths for each flow of interest beforehand and configures forwarding devices with rules to update the path tracing tuple depending on their forwarding decision.

As an example, flow Blue in Figure 3.6 has three possible 3-hop paths to reach its destination. The first path comprises devices [1,2,5]; the second is composed of devices [1,3,5], and the third of the devices [1,4,5]. Table 3.1 details the gradual update carried out by every device visited by a packet from this flow for these three paths. Note that the main difference between the paths is the second hop. In both scenarios, the `Path_id` field has the same value at the first hop. When reaching the second hop, packets following the first path only have their `length` field incremented by one, while the `code` field is left unmodified. Packets in the second and third paths, however, have both their `code` and `length` values modified at this hop, following pre-configured rules in Nodes 3 and 4. At the end of the traversal of the packet through the network, the egress device updates both fields (as well as the `destination` field) and temporarily stores the `Path_id` tuple in its memory.

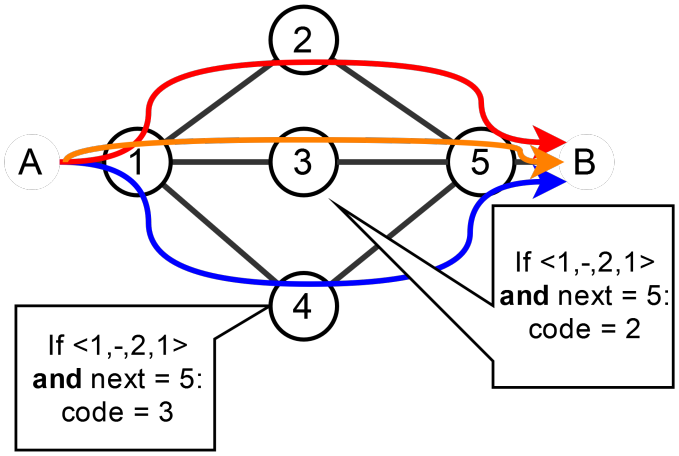


Figure 3.6 – VERMONT's path tracing mechanism. From the author.

Reports sent at the end of an epoch by an egress device include the `Path_id` tuple. They are analyzed upon arriving in the control plane application. A dictionary is used to decode tuple values into an ordered list of forwarding devices, indicating the path taken by packets. The full description of this mechanism is out of the scope of this thesis but is available at (MARQUES; LEVCHENKO; GASPARY, 2020).

Table 3.1 – Path tracing mechanism example.

Path	Hop 1	Hop 2	Hop 3
1,2,5	$< 1, -, 1, 1 >$	$< 1, -, 2, 1 >$	$< 1, 5, 3, 1 >$
1,3,5	$< 1, -, 1, 1 >$	$< 1, -, 2, 2 >$	$< 1, 5, 3, 2 >$
1,4,5	$< 1, -, 1, 1 >$	$< 1, -, 2, 3 >$	$< 1, 5, 3, 3 >$

3.2.3 Employing Different Telemetry Headers and Storing Data in Network Devices

VERMONT supports a number of telemetry fields, as enumerated and briefly described in Table 3.2. For the system's operation, values observed in telemetry fields are stored in ingress, egress, or both devices. As an example, `Flow_id` is used by egress devices to store and update data regarding a specific flow of interest. Values of telemetry fields are also used in reports, as the control plane requires these pieces of information to reason about the violation of properties.

Table 3.2 – Description of telemetry fields used by network devices and in reports sent to the control plane application.

Field/Variable	Description
`Epoch`	Indicates the epoch a packet entered the network.
`Path_src`	Identifies the ingress device of the packet.
`Path_length`	Counts the number of devices that were visited by the packet in its path.
`Path_code`	Encodes the path taken by the packet.
`Path_dst`	Indicates the destination of the packet.
`Flow_id`	Indicates the flow to which the packet belongs. Composed of subfields `Path_src`, `Path_length` and `Path_code`.
`Loop_identifier`	Indicates which devices where visited by the packet.
`Loop_detected`	Indicates whether the packet has been routed through a loop.
`Node_id`	Identifies the node sending a loop report.

As previously mentioned in Chapter 3, VERMONT dynamically inserts in the packets of a flow only the telemetry fields needed for verifying properties of interest. For this task, it employs the Type-Length-Value (TLV) technique, which allows for the creation of a header composed of varying subheaders embedded in each packet of the network. Table 3.3 indicates the fields required for the verification of each property currently supported by VERMONT. When a flow of interest is being monitored for two or more properties that share the need for common telemetry fields, those will be collected and appear only once in a telemetry header. Table 3.4 presents the usage of each field by devices and reports during VERMONT's execution.

As an example, consider a regular network that employs the Ethernet and IPv4

Table 3.3 – Telemetry header fields required to verify each property.

	Property					
Field/Variable	R	W	RPL	PP	DP	LF
`Epoch`	✓	✓	✓	✓	✓	✓
`Path_src`	✓	✓	✓	✓	✓	✓
`Path_length`	-	-	✓	-	-	-
`Path_code`	-	✓	✓	✓	✓	-
`Path_dst`	✓	✓	✓	✓	✓	✓
`Flow_id`	✓	✓	✓	✓	✓	✓
`Loop_identifier`	-	-	-	-	-	✓
`Loop_detected`	-	-	-	-	-	✓
`Node_id`	-	-	-	-	-	✓

R - Reachability; W - Waypointing; RPL - Restricted Path Length;
PP - Path Preference; DP - Disjoint Paths; LF - Loop Freedom.

Table 3.4 – Usage of telemetry fields by network devices and in reports sent to the control plane application.

	Devices		Reports	
Field/Variable	Ingress Device	Egress Device	End-of-Epoch	Loop
`Epoch`	✓	✓	✓	✓
`Path_src`	-	-	✓	✓
`Path_length`	-	-	✓	-
`Path_code`	-	-	✓	-
`Path_dst`	-	-	✓	✓
`Flow_id`	-	✓	✓	✓
`Loop_identifier`	-	-	-	✓
`Loop_detected`	-	-	-	-
`Node_id`	✓	✓	-	✓

headers. In this network, should the operator be interested in verifying all properties for a flow of interest, all packets from this flow would carry all headers available for (the current proof-of-concept implementation of) VERMONT. As seen in Figure 3.7, the packet contains all fields previously mentioned for VERMONT's properties verification, besides the regular packet headers. The specification of each of these headers (i.e, the fields that compose each of them) is presented in Section 3.2.

3.2.4 Finding Violations and Analyzing Reports

The previous subsections shed light on mechanisms employed in the data plane to enable live monitoring of network properties. They ultimately result in reports sent to the control plane application. In this subsection, we describe how VERMONT analyzes these reports in the control plane to detect property violations.

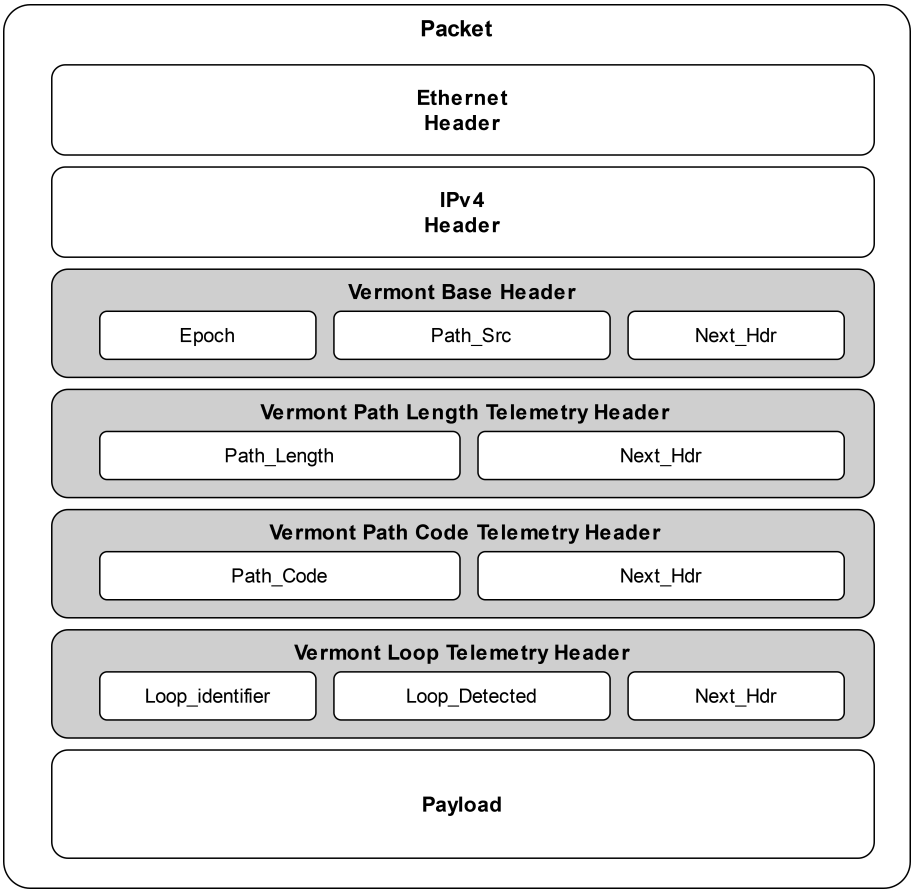


Figure 3.7 – Complete telemetry header embedded into an Ethernet/IPv4 packet. From the author.

Reachability. To identify a violation in reachability, VERMONT monitors arriving report packets from a flow of interest. A violation is determined by a fixed amount of time that separates the arrival of two consecutive reports containing information about the flow. We note that reports *are not* generated whenever there are problems inside the network (e.g., misconfigured rules, contentions) that prevent the flow of interest from reaching its destination. Algorithm 1 formalizes how VERMONT processes arriving reports to detect reachability violations. The proposed detection mechanism is based on a countdown timer instantiated when the property is first defined. This timer is reset upon the arrival of any report regarding the flow of interest associated with the property. If the timer expires (`onTimerExpiration` function), VERMONT immediately generates an alert indicating the violation. The value of parameter `maxInterval` can be adjusted considering the maximum acceptable duration for disruptions to the flow of interest.

Waypointing. To identify the devices visited by a packet, VERMONT uses the path tracing mechanism described in Section 3.2.2. The PathID value stored in each packet (of a flow of interest) is retrieved on the egress node. This value is stored and each unique value is sent once to the control plane at the end of an epoch. With them, the control

Algorithm 1 Pseudo-algorithm executed by the control plane application to detect the violation of the Reachability property.

```
Require: Report, Flows Of Interest, Timer
1: function onPropertyStart(maxInterval)
2:     Timer.start(maxInterval)
3: function onReportReceived
4:     Timer.reset()                          ▷ No violations detected
5: function onTimerExpiration
6:     Notification.raise()                   ▷ Reachability violation detected
7:     Timer.reset()
```

plane application is able to infer the specific route the packets from that flow took and verify waypoints. As an example, consider flow A has two ways to reach its destination. If half of the packets pertaining to flow A used the first route and the other half used the second route, the control plane would receive two report packets, indicating each one of the routes. Algorithm 2 illustrates the processing of the reports to detect waypointing violations.

Algorithm 2 Pseudo-algorithm executed by the control plane application to detect the violation of the Waypointing property.

```
Require: Report, Flows Of Interest, RequiredWaypoint
1: function PathDecoder(PathID)
2:     Return Decode(PathID)
3: function onReportReceived
4:     Path ← PathDecoder(Report.PathID)
5:     if RequiredWaypoint ∈ Path then
6:         Pass                               ▷ No violations detected
7:     else
8:         Notification.raise()               ▷ Waypointing violation detected
```

Restricted Path Length. Packets carry a field (`Path_length`) that counts the number of devices that were visited. This field is part of the path-tracing mechanism described earlier. The verification consists simply of comparing the value of this field with the one specified in the property definition. Algorithm 3 formalizes how VERMONT processes and detects violations to the Restricted Path Length property.

Path Preference. VERMONT is able to infer the path taken by a flow of packets with the path-tracing mechanism. With the path the packet took, one must only compare it against the desired path to verify if this property held or not. In Algorithm 4, we illustrate the processing made by VERMONT when receiving a (report) packet from a flow of interest that has the property Path Preference applied.

Disjoint Paths. VERMONT can identify the devices visited by each flow through

Algorithm 3 Pseudo-algorithm executed by the control plane application to detect the violation of the Restricted Path Length property.

```
Require: Report, Flows Of Interest, MaxPathLength
1: function onReportReceived
2:     PathLength ← Report.Path_length
3:     if PathLength ≤ MaxPathLength then
4:         Pass                                          ▷ No violations detected
5:     else
6:         Notification.raise()          ▷ Restricted Path Length violation detected
```

Algorithm 4 Pseudo-algorithm executed by the control plane application to detect the violation of the Path Preference property.

```
Require: Report, Flows Of Interest, ExpectedPath
1: function PathDecoder(PathID)
2:     Return Decode(PathID)
3: function onReportReceived
4:     Path ← PathDecoder(Report.PathID)
5:     if Path = ExpectedPath then
6:         Pass                                          ▷ No violations detected
7:     else
8:         Notification.raise()              ▷ Path Preference violation detected
```

the path-tracing mechanism. Verification then consists in computing the set intersection. If the intersection is the empty set, the property is satisfied. Otherwise, violated. Algorithm 5 formalizes the verification executed by VERMONT when receiving reports from the flows of interest.

Algorithm 5 Pseudo-algorithm executed by the control plane application to detect the violation of the Disjoint Paths property.

```
Require: Reports, Flows Of Interest
1: function PathDecoder(PathID)
2:     Return Decode(PathID)
3: function onReportReceived
4:     Path1 ← PathDecoder(Reports.Flow1.PathID)
5:     Path2 ← PathDecoder(Reports.Flow2.PathID)
6:     if Path1 ∩ Path2 = ∅ then
7:         Pass                                          ▷ No violations detected
8:     else
9:         Notification.raise()              ▷ Disjoint Paths violation detected
```

Loop Freedom. The Loop Freedom property differs from the others since its verification is carried out entirely in the data plane. The detection of loops is done using the `Loop_identifier` field in the telemetry header. This field acts as a bitmap, where each index corresponds to a unique forwarding device in the network. Whenever a device receives a packet, it checks if its respective index in the field is unmarked. In the negative case (i.e., the index is unmarked), the device marks the field on its corresponding index,

enabling future detection should the packet return to itself. In the positive case, where the index is already marked, it means that this particular packet has already visited the device previously (i.e., the packet has been forwarded via a loop). When a loop is detected, a report is immediately generated containing the fields indicated in Table 3.2. In the control plane application, loop reports are analyzed to pinpoint which device(s) forwarded the packet via unexpected port(s). Since the control plane verification is merely an indication of the device which incorrectly forwarded the packets, its algorithm was suppressed.

4 IMPLEMENTATION AND EVALUATION

This chapter presents the main implementation aspects of our proof-of-concept prototype, as well as provide sufficient resources to characterize VERMONT's performance. The focus of the implementation presented in this chapter is on the data plane, as the majority of VERMONT's code resides there. The subsequent evaluation is done using this prototype, presenting its performance and resource usage and comparing it to other relevant approaches.

4.1 Implementation

A series of constructs were produced to develop the prototype for VERMONT. These constructs enable the multiple features presented earlier in Chapter 3. We introduce the constructs implemented in P4 for execution on network switches. The implementation of the Python-enabled constructs in the data plane is straightforward and follows the specification in Section 2.3 rigorously, and, as such, is not presented. Any relation these constructs have with the features already mentioned is explained as each construct is introduced.

The presentation of each of the developed constructs is presented in the order they appear in our P4 implementation. In Figure 4.1, we present an overview of how the algorithms are organized inside a switch that implements VERMONT. The steps are executed from top to bottom, left to right. We begin with the implementation of the headers, then we move to the Ingress Pipeline and we end later in the Egress Pipeline.

The P4 constructs used for the prototype implementation of VERMONT begin with the declaration of the headers used by our entire program. We implemented two main headers used in every possible VERMONT-enabled scenario and other headers tailored-made for multiple different scenarios. The first header we introduce is VERMONT's base telemetry header. This telemetry header comprises the epoch that the packet reached the network, the source of the packet, and a code indicating the following header in the packet. In Algorithm 6, we present its implementation in P4.

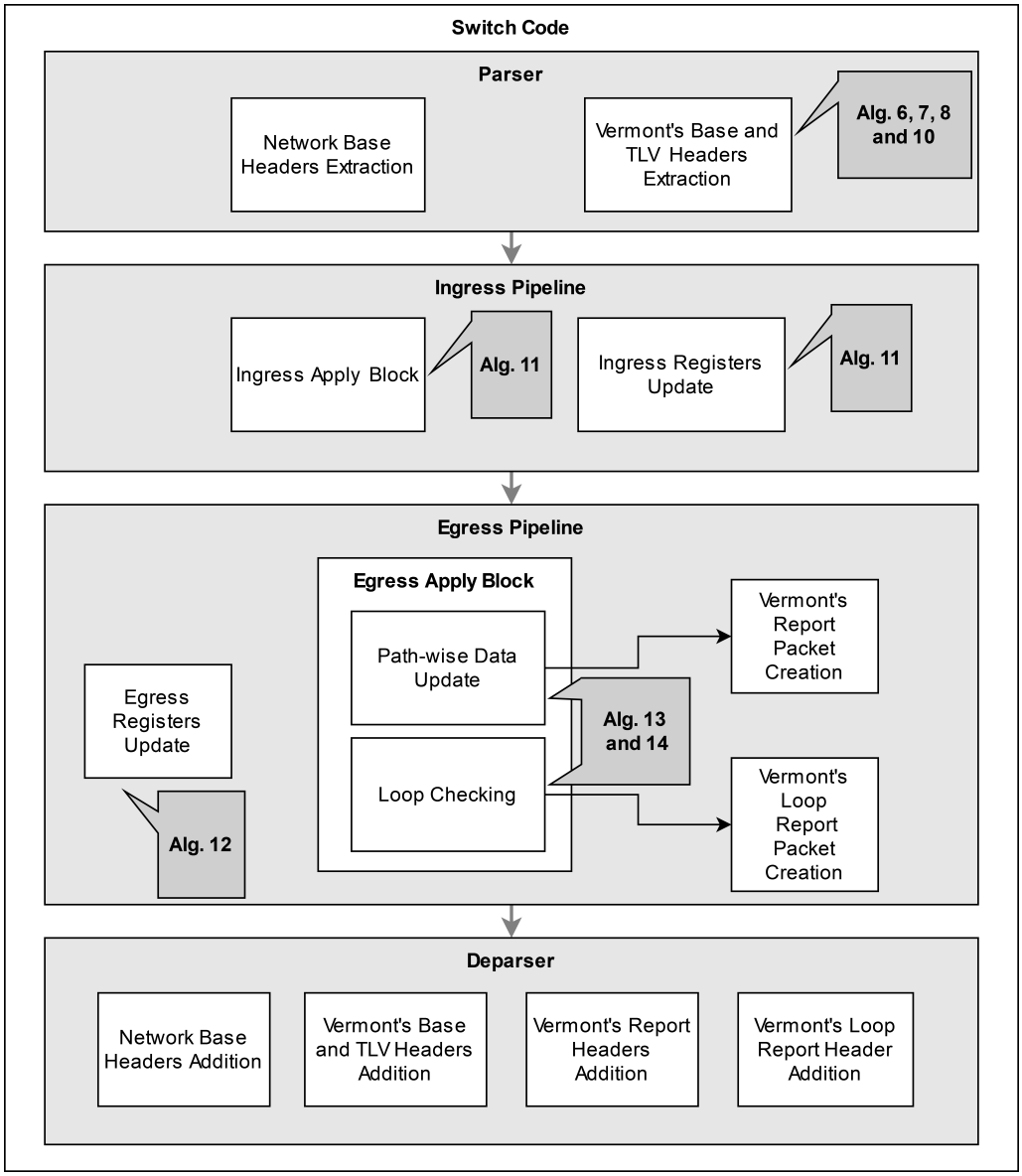


Figure 4.1 – VERMONT enabled switch pipelines. From the author.

Algorithm 6 VERMONT's base telemetry header.

```
1:  #include <core.p4>                                  ▷ Includes of the program
2:  #include <v1model.p4>
3:
10: header ethernet_h {...}                            ▷ Headers used in the network
16: header arp_h {...}
28: header ipv4_h {...}
45:
46: header vermont_base_telemetry_header {
47:     bit <32> epoch;
48:     bit <10> path_src;
49:     bit <8> next_header;
50: }
51: ...                                                ▷ Vermont't TLV Headers
```

As different scenarios emerge in the possible uses of VERMONT, new headers are necessary to indicate different aspects of the traversal of the packet through the network. In Section 3.2.3, we introduced how the TLV technique is used in our approach for using only the exact amount of bytes inside each packet. As such, every property requires a different set of telemetry fields to be collected (to be verified). In Algorithm 7 we present the implementation of each of these headers, one by one. When using single-value TLV headers, `vermont_base_telemetry_header` is composed by another field, called `length`, which indicates the amount of TLV fields a packet is holding. Each field is preceded by another field called type, which indicates the next field presented.

Algorithm 7 VERMONT's base TLV headers.

```
51: ...                          ▷ Vermont's base header and other headers used in the network
52:
53: header vermont_pathlength_telemetry_header {
54:     bit <8> type;
55:     bit <6> path_length;
56: }
57:
58: header vermont_pathcode_telemetry_header {
59:     bit <8> type;
60:     bit <6> path_code;
61: }
62:
63: header vermont_loopidentifier_telemetry_header {
64:     bit <8> type;
65:     bit <6> loop_identifier;
66: }
67: header vermont_loopdetection_telemetry_header {
68:     bit <8> type;
69:     bit <6> loop_detection;
70: }
71:
72: ...                                          ▷ Vermont's report headers
```

In regular TLV headers, each header only comprises one field, but since in our implementation multiple of these fields only appear with at least one other, we chose to aggregate them in our headers. For example, the field `Loop_identifier` will only

appear in a packet along with the field `loop_detected`, and as such, both are aggregated in one header, called `vermont_loop_telemetry_header`. In Algorithm 8, we introduce each of the headers that implement the TLV technique inside VERMONT's prototype. In our implementation, since we combined the headers, these types are not shown, but in an extension of VERMONT, this technique is ready to be used.

Algorithm 8 VERMONT's aggregated TLV headers.

```
51: ...                          ▷ Vermont's base header and other headers used in the network
52:
53: header vermont_pathlength_telemetry_header {
54:     bit <6> path_length;
55:     bit <8> next_header;
56: }
57:
58: header vermont_pathcode_telemetry_header {
59:     bit <16> path_code;
60:     bit <8> next_header;
61: }
62:
63: header vermont_loop_telemetry_header {
64:     bit <16> loop_identifier;
65:     bit <8> loop_detected;
66:     bit <8> next_header;
67: }
68:
69: ...                                                    ▷ Vermont's report headers
```

After the packets reach the end of their journey inside a VERMONT-enabled network and an epoch reaches its end, a report is generated for the control plane. As such, the header corresponding to the information of such report packet is also implemented in our P4 code. In the case of a loop being detected in the network, a loop report is generated. Their implementations are shown in Algorithm 9, and the fields for both report headers were presented earlier in Section 3.2.3. Our implementation is also composed of a structure holding metadata about packets that transfer information from the ingress pipeline to the egress pipeline. This structure merely contains information already in the packet or values to be inserted in registers, and is suppressed in this explanation.

Algorithm 9 VERMONT's report headers.

```
70: ...                                                    ▷ Vermont's TLV headers
71:
72: header vermont_report_header {
73:     bit <32> epoch;
74:     bit <32> egress_epoch;
75:     bit <32> flow_ID;
76:     bit <10> path_src;
77:     bit <6> path_length;
78:     bit <16> path_code;
79:     bit <16> path_dst;
80: }
81:
82: header vermont_report_loop_header {
83:     bit <32> epoch;
84:     bit <32> flow_ID;
85:     bit <10> path_src;
86:     bit <6> path_length;
87:     bit <16> path_code;
88:     bit <8> node_ID;
89:     bit <16> loop_identifier;
90: }
91:
92: struct headers {...}          ▷ Structure containing headers used by the network
102:
103: struct custom_metadata_t{...}                    ▷ Custom metadata structure
173:
174: parser ParserImpl(...) {...}                          ▷ Parser implementation
221: ...                                                              ▷ Pipelines
```

As every packet has its headers parsed by the switch, the parser removes every header in a determined order. The implementation of the parsers for the aggregated headers is suppressed since they follow the standard P4 structure. For the implementation of the base TLV headers, presented in Algorithm 7, the parsers are different, and must parse each field independently. We present a non-exhaustive example for one of the fields used in our implementation in Algorithm 10.

Algorithm 10 VERMONT's parsers for original TLV headers.

```
70: ...                                                    ▷ Vermont's report headers
173:
174: parser ParserImpl(...) {
175:     state parse_hdrs {
176:         cmd.len = cmd.len - 0x1;                      ▷ Amount of fields to be retrieved
177:         bit<8> nexthdr = pkt.lookahead<bit<8»();
178:         transition select(nexthdr){
179:             TYPE_PATH_LENGTH : parse_path_length;
180:             TYPE_PATH_CODE : parse_path_code;
181:             TYPE_LOOP_ID : parse_loop_id;
182:             TYPE_LOOP_DTC : parse_loop_detection;
183:             default : accept;
184:         }
185:     }
186:
187:     state parse_hdrs_aux {
188:         transition select(cmd.len){                   ▷ Check amount of remaining fields
189:             0x0 : parse_next_header;        ▷ If no additional fields, parse next header
190:             default : parse_hdrs;
          }
        }
191:
192:     state parse_path_length{
193:         pkt.extract(hdrs.vermont_pathlength_telemetry_header);
194:         transition parse_hdrs_aux;
        }
195: ...
221: }                                                      ▷ End of parser implementation
261:
262: ...                                                                  ▷ Pipelines
```

Every switch in a VERMONT-enabled network is equipped with a series of registers that allow for the correct functioning of our approach. Some of these registers are for identifying unique switches in the network, maintaining the current epoch value, or storing the ingress epoch value of each flow. These registers are shown in Algorithm 11. The `node_ID` register is set individually for each switch, and the `e_epoch` register is updated every epoch. The value `nflows` before the `i_epoch` register represents the

maximum amount of flows that will be tracked by a single switch in an epoch, and this number may be configured differently for each individual network.

Algorithm 11 VERMONT's base registers.

```
256: ...                                                    ▷ Headers, structures and parser
257: control ingress(...) {                                 ▷ Ingress pipeline
258:
259:     register < bit <10> > (1) node_ID;        ▷ Registers used in the ingress pipeline
260:     register < bit <32> > (1) e_epoch;
261:     register < bit <32> > (nflows) i_epoch;
262:
263:     action drop() {...}
271:
272:     action ipv4_forward(...) {...}
275:
276:     table ipv4_lpm {...}
287:
288:     action set_flow_ID(...) {...}
291:
292:     table flow_ID {...}
303:
308:
309:     apply {                                             ▷ Ingress apply block
310:         ...                                             ▷ Ingress workflow
360:     }
361: }
362:
363: ...                                                     ▷ Egress pipeline
```

The switches are also equipped with registers to hold information about the flows that use them as their egress nodes. As such, every switch has a set of these registers that store data used to be reported to the control plane at the end of an epoch. Algorithm 12 presents these registers. After the generation of an end-of-epoch report, these registers are cleaned and reset.

As the packet travels through the network, its `path_code` field is updated as explained in Section 3.2.2. For this task, the control plane inserts the corresponding tables for match-action into the switches at runtime, and each switch updates the value using these tables. Algorithm 13 presents the implementation of this match-action stage, con-

Algorithm 12 VERMONT's egress registers.

```
367: ...                                                          ▷ Ingress pipeline
368:
369: control egress(...) {                                        ▷ Egress pipeline
370:
371:     register < bit <32> > (nflows) e_egress_epoch;
372:     register < bit <10> > (nflows) e_path_src;
373:     register < bit <6> > (nflows) e_path_length;
374:     register < bit <16> > (nflows) e_path_code;
375:
376:     ...                                          ▷ Egress tables and actions
```

taining the table and the action used for this behavior. In this algorithm, lines 478 to 480 represent the action that executes the modification of the `path_code` field, while lines 482 to 493 present the table used and its keys.

Algorithm 13 VERMONT's match-action stage for path code updates.

```
389: ...                                          ▷ Egress pipeline start and registers
390:
391:     action rewrite_mac_addrs(...) {...}
406:
407:     table mac_addrs {...}
417:
478:     action set_path_ID(bit<16> new_path_code) {
479:         hdrs.telemetry.pathcode.path_code = new_path_code;
480:     }
481:
482:     table update_path_ID {
483:         key = {
484:             hdrs.telemetry.base.path_src: exact;
485:             hdrs.telemetry.pathlength.path_length: exact;
486:             hdrs.telemetry.pathcode.path_code: exact;
487:         }
488:         actions = {
489:             set_path_ID;
490:             NoAction;
491:         }
492:         default_action = NoAction();
493:     }
494:
495:     ...                                          ▷ Egress pipeline apply block
```

Most of VERMONT's implementation resides in the egress pipeline. These behaviors are presented in Algorithm 14. First, the node checks whether it is the packet's ingress node. The switch initializes the telemetry header to base values if it is. Then, the switch updates the `path_code` field and increases the `path_length` field. The next step is to check if this packet has already visited this specific node. If it has, VERMONT immediately generates a loop report using available data. In this algorithm, line 587 executes the behavior presented in Algorithm 13, while lines 588 and 589 increase the value of the field `path_length`. In lines 591 and 592, VERMONT verifies, through a binary OR operation, if this node was already visited.

Algorithm 14 VERMONT's path-tracing mechanism update and loop detection.

```
526: ...                                                    ▷ Egress tables and actions
527:
528:     apply {                                                ▷ Egress apply block
529:         ...                                            ▷ Other egress algorithms
530:
587:         update_path_ID.apply();
588:         hdrs.telemetry.pathlength.path_length =
589:                 hdrs.telemetry.pathlength.path_length + 1;
590:         cmd.loop =
591:             hdrs.telemetry.loop.loop_identifier
592:                 | ((bit<16>) 1) « ((bit<8>) cmd.node_ID);
593:
594:         ...                                            ▷ Other egress algorithms
904:     }                                                      ▷ End of apply block
905: }                                                       ▷ End of egress pipeline
906:
907: ...                                                        ▷ Deparser and EOF
```

As soon as these steps are complete, VERMONT updates every presented register with the information of this packet, considering its flow identification. The last step is to check whether a new epoch has started. If it has, VERMONT generates an end-of-epoch report and cleans all suitable registers. Since these updates are simple modifications, their implementations are not presented.

4.2 Evaluation

We evaluated VERMONT using three sets of experiments. The first set evaluated the effectiveness of VERMONT in detecting property violations in a realistic network scenario. The second set compares the scalability of VERMONT with a state-of-the-art approach, Minesweeper (BECKETT et al., 2017). The third set focuses on the evaluation of costs related to deploying VERMONT at scale, considering six representative WAN topologies. In the following paragraphs, we present our testbed, describe the motivation for our evaluation, and present the steps taken in each set of experiments.

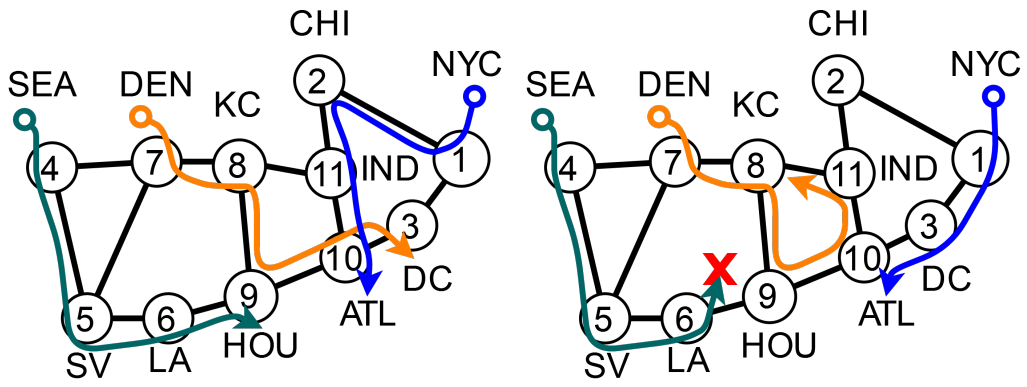
As previously presented, we developed a prototype of VERMONT for the reference P4 software switch (BMv2 – Behavioral Model version 2). Our prototype is available at (VASSOLER; MARQUES; GASPARY, 2022). We used as a testbed Mininet-emulated networks along with BMv2. Evaluation was carried out on an overprovisioned Ubuntu 20.04 (Linux) machine with an Intel Xeon Gold 6253CL CPU @ 3.1GHz 30-core 30-thread processor, 120GB RAM, and 30GB SSD storage, which minimizes noises associated with the experimental toolset. Network traffic was generated using tcpreplay, composed of synthetic UDP traces. VERMONT was configured with an epoch length of 2^{16} = $\approx$65 milliseconds.

4.2.1 Verifying Properties in a WAN: The Case of Abilene

We performed a series of experiments representing real world scenarios in order to evaluate the effectiveness of VERMONT. To this end, consider the flows and topology illustrated in Figure 4.2a. We chose the Abilene topology for this evaluation. In the context of monitoring this infrastructure, we assume three possible property violations. The first is the reachability property, applied to the flow Teal. For this property, at least one packet of the flow Teal coming from SEA and addressed to HOU must indeed reach HOU at every epoch. In this example, the assigned entry point of this flow is Node 4 and the leaving point is Node 9. The second property of interest is Loop Freedom, applied to the flow Orange in the figure. For this property, it is expected that the packets entering the network from DEN (through Node 7) and leaving the network through DC (through Node 3) do not experience any loops. The third and last property of interest is Waypointing, applied to the flow Blue in the figure. This property expects that the packets of the flow will, at any given time, reach Node 11 on their way from NYC to ATL.

To generate failures that would lead to property violations, we configure incorrect rules in points of the network, making some devices not work as expected. This setup is exemplified in Figure 4.2b. Specifically, Node 6 has problems in its forwarding interfaces, dropping all packets received during random windows of time, while Nodes 1 and 10 have misconfigured forwarding rules and send packets to locations different from the expected ones. As a result, the network violates the reachability property for flow Teal, given that its packets are discarded by Nodes 6. The network also violates the Loop Freedom property for flow Orange, as its packets are trapped in a loop in the network (i.e., 7-8-9-10-11-8). Lastly, the waypoint property is violated for flow Blue, as its packets are routed through Nodes 1, 3, and 10, but not Node 11.

Next, we detail how these properties are verified. As soon as packets from flow Teal enter Node 6, they are discarded, and VERMONT does not generate reports with its telemetry data. Table 4.1 details the reports received by VERMONT during its execution. The table shows a gap in the epochs for flow Teal between epochs 324 and 331. Assuming the property has parameter `maxInterval` set as the length of one epoch, this represents a reachability violation. In the case of flow Orange, when the packets return to Node 8, which reads the `Loop_identifier` field in the telemetry header, the device identifies a loop. The table presents the path taken by the packets and the device that reported the loop. The control plane, receiving the notification, verifies that a loop was detected. Finally, packets from flow Blue never visit Node 11, and VERMONT end-of-epoch reports relay this information using the `Path ID` tuple. The control plane identifies a violation in waypointing by noting, as seen in the table, that the `path_code` field changed its value between epochs 323 and 343, indicating a different path for the flow that does not include Node 11.



(a) Expected scenario (b) Actual scenario

Figure 4.2 – Flows and topology used for the evaluation. From the author.

Table 4.1 – Received reports by VERMONT's control plane.

Flow	Epoch	Path Src	Path Dst	Path Code	Loop Id.	Rep. Node
Teal	323	4	9	1	-	-
Teal	324	4	9	1	-	-
Teal	331	4	9	1	-	-
Teal	332	4	9	1	-	-
Blue	323	1	10	1	-	-
Blue	343	1	10	2	-	-
Orange	325	7	3	-	7, 8, 9, 10, 11	8

4.2.2 Scalability Analysis: VERMONT vs. Minesweeper

Following our evaluation demonstrating the effectiveness of VERMONT, we also evaluated its scalability regarding processing demands. Next, we present VERMONT's computation time for varying workloads and compare it to Minesweeper. By extending the previously presented scenario, we chose an increasing number of properties to be simultaneously verified by both approaches. The workload consists of pseudo-randomically generated properties of varying types associated with traffic in the network. As seen in Figure 4.3, by comparing our achieved results to Minesweeper's, VERMONT verifies all properties in a few seconds, while Minesweeper takes minutes to finish the verification for a scenario with 10^3 properties. This shows that Vermont is at least one order of magnitude faster than Minesweeper. These results are explained by the approaches taken by VERMONT and Minesweeper. Being a dynamic verification approach, VERMONT only analyzes pieces of information pertaining to actual traffic flowing through the network. Minesweeper is a static verification approach, and as such, has the necessity to evaluate all possible paths, leading to the branch explosion problem.

4.2.3 Resource Usage: Network and Physical Devices

To conclude our evaluation, we explore other resource costs resulting from VERMONT's approach to property verification. We chose two main aspects to explore the other costs involved in the execution of our INT-based approach. These are the memory used by VERMONT on the network devices and its packet header overhead.

To create a robust evaluation, we applied our approach to multiple networks. We chose to employ the REPETITA dataset (GAY; SCHAUS; VISSICCHIO, 2017), consist-

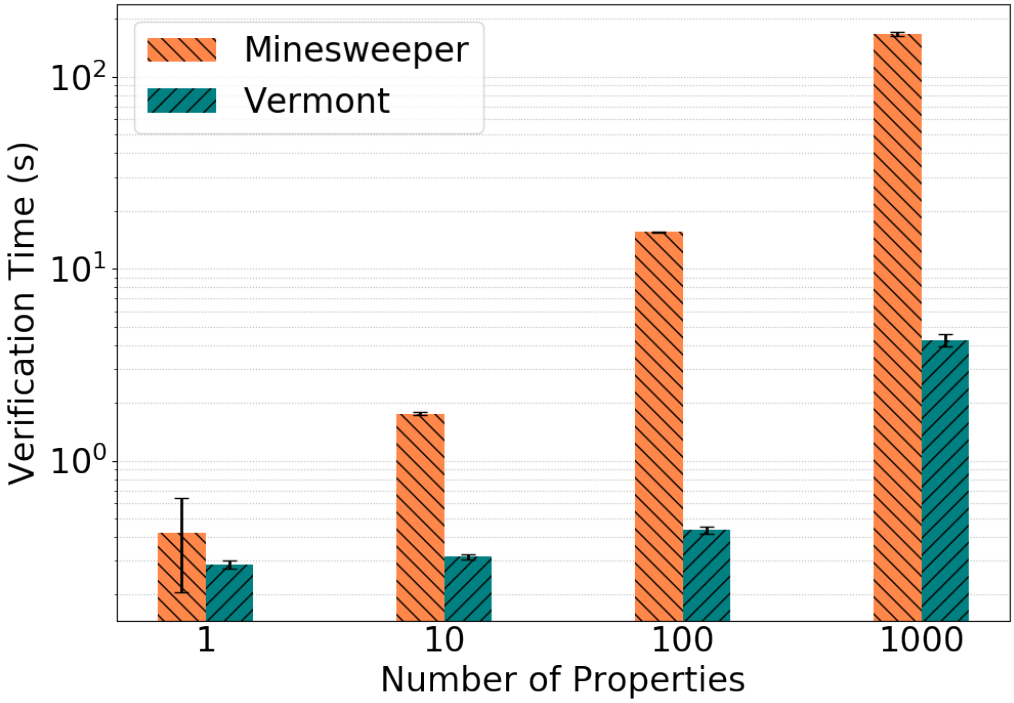


Figure 4.3 – Time to verify properties. From the author.

Table 4.2 – Metadata for the network topologies used on the performance evaluation.

Network	Label	Nodes	Links	Avg. Path Length
Bellcanada	BC	48	130	5.3
Us Signal	US	61	158	6.0
VTLWavenet	VW	92	192	13.1
TATA	TT	145	388	9.9
Cogent	CG	197	490	10.5
Sprintlink	SL	315	1944	4.0

ing of more than 260 networks. For clarity, we focus on six representative networks and show VERMONT's resource usage on them. The details of each used network are presented in Table 4.2. The network topologies vary from 48 to 315 nodes and 130 to 1,944 links. We present the results in Table 4.3, which were obtained via analytical modeling. Our models considering the memory usage of an implementation in P4 of VERMONT targets the RMT architecture (BOSSHART et al., 2013), while the resource usage concerning the packet overhead considered a standard Ethernet Maximum Transmission Unit (MTU). Every network has an associated number of packets per second (pps) that flow through it. These values vary from 831,790 pps in the smallest network to 19,253,769 pps. The values for every used network can be found in the dataset, and those were the values used for the analytical models.

Memory Usage. Table 4.3 presents the amount of memory used by each device of the previously mentioned networks when applying VERMONT. We focused our efforts on assessing the amount of Ternary Content Addressable Memory (TCAM) and Static

Table 4.3 – VERMONT's memory (Mbits and %RMT) and header space (Bytes and %MTU) usage.

Network	SRAM	TCAM	Header Space Usage
BC	9.36 (2.53%)	0.29 (0.72%)	34 (2.26%)
US	11.98 (3.24%)	0.37 (0.92%)	36 (2.4%)
VW	18.5 (5.00%)	0.65 (1.62%)	40 (2.66%)
TT	29.95 (8.09%)	1.18 (2.95%)	48 (3.2%)
CG	40.77 (11.01%)	1.61 (4.02%)	54 (3.6%)
SL	67.18 (18.16%)	2.89 (7.22%)	69 (4.6%)

Random Access Memory (SRAM) that VERMONT demands. In the largest network (SL), VERMONT used 2.89Mb TCAM and 67.18Mb SRAM, representing 7.22% and 18.16% of the amount available. These results demonstrate the low memory requirement, even considering the use of this resource grows proportionally to the network size.

Packet Header Overhead. Table 4.3 also presents the header space required to run VERMONT for each evaluated network. It is important to note that this is the number of bytes demanded in each packet when all properties are verified for one flow, i.e., the worst-case scenario. The maximum value was 69 bytes, representing only 4.6% of a standard Ethernet MTU. Packets of flows for which fewer properties are to be monitored will be even less penalized. This is due to the telemetry header being tailor-made and dependent on network size.

4.3 Discussion and Limitations

In this section, we discuss important topics related to VERMONT's design, implementation, and limitations.

Verifying properties using production traffic. Using production traffic to monitor its associated properties enables unparalleled accuracy and representativeness for network verification. However, doing so creates a dependence on traffic to verify properties (i.e., a property is only verified when the associated flow of interest is present in the network). As such, we consider VERMONT to be an on-demand property verification approach, where the violations are reported as soon as they happen to production traffic. We note that a more speculative analysis (e.g., trying to evaluate, ahead of time, whether violations would arise should traffic be present) could be performed by introducing probing to our proposed approach, which we leave as future work.

Gradual deployment. In VERMONT's design, we assume that all devices from the

network are equipped with monitoring capabilities enabled by data plane programmability. Devices not compatible with VERMONT's approach are left as grey boxes and will not interfere with the functioning of the approach since our proposed custom header is in Layer-4. Strategically placing VERMONT's enabled boxes in a network is feasible and would work for a variety of properties. Still, a more extensive analysis of VERMONT's trade-offs in such scenarios is left as future work.

Porting our prototype to other targets. Although VERMONT was developed for P4-enabled switches, applying our approach to other targets is possible and highly feasible. We focused our efforts on developing VERMONT using P4 constructs that are readily available in most, if not all, known programmable platforms. As such, targeting other devices should be achievable with small modifications to our P4 code.

From network policies to traffic properties. In this work, the starting point is the formal definition of available properties. From this definition, we were able to equip VERMONT with assets capable of verifying these properties. We recognize that when deployed, using this level of abstraction to configure VERMONT may not be the most appropriate. We advocate that the network operators would benefit from a higher-level solution that could ease the translation of their needs to the features and properties supported by VERMONT. As such, a graphical user interface (GUI), a natural-based, or an intent-based language approach could benefit the end users of our solution. This venue of research is also left as future work.

Security assurances. This work assumes that every device inside the network is not compromised and has correct functioning/responses when regarding VERMONT. When one or more devices are not working properly with VERMONT, *i.e.*, do not execute their actions regarding flows of interest or execute spurious operations due to malice, VERMONT's effectiveness can not be assured. This happens because every device in the network that is working alongside VERMONT has one or more non-transferable responsibilities regarding one or more flows of interest. We leave the study of these implications and workarounds with this issue open as future work.

5 CONCLUSION

In this thesis, we proposed VERMONT, an approach that uses INT-based monitoring techniques to verify networks in production time. During our work, we concluded that proper verification of network-wide properties done dynamically (and using inputs from the network) is feasible and efficient. Our efforts show that an approach that considers solely the data collected in the data plane as the packets traverse the network is enough to evaluate whether a property is being violated or not in a given window of time. This approach also does not rely on probing the network for the inputs and uses only production packets.

Using monitoring campaigns in the data plane, VERMONT is able to collect data from the packets of determined flows of interest. This data is summarized as reports, which are sent to the control plane and used to verify properties. These reports are generated whenever a window of time (also called epoch) ends or a crucial violation is detected in the network (loops). By summarizing the data generated in the network, we were able to reduce the total amount of packets sent to the control plane for analysis. This summarization is done in the data plane, as the egress devices of each flow collect and aggregate monitored data.

Our approach presents fast verification time, being at least one order of magnitude faster than a widely recognized and relevant counterpart, Minesweeper (BECKETT et al., 2017). Our work represents substantial advances in verification time when compared to static verifiers. It, therefore, opens an avenue for research of next-generation verification approaches, monitoring properties continuously and with low resource demands.

During the evaluation, we observed that our approach has a modest resource usage, requiring less than 20% of the total SRAM of a modern programmable device (i.e., RMT) in the largest evaluated network and less than 5% of the standard MTU. For a large network infrastructure, it is expected that our approach will behave similarly to the largest evaluated scenario and should not be impaired at all by the usage of VERMONT.

We also consider our approach to be a complement of the static verification approach rather than a direct substitute. The main idea is to use VERMONT as an auxiliary program that verifies violations arising at runtime, while other approaches (BECKETT et al., 2017; STEFFEN et al., 2020) are used prior to the execution to eliminate possible errors created beforehand.

As future work, we plan on evolving VERMONT to go beyond detection by also

correcting violations found during runtime, exploring mitigation aspects of these violations while maintaining proper verification at all times. VERMONT can also be expanded to incorporate visualization features that allow for the network operators to promptly identify violations as they arise in a front-end system. As a last venue of future work, VERMONT could be automated to identify and configure itself to any network. In our proof-of-concept design, this process was carried out manually, but such an approach would not be practical in real setups.

REFERENCES

ALBAB, K. D. et al. Switchv: automated sdn switch validation with p4 models. In: **Proceedings of the ACM SIGCOMM 2022 Conference**. [S.l.: s.n.], 2022. p. 365–379.

BASAT, R. B. et al. **PINT: Probabilistic In-band Network Telemetry**. 2020.

BECKETT, R. et al. 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: Association for Computing Machinery, 2017. (SIGCOMM '17), p. 155–168. ISBN 9781450346535. Available from Internet: <https://doi.org/10.1145/3098822.3098834>.

BOSSHART, P. et al. P4: Programming protocol-independent packet processors. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 3, p. 87–95, jul. 2014. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/2656877.2656890>.

BOSSHART, P. et al. Forwarding metamorphosis: Fast programmable match-action processing in hardware for sdn. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 43, n. 4, p. 99–110, aug 2013. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/2534169.2486011>.

BROCKNERS, F. et al. **Requirements for In-situ OAM**. [S.l.], 2017. Work in Progress. Available from Internet: <https://datatracker.ietf.org/doc/draft-brockners-inband-oam-requirements/03/>.

BüLBüL, N. S.; ERGENç, D.; FISCHER, M. Towards sdn-based dynamic path reconfiguration for time sensitive networking. In: **NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium**. [S.l.: s.n.], 2022. p. 1–9.

DUMITRESCU, D. et al. Bf4: Towards bug-free p4 programs. In: **Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication**. New York, NY, USA: Association for Computing Machinery, 2020. (SIGCOMM '20), p. 571–585. ISBN 9781450379557. Available from Internet: <https://doi.org/10.1145/3387514.3405888>.

FANTOM, W. et al. A neat way to test-driven network management. In: **NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium**. [S.l.: s.n.], 2022. p. 1–5.

FEAMSTER, N.; REXFORD, J.; ZEGURA, E. The road to sdn: An intellectual history of programmable networks. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 44, n. 2, p. 87–98, abr. 2014. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/2602204.2602219>.

FREIRE, L. et al. Uncovering bugs in p4 programs with assertion-based verification. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2018. (SOSR '18). ISBN 9781450356640. Available from Internet: <https://doi.org/10.1145/3185467.3185499>.

FREIRE, L. et al. Uncovering bugs in p4 programs with assertion-based verification. In: **Proceedings of the Symposium on SDN Research**. New York, NY, USA: Association for Computing Machinery, 2018. (SOSR '18). ISBN 9781450356640. Available from Internet: <https://doi.org/10.1145/3185467.3185499>.

GAY, S.; SCHAUS, P.; VISSICCHIO, S. **REPETITA: Repeatable Experiments for Performance Evaluation of Traffic-Engineering Algorithms**. arXiv, 2017. Available from Internet: <https://arxiv.org/abs/1710.08665>.

KFOURY, E. F.; CRICHIGNO, J.; BOU-HARB, E. An exhaustive survey on p4 programmable data plane switches: Taxonomy, applications, challenges, and future trends. **IEEE Access**, Institute of Electrical and Electronics Engineers (IEEE), v. 9, p. 87094–87155, 2021. Available from Internet: <https://doi.org/10.1109\%2Faccess.2021.3086704>.

KHURSHID, A. et al. {VeriFlow}: Verifying {Network-Wide} invariants in real time. In: **10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)**. [S.l.: s.n.], 2013. p. 15–27.

KIM, C. et al. In-band network telemetry via programmable dataplanes. In: **ACM SIGCOMM**. [S.l.: s.n.], 2015.

LI, Y. et al. A survey on network verification and testing with formal methods: Approaches and challenges. **IEEE Communications Surveys and Tutorials**, v. 21, n. 1, p. 940–969, 2019.

LIU, J. et al. P4v: Practical verification for programmable data planes. In: **Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication**. New York, NY, USA: Association for Computing Machinery, 2018. (SIGCOMM '18), p. 490–503. ISBN 9781450355674. Available from Internet: <https://doi.org/10.1145/3230543.3230582>.

MARQUES, J. A.; LEVCHENKO, K.; GASPARY, L. P. Intsight: Diagnosing slo violations with in-band network telemetry. In: . New York, NY, USA: Association for Computing Machinery, 2020. (CoNEXT '20). To Appear.

MCKEOWN, N. et al. Openflow: Enabling innovation in campus networks. **SIGCOMM Comput. Commun. Rev.**, Association for Computing Machinery, New York, NY, USA, v. 38, n. 2, p. 69–74, mar. 2008. ISSN 0146-4833. Available from Internet: <https://doi.org/10.1145/1355734.1355746>.

NEVES, M. et al. Verification of p4 programs in feasible time using assertions. In: **Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies**. New York, NY, USA: Association for Computing Machinery, 2018. (CoNEXT '18), p. 73–85. ISBN 9781450360807. Available from Internet: <https://doi.org/10.1145/3281411.3281421>.

P4ORG. In-band network telemetry (int) dataplane specification. P4ORG, 2020.

SHUKLA, A. et al. P4consist: Toward consistent p4 sdns. **IEEE Journal on Selected Areas in Communications**, v. 38, n. 7, p. 1293–1307, 2020.

SHUKLA, A. et al. Runtime verification of p4 switches with reinforcement learning. In: **Proceedings of the 2019 Workshop on Network Meets AI & ML**. [S.l.: s.n.], 2019. p. 1–7.

SILVA, A. S. da; SCHAEFFER-FILHO, A. Networds: Enabling the understanding of network property violation occurrences. In: **NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium**. [S.l.: s.n.], 2022. p. 1–9.

SOURI, A. et al. A systematic literature review on formal verification of software-defined networks. **Transactions on Emerging Telecommunications Technologies**, v. 31, n. 2, p. e3788, 2020. E3788 ETT-18-0271.R3. Available from Internet: <https://onlinelibrary.wiley.com/doi/abs/10.1002/ett.3788>.

STEFFEN, S. et al. Probabilistic verification of network configurations. In: **Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication**. New York, NY, USA: Association for Computing Machinery, 2020. (SIGCOMM '20), p. 750–764. ISBN 9781450379557. Available from Internet: <https://doi.org/10.1145/3387514.3405900>.

STOENESCU, R. et al. 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.

TIAN, B. et al. Aquila: a practically usable verification system for production-scale programmable data planes. In: **Proceedings of the 2021 ACM SIGCOMM 2021 Conference**. [S.l.: s.n.], 2021. p. 17–32.

VASSOLER, G.; MARQUES, J. A.; GASPARY, L. P. **VERMONT: An In-band Telemetry-based Approach for Continuous Network Property Verification [https://github.com/gabrielvassoler/Vermont]**. 2022. Available from Internet: <https://github.com/gabrielvassoler/Vermont>.

ZENG, H. et al. A survey on network troubleshooting. **Technical Report Stanford/TR12-HPNG-061012, Stanford University, Tech. Rep.**, 2012.

ZHENG, N. et al. Meissa: scalable network testing for programmable data planes. In: **Proceedings of the ACM SIGCOMM 2022 Conference**. [S.l.: s.n.], 2022. p. 350–364.

APPENDIX A — RESUMO EXPANDIDO

As redes de comunicação devem ser resilientes. Espera-se que eles superem desafios operacionais, mantendo funcionamento adequado. Essas expectativas de comportamento podem ser traduzidas em propriedades de rede. Exemplos de tais propriedades, importantes para a operação de redes são: alcançabilidade de ponta a ponta, *waypointing* de caminho, limite de comprimento de caminho e inexistência de loops (BECKETT et al., 2017). Uma infinidade de problemas pode causar a violação dessas propriedades. Por exemplo, um *bug* em um aplicativo de roteamento pode induzir a criação de caminhos incorretos que não provém alcançabilidade de ponta a ponta. Da mesma forma, um único dispositivo de plano de dados mal configurado pode fazer com que os pacotes sejam encaminhados em um *loop*. Considerando que os serviços e aplicações modernos geralmente são compostos por muitos componentes distribuídos em vários nós folhas ou até mesmo redes, deixar esses tipos de problemas despercebidos pode resultar rapidamente em violações de propriedades de rede, tempo de inatividade prolongado dos serviços e, consequentemente, grande prejuízo financeiro.

Uma área importante para resolver esses problemas é a de verificação de redes. Esta disciplina evoluiu significativamente nos últimos dez anos, com a proposta de diversos trabalhos. Estes trabalhos visam analisar a exatidão das aplicações do plano de controle e do código de *switches*, bem como verificar as propriedades de rede (BECKETT et al., 2017; LIU et al., 2018; DUMITRESCU et al., 2020; STEFFEN et al., 2020; FANTOM et al., 2022; BASAT et al., 2020). A maioria das abordagens existentes funciona apenas com dados estáticos, sendo estes geralmente descrições de topologia, arquivos de configuração e código dos *switches*. Essas abordagens não levam em consideração, no entanto, o tráfego real de rede ou o conteúdo das tabelas de *match-action* no plano de dados em tempo de execução. Como resultado, elas não podem detectar violações de propriedades que podem surgir apenas durante a operação da rede. Essa limitação é exacerbada no contexto das redes definidas por software (SDNs), uma vez que o código e a configuração estão sujeitos a alterações mais frequentes, de forma a lidar com a dinâmica do tráfego e das demandas de serviço. Essas abordagens de verificação estática fornecem uma lacuna de pesquisa considerável, deixando de determinar se as redes operam, ou não, corretamente em tempo de execução, dado um ambiente altamente dinâmico.

Embora a verificação estática das propriedades da rede tenha recebido atenção considerável, a verificação dinâmica baseada em tráfego real foi abordada por poucos

estudos. Abordagens como a proposta por P4Consist (SHUKLA et al., 2020) analisam o tráfego gerado artificialmente na rede para verificar as propriedades especificadas sob demanda por um operador humano. Esse tipo de análise captura uma visão completa do funcionamento interno da rede do ponto de vista de um pacote, mas impõe uma carga significativa à rede. Trabalhos recentes baseados em In-band Network Telemetry (INT) para monitoramento de rede (KIM et al., 2015; MARQUES; LEVCHENKO; GASPARY, 2020; TIAN et al., 2021) sugerem uma maneira mais eficiente e acurada de verificação de propriedades dinâmicas. Essas abordagens de monitoramento permitem visibilidade de alta granularidade sobre o que está acontecendo dentro de uma rede enquanto impõem pouca sobrecarga nos pacotes de produção e não requerem a geraração de sondas de tráfego (*probes*).

Visando aproveitar a oportunidade mencionada para avançar no campo da verificação de propriedades, nesta dissertação, propomos VERMONT, um sistema para monitoramento contínuo de propriedades de rede e detecção rápida de violações em tempo de execução. Usando coleta eficiente de metadados baseada em INT, VERMONT agrega e correlaciona informações para detectar violações em sua fonte e relatá-las a um aplicativo externo de gerenciamento no plano de controle. Mais especificamente, o tráfego de interesse é monitorado por época, e cada pacote monitorado carrega apenas informações essenciais para a verificação, atualizadas à medida que o pacote atravessa a rede. Nos dispositivos de saída, VERMONT decide, com base nas informações contidas nos pacotes, se um pacote de relatório deve ser enviado aos servidores de verificação. Os relatórios gerados são analisados no plano de controle para verificar propriedades como alcançabilidade de ponta a ponta, *waypointing* e inexistência de *loops*.

As *principais contribuições de pesquisa* desta dissertação são:

- Um sistema distribuído de rede capaz de coletar e analisar metadados do plano de dados para monitorar as propriedades da rede em tempo real. Este sistema divide fatias de tempo dentro da rede e é capaz de verificar com precisão se as propriedades definidas estão sendo atendidas ou não.
- *Design* e implementação de um sistema de prova de conceito que permite aos operadores expressar as propriedades da rede que são traduzidas para campanhas de monitoramento baseadas em INT.
- Avaliação do VERMONT e comparação com uma abordagem de ponta, considerando desempenho e custos.

Com base nos resultados obtidos, esta abordagem apresenta tempo de verificação rápido, sendo pelo menos uma ordem de grandeza mais rápida do que sua contraparte mais direta, Minesweeper (BECKETT et al., 2017). Nosso trabalho representa avanços substanciais na verificação quando comparado aos verificadores estáticos, assim como abre caminho para a próxima geração de abordagens de verificação, monitorando propriedades continuamente, com baixa demanda de recursos.

Durante a avaliação, descobrimos que nossa abordagem tem um uso de recursos modesto, requerindo menos de 20% da SRAM total de um dispositivo programável moderno (i.e., RMT) na maior rede avaliada e menos de 5% do MTU padrão. Para uma infraestrutura de data center comum, espera-se que nossa abordagem se comporte de maneira semelhante ao maior cenário avaliado e que ela não seja prejudicada pelo uso do VERMONT.

Também consideramos nossa abordagem como um aditivo da abordagem de verificação estática, em vez de um substituto direto. A idéia principal é usar o VERMONT como um programa auxiliar, que verifica violações que surgem em tempo de execução, enquanto outras abordagens (BECKETT et al., 2017; STEFFEN et al., 2020) são usadas antes da execução para eliminar possíveis erros criados na configuração da rede.