TÉCNICO LISBOA

NF Composition Synthesis

Pedro Miguel Drogas Moreira

Thesis to obtain the Master of Science Degree in

Computer Science and Engineering

Supervisors: Prof. Luís David Figueiredo Mascarenhas Moreira Pedrosa
Prof. Fernando Manuel Valente Ramos

Examination Committee

Chairperson: Prof. Daniel Simões Lopes
Supervisor: Prof. Luís David Figueiredo Mascarenhas Moreira Pedrosa
Member of the Committee: Prof. Miguel Filipe Leitão Pardal

October 2023

Declaration
I declare that this document is an original work of my own authorship and that it fulfills all the requirements of the Code of Conduct and Good Practices of the Universidade de Lisboa.

Acknowledgments

First and foremost, I would like to thank Prof. Luís Pedrosa and Fernando Ramos. I am fortunate to have had the privilege of working under their supervision.

I would also like to thank my colleague Francisco Pereira for all his time spent giving me feedback and for his assistance throughout this journey.

Last but not least, would also like to thank my mother and my grandmother for their encouragement and their emotional support, patience and sacrifices they made to make this all possible, as well as all my friends for making this journey a lot better. Thank you!

This work was supported by the SALAD-Nets CMU-Portugal/FCT project (2022.15622.CMU), the European Union (ACES project, 101093126), by national FCT funds (Myriarch project, 2022.09325.PTDC), and INESC-ID (via UIDB/50021/2020).

Abstract

Recent work has been exploring software network functions (NFs), a collection of techniques that enable processing packets entirely in software, allowing commodity servers to replace inflexible network equipment. Processing packets at line-rate has been a challenge, but recent techniques like kernel-bypass have enabled packet processing at throughputs of 10s of Gbit/s and latencies of 100s of ns, making such options competitive with dedicated hardware.

In some cases, however, developers and operators may wish to use an NF that combines features from multiple existing NFs. When no single NF already exists with such a combination of features, they face the challenge of merging code-bases, an often daunting task where they must become familiar with the code-bases of each constituent NF, and effectively build a new one while splicing in and refactoring pieces of code from different sources to merge the desired functionality. This is further compounded by the effort to maintain these code-bases as the constituting NFs evolve their own code-bases over time.

Recent work in verifying NFs [1] explored using symbolic analysis to build a sound and complete representation of the entire behavior of an NF. In that case, this representation was used to check the behavior against a specification for correctness. In this thesis, we propose using this technique to merge of two or more NFs to combine their functionality. In many cases, such functionality can be seen as logically orthogonal and the merger can be fully automated. For more complex cases, we explore novel forms of conflict resolution.

Keywords

Software Network Functions, Program Analysis, Program Synthesis

Resumo

Diversos trabalhos têm explorado funções de rede implementadas em software (NFs) que permitem o processamento de pacotes, permitindo assim que servidores comuns substituam equipamentos de rede inflexíveis. Desenvolver NFs com alta taxa de processamento de pacotes tem sido um desafio, mas técnicas recentes, como o uso de bibliotecas que permitem contornar o kernel, permitiram o processamento de pacotes a taxas de 10s de Gbit/s e latências de 100s de ns, tornando essas opções competitivas com hardware dedicado.

Em alguns casos, no entanto, os desenvolvedores podem desejar usar uma NF que combine recursos de várias NFs existentes. Assim, os mesmos enfrentam o desafio de combinar o código de ambas, uma tarefa bastante complexa na qual precisam de se familiarizar com o código de cada NF constituinte e efetivamente construir uma nova enquanto incorporam e melhoram trechos de código para sintetizar a funcionalidade desejada. Isso é ainda mais complicado pelo esforço necessário para manter o código à medida que as NFs constituintes evoluem o seu próprio código ao longo do tempo.

Trabalhos recentes na verificação de NFs [1] exploraram o uso de análise simbólica para construir uma representação precisa e completa de todo o comportamento de uma NF. Nesse caso, essa representação foi usada para verificar o comportamento em relação a uma especificação de modo a encontrar erros. Nesta tese, propomos usar esta técnica para fundir duas ou mais NFs para combinar as suas funcionalidades. Esta fusão pode ser totalmente automatizada, caso as funcionalidades de ambas as NFs sejam ortoginais. No entanto, para casos mais complexos, desenvolvemos novas formas de resolução de conflitos.

Palavras Chave

Funções de rede em software, Análise de programas, Síntese de programas

Contents

List of Figures

List of Tables

Listings

Acronyms

NF	Network Function
IPS	Intrusion Prevention System
BDD	Binary Decision Diagram
libVig	Vigor's library
DPDK	Data Plane Development Kit
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
IPv4	Internet Protocol version 4
WAN	Wide Area Network
LAN	Local Area Network
DFS	Depth-First Search
SMT	Satisfiability Modulo Theories
SOTA	State-of-the-art

1

Introduction

Contents

Today, both the network control and data planes can be software defined, giving rise to new challenges. At the scale of modern networks, the impact of program and configuration bugs is particularly relevant in that aspect, as a single error can have tremendous impact at a network scale. Recent works used two complementary techniques for proactively finding and preventing bugs in configurations - verification and synthesis. Given a collection of router configurations and a high-level specification of what the network should do, verification aims to ensure that the configurations implement this high-level specification correctly for all possible network behaviors. Synthesis, on the other hand, starts from a high-level specification of what the network should do, aiming to produce a collection of configurations or programs that faithfully implement the specification for all possible dynamic network conditions. Other important challenge resides on Network Function (NF) service chains: typically network traffic traverses a sequence of NFs. For example, a packet may go through a firewall, then through an Intrusion Prevention System (IPS), and then through a load balancer, before reaching its destination. A closer look into these NFs shows that many of them process the packets using very similar processing steps. For example,

most NFs parse packet headers and then classify the packets on the basis of these headers, while some NFs modify specific header fields, or also classify packets based on Layer 7 payload content. In this context, Composing different NFs to obtain a single NF with the same functionality, while gaining from the potential sharing of resources, is a particularly challenging problem we propose to tackle in this thesis.

1.1 Main contribution

With this goal in mind, we developed a NF static analysis tool that automatically *combines* symbolic representations of network functions. The challenge is that in many cases naively merging multiple NFs breaks the original semantics. Indeed, NFs may produce different actions for the same packet states, or even modify the same fields of the packet.

To address this problem, our tool relies on user-defined conflict resolution policies, allowing the developer to define a set of rules to ensure the correctness of the combined functionality.

Our contribution to the network function combination state of the art is, therefore, a tool that allows for an automatic combination of network functions, relying on conflict resolution policies when necessary. As starting point, we leveraged Maestro, a system that provides a complete and sound representation of the constituent NFs, as a byproduct of Vigor's verification process.

1.2 Organization

This dissertation is divided into five chapters.

The following chapter presents background material and several state-of-the-art systems that employ verification, synthesis, and composition, followed by an exposition of the combination of challenges and solutions in this field. We also present the state of the art in NF composition, as motivation to build a new tool.

In Chapter 3 we present our solution. We start by looking at the challenge as a program composition problem. Then we detail how our proposed tool automatically combines representations of network functions into a new one that represents the combination of the functionality, using conflict resolution policies when necessary.

Chapter 4 presents the evaluation of our proposed tool: it compares the complexity and effort needed to combine NFs, both by hand and using our tool, as well as performance evaluations of the resulting NFs.

The last chapter shares the conclusions taken from this work, as well as some limitations and future work ideas.

2

Related work

Contents

In this section, we start by providing a background about software defined networks, and how software network functions can out stand in terms of functionality, security, performance and flexibility. Then we explore some techniques of data and control plane verification and program synthesis.

Finally, we address several techniques of composition of network policies and network functions. We end the section with a brief summary on how these techniques help to achieve our main goal, and a comparison between the state of the art on network function composition and our thesis.

2.1 Software defined networks

Computer networks can be divided into two planes of functionality: the data and control planes. The data plane corresponds to the networking devices, which are responsible for (efficiently) forwarding data. The control plane represents the protocols used to populate the forwarding tables of the data plane elements.

Inserting new features in those networks is very difficult since it would imply a modification of the control plane of all network devices through the installation of new firmware and, in some cases, hardware upgrades. Hence, the new networking features are commonly introduced via expensive, specialized, and hard-to-configure equipment (also known as middleboxes) such as load balancers, intrusion detection systems and firewalls, among others. These middleboxes need to be placed strategically in the network, making it even harder to later change the network topology, configuration, and functionality. Whereas traditional networks use dedicated hardware devices (i.e., routers and switches) to control network traffic, SDNs (Software-defined networks) [5] control the network via software that runs in an external (to the devices) entity.

SDN refers to a network architecture where the forwarding state in the data plane is managed by a remote control plane decoupled from the data plane. In other words, the control plane is physically separated from the data/forwarding plane in the SDN controller, a logically centralized entity that communicates with the underlying hardware infrastructure to dictate how traffic is forwarded on the network. This type of architecture brings several advantages such as increased control and greater flexibility.

The following subsections will discuss the structure of software-defined networks using a bottom-up approach.

2.1.1 Data plane

An SDN infrastructure [5], similar to a traditional network, is composed of a set of networking equipment (switches, routers, and middlebox appliances). The main difference resides in the fact that those traditional physical devices are now simple forwarding elements without embedded control or software to make autonomous decisions.

The data plane is mainly composed of those forwarding devices that are interconnected through wireless radio channels or wired cables. In every device a southbound APIs is present, acting like the connecting bridge between the control and forwarding elements, and as such is the crucial element for separating control and data plane functionality.

2.1.2 Control plane

The control plane can be seen as the “network brain”. The network intelligence is removed from the data plane devices to a logically centralized control system. All control logic is implemented in applications

that run on top of this controller through another application programming interface (API).

The controller [5] is the critical element in an SDN architecture as it is responsible for generating the network configuration based on the policies defined through network applications by the network operator, that no longer needs to deal with the lower-level details of connecting and interacting with forwarding devices.

Furthermore, as those abstractions provided by the control platform can be shared, all applications can take advantage of the same network information, leading to more consistent and effective policy decisions.

2.2 Software network functions

Networks today are responsible for more than just forwarding packets, as they can be composed of middleboxes with a wide range of functionality, that includes security (firewalls and IDSs), performance (WAN accelerators), or support for new applications and protocols (TLS proxies).

The move from hardware middleboxes to software network functions has proven more challenging than expected. They have several advantages such as increased elasticity in terms of scaling, operational simplicity, and faster innovation, but brought challenges related to NF performance and its development process.

Several authors [6] have argued that developing new NFs remains a tedious process, requiring that developers repeatedly rediscover and reapply the same set of optimization and that current techniques for providing isolation between NFs (using VMs or containers) incur high-performance overheads. Their system, Netbricks [6], tries to solve those problems by providing both a programming model (for building NFs) and an execution environment (for running NFs).

Rather than having developers deal with a large set of modules – trying to determine which best suits their needs in terms of optimization and generality – NetBricks focuses on a core set with well-known semantics and highly-optimized implementations by limiting the set of such modules to core functions such as packet parsing, processing payloads, byte stream processing. In order to give those modules great flexibility and generality, they are allowed to be customized through the use of User-Defined Functions (UDFs). So, network functions are defined as a directed graph with those abstractions as nodes. The programming model is built around a core set of high-level but customizable abstractions for common packet processing tasks with the use of safe languages (Rust) to provide memory isolation equivalent to what is provided by a hardware MMU. Its execution environment was designed to provide the same memory isolation as containers and VMs, without incurring the same performance penalties. We share the concerns of NetBricks, but instead of a new programming model we propose using a synthesis approach to assist NF development.

It should be noted that softwarization and virtualization of network functions requires more than just memory isolation. The semantics of the physical network devices that the NFs are replacing should be maintained. This is a challenge in its own right, particularly in the case of NF composition we are addressing.

2.3 Formal verification

As we move from hardware middleboxes to software NFs the need to guarantee program correctness becomes a common concern. Formal verification can assist developers in this process. In the case of configuring networks the task is particularly arduous because policy requirements (for resource management, access control, etc.) can be complex and configuration languages are low-level. Consequently, configuration errors that compromise availability, security, and performance are common. The same applies to software network functions that promise to help networks meet ever-increasing application demands. However, unvefiried changes on their code can lead to serious faults on large networks. Formal verification can help by assessing whether a software satisfies some requirements (properties) by checking those properties against a verifiable representation of it.

Verification can be based on two main approaches. First, theorem proving. Network verification tools such as Verifast [7] use this method, where program verification is made using lemma functions, that is, functions that serve only as proofs that their precondition implies their postcondition, as in a contract. A program is guaranteed to be correct if the verifier checks that lemma functions terminate and do not have side effects.

The other approach is symbolic execution. It is used in verification tools like KLEE [8], where the program to be verified is replaced with specific operations (forming a symbolic representation) that will deal with symbolic inputs. When program execution is based on a symbolic value, the system (conceptually) may follow several branches, with each path maintaining a set of constraints called the path condition, which must hold on to the execution of that path. When a path terminates or hits a bug, a test case can be generated by solving the current path condition for concrete values. That approach is also used in Verifast, in conjunctiomn with theorem proving. One of the common problems of symbolic execution in verification is the exponential number of paths through code, that can lead to path explosion. The number of states can grow quite quickly in practice: often even small programs generate tens or even hundreds of thousands of concurrent states during the first few minutes of interpretation, and consequently consume many computational resources. For instance, tools like Vigor [1] address this problem by using models that are validated a posteriori using lazy proofs. As the use of those models provided by Vigor helps on building simple and complete symbolic representations, we will leverage it in the solution proposed in this project: we will use Vigor analysis engine to produce the symbolic representations of

the constituting network functions in order to apply a composition algorithm.

2.3.1 Data plane verification

Data plane verification in traditional networks works by taking as input a model of the processing performed by each network box, the links between boxes, and a snapshot of the forwarding state. Then, queries are made about the network without resorting to dynamic testing, in order to verify reachability, absence of loops, etc. The problem is that if we view packets as variables being passed between different network boxes, static network analysis becomes akin to software testing. In this context, the power of symbolic execution lies in its ability to relate the outgoing packets to the incoming ones - even if all the incoming packet headers are unknown, a symbolic execution engine can detect which header fields are allowed in each part of the network, which ones are invariant, and can tell how the modified headers depend on the input when they are changed – the trade-off is scalability.

Stoenescu et al. proposed SymNet [9], a network static analysis tool based on symbolic execution. SymNet injects symbolic packets and tracks their evolution through the network. It statically analyzes an abstract data plane model that consists of the SEFL code (the language used to express data plane processing and used for verification) for every node and the links between nodes. SymNet can check networks containing routers with hundreds of thousands of prefixes and NATs in seconds, while verifying packet header memory-safety and covering network functionality such as dynamic tunneling, stateful processing, and encryption. SEFL uses a packet layout that mimics real implementations of packets. Packet headers are variables, but each header has an absolute offset at which it is allocated. All SEFL headers must be allocated individually, and allocation and deallocation commands must include the explicit size of the header field, not allowing symbolic sizes, to ensure traceability.

Symnet starts execution by creating an initial empty packet, with no header fields or metadata, and then executes code to create a symbolic packet of the given type. Each path in the network symbolic execution is given as input to instructions (that can modify the state associated) and must be tied to an active packet passing through the network. If a code path does not result in packets, it should not be symbolically executed. For each value, on each path, Symnet holds a list of constraints that apply to that value. Assignment operations modify the top of the current value stack, and a complete history of the values associated with a symbol are kept to detect network loops, and to check whether fields change across network hops. A path finishes execution when the Fail instruction is called, when a constraint does not hold on to any of its variables, or when it reaches a port with no outgoing links.

Other works such as Vigor [1] and VigNAT [10] focus on designing and implementing software network functions that are proven to be correct, being the former a generalization of the latter. VigNAT is a network address translator (NAT) written in C and proven to be semantically correct according to RFC 3022, as well as crash-free and memory-safe. This network function is verified by Vigor, a software stack

and toolchain for building and running NFs that are guaranteed to be correct while preserving competitive performance and developer productivity. Vigor targets NFs implemented on top of standard network I/O frameworks (e.g. DPDK). This tool takes as input an NF implementation and a specification that the implementation must satisfy, and it automatically produces either a proof that the implementation satisfies the given specification or a counterexample. It also verifies low-level properties like memory safety, crash freedom, hang freedom, and absence of undefined behavior.

Developers can write NF code on top of DPDK and use data structures from Vigor's library (libVig) to implement stateful parts of the NF— i.e., the state that persists across packets. The API for accessing libVig's data structures that ensures persistence of state across packets (libVig's functions) is verified against a written beforehand contract which states that given some preconditions, an invocation to the function leads to the post-conditions. One of the main features of Vigor is the ability to verify the entire NF stack, including core implementations, NIC drivers, DPDK functions, and the underlying OS. For that to be possible, the authors designed a custom operating system that can be automatically verified - an NF-specific operating system, the NFOS, that performs the necessary setup for the packet I/O framework to take over.

Other languages beyond C have been used for controlling packet forwarding planes in network devices. One example is P4 [11], a domain-specific language with a number of constructs optimized for packet processing. In that context, VERA [12] is a tool that verifies P4 programs using symbolic execution. It automatically uncovers a number of common bugs including parsing/deparsing errors, invalid memory accesses, loops, and tunneling errors. It does so by translating P4 to SEFL, and relies on symbolic execution with Symnet to analyze the behavior of the resulting program. It performs an exhaustive verification on a snapshot of a running P4 program (i.e. the program and a snapshot of all its table rules): it uses the parser of the P4 program to generate all parsable packet layouts (e.g. header combinations) and makes all header fields symbolic (i.e. they can take any value). It then tracks the way these packets are processed by the program, following all branches to completion. To improve scalability, Vera introduces a novel match-forest data structure that concurrently optimizes both update and verification time. Even if a snapshot of a P4 program is bug-free, there is no guarantee that the same holds for other snapshots (i.e. sets of table rules). With Vera, users can explore multiple table snapshots by specifying symbolic table rules instead of concrete ones.

Another tool used to verify programmable data planes using P4 is called P4V [13]. The design of P4V is based on classic verification techniques but adds several key innovations including a novel mechanism for incorporating assumptions about the control plane and domain-specific optimizations which are needed to scale to large programs. It starts by building a first-order formula that captures the execution of the program in logic, leveraging the fact that P4 programs denote functions on finite sequences of bits (i.e., packets) parameterized on finite-state (i.e., switch registers), and then use an automated

theorem prover to check whether there exists an initial state that leads to a violation of one or more correctness properties. Another challenge was that a P4 program does not fully specify the semantics of a data plane, which makes it impossible to fully verify many programs without additional knowledge of the control plane. They worked around this problem by delaying verification until the forwarding rules are known. By combining the program and the forwarding rules, one obtains a deterministic program that can be verified.

2.3.2 Control plane verification

Control plane verification is a hard problem that includes checking the correctness of the control plane configuration, proving convergence after link additions or failures, and characterizing the transient behavior until convergence is reached. Some of the earliest network diagnostic tools such as traceroute and ping can help find configuration errors by analyzing whether and how a given packet reaches its destination. These tools are simple and have high network design coverage—they can analyze forwarding for any network topology or routing protocol. But they have poor data plane coverage.

Tools such as Veriflow [14] have better data plane coverage. It is implemented as a shim layer between the controller and the network, and monitors all communication in either direction, intercepting all the rules and verifying them before they reach the network. Real-time response is achieved by confining its verification activities within those parts of the network that are affected when a new forwarding rule is installed. For this purpose, the network traffic is divided into a set of equivalent classes (EC) of packets, i.e, packets that experience the same forwarding actions throughout the network. Scalability is achieved because each change on the network will only affect a small number of ECs. For each EC Veriflow builds a forwarding graph, traversing it to check for invariants. An invariant is specified as a verification function that takes as input the forwarding graph for a specific EC, performs arbitrary computation, and can trigger resulting actions. Up to a certain level of detail, the forwarding graph is an exact representation of the forwarding behavior of the network. Therefore, invariant modules can check a large diversity of conditions concerning network behavior.

While a step in the right direction, the data plane coverage of this tool is still less than ideal, because they analyze only the data plane that is currently derived from the control plane in the network. That is, they can only find errors after the network has produced the erroneous data plane. Other control plane analysis tools such as Batfish [15] take the network configuration (i.e., its control plane) and a specific environment (e.g., a link-failure scenario) as input and translate them into a variant of DataLog that will produce logical relations that represent the data plane. Then, a constraint solver checks the properties of the resulting data plane and produces concrete packets that violate those properties. Therefore, developers can query BatFish for correctness properties, or even find errors without any queries, due to its capability of checking, by default, several errors including multipath and reachability. Still, each

run of Batfish allows users to explore at most one data plane, and given the large number of possible environments, it is intractable to guarantee correctness for all possible data planes.

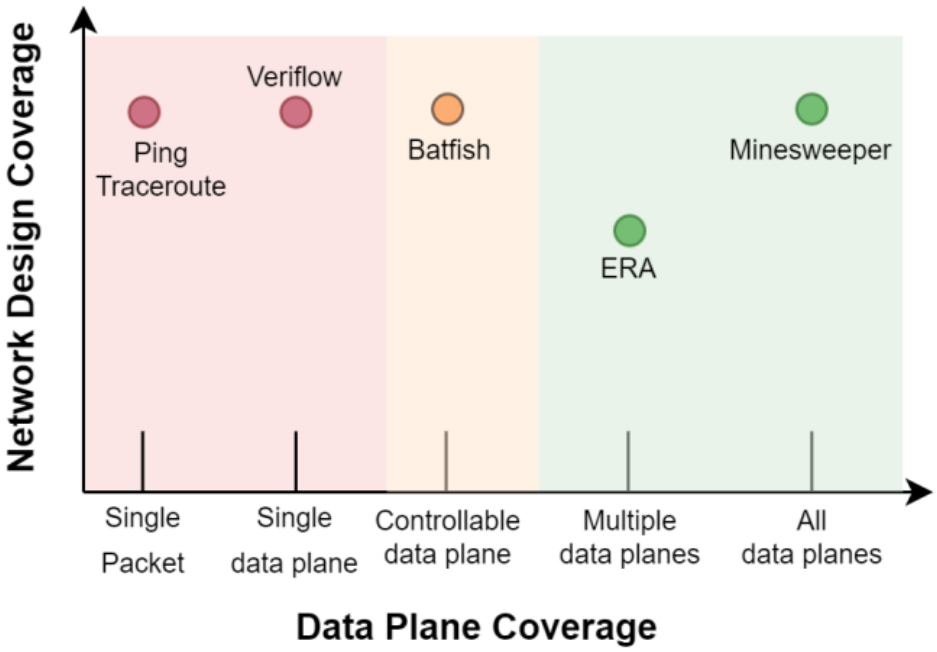

Figure 2.1: Landscape of network analysis tools

As we can observe in Fig 2.1, many control plane analysis tools have gone from testing individual controllable data planes to reasoning about many data planes that can be generated by the control plane. However, this type of tool trades network design coverage for higher data plane coverage. ERA [16] is such an example. This solution represents a concrete set of control plane messages using binary decision diagrams (BDDs) and propagates this set along a path through the network by transforming the set as dictated by the network configuration. A control plane model that captures the key behaviors of various routing protocols is designed and routers are viewed as functions that take as input route announcements and produce a set of route announcements for its neighbor. Those route announcements are represented by the decision diagrams which representation is shrunk by identifying equivalent classes that are treated identically in the network. Each equivalence class is given an integer index, and the reachability analysis is transformed to arithmetic operations directly on sets of these indices. In this way, ERA can efficiently check reachability in certain large symbolic environments, but not all.

Beckett et al. tried to solve the trade-off problem of network design and high data plane coverage by developing a configuration tool called Minesweeper [17], that has both high network design coverage in that it works for a large collection of network protocols, features, and topologies as well as high data plane coverage in that it can verify a large number of properties for all possible data planes that

might emerge from the control plane. It translates network configuration files into a logical formula that captures stable states to which the network forwarding will converge. Then, the formula is combined with logical constraints that describe a desired property and if the formula is satisfiable, then exists a state of the network in which the property does not hold.

2.4 Synthesis

In the following subsections we address a general definition of program synthesis, as well as some novel applications of synthesis in the network control and data planes.

2.4.1 Traditional program synthesis

Program Synthesis [18] is the task of automatically finding programs from the underlying programming language that satisfy user intent expressed in some form of constraints. Unlike typical compilers that translate a fully specified high-level code to low-level machine representation using a syntax-directed translation, program synthesizers typically perform some form of search over the space of programs to generate a program that is consistent with a variety of constraints (e.g. input-output examples, demonstrations, natural language, partial programs, and assertions).

A synthesizer is characterized by three key dimensions: the kind of constraints that it accepts as an expression of user intent, the space of programs over which it searches, and the search technique it employs. The user intent can be expressed in various forms including logical specification, examples, traces, natural language, partial programs, or even related programs. The space should be large/expressive enough to include a large class of programs for the underlying domain while being restrictive enough so that it is amenable to efficient search. Finally, the search technique can be based on enumerative search, deduction, constraint solving, statistical techniques, or some combination of these. According to approaches such as Automated program repair [19], its challenge lies in two main components: intractability of the program space (program synthesis is undecidable, thus almost all successful synthesis approaches perform some kind of search over the program space and this search itself is a hard combinatorial problem) and diversity of user intent (because most of the software rely on weak specifications, like a test-suite, and the generated patches may not generalize to tests outside of it).

One problem of these traditional techniques is that they are not able to learn from past mistakes. An alternative was proposed by Feng et al. [2], in the form of a new conflict-driven synthesis algorithm that is capable of learning from its past errors. Given a spurious program that violates the desired specification, it identifies the root cause of the conflict and learns new lemmas that can prevent similar mistakes in the future. The synthesis algorithm is composed of three components (as shown in Fig. 2.2) – given a partial program P with holes (representing unknown program fragments), the Decide component selects

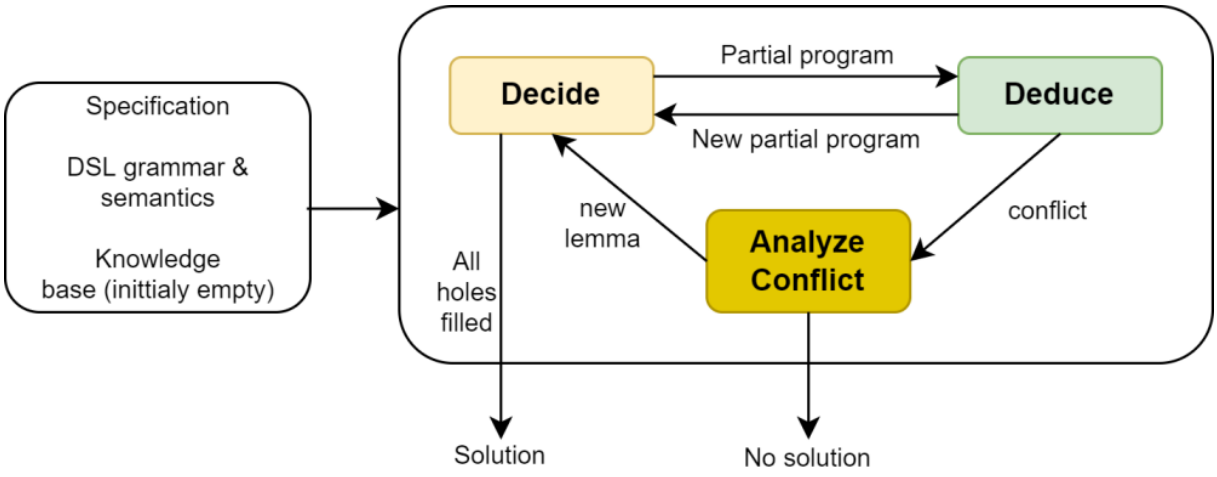

Figure 2.2: Conflict-driven synthesis algorithm overview based on [2]

which hole to fill and determines how to fill it using the constructs of a DSL (domain-specific language); the Deduce component makes new inferences based on the syntax and semantics of the DSL as well as a knowledge base, which keeps track of useful "lemmas" learned during the execution of the algorithm; when the Deduce component detects a conflict (meaning that the partial program is infeasible), the goal of the AnalyzeConflict component is to identify the root cause of failure and learn new lemmas that should be added to the knowledge base (because the decisions made by the Decide component need to be consistent with the knowledge base, these lemmas prevent the algorithm from making similar bad decisions in the future).

2.4.2 Data plane synthesis

When implementing network functions (NFs), developers are often confronted with a choice – use one of a wide variety of programmable networking devices, such as programmable switches and SmartNICs, to trade-off some flexibility for the ability to process packets at full line rate, or implement network functions in software and face the challenge of performance.

Considering the first type, writing packet-processing programs for programmable switch pipelines is challenging because a program either runs at line rate if it can fit within pipeline resources or does not run at all. It is the compiler's responsibility to fit programs into pipeline resources. Chipmunk [20] is a compiler that uses a program synthesis engine, SKETCH, to transform high-level programs down to switch machine code. SKETCH takes two inputs: a specification and a sketch (a partial program with holes representing unknown values in a finite range of integers). Sketches constrain the synthesis search space by only considering for synthesis those programs in which each sketch hole is filled with an integer belonging to the hole's range, then filling all holes with integers so that the completed sketch

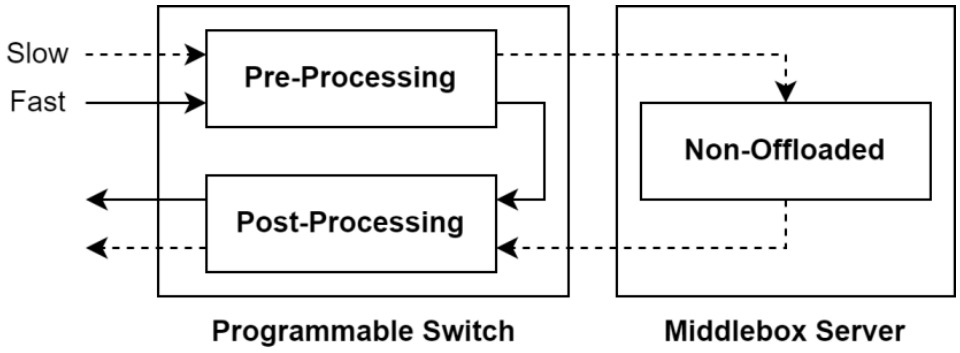

Figure 2.3: Packet flow on offloaded middlebox

meets the specification, assuming it is possible to meet the specification, or returning information that it is impossible to do so. However, naively formulating code generation as program synthesis can lead to long compile times. Hence, Chipmunk uses a new domain-specific synthesis technique, slicing, which considerably reduces compile time, as each slice can be synthesized in parallel and its implementation only satisfies a subset of the specification.

As it is not always possible to fully offload a network function to a programmable switch, researchers have proposed offloading only part of a software middlebox to a programmable switch to achieve some performance gains, while keeping the other part of the logic in the software middlebox. However, this usually requires manually selecting the middle-box components to offload and rewriting the offloaded code in P4, the domain-specific language for programmable switches. Gallium [21] is a compiler that automates this process, transforming an input software middlebox into two parts — a P4 program that runs on a programmable switch, and an x86 non-offloaded program that runs on a regular middlebox server.

The objective (as shown in Fig. 2.3) is to automatically partition the input middlebox program into three parts, i.e., a pre-processing partition, a non-offloaded partition, and a post-processing partition. The pre-processing and post-processing partitions run on the programmable switch as a single P4 program, and the non-offloaded partition executes on the middlebox server. Packets coming to the middlebox go through the pre-processing, non-offloaded, and post-processing partitions sequentially. Because a programmable switch is efficient at packet processing, this offloading scenario can improve middlebox performance if enough instructions are offloaded to the switch. Further, if the non-offloaded partition is not involved in processing a packet, the packet can simply skip the middlebox server, reducing latency and processing cycles in the server. Gallium ensures that the combined effect of the P4 program and the non-offloaded program is functionally equivalent to the input middlebox program, the P4 program respects the resource constraints in the programmable switch, and the run-to-completion semantics are met under concurrent execution.

In our solution we will also explore data plane synthesis to generate an NF that represents the composition of multiple input NFs.

2.4.3 Control plane synthesis

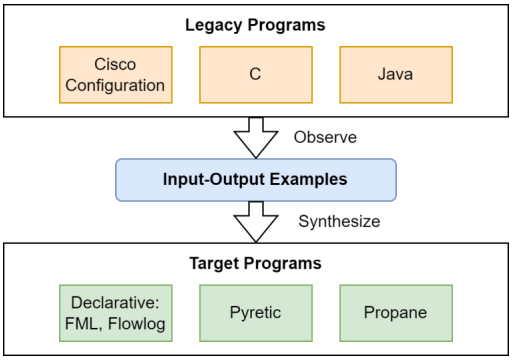

Figure 2.4: Facon overview [3]

In recent years, there has been a proliferation in domain-specific languages (DSL) for the network control plane. These languages enable us to exploit the programmability of these networks, while still providing correctness guarantees through verification and analysis of DSLs. Despite these advantages, many DSLs have not gained widespread adoption in practice. There are many potential reasons (e.g., performance concerns, limited expressiveness, etc.), but one major hurdle is the learning curve associated with a new programming paradigm. Those DSLs would require significant changes in programmers' reasoning process, due to the semantic differences between imperative and declarative programming, for instance.

Facon [3] is a tool that automatically generates programs in arbitrary DSLs based on input/output examples. Rather than learning new DSLs and rewriting programs when migrating to a new platform, Facon automatically generates DSL programs by learning from input-output programs. These examples can come from either an operator or from the I/O trace of an equivalent legacy system. Facon then follows the syntax of the DSLs and generates a program that satisfies all given input-output examples.

Since input/output examples apply to any network protocols, this approach can be generalized, hence enabling the migration of legacy networks to new DSLs, or to transform one DSL to another.

2.5 Composition

Managing a network requires support for multiple concurrent tasks, from routing and traffic monitoring, to access control and load balancing. Software-Defined Networking allows applications to realize

these tasks directly, by installing packet-processing rules on switches. Although SDNs can greatly simplify network management by offering programmers network-wide visibility and direct control over the underlying switches from a centralized controller, they provide limited support for creating modular applications, that is, building applications out of multiple and independent modules that jointly manage network traffic, and the same applies to programmable data planes. One important challenge in the case of network functions is to ensure that the packet-processing rules installed to perform one task do not override the functionality of another. In other words, the composition of the multiple NFs should maintain the functionality of each NF in separate, and not break the original semantics.

2.5.1 Composition of network policies

Although well-known SDN controllers like OpenFlow and NOX provide a standard interface for manipulating the rules on switches, the programming model provided has several deficiencies that make it difficult to use in practice. In the perspective of modular programming, the interactions between concurrent modules is not easy to specify: if two or more programs install overlapping rules on the switch when a packet arrives, the switch is free to process the packet using either rule, something not desirable.

Frenetic [22], a high-level language for programming distributed collections of network switches, introduced the concept of parallel composition, which gives each module the illusion of operating on its own copy of each packet. To address this challenge this language is organized in two levels – a set of source-level operators draw on declarative database query languages for constructing and manipulating streams of network traffic, and a run-time system handles all the details of installing and uninstalling rules on switches. All of the high-level and intuitive operators and primitives are designed to consider what the programmer wants, rather than what is best for the hardware. The semantics can be stated independently of the context in which they are used, and consequently the programmer is able to reuse the same modules to build different modular programs. Frenetic is also able to suppress superfluous packets that can arrive at the controller due to lack of installed rules in a certain moment (race conditions), while retaining a simple programming model as those types of problems are not exposed to the programmer.

The authors performed an experimental evaluation based on the implementation of several applications in Frenetic, and compared them against equivalent NOX programs, considering three metrics: lines of code (a measure of the complexity of each program), traffic to the controller, and total traffic (on every link in the network). The results demonstrated that the performance of Frenetic's run-time system is competitive with hand-written NOX programs, while Frenetic provides substantial code savings by enabling easy composition of the monitoring and forwarding modules.

Pyretic [23] is another language and system that enables programmers to specify network policies at a high level of abstraction, compose them together in a variety of ways, and execute them on abstract

network topologies. It complements Frenetic by introducing a new kind of composition - sequential composition - that allows one module to act on the packets already processed by another model. The parallel and sequential composition operators offer simple, yet powerful, ways to combine policies generated by different modules and allow policies to operate on abstract locations that map to ones in the physical network.

Besides building efficiently independent modules and combine them, modular programming requires a way to constrain what each module can see (information hiding) and do (protection). Towards that goal, Pyretic offers topology abstraction with network objects. An abstract topology can be a subgraph of the real topology, like one big virtual switch spanning the entire physical network. To implement this abstraction, each packet flowing through the network is a dictionary that maps field names to values. These fields include entries for the packet location (either physical or virtual), standard OpenFlow headers, and other custom data. In order to present the illusion of a packet traveling through multiple levels of abstract networks, each packet holds a stack of values instead of a single bitstring, and when a packet leaves a virtual switch, the run-time system pops a field of the appropriate stack.

2.5.2 Composition of network functions

Openbox [4] is a software-defined framework for network-wide development, deployment, and management of network functions, as it effectively decouples the control plane of NFs from their data plane, similarly to SDN solutions that only address the network's forwarding plane. It consists of three logic components (as shown in Figure 2.5) – user-defined applications, a logically-centralized OpenBox controller, and the Openbox instances.

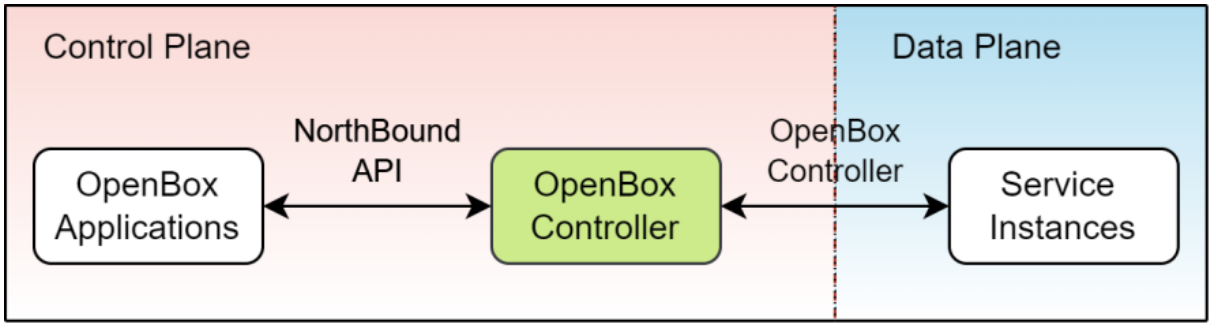

Figure 2.5: OpenBox overview (from [4])

The data plane consists of OpenBox service instances (OBIs), which are low-level packet processors that can be implemented in hardware or software. An OBI receives a processing graph from the controller and applies it on traversing packets. It can also answer queries from the controller and report its load and system information. Software implementations can run in a VM and be provisioned and scaled on demand. The OpenBox communication protocol is used by OBIs and the controller to com-

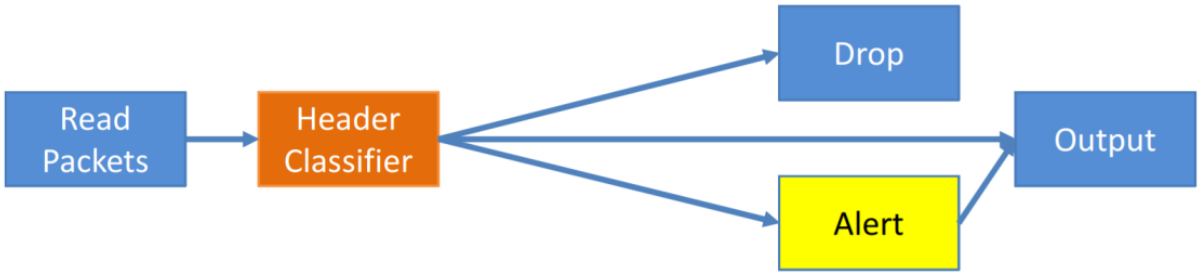

Figure 2.6: Firewall (from [4])

municate with each other. The protocol defines a set of messages for this communication and a broad set of processing blocks that can be used to build network function applications. Abstract processing blocks are defined in the protocol specification and have their own parameters. In addition, similarly to Click [24] elements, blocks may have read handles and write handles in order to request or write information about configuration. The controller (OBC) is a logically centralized software server. It is responsible for managing all aspects of the OBIs: setting processing logic, and controlling provisioning and scaling of instances. The OBC provides an abstraction layer that allows developers to create network-function applications by specifying their logic as processing graphs. At the top of the stack, there are several applications and each one of them defines a single network function (NF) by statement declarations. Each statement consists of a location specifier, which specifies a network segment or a specific OBI, and a processing graph associated with this location. Applications are event-driven, where upstream events arrive at the application through the OBC.

An interesting feature of OpenBox is the abstraction of packet processing applications designed to provide a framework for the development and deployment of a wide range of network functions. Packet processing is abstracted as a processing graph, which is a directed acyclic graph of processing blocks. Each processing block is mapped to a compound set of Click elements, or to a new element implemented if no Click element was suitable, and represents a single, encapsulated logic unit to be performed on packets. This includes header field classification or header field modification, and has a single input port (except for a few special blocks) and zero or more output ports. When handling a packet, a block may push it forward to one or more of its output ports. Each output port is connected to an input port of another block using a connector.

The figures above shows sample processing graphs for a firewall network function (Fig. 2.6) and an IPS network function (Fig. 2.7). For instance, the firewall reads packets, classifies them based on their header field values, and then either drops the packets, sends an alert to the system administrator and outputs them, or outputs them without any additional action. Each packet will traverse a single path of this graph.

Consider both network functions shown above running at the same physical location in the data

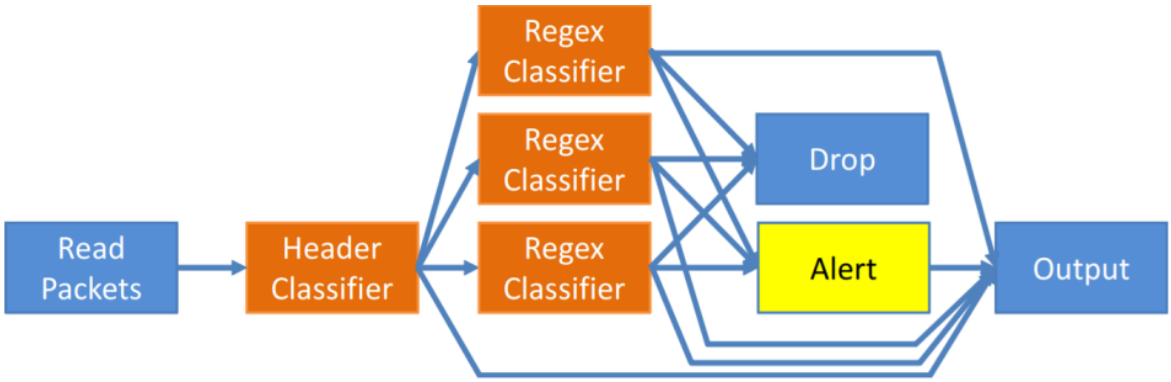

Figure 2.7: IPS (from [4])

plane. With OpenBox it is possible to merge the two graphs into one, such that the logic of the firewall is first executed, followed by the execution of the IPS logic. Additionally, it is possible to reduce the total delay incurred on packets by reducing the number of blocks each packet traverses. The result of this merger process is shown in Figure 2.8: packets go through one header classification instead of two, and execute the logic that corresponds to the result of this classification.

In order to model the merge algorithm, abstraction blocks are classified into five classes: (i) Terminals (T): blocks that start or terminate the processing of a packet; (ii) Classifiers (C): blocks that, given a packet, classify it according to certain criteria or rules and output it to a specific output port; (iii) Modifiers (M): blocks that modify packets; (iv) Shapers (Sh): blocks that perform traffic shaping tasks such as active queue management or rate-limiting; and (v) Statics (St): blocks that do not modify the packet or its forwarding path, and in general do not belong to the classes above.

The algorithm works in four stages: (i) It normalizes each processing graph to a processing tree so that paths do not converge; then, (ii) it concatenates the processing trees in the order in which the corresponding NFs are processed; (iii) all paths of the resulting tree are examined in order to detect redundancies; and (iv) the last stage takes place after the merge process is completed. It eliminates copies of the same block and rewires the connectors to the remaining single copy so that eventually the result is a graph. Correctness has to be maintained: a packet must go through the same path of processing steps such that it will be classified, modified, and queued the same way as if it went through the two distinct graphs. Static operations such as alerts or logs are also executed on the same packet, as they would without merging.

2.6 Summary

In this section we presented background material and several state-of-the-art systems that employ verification, synthesis, and composition, as summarized in Fig. 2.9. As we have seen, developing and

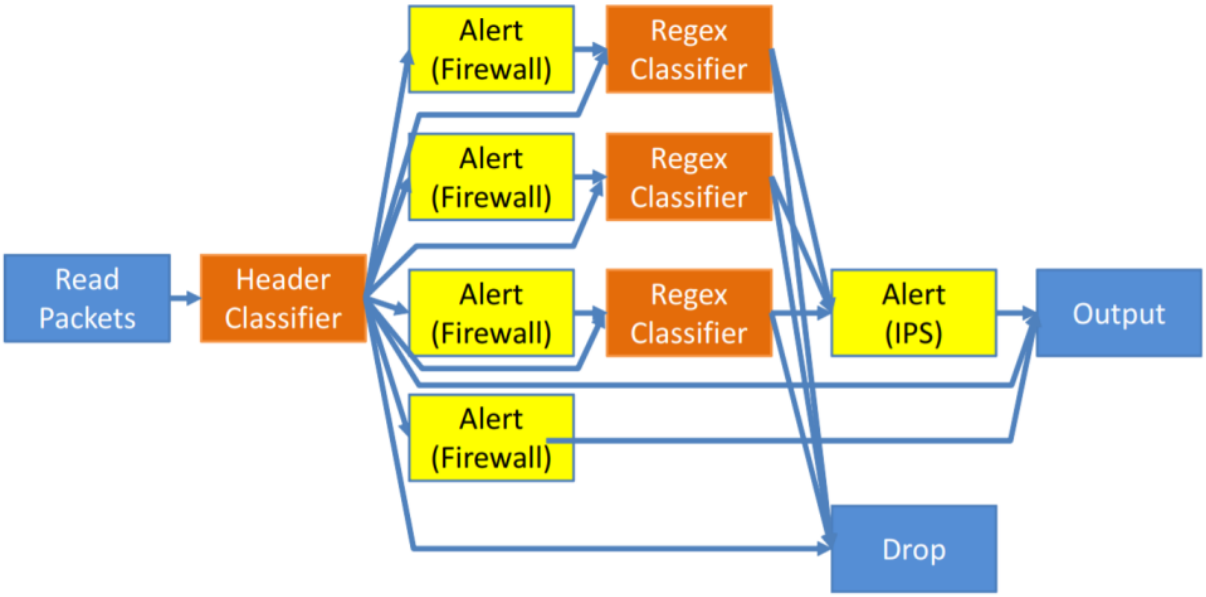

Figure 2.8: The result of the graph merge algorithm for the two processing graphs shown in Figures 2.6 and 2.7 (from [4])

configuring networks correctly is challenging. Many configuration primitives – if used incorrectly – can lead to unexpected consequences.

We described two dual approaches to address this challenge: verification and synthesis. Verification looks at the problem by looking at a given input network (code, configuration, forwarding rules, etc.) and checking if it always satisfies a given property P. Synthesis, in turn, attempts the opposite approach: given some specification of properties P that must hold, generate an implementation that always satifies P, *by construction*.

Network verification can be performed in a highly general and scalable manner. While verification can be effective for finding problems in a network after the configurations have been implemented, it provides no insight into how to write the configurations (or programs that generate them) in the first place, or how to evolve them when the network changes. Unfortunately, this means that writing configurations or network programs remains a daunting task.

We subsequently explored a variety of tools that synthesize networks (configurations, data-plane code, etc.). This approach goes a step further in aiding developers and operators in building networks that meet their intent. Much of the work in this space has focused on the network control-plane, synthesizing network configurations. In this thesis, we will use synthesis, not to synthesize network configurations, but to synthesize data-plane packet processing code.

When developing software, one rarely writes an entire program as a single function, or even as a single module/class. Instead, programs are decomposed into separate modules with separate concerns,

and the modules present an interface to the outside world (e.g., a collection of functions and their types) that hides many implementation details. In this thesis we will explore new ways to generalize this concept for network functions, by allowing developers to merge functionality from multiple NFs.

With this in mind, we conclude our analysis of the state-of-the-art with a series of works on network composition, including composition of network policies and network functions. As in OpenBox, the focus of our solution will be in composing the network functions, but with some key differences – Openbox focuses on NFs that are built modularly using click [24] modules. Composition in this scenario is not so much of a challenge unto itself, but rather an optimization problem, to reduce redundant code shared among several co-located NFs. In this thesis we will explore composing NFs developed in C, using the Vigor framework [1]. This gives developers more freedom and expressivity when creating NFs, but also brings less structure to the problem, which raises new challenges in the composition process itself.

Tools	Control Plane			Data Plane			Input	Output	Used Technique
	Verification	Synthesis	Composition	Verification	Synthesis	Composition			
Veriflow	✓						Openflow rules	Pass/fail	Real-time check
Batfish	✓						Network configuration	Verification proof	Query-based
Minesweeper	✓						Network configuration	Verification proof	Logical formulas
Facon		✓					High-level network program	SDN code	Examples observation
Frenetic			✓				High-level network program	SDN code	Query-based
Pyretic			✓				High-level network program	SDN code	Query-based
Vigor				✓			C	Verification proof	Symbolic execution
VERA				✓			P4	Verification proof	Symbolic execution
VigNAT				✓			C	Verification proof	Symbolic execution
SymNet				✓			Network model	Verification proof	Symbolic execution
Chipmunk					✓		P4	"Hardware" code	Slicing
OpenBox						✓	Click	Click	Undirected graphs
Proposed Solution					✓	✓	C	C	Symbolic composition

Figure 2.9: SOTA systems

3

Architecture

Contents

This thesis expands the Network function composition state of the art by presenting a tool designed to combine multiple NFs. Its goal is to allow developers to effortlessly combine their NFs (in our case, written in the Vigor framework) into new ones that represent a combination of functionality. This chapter, which is divided into six sections, aims to describe how this tool was built to provide a solution for network function composition. In the first section, we start by presenting Network Function Composition as a program composition problem, followed by a simple example that illustrates the main idea behind the process of composition. The next sections expose an architectural overview of our solution, as well as a deep dive into each component of the tool's architecture. Alongside the overview, we provide a running example of two NFs to be composed with our tool's pipeline, for a better understanding of the

whole process and how each component of the pipeline works individually, as well as in conjunction.

3.1 NF combination as a program composition problem

The move from hardware middleboxes to software network functions has brought several advantages, such as increased efficiency, flexibility, and innovation. Consequently, this evolution led to the need to combine different functionalities of different network functions. Here the developers face the tedious challenge of merging the code-base of different network functions while ensuring that all the semantics of each functionality remains the same.

Our goal is to automate the difficult process of combination of NFs. We will explore the composition of network functions as a program composition problem, and leverage previous State-of-the-art (SOTA) systems, Vigor [1] and Maestro [25], to achieve our goal.

3.1.1 Program composition

A combination of several NFs can be seen as a program composition problem. Composing multiple programs or pieces of code to achieve a desired outcome. This requires a deep understanding of the individual components being combined, as well as knowledge of design patterns, best practices, and relevant programming paradigms.

The problem starts with the program input which, when dealing with NFs, is the network traffic. Combining all NF modules in a sequential pipeline of functionality is itself insufficient. To understand this, let's take as an example two known network functions: a firewall (Listing 3.1), which has a set of rules to manage incoming and outgoing traffic; and a policer (Listing 3.2) that limits a specific user's download and/or upload rate according to a previously established contract. A naive approach to combine them would be forming a simple chain: all the traffic going through the firewall would then go through the policer. That kind of solution has a problem: the policer would only have access to a portion of the traffic that went through the firewall, and it would limit less traffic than it should, perhaps breaking the operator's goals. Swapping this topology wouldn't solve the problem either, as the firewall would only have access to traffic with a limited rate, not being able for instance to detect attacks that rely on high traffic rates, or even just missing key packets that establish state.

Different functionalities can lead to different actions or to different traffic states, and we must ensure that, in the end, all functionality of the combined version is semantically identical and produce the same results as their independent versions.

Listing 3.1: Pseudocode of a firewall written in Vigor [1]

```
int nf_process(int device, pkt_t p, time_t now) {
    struct eth_hdr_t eth_header = get_eth_hdr(p);
    struct ipv4_hdr_t ipv4_header = get_ipv4_hdr(eth_header);
    struct tcpdup_hdr_t tcpudp_header = get_tcpudp_hdr(ipv4_header);

    if (device == WAN) {
        // inverse ports and IPs for the "reply" flow
        struct Flow flow = {
            src_port: tcpudp_header.dst,
            dst_port: tcpudp_header.src,
            src_ip: ipv4_header.dst,
            dst_ip: ipv4_header.src
        };

        if(!map_constains(map, flow)) {
            return WAN; //drop
        }

        int dst_device = map_get(map, flow);
        return dst_device;
    } else {
        // store flow
        struct Flow flow = {
            src_port: tcpudp_header.src,
            dst_port: tcpudp_header.dst,
            src_ip: ipv4_header.src,
            dst_ip: ipv4_header.dst
        };

        map_put(map, flow, device);
        return WAN; //forward
    }
}
```

3.1.2 Policy-driven NF composition

One of the advantages of using Vigor's [1] networks functions is that they are formally verified. Their correctness is ensure by generating a complete and sound representation of the NF's functionality using symbolic execution: Binary Decision Diagram (BDD). These representations are crucial to overcome the program composition problem, as they represent all possible outcomes to every type of incoming traffic, to ensure the correctness of the functionality. Even when combined consider as if each one of the NFs had access to a copy of the traffic.

Let's consider two network functions as a module that receives a network packet, that apply changes to it, and produces a specific action (drop, forward or broadcast) as it is shown in Fig.3.1.

In an optimal case, one can have 2 network functions whose functionalities are completely compatible. This occurs when they modify different fields of the packet and produce the same actions. This

Listing 3.2: Pseudocode of a policer written in Vigor [1]

```
struct Map* map;

int nf_process(int device, pkt_t p, time_t now) {
    struct eth_hdr_t eth_header = get_eth_hdr(p);
    struct ipv4_hdr_t ipv4_header = get_ipv4_hdr(eth_header);
    struct tcpdup_hdr_t tcpudp_header = get_tcpudp_hdr(ipv4_header);

    if (device == LAN) {
        return WAN; // not policing outgoing packet
    } else {

        Struct RateChunk current = { size: p.size, time: now() };
        if (!map_contains(map, ipv4_header.dst)) {
            map_put(map, ipv4_header.dst, current); // incoming packet from WAN
            return LAN; //forward
        } else {
            struct RateChunk past = map_get(map, ipv4_header.dst);
            int rate = get_rate(past, current);
            if (rate > MAX_RATE) {
                return WAN; //drop
            } else {
                return LAN; //forward
            }
        }
    }
}
```

simplifies the procedure: our tool is able to do to a fully automated merge of both NFs.

In more complex cases, when have several network functions that can interact with the same packet fields, or even produce different actions for the same type of packet, the problem is more difficult. To address it, we propose giving the developer the opportunity to define his own conflict resolution rules, in order to ensure that the new functionality works exactly as he planned. As shown in Fig. 3.2, our tool pairs all compatible outcomes from both NFs and uses merge mechanisms defined by the user to produce a new and correct output.

This whole process is better understood by means of an example. Let's go back to the firewall and the policer. In the merge process, we need to ensure that each one of them has access to the whole (unmodified) incoming packet to ensure their correctness. As shown in Listing 3.3, we encapsulate each functionality in a function (to simplify the process) and give it access to the received packet (this is possible by creating a copy) that returns the chosen action from each one of the NFs. In the end, we apply a simple conflict resolution mechanism that serves as example of a user-defined rule: all traffic is dropped if its above the rate limit or if its from an unknown connection.

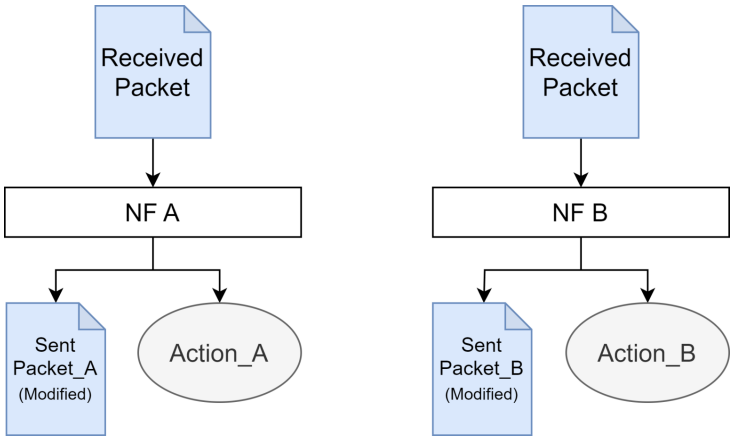

Figure 3.1: Generic network functions A and B

3.2 Overview

Out tool's architecture is a pipeline composed of three modules: the Analysis Engine, the Merge Engine, and the Synthesis Engine. Each module tackles a specific challenge in the problem of the automatic composition of NFs. The pipeline is comprehensively explained in this chapter. The architecture is shown in Fig. 3.3.

The Analysis Engine is based on Maestro [25], which extends the Vigor [1] symbolic analysis, which in turn builds on KLEE [8]. The engine takes as input an NF implementation and builds a sound and complete algebraic representation of the NF's functionality, resembling a binary decision tree that contains all possible paths of the program, represented by high-level directives. This is performed independently for each of the input NF.

The module that follows, the Merge Engine, takes both representations as input and returns a representation of the combination of both functionalities. In the presentation that follows, we first address an optimal scenario, where our tool can fully automate the merge between two NFs (using a naive conflict resolution process). Then, we will consider a more complex case, that receives a user-defined conflict resolution policy to complete the merge.

Finally, the Synthesis Engine (code generation and synthesis tools, based on Maestro [25]) is used to output the final implementation.

3.3 Analysis engine

As explained before, the composition of NFs raises different problems, from their sequential combination to their different resulting actions for identical packet inputs. In this thesis we address both problems leveraging Vigor's NF properties.

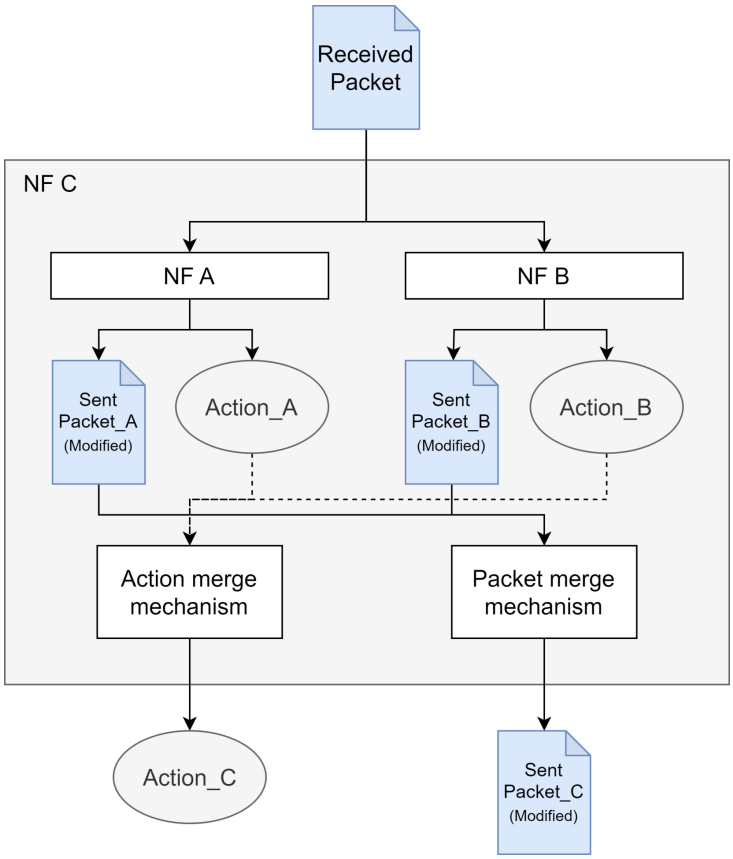

Figure 3.2: Combination of network functions in 3.1

We recall that Vigor is a software stack and toolchain for building and running NFs that are guaranteed to be correct, while preserving performance. Developers write the core of the NF in C, on top of a standard packet-processing framework Data Plane Development Kit (DPDK), putting the persistent state in data structures from Vigor's library (libVig). The Vigor toolchain then automatically verifies that the resulting software stack correctly implements a specification, using complete and sound mathematical representations of the NF that will be the key for the merge process.

In this section, we begin describing how Vigor NFs are built and refer to important implementation details that directly influence the composition process. Finally, we describe Binary Decision Diagrams in detail, and in what way they can be used to overcome our composition challenges.

3.3.1 Using Vigor's framework to create NFs

When using Vigor's framework to create NFs, developers should only be concerned with creating the packet processing logic, which is encapsulated in a function: `nf_process`. In the presence of NFs that need to store their state (like a Firewall that needs to store current active connections) one can use several data structures provided by libVig. Since the correct behavior of a stateful NF depends directly

Listing 3.3: Merge proposal between Firewall and Policer

```
int nf_process(int device, pkt_t p) {

    int dst_device_firewall = nf_process_fw(device, p);

    //its safe to do that as firewall dont change packet
    pkt_t p_copy = mk_copy(p)
    int dst_device_policer = nf_process_pol(device, );

    //merge changes
    merge_changes(p,p_copy);

    //Simple conflict resolution mechanism
    if(is_drop(dst_device_firewall, device) ||
        is_drop(dst_device_pol, device)){
        drop(p);
    } else {
        forward(p, dst_device_firewall); //assume both forward to the same device
    }
}
```

on its structures, their initialization is encapsulated in another important method: `nf_init`.

Function nf_init. This function is executed before `nf_process` and is responsible for the initialization of all data structures or other variables used by the NF. It does not receive any input, and its return indicates whether all of them were correctly initialized or not. The main data structures are:

- **map** The map is used to store integers associated with a key of arbitrary type. This is internally implemented as a hash table.
- **vector** The vector is used to store arbitrary type data associated with an integer index.
- **dchain** The dchain (double-chain) is a double linked-list of integers. The values of the stored inte-

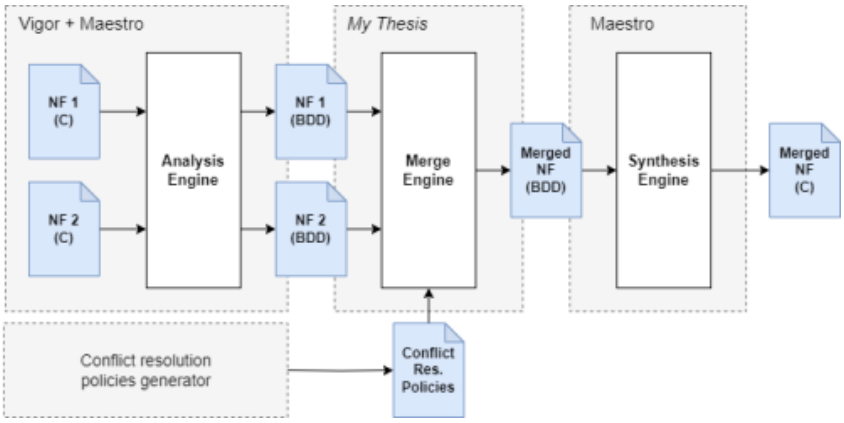

Figure 3.3: Architecture of our solution

gers are not chosen by the user but by the dchain itself, which is configured to store a limited total number of integers. When using this data structure, one can use the libVig API to allocate (store) a new index, verify if a specific index is already allocated, or rejuvenate a previously allocated index. When prompted to allocate a new index, the dchain marks one of its integers as reserved and returns it to the user. Each index is associated with a timestamp and is timed out if it passes a certain fixed amount of time without being asked to rejuvenate by the user. When an index is timed out, it becomes unallocated.

In Listing 3.4 we define the data structure initialization method for both the firewall and the policer from the previous section. The firewall uses the map to store current active connections, and policer to store information to calculate the incoming packet rate. If the initialization of their data structures fails, the program aborts. When combining stateful NFs, one must ensure that the new NF contains all of the data structures of each one of the constituent NFs.

Function nf_process. Here is where all packet process logic is encapsulated. Listings 3.1 and 3.2 represents a simplified version of the packet processing method of Vigor's firewall and policer (respectively). The function receives as input the packet, the arrival time, and the source device. During its process, the developer may change certain fields of the packet, as libVig provides methods that allow the direct modification of Ethernet, Internet Protocol version 4 (IPv4), and Transmission Control Protocol (TCP)/User Datagram Protocol (UDP) headers. Different NFs may change the same fields of the packet, or access some fields that the other does not: a firewall uses three specific chunks of the packet (Ethernet, IPv4, and TCP header), but the Policer only needs the TCP header. During the combination process, one must decide which changes persist, and how to ensure that the resulting combination covers all packet fields used by each one of the constituent NFs.

Besides packet changes, stateful NFs are continuously accessing specific indexes on its own data structures and storing information. When combining stateful NFs with a large number of data structures (like maps or vectors), it would be optimal if we could reduce the number of accesses to those structures, and consequently, have a positive impact on NF performance. Our tool serves as a starting point for future work, where both network functions may take advantage of resource sharing by merging data structures whose accesses are made to the same index.

After packet processing the function returns a decision regarding the packet's fate. If dropped, the function returns the same device number it received as input. If the packet is forwarded, it returns the destination device number or a simple constant indicating the broadcast procedure. Although, different NFs can choose different actions or even forward the packet to different devices, e.g., the firewall could forward a packet from a known connection, but Policer could drop it because it's out of the rate limit. It is up to the developer to decide the correct actions to the desired new functionality.

Our tool takes advantage of the Vigor's verification framework that uses symbolic execution. In this

process, NF's control flow is followed symbolically, exploring different paths it can take. At branches (if-else statements), the execution splits, considering both true and false conditions. As the program executes symbolically, it collects constraints on the symbolic values based on the conditions encountered. These constraints are logical expressions that define the conditions for reaching specific program states. This will result in a sound and complete representation of the program - the BDD - that won't be used in the verification process (our goal is not to synthesize a verified NF) but instead as input to the merge engine that is responsible for the merge process.

Listing 3.4: nf_init code (from Firewall and Policer)

```
1 bool nf_init(void){
2     map = initialize_map(map, capacity);
3     return map != NULL;
4 }
```

3.3.2 Binary Decision Diagrams

The byproduct of Vigor's verification process, the Binary Decision Diagram, is a binary decision tree that represents the NFs, whose nodes correspond to either calls to packet management functions, calls to libVig functions, or conditional statements. Every time a conditional node is added to the tree, it bifurcates into two subtrees, one corresponding to the flow of the program in which the condition is true, and the other to the scenario where the condition is false.

Using symbolic representations and symbolic values instead of concrete ones is like giving each program the ability to understand and work with its own set of data without affecting the data of other programs. Each node in the tree operates in its own symbolic context, without changing the actual values of other nodes. Furthermore, it eases the process of finding compatible pieces of code, an important step in the merge process described in the next section.

Now that we understand the importance of using those representations in the composition process, it is important to be familiar with the 3 main types of nodes and their impact in the merge process.

- **Call nodes**. These nodes in the BDD represent calls to packet management functions or calls to libVig functions.
- **Branch nodes.** These nodes are simply conditional statements made with klee expressions. They represent decision points in the program's control flow. These are points in the code where the program can take different paths.
- **Return nodes.** Represent the return of both initialization and packet process methods. When used

in `nf_init` indicates whether the data structure initialization was successful or not. In `nf_process` indicates whether the packet is dropped or not, and if its forwarded, to which devices.

To better understand how each type of node influences the composition process, we rely on the BDDs from our running examples introduced earlier: the firewall (Fig.3.6) and the policer (Fig.3.5).

Represented in blue are the branch nodes. One symbolic value represents one of `nf_process` inputs: `VIGOR_DEVICE`. This value is the source device of the packet. It bifurcates in *TRUE* or *FALSE* because both NFs assume that there are two devices (0 and 1) that are often used to represent a Local Area Network (LAN) or a Wide Area Network (WAN). The other branch nodes belongs to the result of data structure operations. The conditions inside the branch nodes are used to decide which paths between two NFs are compatible, and therefore to find potential conflicts.

The nodes colored in white represent packet management functions. These nodes are used to store all modifications made to the packet. All methods provided by Vigor's framework to access specific packet fields are built on top of the `packet borrow` method. Taking a look at the Policer's BDD, the first `packet_borrow` method receives as input the packet itself (represented by the symbol `packet_chunks`) and the size, that is 14 bytes for this type of header. For each one of those method calls, there is always one corresponding `packet_return` that will store all the changes made to that specific packet chunk. When running symbolically, `packet_borrow` methods will store an expression of N bytes that represents the initial chunk state, which in the end will be compared to the expression of the corresponding `packet_return`, to detect potential changes to that field. To ease this process, those methods rely on an intern counter whose purpose is to be used as an offset. If you observe the firewall BDD, it uses the TCP/UDP header (with 20 bytes of size). To ensure that the corresponding `packet_borrow` is pointing to the correct chunk, its current offset has to be set to 34 bytes (from the Ethernet header and the IPv4 header). The counter is then decreased when reaching the corresponding `packet_return` method. As is evident, different NFs can use different chunks of the packet, with different sizes, and can also modify them. When combined, a new set of chunks must not break the semantic of both NFs, one of the main challenges.

The data structure methods are visually denoted by the gray nodes. Every node in the BDD is identified by a unique ID. Functions like `map_contains` and `map_get` generate symbols whose name is related to the node ID it belongs. In the firewall, the `map_contains` method generates a symbol - `map_has_this_key_07` - that indicates if the call made by node 7 found an object to the corresponding key or not. This symbol will then be used by a branch node and originate two new paths of the program. This type of node raises other challenges to the combination process: it is necessary to maintain the nodes' original sequence from the respective BDDs they originated from, and ensure that BDDs do not have nodes with identical IDs that generating identical symbols referencing different functions.

Finally, the green and red nodes represent the end of each path of the BDD: the action returned

by the `nf_process` method. They are identical to their original code structure: store an action (forward, drop or broadcast) and the device in case of not dropping the packet. When combining NFs, they can generate different actions for the same class of packets, or even route them to different devices. It's important to ensure that the new actions resulting from the combination process do not break the semantics of each constituent NF.

The `nf_init` method is also represented with its own BDD, as in Fig. 3.4. Here arise again the challenges of node dependency and ID conflicts addressed above. Taking a closer look to this BDD, the first node with ID 0 represents the map allocation method that generates a symbol corresponding to whether the initialization was successful or not, used as a branch node (ID 1). First, is necessary to ensure that each node ID is unique in the context of both NFs. Second, the merge of all nodes into one single BDD must not break the semantics of the initialization process, i.e, each branch node needs to be inserted after the initialization method that generated its symbol: there is no point in checking whether the initialization was successful before it happens.

Now that we detailed the basics of Vigor's NFs and its symbolic representations, as well as the challenges behind them, we will introduce, in the next section, our solution to the combination process.

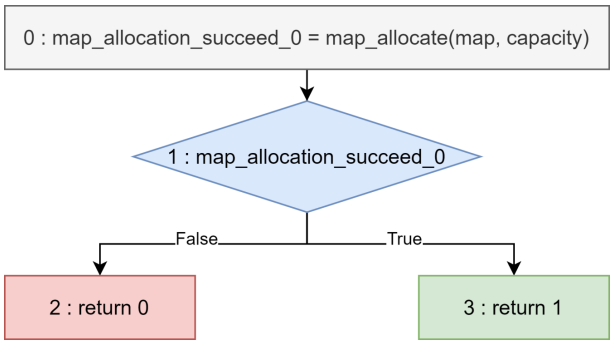

Figure 3.4: nf_init BDD from Policer and Firewall

3.4 Merge engine

The main goal of this module is to combine two or more representations of network functions into a new one representing a combination of both, while preserving the correctness of each one of the constituent modules. To achieve this, we start with the sound and complete representation of both constituent NFs, detailed in the previous chapter.

In the first subsection, we will focus on the merge process, go through each of its steps, and present implementations details. Then, we will explore each type of conflicts that can happen and how to define conflict resolution policies to solve them.

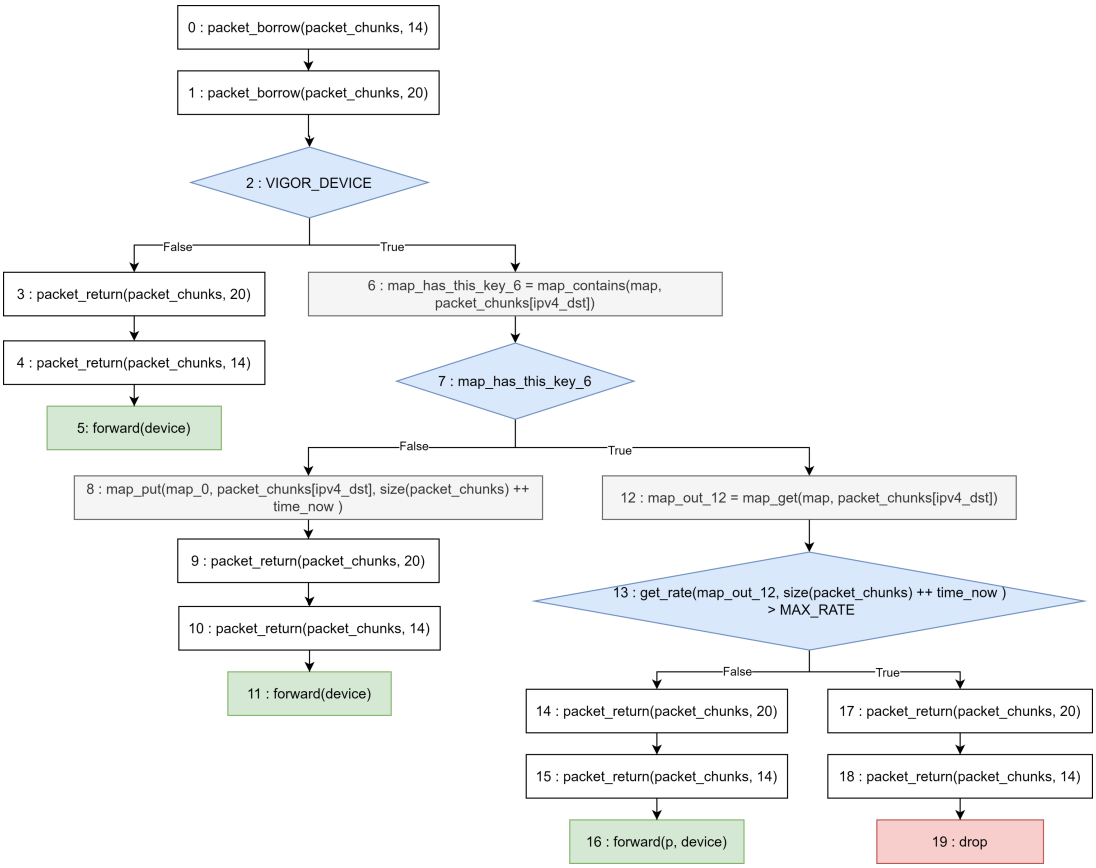

Figure 3.5: Policer BDD

3.4.1 Automated merge process

In this section we present our proposal to fully automate the merge of two NF representations, in the absence of conflicts. We use firewall and policer as running examples, leaving resolving conflicts for the next section. For that reason, we use a naive resolution approach for potential conflicts here: prioritizing firewall's decisions.

Pre-merge phase. In the previous section, we observed that one of the problems of the merge process relies on the node ID. Each node is identified by a unique ID number (unique in the context of its representation) and it is used to identify dependencies between nodes. Taking a look at Fig.3.4, this representation will be the same for both firewall and policer, as each one of them only uses a map. This originates a problem. In the merge process, we would have 2 branch nodes whose condition would be the same: `map_allocation_succeed_0`. Although they are identical, they refer to different map initializations, and we need both to verify the success of the initialization of each one of the maps. For that reason, before the merge process, our tool ensures the uniqueness of the nodes between the constituent BDDs. Consider two BDDs, BDD_A with N nodes and BDD_B. The first step is changing the node IDs of BDD_B , starting in $N + 1$.

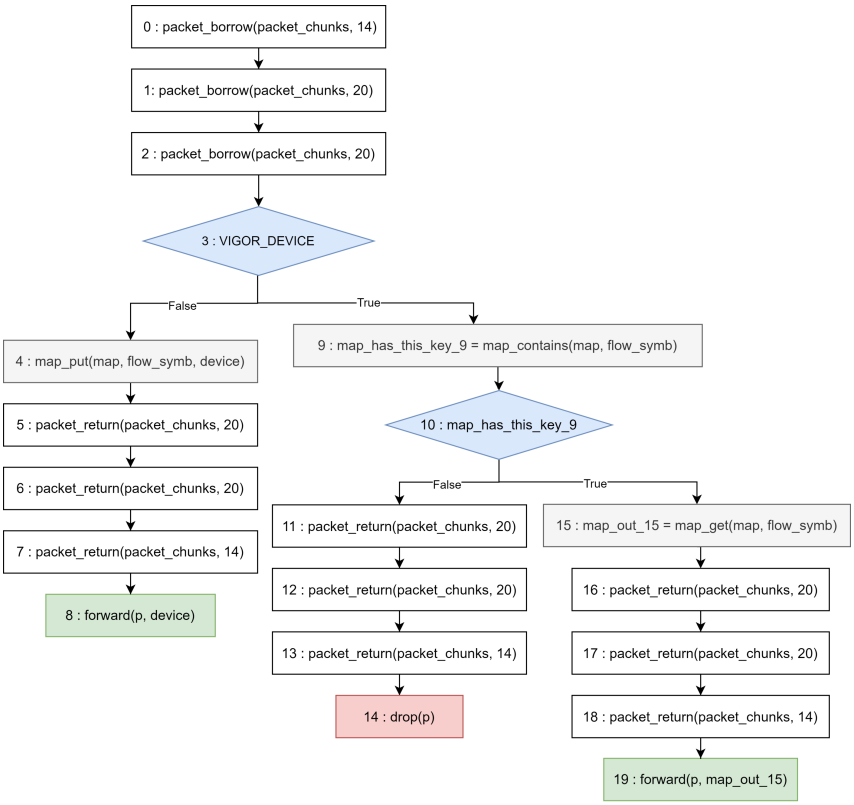

Figure 3.6: Firewall BDD

The process of updating the node IDs in a BDD is based on a Depth-First Search (DFS) algorithm, which iterates through each node and updates the respective identifier field. In more complex cases, such as nodes that generate symbols used by others, that, for this reason, contain the ID of the parent node in their name, it is necessary to recreate the names of these new symbols with the new parent node ID, and these translations are recorded in a translation table. Each time the algorithm updates a node's ID, it also checks whether the symbols used in the node's expressions need to be updated or not. The result of applying this process to the `nf_init` representations of both NFs is represented in Fig. 3.7. In our example, we chose to make the change on policer's BDD and the result is listed in Fig.3.8 .

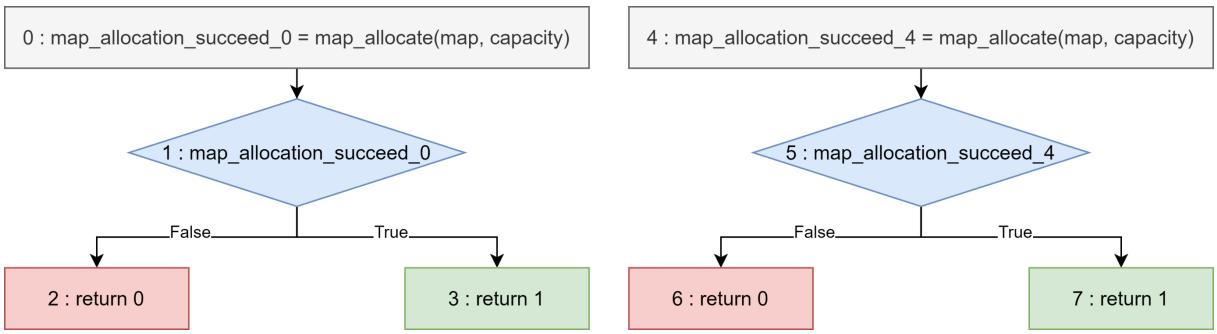

Figure 3.7: nf_init BDDs from policer and firewall with unique IDs

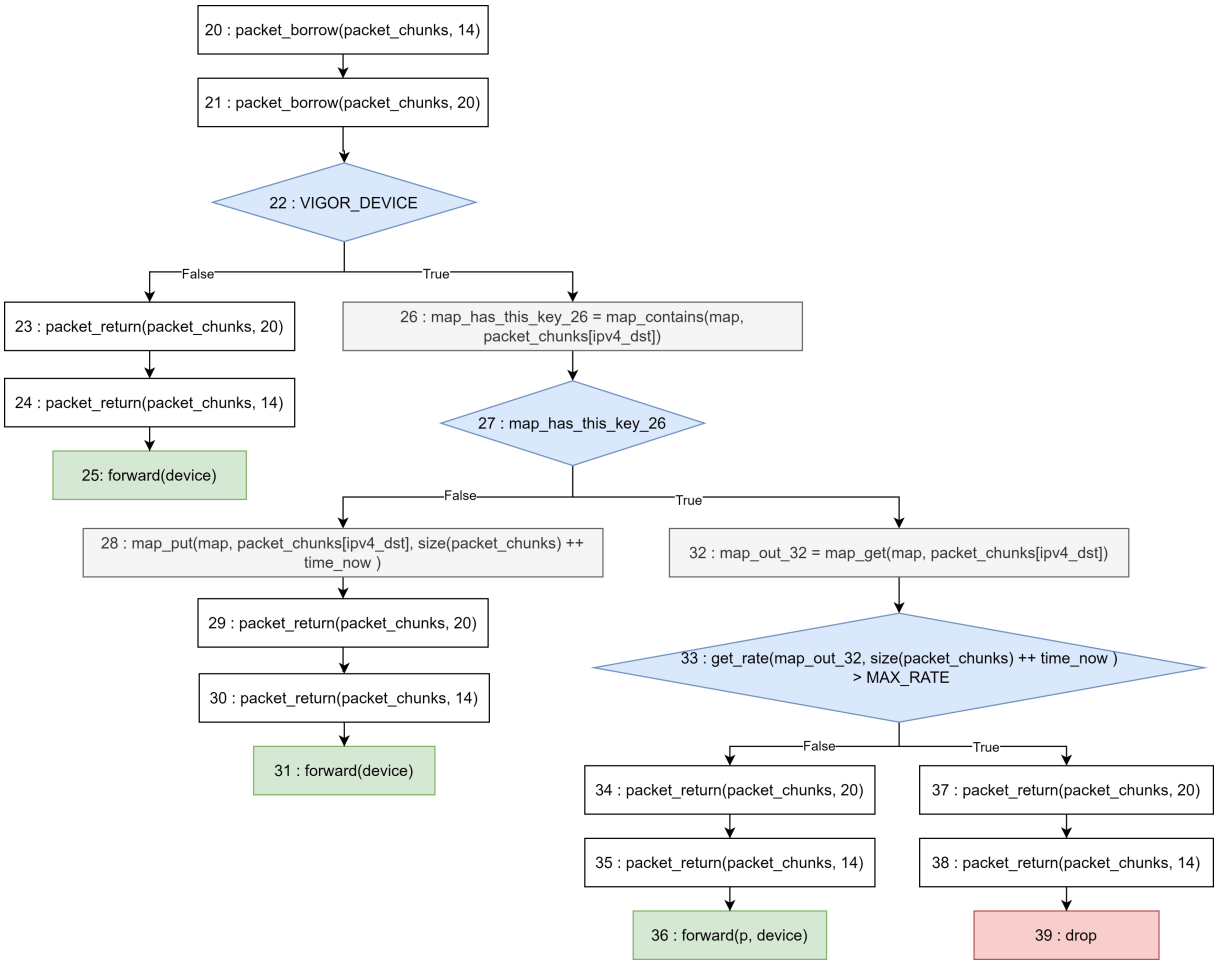

Figure 3.8: BDD of Policer's nf_process method after changing IDs

Path compatibility. Finding pairs of compatible paths is the first and most important step of the merge process. It is crucial for the conflict resolution process: one can see a network function as a module that returns a modified packet and an action, and this way we can pair all compatible actions and modified states of the packet to solve any conflict that may happen. The question is, what makes two paths compatible? Before answering this question, one must understand the concept of path constraints. Path constraints are a conjunction of all branch nodes (that represent if-else statements) that are present throughout a complete path of the program.

Assuming that, one can say that two paths are compatible if and only if the conjunction of their path constraints is SAT. In other words, if there's a class of packets that both paths can process. We solve this problem by resorting to Z3, a Satisfiability Modulo Theories (SMT) solver, that is responsible for finding solutions to logical formulas. These logical formulas can include booleans, bit-vectors, and arithmetic and logical operations. To use the solver's capabilities to verify path compatibility, one must first build the statement to be given to Z3. We want the query given to Z3 to encode the following problem: for a given

conjunction of all constraints from two different paths, find at least one possible packet that satisfies them all.

To better understand this property, let's explore 2 different scenarios. In Fig. 3.9 we have two different paths. The right path belongs to the policer and the left one to the firewall. We will denominate the Policer path as P_{pol}, and the Firewall path as P_{fw} throughout the subsection.

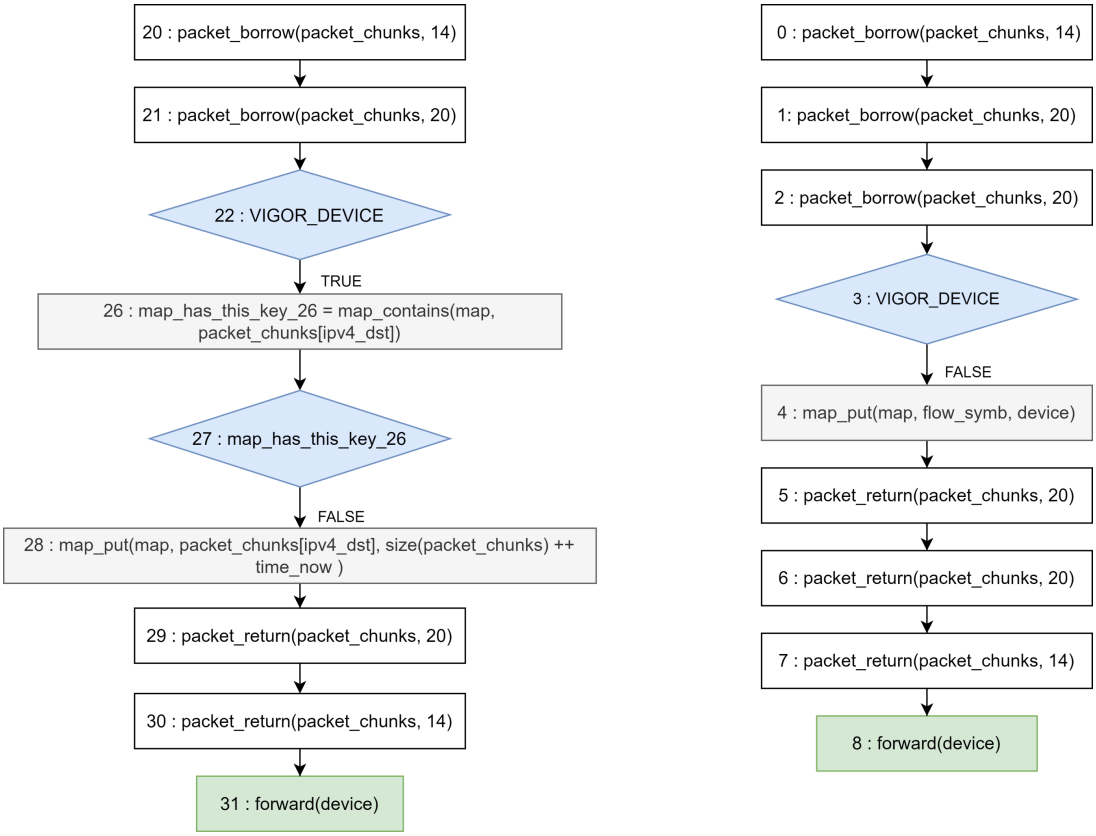

Figure 3.9: Incompatible paths from policer (right) and from firewall (left)

By the constraints on the P_{pol}, one can say that it deals with packets coming from a WAN device ($VIGOR\_DEVICE$) that doesn't belong to any of the current connections ($\neg map\_has\_this\_key\_26$) saved in its state. As for the P_{fw}, its purpose is to register in the firewall's state new incoming ($\neg VIGOR\_DEVICE$) connections. To do a compatibility check, we resort to Z3 to check if a packet or a class of packets that can be processed in those conditions exist:

$$VIGOR\_DEVICE \wedge \neg VIGOR\_DEVICE \wedge \neg map\_has\_this\_key\_26 \tag{3.1}$$

In other words, we are asking the solver if there is a packet whose source device is simultaneously a LAN and a WAN device, and that does not make sense. We conclude that it is impossible to find a packet suitable to those conditions, and for that reason the paths are incompatible.

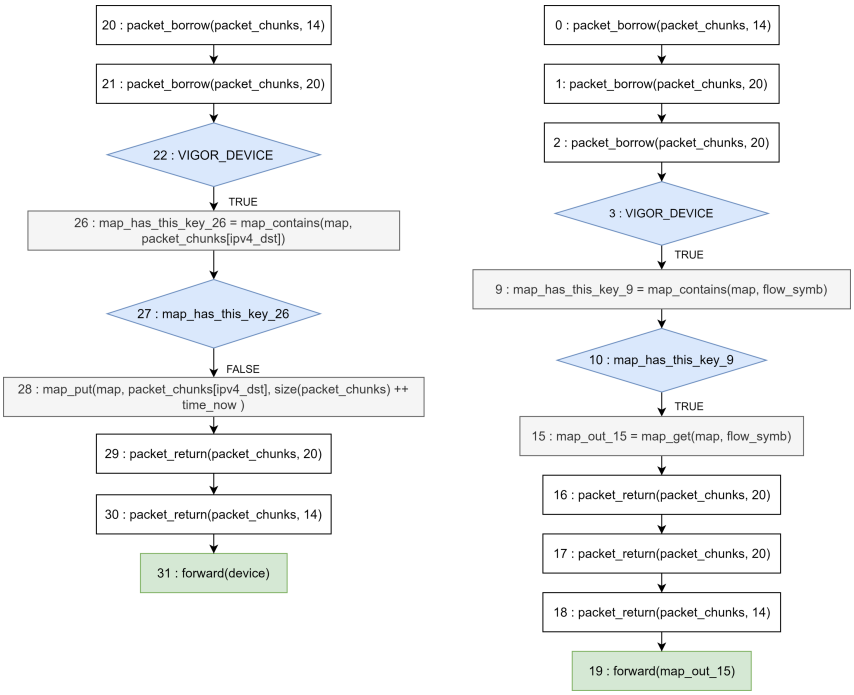

Figure 3.10: Compatible paths from policer (right) and from firewall (left)

Let's take a look at another pair of paths, as shown in Fig. 3.10. In this case, both P_{pol} and P_{fw} share the same source device ($VIGOR\_DEVICE$). Regarding P_{pol} , its next condition ($\neg map\_has\_this\_key\_26$) implies that there's no object in the policer's map whose key is the packet's IPv4 destination address. As for P_{fw} , the firewall's map contains ($map\_has\_this\_key\_9$) the destination device (map_out_15). The resulting condition that will be passed to Z3 to perform the compatibility check is presented blow:

$$VIGOR\_DEVICE \wedge \neg map\_has\_this\_key\_26 \wedge map\_has\_this\_key\_9 \tag{3.2}$$

The conditions $\neg map\_has\_this\_key\_26$ and $map\_has\_this\_key\_9$ belong to data structure related methods (nodes with ID 26 and ID 9, respectively) that access different structures, and for that reason, they are completely orthogonal, i.e., they are independent or unrelated to each other. The compatibility of paths is not influenced by conditions that are solely related to the manipulation of data structures in each of the NFs, as this belongs to the internal context of both. On the contrary, the conditions that are crucial for distinguishing paths are those related to the data shared by both NFs: the packet itself, the device, and the time of arrival. For that reason, sharing the same source device is enough to ensure the compatibility of these paths.

Path combination. After identifying all possible combinations of compatible paths, the next step of the merge process is choosing which nodes from each path will be used in the new final representation and the order of their insertion. In Fig. 3.11 we observe the order of insertion of each node group in the

new BDD.

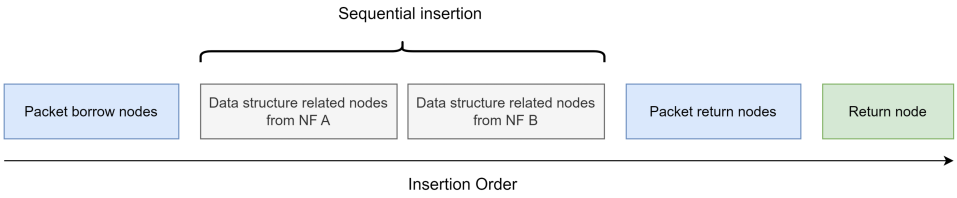

Figure 3.11: Insertion order of the nodes on the new BDD

The first group of nodes that will be inserted in the new BDD are the call nodes of `packet_borrow` methods that represent a chunk of N bytes from the packet, as explained in previous sections. The problem here resides in the following: different NFs may use different chunks from the packet with different sizes. To prevent that, our solution starts by doing an alignment check between the packet borrow nodes from each path. As the size of each packet borrow is represented as a klee expression as a byproduct of the verification process, we resort again to the z3 solver to perform any size comparison, even if those expressions represent constants. In Fig. 3.12 we have 2 different scenarios to help understanding.

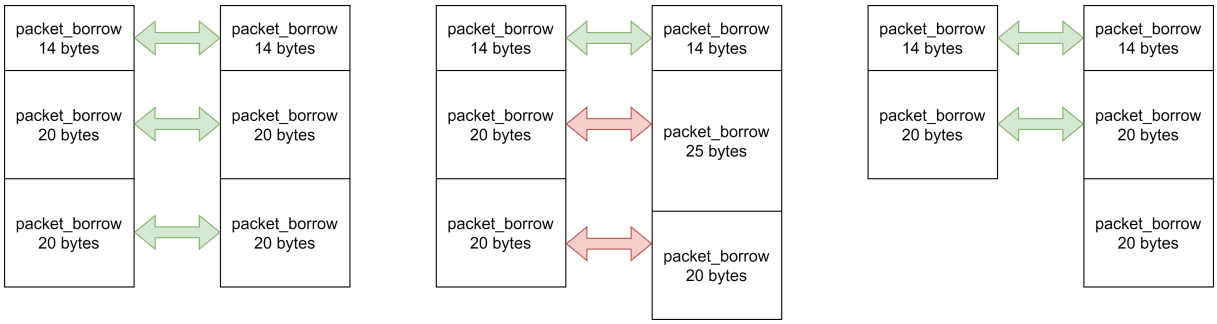

Figure 3.12: Perfect alignment scenario (left), chunk size mismatch (middle), and scenario from the running example (right)

On the left side, we have what we would expect for every NF we are dealing with our tool - the 14-byte borrow represents the Ethernet Header, the other ones the IPv4 and TCP/UDP Header, respectively. Each one of the NFs is using the correct functions provided by libVig that provide the right structures to deal with every field within each chunk. On the right side, we have an example of trying to merge 2 NFs where one of them uses directly the `packet_borrow` function to access a 25-byte chunk. In cases like that, our tool will not continue the merge process and will abort, informing the user that there is a mismatch between pairs of chunk sizes. The last scenario (on the right) happens between the example of compatible paths in Fig. 3.10. The policer just needs to use fields from the Ethernet and IPv4 header in its functionality, but the firewall needs the source and destination ports from the TCP/UDP header to be able to register new or verify older connections. To ensure correctness our tool must ensure that every path contains every chunk needed after the combination, and for that reason, all three chunks

must be used. In Fig. 3.13 we show which chunks and in what order will be inserted in the final BDD after this first stage. As the first two pair of chunks are perfectly aligned, by default the chosen ones are the nodes with ID 20 and 21. The last chunk comes from P_{pol} (node with ID 2) because it is the only path that has it.

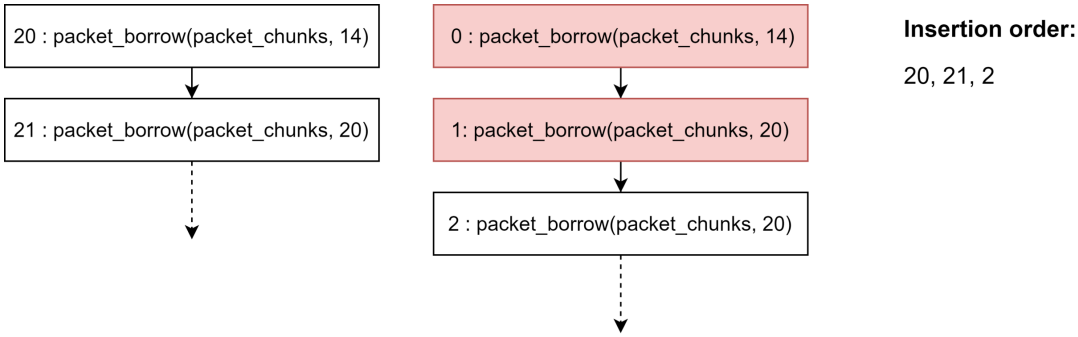

Figure 3.13: Merge of packet_borrow nodes from the P_{fw} and P_{pol}

The next group of nodes being added to the new BDD (Fig. 3.11) are the data structure related nodes and the nodes that only interact with the internal context of the NF. As we have seen earlier, this type of node mostly represents functions that deal directly with the data structures used in libVig, as well as any branch node with symbols that depend on them. In this stage, our tool proceeds to add the nodes from both paths in a sequential way to preserve dependencies that might happen between nodes, as we see in Fig. 3.14. The conditional nodes with ID 27 and 10 contain symbols that are generated from nodes 26 and 9, respectively, and for that reason they cannot appear before these operations. After this stage, the current insertion order of the nodes in the final BDD is: 20, 21, 2, 22, 26, 27, 28, 9, 10, 15.

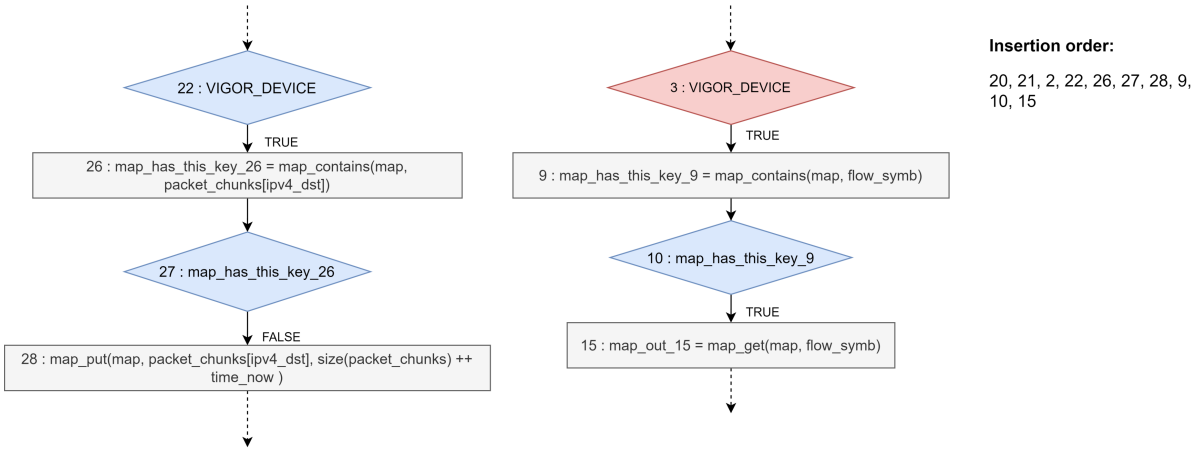

Figure 3.14: Merge of data structure related nodes and other types from the P_{fw} and P_{pol}

At the end of each path, that corresponds to the leaves of the BDD, we have the last two groups of nodes. The first one is responsible for storing all the changes made to the packet (the group of nodes

representing `packet_return` functions). Here, our tool ensures that the chosen nodes contain all the modifications made from both paths. But a question remains: how do we detect packet modifications?

Each `packet_borrow` node stores an expression representing the chunk of N bytes from the `packet_chunks` symbol , which represents the initial content of the packet and which offset is defined by the internal counter used, as explained in the previous section. The corresponding `packet_return` node also stores an expression representing the chunk that may or may not have modifications to the packet. In Fig. 3.15, we show how to proceed when both NF paths contain packet changes. Instead of choosing one of the packet returns, our tool creates a new one with a new chunk expression. To build this new expression, our tool proceeds to iterate over the bytes from both `packet_return` expressions, and using the Z3 solver tries to identify if the byte was modified, comparing with the byte from the corresponding `packet_borrow`. If so, this byte is concatenated to the new expression. For untouched bytes, our tool by default chooses the byte from P_{fw} .

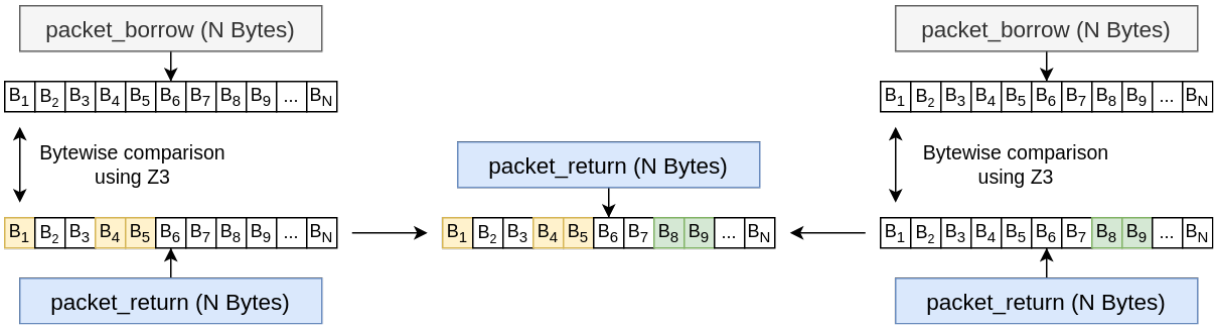

Figure 3.15: Packet content merge process

Applying this process to our running example on Fig. 3.10, the first step would be merging all the changes to the last chunk used, which in this case represents the TCP/UDP header. This chunk is only used by the firewall. So, for that reason, we are able to skip the verification of changes and choose directly the `packet_return` node (ID 16) from P_{fw} . Regarding the Ethernet and IPv4 header, as both paths use those chunks, for each of them we must create a new `packet_return` node whose expression stores all the changes made from both paths, following the process on Fig. 3.15 . As none of the paths apply changes to these 2 last chunks, our tool choose arbitrarily each byte concatenated to the packet chunk expression of each the new nodes: the node with ID *new* (that represents the merge between the nodes with ID 29 and 17), and *new_2* (that represents the merge between the nodes with ID 30 and 18). Fig 3.16 shows the order of insertion of these nodes until now.

The last node from each path represents the return of the `nf_process` method. The node with ID 31 forwards the packet to the destination device represented with the symbol `device`, and the node with ID 19 forwards to the device `map_out_15`. The focus of this section is not detect and resolve conflicts, and for that reason, using the naive resolution approach described earlier, we choose the node with ID 31.

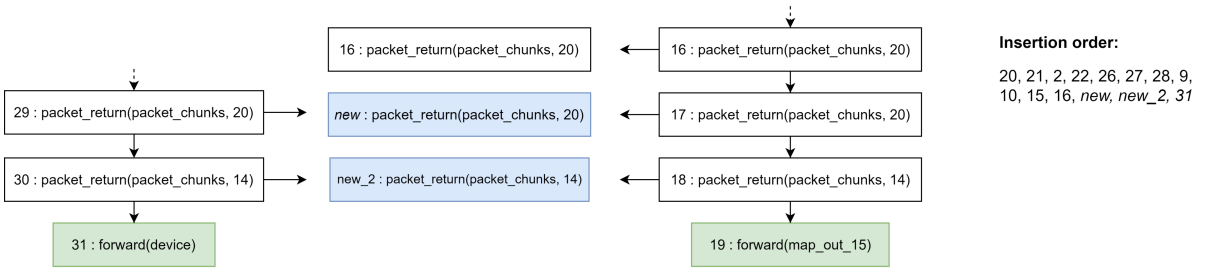

Figure 3.16: Resulting nodes from process in fig. 3.15 and final insertion order list

Building the final BDD (nf_process). At this stage of the merge process there is a list of nodes waiting to be inserted in the final BDD. The final process is simple: go through the list of nodes and insert each one of them. However, this raises a question: When inserting a new node in a binary decision tree, one can perform a depth-first search to find a suitable place for the node; but when reaching a conditional branch, how do we know which path it should take?

Here we introduce the concept of node constraints, the set of constraints of each node is represented by the conjunction of conditions from branch nodes that precedes it. This concept is better visualized if applied to our running example from Fig. 3.10. In Table 3.1 we represent the node constraints for each one of the chosen nodes as well as their insertion order.

	Constraints	Nodes (ID)
	none	20, 21, 22, 2
C_1	$VIGOR\_DEVICE$	26, 27, 9, 10
C_2	$VIGOR\_DEVICE \wedge map\_has\_this\_key\_26$	28, 31
C_3	$VIGOR\_DEVICE \wedge map\_has\_this\_key\_9$	15, 16
C_{new}	$VIGOR\_DEVICE \wedge map\_has\_this\_key\_9 \wedge map\_has\_this\_key\_26$	new, new_2
Insertion order	20, 21, 2, 22, 26, 27, 28, 9, 10, 15, 16, new, new_2, 31	

Table 3.1: Node constraints for each one of the chosen nodes from the running example

As the node insertion process involves too many steps, we will simplify it by introducing an intermediate step for explanation. In Fig.3.17 we represent the insertion of the node with ID 9 in the new final BDD. For simplicity, let's assume that this was the first pair of paths to be processed and, for that reason, the root of the new BDD starts in the node with ID 20.

The insertion process proceeds to do a depth-first search starting at the root of the new BDD. Then, it goes through each one of the call nodes until it finds the first conditional branch (with id 22). Here, our tool resorts to the Z3 solver to choose which path the node should take. This conditional branch denotes two main classes of packets: packets whose source device is classified as a LAN ($false$ branch) or a WAN ($true$ branch) device. Looking at the constraints from the node with ID 9, C_1 , we observe that it processes packets coming from WAN devices, and for that reason, the algorithm will continue the insertion process on the $true$ branch.

Insertion order: 20, 21, 2, 22, 26, 27, 28, **9**, 10, 15, 16, new, new_2, 31

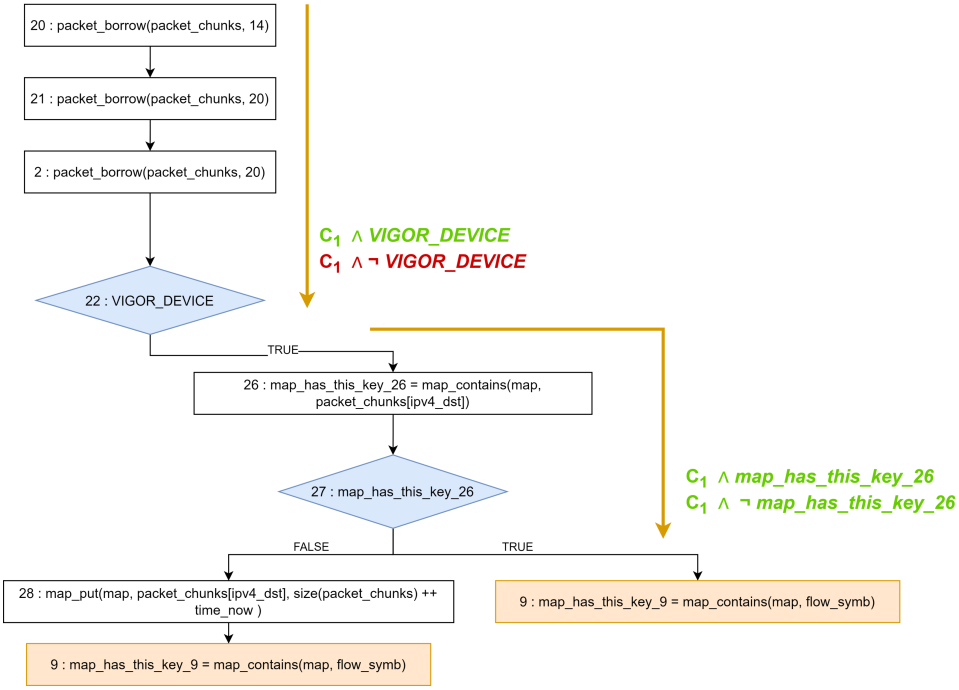

Figure 3.17: Intermediate step of the node insertion algorithm using node with ID 9 as example

Iterating through the next nodes, it finds another conditional branch (node with id 27), that represents whether the operation on node 26 (verify the existence of a specific key on the firewall's map) was successful or not. In this case, the new node can coexist in both scenarios: as C_1 is related to the packet's source device, which is completely orthogonal to the firewall's map. Thus, it is compatible with the node 27 condition, whether it is evaluated in $true$ or $false$: the algorithm will continue its execution in each branch.

Going through the path on the right ($true$ branch) the algorithm detects that node 27 has no next node, and so it will insert the new node (ID 9). Regarding the left path ($false$ branch), the new node is inserted after the node with ID 28, following the same logic. In general, the insertion algorithm will continue its execution until it discovers an incomplete end of a path (meaning a node that doesn't have a following node), or a node representing a `packet_return` method. In the new BDD, it is ensured that the last groups of nodes in each path are the nodes of the `packet_return` methods, followed by the return node, to maintain the semantics of each constituent NF.

After repeating this process to all pairs of compatible paths, the result achieved is represented in Fig.3.18. Here, the final result still has the old node IDs, for simplification and to ease the understanding of where each node belongs.

The new BDD uses 3 headers (Ethernet, IPv4, and TCP/UDP). For packets coming from a LAN

device ($false$ branch on the node with ID 22), we observe the firewall's functionality of registering the flow in its data structure, which we will denominate map_1. On the other branch, the combined functionality applied to packets from WAN devices is described. The first nodes (ID 26 and 27) denote the policer's functionality, which verifies on the policer's map (map_2) if there are already previous packets to estimate the traffic rate (node with ID 33), and if not, it is stored in map_2 (node with ID 28). After that, we observe the firewall's behavior for packets coming from WAN devices - verify if there is registered a corresponding LAN device for that connection (nodes with ID 9 and 10). Notably, the packet can be forwarded even if it's outside the limit rate ($true$ branch on the node with ID 33), and for that reason, the policer functionality is compromised. This happens because, as said earlier, the focus of this subsection was only to show the main steps of the merge process, and for that reason, we used a naive approach to resolve conflicts between both NFs. The next section will address user-defined conflict resolution policies.

Building the final BDD (nf_init). Just like the function responsible for packet processing, the one responsible for initializing data structures is also represented by a BDD, as shown in Fig. 3.7. Therefore, the final implementation must also present the combination of the structure initialization for both NFs. The combination process is very similar to the one described earlier but with some slight simplifications.

Initially, it is equally important to ensure the uniqueness of the IDs in the context of both NFs. As described earlier, nodes related to data structures generate symbols that represent the result of the executed method, which in turn contains in its name the ID of the parent node that generated it (such as nodes with ID 0 and ID 4). Focusing on this example, if both nodes had the same ID, they would generate identical symbols that would be used to represent different operations.

The insertion algorithm used for the combination process is the one detailed in the previous subchapter. The main difference resides in the node insertion list: in this case, it is enough to iterate through the nodes of each NF (sequentially) and insert them.

This type of BDD is simpler than the one representing packet processing because their nodes are only related to data structure initialization methods and their respective conditional nodes that represent the result of that initialization. Due to the absence of conflicts, it is not necessary to find compatible paths: it should be noted that because the nodes of each BDD are related to different data structures, all paths of both BDDs are orthogonal to each other. Thus, the paths of the new BDD represent all the possible combinations made between the paths of the constituent NFs, as presented on the BDD on the left of Fig. 3.19. Taking a closer look at this representation, it is notable that even if the initialization of the first map fails ($false$ branch on node ID 1), it proceeds to the initialization of the remaining data structures.

An NF can only function if its data structures are correctly initialized. For that reason, the final stage applied to the BDD is to remove redundant code: Our tool iterates through each path that represents the failed initialization of each data structure and removes all the nodes except the one that represents the

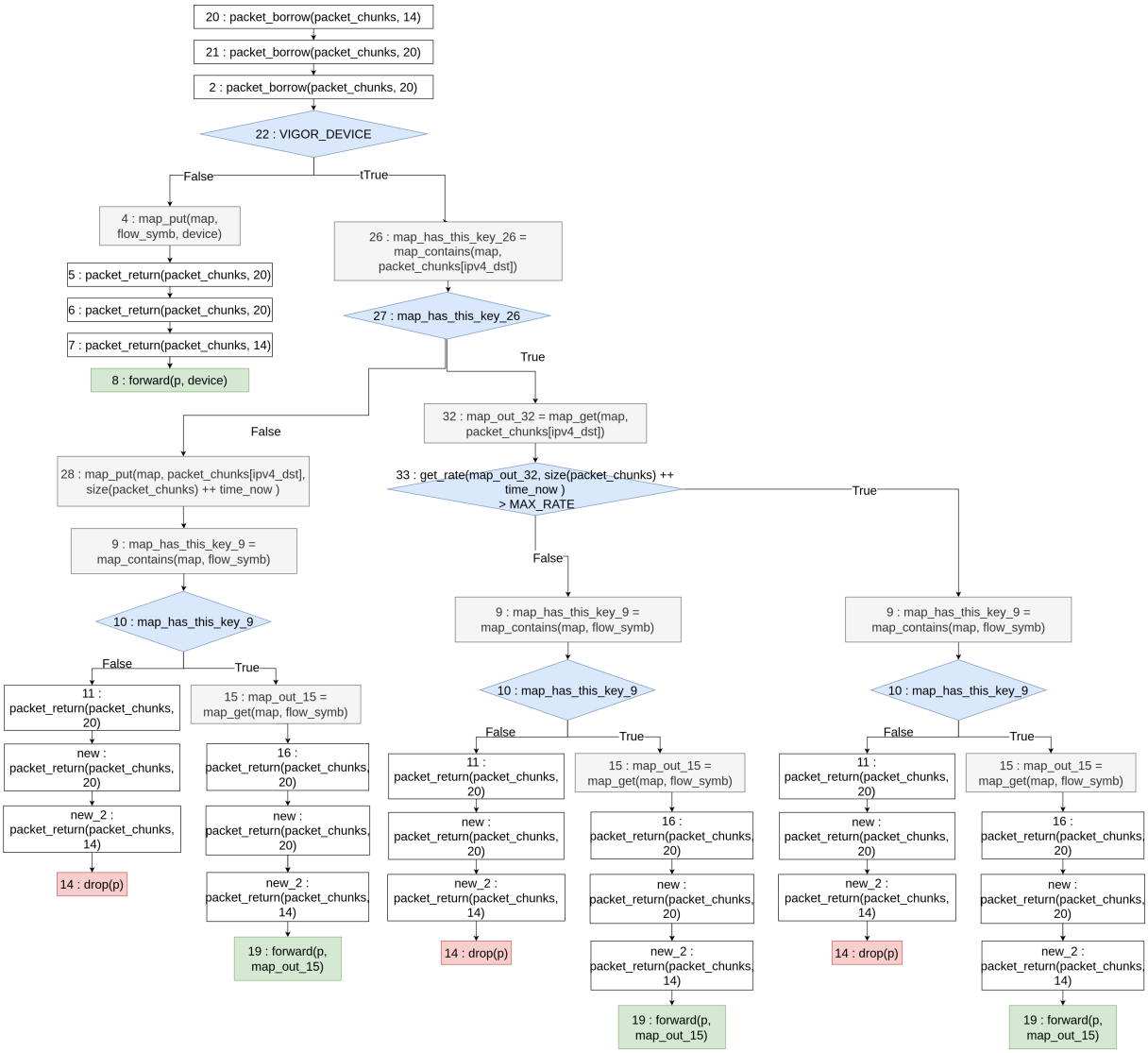

Figure 3.18: Final BDD representing the combination of Vigor's firewall and policer

return of the function. The result is presented in the right side of Fig. 3.19.

3.4.2 User-defined conflict resolution policies

In the previous sub-chapter, we addressed the scenario where our tool was able to merge both network functions automatically - and for that, we assumed a naive resolution approach to solve any conflicts between the Firewall and Policer to simplify the whole process - but this is not always possible.

As said previously, one can look at a network function as a module that receives a packet and returns a modified state of the packet and a specific action, forward, drop, or broadcast. Sometimes, in the combination process, we come across paths that can lead to different packet states or even different

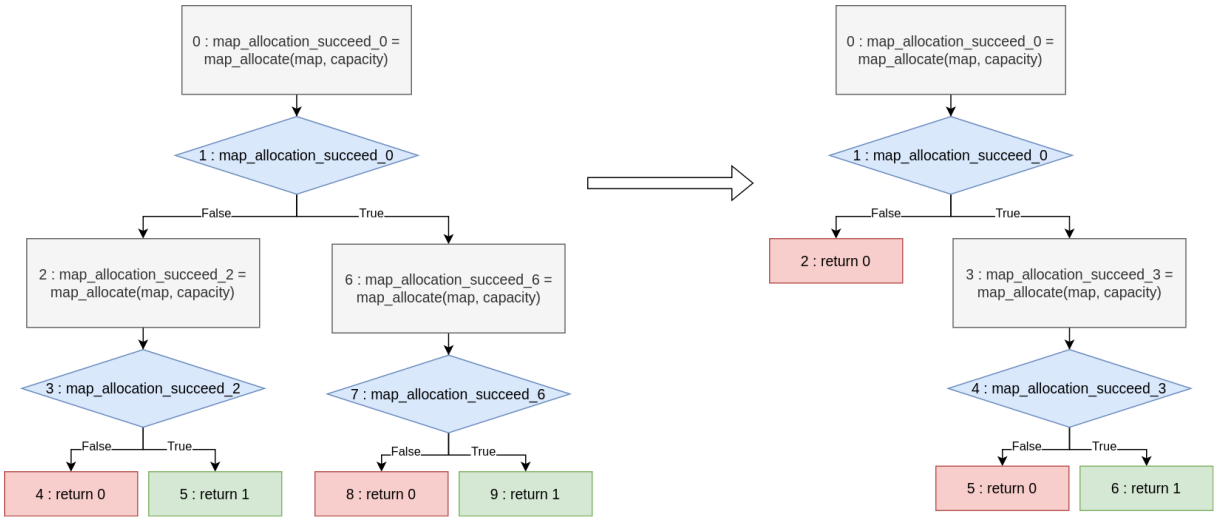

Figure 3.19: Final data structure initialization BDDs

	BDD 1			
BDD 0	Action	FORWARD	DROP	BROADCAST
	FORWARD	BDD_n	BDD_n	BDD_n
	DROP	BDD_n	—	BDD_n
	BROADCAST	BDD_n	BDD_n	—

Table 3.2: State conflict matrix representation

actions. In this case, the merge process cannot be completed, as our tool cannot to infer which packet state or which action suits better for that combination. That is where we rely on the user to define a set of rules to resolve these conflicts.

Before detailing the conflict resolution process, one must understand in detail what kind of conflicts can happen:

Packet content conflicts. This type of conflict occurs when both network functions apply changes to the same packet fields. For example if they write different source IPs in the IPv4 header.

Packet process action conflicts. Occurs when network functions make different decisions about the fate of a specific packet state. For example, if one of them drops the packet and the other one broadcasts it for every device.

If the tool is not able to complete the merge process by itself, the user must provide a configuration file that contains all the rules used by the tool to overcome any type of conflicts that might happen. This configuration is mainly built in two parts: a state configuration matrix and a simple Boolean-type field.

The state configuration matrix (Table 3.2) contains all possibilities of state combinations. For each one of them, the user may choose from which representation BDD_n belongs the correct action the tool must choose.

Choosing a matrix to resolve those types of conflicts gives the user the liberty to create any kind of

rule they desire. For example, if the user wants to drop all packets if the BDD_1 drops, he just have to fill the second column to prioritize drops from this BDD. Even when both NFs decide to forward the packet, despite the action being the same, they can forward it to different devices: in this case, the user must choose the correct device.

Regarding packet field changes, our tool uses the Boolean type field to know which changes it should prioritize when both NFs modify the same bytes. As explained in the previous chapters, all the changes are stored in the packet_return nodes. One knows which changes were made by comparing the packet chunk symbolic expression from packet_borrow with the respective packet_return at the end of the path. Our tool can detect changes with byte granularity. Though, bit-level granularity would be useful as well to handle packet fields smaller than 1 byte. We leave handling such cases to future work.

Having identified the potential types of conflicts that may arise, and how to build a configuration file to help our tool in the merge process, let's revisit our running example regarding the Firewall and the Policer. But first, we need to build a reasonable configuration that ensures the correctness of both functionalities:

- As neither of the NFs apply changes to its packet chunks, one can choose arbitrarily a BDD to prioritize. In this case, we chose to prioritize firewall's behaviour.
- In both NFs, the actions applied to packets sent from a LAN device are always the same: forward the packet to the WAN device. We assume that the device number corresponding to the WAN and LAN devices are the same for both NFs, and for that reason the packet will always be forwarded to the same device. Thus, we can choose arbitrarily which device number must be prioritized.
- The actions applied to packets sent from a WAN device diverge: firewall drops all packets from unknown flows, and the policer drops the ones that exceed the max limit rate. However, not all packets coming from unknown connections exceed the maximum traffic limit. Moreover, there may be packets coming from known connections that need to be discarded as they have exceeded the traffic limit. To maintain the semantics of each NF, we can choose to drop the packets if at least one of them drops.

The resulting conflict matrix is represented below:

	BDD_1 (Firewall)			
BDD_0 (Policer)	Action	FORWARD	DROP	BROADCAST
	FORWARD	BDD_0	BDD_1	BDD_1
	DROP	BDD_0	—	BDD_0
	BROADCAST	BDD_0	BDD_1	—

Our tool receives as input this configuration in JSON format, and so the configuration file that represents all the resolution policies addressed is represented in Listing 3.5. The JSON object contains the

conflict matrix, the boolean `prior_changes` that represents which NF prioritize when both change the same byte in the packet, and the other options that provide the user a graphical representation of the BDD, namely the ability to choose the colors from the nodes from each one of the constituent NFs. The user doesn't need to know how to build a complete JSON file, as our tool already provides a UI that helps the user to create the conflict resolution file. The UI used to generate this file can be visualized in the appendix A.

Listing 3.5: Configuration file regarding user-defined conflict resolution policies

```
#configuration file for our composition tool
#firewall bdd -> 1
#policer bdd -> 0
{
  "conflict_matrix": [
    [0, 1, 1],
    [0, -1, 0],
    [0, 1, -1]
  ],
  "prior_changes": 1,
  "enable_gviz": true,
  "bdd1_color": "#ff0000",
  "bdd2_color": "#c0ff14"
}
```

To provide a better understanding of the differences between the merge with and without user-defined conflict resolution policies, we will take as an example another pair of compatible paths that are represented in Fig 3.20. The left path belongs to the Policer and represents the scenario where an incoming connection is sending a packet with a rate above the limit, and for that reason that packet is dropped. On the other hand, the firewall's path on the right side is forwarding known connections. Using the conflict matrix defined above our tool concludes that the packet should be dropped if at least one of the paths drops it and the `return_process` node chosen to be inserted in the new BDD will be the one that belongs to the Policer. During the merge process, there is a step where each one of the `packet_return` nodes needs to have its expression reconstructed to include all modifications from both NFs (previously addressed in Figure 3.15). Despite not happening in this specific scenario, because none of the paths perform modifications to the packet, if both paths make modifications to the same byte, our tool resorts to the configuration file to decide which change should be concatenated to the new `packet_return` expression.

After applying the process from the previous subsection, but adding the user-defined conflict resolution policies, we obtain the result represented in Fig. 3.21.

This new representation of functionality combination differs significantly from a sequential combination process in several aspects. Specifically, in the case of the firewall and the policer, when combined

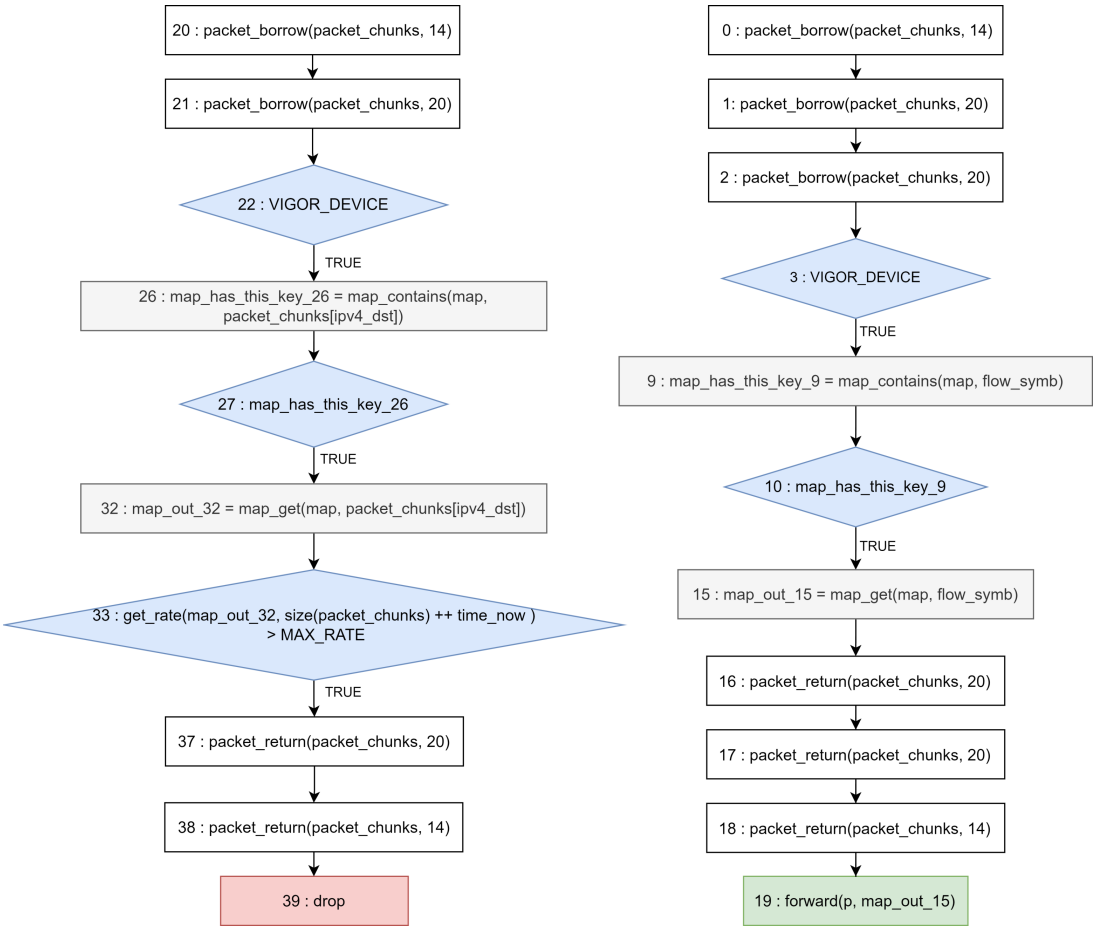

Figure 3.20: Compatible paths from policer (left) and firewall (right) with a conflict an action conflict

sequentially, one of the issues was that the firewall had access to traffic limited by the policer, or due to the policer only having access to traffic allowed by the firewall. Looking again at our implementation, we can observe that when a packet originates from a LAN device, it is registered in the firewall's map. However, when it comes from a WAN device, regardless of not being recognized by the firewall functionality and discarded for that reason (according to the conflict resolution policies that have been defined), we can see that it is still registered by the policer functionality and accounted for the traffic quota, allowing the policer to have access to all traffic, even if it is discarded. Furthermore, since all packet modifications are defined at the end of each path, each node uses expressions based on symbols instead of directly altering the packet's content. This means that each node has access to the original state of the packet, eliminating the need to create copies of the packet for each functionality.

3.5 Synthesis engine

After generating the combined implementation of two NFs using our tool, the next and last step is to resort to Maestro's synthesizer to generate the implementation in C code.

This engine is able to generate a sequential implementation of an NF written in C based on its BDD. The whole process is to transform the BDD in an AST tree and then traverse it and generate C code. This process of translation is straightforward, as the initial code is already in C, and therefore there is no translation between different languages or platforms. The generated C code will, however, be different from the original implementation, as it results in the combination of two NFs.

3.6 Summary

In this section, we start by addressing the NF combination as a program composition problem and discuss its main challenges, as well as how our solution plans to overcome the conflict resolution problem using a policy-driven merge process.

Next, we presented our solution's architecture, from the generation of the BDDs of the constituent NFs, to the synthesis of C code of the final representation, going through all of the implementation details, and using our running examples (firewall and policer) for a better demonstration of the process.

In the next section, we evaluate the combinations performed by our tool, and compare them with manual combinations.

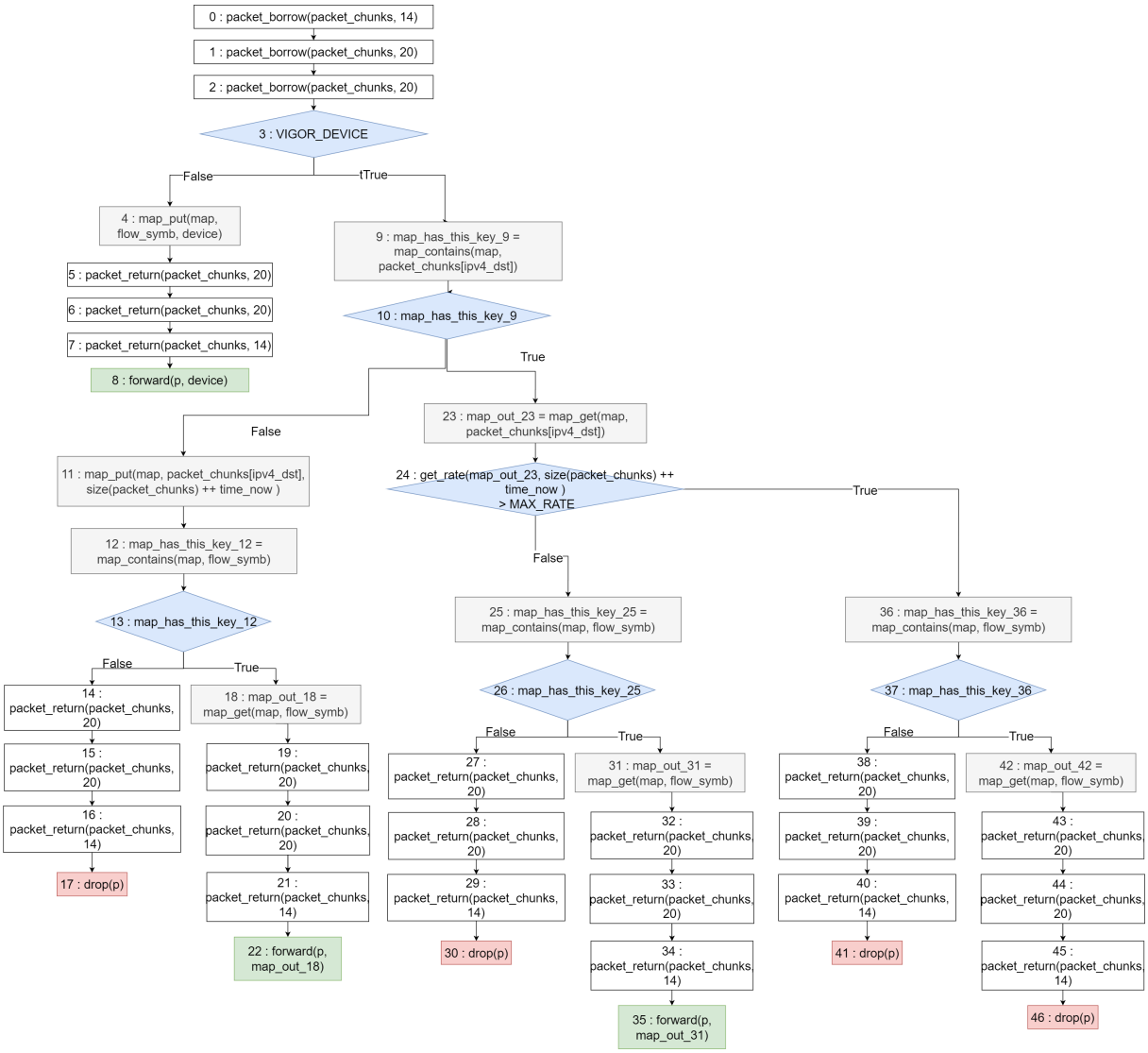

Figure 3.21: Final BDD representing the combination of Vigor's firewall and policer using user-defined conflict resolution policies

4

Evaluation

Contents

In this chapter, we evaluate a set of NF combinations generated by our tool. We aim to provide answer to these questions:

1. How much human effort is saved by automating the merge of network functions?

2. How well do the composed NFs perform, compared with other alternatives?

In the first section of this chapter, we use NFs provided by Vigor to generate a set of combinations and show their respective user-defined conflict resolution policies used in the merge process, as well as the lines of code and spending time of the handmade versions and tool generated ones. This is both used as an example (as each one of the provided NFs constitutes a unique example in the process of combination) and as a means to provide the answer to the first question.

The last question is related to performance, and so it is important to understand this concept in the context of NFs, and how is it measured. Performance in NFs is associated with two concepts: throughput, i.e, the number of packets processed per second and latency, i.e., the time a single packet takes to go to the NF and come back. Thus, the following section explains the methodology behind acquiring values for both throughput and latency throughout the evaluation.

Finally, we close this chapter by evaluating the performance of each combined NF, comparing it to their respective handmade implementations, and thus answering the second question.

4.1 Microbenchmarks

At the time of writing, Vigor provides seven NFs: vignop, vigpol, vigfw, vigbridge, viglb, vighhh and vignat. In this section, we will pick 3 of them (vigpol, vignat, and vigfw), and explain its functionality, and show how our tool can be used to combine them.

Although pseudo-code for each NF is provided, it is used as a means to illustrate their functionality. It is not complete and does not incorporate all the features present in the NFs. Furthermore, state-related data structures and calls to the libvig API are largely simplified. These simplifications, however, only omit details that do not influence the combination process.

4.1.1 Firewall and Policer

Vigor's implementation of a firewall, vigfw, only allows packets from the WAN associated with flows started by packets that came from LAN. Its pseudo-code is illustrated in Listing 3.1. Vigfw uses a map to store the flows and accesses it with flow information, i.e., IP addresses and TCP/UDP ports. However, packets that come from the WAN trigger accesses to this map with inverted IP addresses and inverted TCP/UDP ports.

A policer aims to limit a specific user's download and/or upload rate according to a previously established contract. Vigpol, Vigor's policer implementation configured with two ports (LAN and WAN), limits only the download rate, identifying a specific user's download by the IPv4 destination address of the packets coming from the WAN. Its pseudo-code can be found in Listing 3.2.

The behavior of each NF is contingent upon the origin of the packets, whether they originate from a LAN or a WAN device. This indicates that the NFs are designed to respond differently based on the traffic's source, whether it be from within the local network or from an external wide area network.

If we assume that the device numbers of LAN and WAN are the same for both NFs, we can create a new network function that only allows incoming traffic from a known flow, but with a limited rate. On the other hand, instead of limiting the download rate of known incoming flows, we can invert the LAN and

WAN devices and generate a new one that only accepts incoming traffic from known flows at any rate, but limits the upload rate.

The user-defined conflict resolution policies will be equal for both versions: we want the packets to be dropped if at least one of the NFs decides to drop it. As for packet changes, there are no conflicts because none of the NFs modifies packet fields. It is important to refer as well that if both NFs forward the packet, as the network topology is the same (one WAN and LAN device), it doesn't matter which decision we choose. The conflict resolution matrix is represented below:

	BDD 1 (Policer)			
BDD 0 (Firewall)	Action	FORWARD	DROP	BROADCAST
	FORWARD	0	1	1
	DROP	0	—	0
	BROADCAST	0	1	—

4.1.2 Nat and Policer

A Network Address Translator (NAT) translates addresses between network address spaces. Their most common use is to allow communication between devices inside a private network with devices on the Internet whilst sharing only a single public IPv4 address.

Vignat, a NAT implementation in Vigor, stores flows coming from the LAN and allocates a port that is used to index that flow. When a reply packet from WAN is sent to that port, vignat checks if the IP source address and TCP source port match the IP destination address and TCP destination port of the stored flow, translating and redirecting it to the LAN if this check holds. The pseudo-code is shown in Listing in 4.1.

As in the previous example, we can make two combinations of a NAT and a Policer, based on whether their LAN and WAN devices match. If we assume that the device numbers of LAN and WAN are the same for both NFs, one can create a new network function that stores flows coming from the LAN but just forwards the packets that are within a specific limit rate. On the other hand, instead of limiting the download rate, we can invert the LAN and WAN devices and generate a new one that performs IP translation of known flows while limiting the upload rate.

The user-defined conflict resolution policies will be equal for both versions: we want the packets to be dropped if at least one of the NFs decides to drop it. Regarding packet modifications, despite NAT making packet modifications due to its translation process, there are no conflicts as the Policer doesn't perform any packet field modification. It is important to refer as well that if both NFs forward the packet, as the network topology is the same (one WAN and LAN device), it doesn't matter which decision we choose. The conflict resolution matrix is represented below:

	BDD 1 (Policer)			
BDD 0 (NAT)	Action	FORWARD	DROP	BROADCAST
	FORWARD	0	1	1
	DROP	0	—	0
	BROADCAST	0	1	—

4.1.3 Results

The comparison between the time spent on the manual merge versus the one spent by our tool is presented in the Table 4.1. Each one of the results regarding the execution time measured in our tool represents the median of 10 experiments conducted. The time was measured from the generation of the BDDs for both constituent NFs to the synthesis of the code from the resulting BDD. Regarding the execution time results for the manual merge, they also encompass the total time dedicated to bug fixing and testing to verify the validity of the outcome.

Execution Time (minutes)	Proposed Tool	Handmade
fw + polup	4.15	42
fw + poldown	4.10	2220
nat + polup	4.28	24
nat + poldown	4.26	120

Table 4.1: Comparison between the execution time (in minutes) of merged NFs with and without the use of the tool

The time required by our tool to perform the merge is quite similar for any of the combinations (about 4 minutes). This is because the functionality of the firewall is very similar to that of the NAT. The only difference lies in the NAT translation process - which requires more nodes in the BDD to be merged - and this is reflected in the small time difference between the combinations that use the firewall (4.15 and 4.10 minutes) and the NAT (4.28 and 4.26 minutes).

The improvement is quite significant when compared to the results of the manual merge. As is evident, the obtained results exhibit significant variation across different combinations. It is noticeable that the `fw+poldown` combination took about 2220 minutes (almost 37 hours) to complete, in comparison to the others. This time difference was mainly due to the programmer expertise: it was the first combination made. The second version of this combination, `fw+polup`, took considerably less time since the code for both NFs was already combined, and only required the adjustment of code related to the Policer functionality to limit uploads.

Regarding combinations involving the NAT, there is also a noticeable time discrepancy in manual merging. The initial version, `nat+poldown`, took approximately 2 hours due to the combination of all code present in multiple files of both NFs. In the second version, `nat+polup`, as explained earlier, the time taken for the combination was about 24 minutes, as it only required an adjustment in the Policer logic to limit upload rates.

The results regarding the lines of code used by each combination are illustrated in Fig. 4.1. In gray we represent the number of lines of code used by the manual merge versions, while in blue are the results from the combinations using our tool. This measurement was conducted using the program *Cloc* [26], which enabled us to perform an accurate and fair count for all cases. In the case of manual merging, in addition to the lines of code from all files within the respective NF's folder, all files used from the Vigor framework were also counted. To obtain precise results, blank lines, comments, and code from other files (such as Makefile) were excluded.

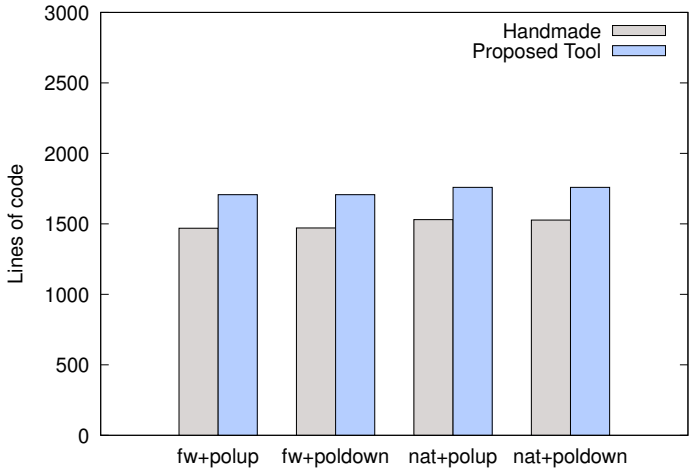

Figure 4.1: Comparison between the number of lines of code of merged NFs with and without the use of the tool

Analyzing the graph it is notable that the combinations generated by the tool exhibit a slightly higher number of lines of code compared to the manual merge versions. This is due to the fact that the tool was not built with the aim of optimizing the code, but rather to ensure the correctness of the functionalities of each NF. As a result, the resulting BDD may contain redundant nodes, which will consequently generate redundant code.

4.2 Benchmark methodology

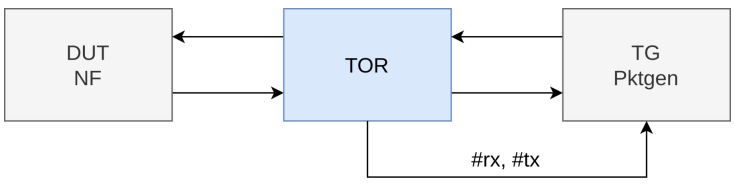

Figure 4.2: Testbed topology used to measure performance, using a traffic generator (TG) and a device under test running an NF (DUT).

Testbed. The used testbed is represented in Fig. 4.2 as per FRC 2544 [27], and following the same methodology used by Pereira *et al.* in Maestro [25]. Both the machines used for traffic generation (TG) and the machine used for testing the NF (device under test, or DUT) are equipped with dual socket Intel Xeon Gold 6226R @ 2.90GHz, 96 GB of DRAM, and Intel E810 100 Gbps NICs. Turbo Boost, Hyper-Threading, and power saving features were disabled, as recommended by DPDK . Both devices connect through a top-of-rack (TOR) switch from which we collect packet counters at the end of each experiment. As mentioned in the beginning of this chapter, the two key concepts related to NF's performance are throughput and latency, and measuring them requires the use of different techniques. Both techniques use DPDK Pkt-Gen [28] to generate and replay traffic, and measure packet loss and latency.

Throughput. To measure throughput, the TG replays a given traffic sample (a PCAP file) in a loop at a given rate via the outbound cable for 5s per experiment. The DUT receives this traffic, processes it, and sends it back via the return cable. We further use the TOR to infer loss at the DUT, and—through comparison with the TG report—to also detect when packets were lost within the TG as well. We use DPDK-Pktgen on the TG to find the maximum rate with less than 0.1% loss. We perform 10 measurements per experiment for statistical relevance and show error bars with min/max values.

Latency. The NFs are subjected to timestamped probe packets, which are later received by the TG and used to calculate the latency.

4.3 Performance Benchmarking

After the development of the tool, several performance studies were conducted, measuring latency and throughput according to different packet types. In the throughput study, we compared the values obtained from manual combinations and those generated by the tool, under varying packet sizes. In the latency study, the values obtained from both manual combinations and the tool were also compared. In this section, the traffic samples used represent a Zipfian distribution of the traffic, rather than a uniform distribution. While the latter suited the scenario under study, it does not correspond to a distribution often encountered in the real world. In a uniform distribution of flows, there is an equal probability of two packets arriving with the same flow IDs. However, in a Zipfian distribution, a large percentage of arriving packets share the same destination address. The Zipfian traffic was generated using the Zipfian parameters found by Pedrosa et al. [29], which were found by analyzing real-world traffic samples from a University network [30]. Nonetheless, we also present the performance results with a uniform distribution in appendix A.

In Figure 4.3, we present the throughput results measured with Zipfian traffic samples. In gray we see the measurements obtained from manual combinations, while in blue, the measurements obtained from combinations generated by our tool. In the left side graph, throughput is compared in millions

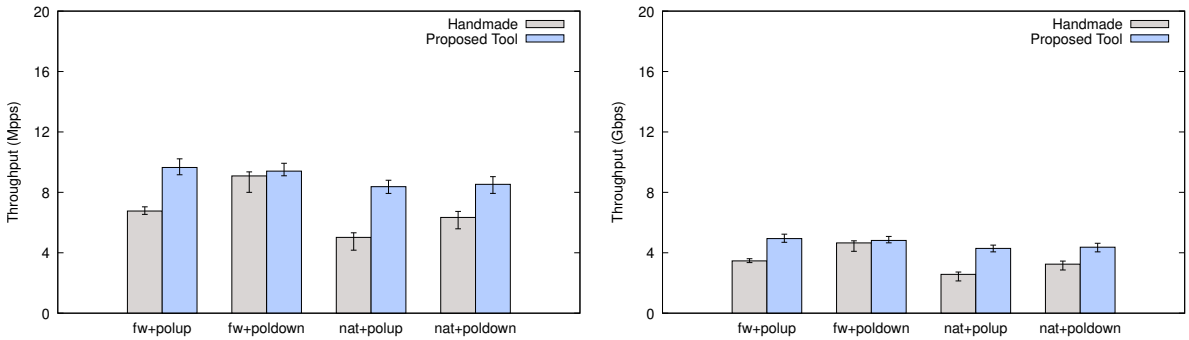

Figure 4.3: Throughput comparison using Zipfian distribution of traffic

of packets processed per second (Mpps), whereas in the right graph, it is interpreted as gigabits per second (Gbps). Comparing the obtained results, it is notable that the combinations generated by the tool exhibit better performance compared to manual ones This is because the code generated by the tool is condensed into a single file with almost no modularity. When code is modularized (as the one from manual combinations), it often involves breaking tasks into separate functions or modules. While this improves code readability and maintainability, it can introduce a small performance overhead due to the extra computational cost of function calls.

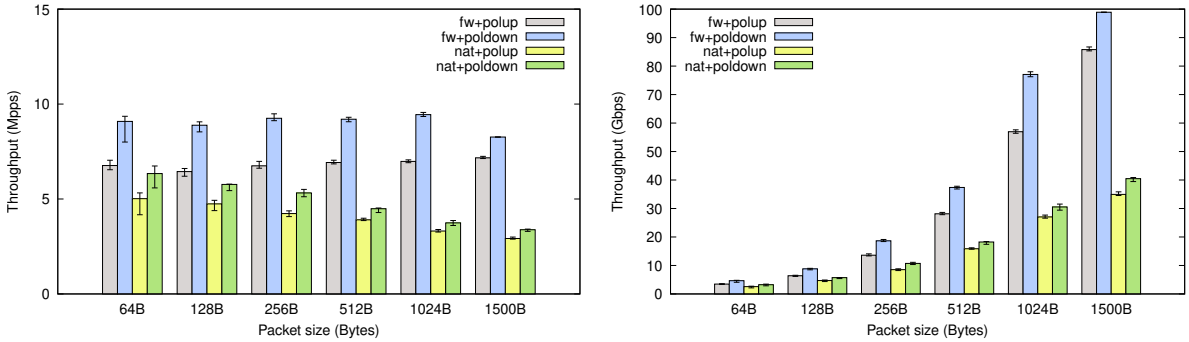

Figure 4.4: Throughput variation on handmade combinations for different packet sizes using Zipfian distribution of traffic

In Figures 4.4 and 4.5 we present the throughput measurements varying the packet size in each traffic sample, for manual combinations and for combinations generated by our tool, respectively.

In the left side graphs, a decrease in throughput is noticeable with an increase in packet size. The throughput, measured in million packets per second (Mpps), tends to increase with larger packet sizes up to a point. However, as the packet sizes continue to increase, they eventually surpass the system's capabilities of packet processing and the CPU becomes a bottleneck. At this point, the throughput will start to decrease because the system can no longer efficiently process the larger packets.

In the case of throughput measured in Gigabits per second (Gbps), it typically increases with larger packet sizes up to a certain point, after which it stabilizes. When increasing the packet size, more data is

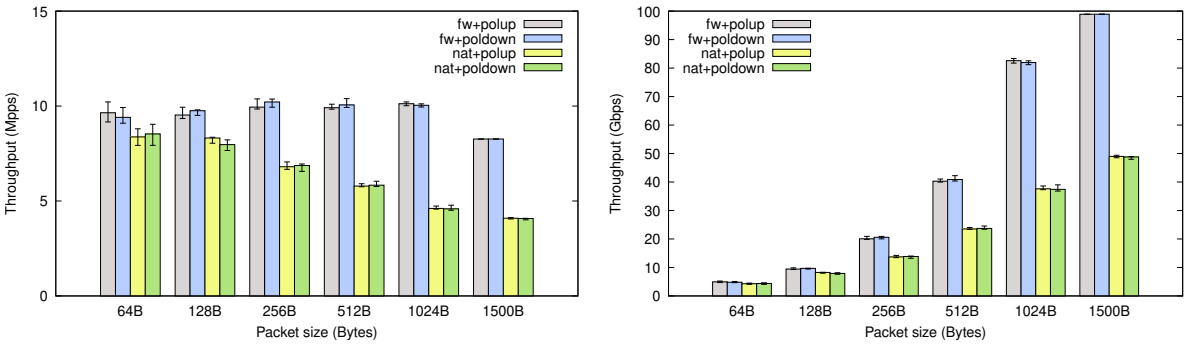

Figure 4.5: Throughput variation on tool combinations for different packet sizes using Zipfian distribution of traffic

being sent in each transmission, and consequently using more of the available bandwidth, until the NIC becomes a bottleneck. In the right side graphs, we can notice the continuous increase of the throughput, as none of the tests can reach the 100 Gbps bandwidth available in the NIC used in this benchmark.

Analyzing the results one can conclude that the combinations using NAT require more computational power due to its IP translation process. As we can observe, the decrease of the throughput in Mpps starts right on packet sizes of 128 bytes, while in the firewall combinations, the CPU starts being a bottleneck in packet sizes of 1500 bytes. This is also noticeable in the throughput measured in Gigabits per second (Gbps), as the firewall combinations doubles the performance for packets with sizes equal to or greater than 512 bytes.

It's worth noting as well that the throughput results obtained for combinations made by our tool are slightly better comparing to the ones from the manual merge.

The latency associated with each combination was also measured and it is presented in the Fig. 4.6. Each graph represents the cumulative distribution functions for each one of the combinations comparing the manual (colored in grey) and the ones created with our tool (in blue). Observing the median of each graph (the median is the point where each CDF curve crosses the 0.5 mark) it is noticeable that our tool's combinations have a slightly smaller median than the ones combined by hand. This indicates that the latency values from our tool implementations tend to be smaller than the ones from the manual combinations. Another analysis that reinforces this conclusion is the heavy tails recorded after 25μ . In the first three graphs, we conclude that the probability of having latency numbers greater than 25μ is slightly higher in the manual combinations, excluding the last graph, where the latency numbers of the manual combination beyond this threshold outperform the ones from those created with our tool.

4.4 Summary

The aim of this section was to evaluate our tool in order to understand how much human effort is saved by using it to combine NFs, and what the performance difference would be when compared to other

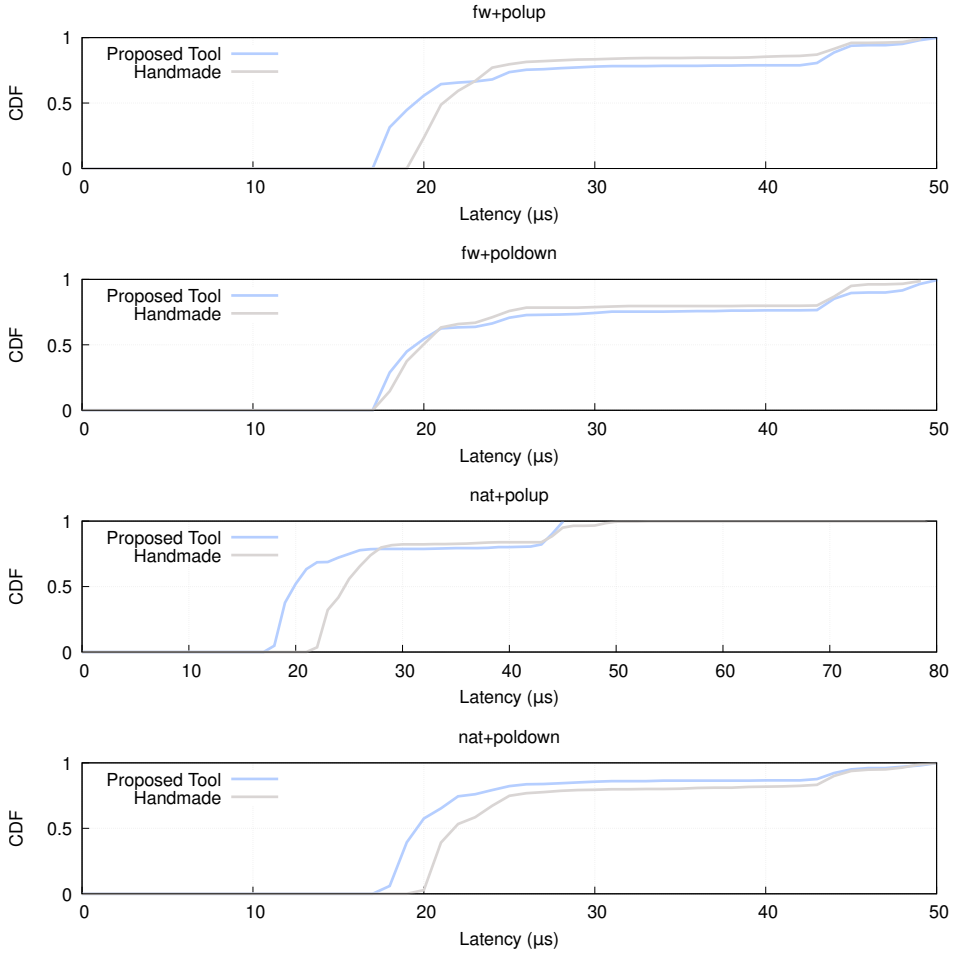

Figure 4.6: Comparison of latency cumulative distribution functions using Zipfian distribution

alternatives when the combination is done manually.

To conduct the tests, four combinations were made between the Vigor's firewall and policer, and the conflict resolution policies used were described.

The first study described considered microbenchmarks such as the number of lines of code and the time spent on performing the combination. It was possible to conclude that our tool promises to save a considerable amount of time in NF combination, especially in cases where the developer is not very familiar with the Vigor framework.

Finally, the metrics used to measure the performance of the combinations and the environment in which they were conducted were described. With the presentation of these results, it was possible to conclude that the implementations made by our tool achieved better results in terms of throughput, with no significant differences in latency compared to NFs combined manually.

Listing 4.1: Pseudocode of a NAT written in Vigor [1]

```
struct Map* map;
struct Dchain* dchain;

void process_packet(int device, pkt_t p) {
    if (device == WAN) {
        if (!dchain_is_index_allocated(dchain, p.tcpudp_dst)) {
            drop(p);
            return;
        }
        struct Flow internal_flow = map_get(map, p.tcpudp_dst);

        if (internal_flow.dst_ip != p.ipv4_src ||
            internal_flow.dst_port != p.tcpudp_src) {
            drop(p);
            return;
        }

        // translate
        p.ipv4_dst = internal_flow.src_ip;
        p.tcpudp_dst = internal_flow.src_port;

        forward(p, internal_flow.internal_device);
    } else {
        // store flow
        struct Flow flow = {
            .src_port = p.tcpudp_src,
            .dst_port = p.tcpudp_dst,
            .src_ip = p.ipv4_src,
            .dst_ip = p.ipv4_dst,
            .internal_device = device
        };

        // allocate a new port
        int allocated_port = dchain_allocate(dchain);
        map_put(map, allocated_port, flow);

        // translate
        p.ipv4_src = EXTERNAL_IP;
        p.tcpudp_src = allocated_port;

        forward(p, WAN);
    }
}
```

5

Conclusion

Contents

5.1 Conclusion

This dissertation contributes to the state-of-the-art network function combination by introducing a tool that automatically combines multiple implementations of NFs while ensuring the correctness of the combined functionality.

Using the Vigor framework and its verification byproducts, our tool can analyze and merge multiple NFs producing a new implementation that works as if each one of the NFs had access to a copy of the input traffic, which output is therefore the result of the combination of both outputs.

This is made possible by resorting to three modules - the analysis engine, merge engine, and synthesis engine. In the NF analysis engine, we resort to Vigor's KLEE symbolic engine to obtain a complete and sound representation of the NF's behavior. With this representation, we make use of Maestro to generate a binary decision diagram that represents all abstract functionality of the NF. After generating

the representation of both constituent NFs, the merge engine (our tool) proceeds to find compatible code paths to merge all changes made by each one of the NFs, and then uses a constraint-based insertion algorithm to build the final BDD.

In more complex cases, when different NFs may change the same fields of the packet or even perform different decisions, our tool resorts to user-defined conflict resolution policies to decide which changes to prioritize, and what to do when both paths have different decisions about the packet's fate. In the end, we use Maestro's code synthesizer to generate the final implementation of the combined NF in C code.

We measured the performance achieved by four combinations made with three different NFs, using our tool and comparing it to the respective handmade implementations. The results show small throughput increases in all NFs and also that latency was mostly unaffected. Our microbenchmarks show that using our tool to merge NFs can save a lot of time for developers, specially those who are not used to Vigor's framework.

5.2 System limitations and future work

5.2.1 Formal verification of the generated implementations

Although our tool uses formally verified sequential implementations of NFs built in Vigor, its generated combined implementations are not formally verified. Verification is not simple, and is outside the scope of this thesis. As such, it is left as future work.

5.2.2 Resource sharing

When combining multiple stateful NFs, one of the operations that has most impact in the performance is the access to the data structures. When merging compatible paths of the code, there could be multiple accesses to identical data structures in the same index. Our work serves as starting point to optimize the combination process by merging compatible data structures. Despite not being able to implement this optimization - due to time constraints - we started analyzing multiple cases and concluded that two data structures of the same type (e.g. vector or map) can be merge in the following condition - assuming two data structures of the same type $Vector_a$ and $Vector_b$, one can combine both into just one structure if and if only for every path in the BDD, all accesses to those structures are done in the same index.

5.2.3 IP options

Our tool was built not considering the IP options field of the IPv4 packet. In future work, one can start tracing the IP options field by its symbolic size, or even create support for the multiple options available in this field.

Bibliography

[1] A. Zaostrovnykh, S. Pirelli, R. Iyer, M. Rizzo, L. Pedrosa, K. Argyraki, and G. Candea, "Verifying software network functions with no verification expertise," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, 2019, pp. 275–290.

[2] Y. Feng, R. Martins, O. Bastani, and I. Dillig, "Program synthesis using conflict-driven learning," in *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 420–435. [Online]. Available: https://doi.org/10.1145/3192366.3192382

[3] H. Chen, A. Wang, and B. T. Loo, "Towards example-guided network synthesis," in *Proceedings of the 2nd Asia-Pacific Workshop on Networking*, ser. APNet '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 65–71. [Online]. Available: https://doi.org/10.1145/3232565.3234462

[4] A. Bremler-Barr, Y. Harchol, and D. Hay, "Openbox: A software-defined framework for developing, deploying, and managing network functions," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 511–524. [Online]. Available: https://doi.org/10.1145/2934872.2934875

[5] L. Peterson, C. Cascone, B. O'Connor, T. Vachuska, and B. Davie, "Software-defined networks: A systems approach," *Systems Approach LLC*, 2020.

[6] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the v out of NFV," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 203–216. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

[7] B. Jacobs and F. Piessens, "The verifast program verifier," Technical Report CW-520, Department of Computer Science, Katholieke . . . , Tech. Rep., 2008.

[8] C. Cadar, D. Dunbar, D. R. Engler *et al.*, "Klee: unassisted and automatic generation of high-coverage tests for complex systems programs." in *OSDI*, vol. 8, 2008, pp. 209–224.

[9] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu, “Symnet: Scalable symbolic execution for modern networks,” in *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016, pp. 314–327.

[10] A. Zaostrovnykh, S. Pirelli, L. Pedrosa, K. Argyraki, and G. Candea, “A formally verified nat,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’17. New York, NY, USA: Association for Computing Machinery, 2017, p. 141–154. [Online]. Available: https://doi.org/10.1145/3098822.3098833

[11] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, and D. Walker, “P4: Programming protocol-independent packet processors,” *SIGCOMM Comput. Commun. Rev.*, jul 2014.

[12] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, “Debugging p4 programs with vera,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 518–532. [Online]. Available: https://doi.org/10.1145/3230543.3230548

[13] J. Liu, W. Hallahan, C. Schlesinger, M. Sharif, J. Lee, R. Soulé, H. Wang, C. Caşcaval, N. McKeown, and N. Foster, “P4v: Practical verification for programmable data planes,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, ser. SIGCOMM ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 490–503. [Online]. Available: https://doi.org/10.1145/3230543.3230582

[14] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey, “Veriflow: Verifying network-wide invariants in real time,” in *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013, pp. 15–27.

[15] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein, “A general approach to network configuration analysis,” in *12th {USENIX} symposium on networked systems design and implementation ({NSDI} 15)*, 2015, pp. 469–483.

[16] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese, “Efficient network reachability analysis using a succinct control plane representation,” in *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, 2016, pp. 217–232.

[17] R. Beckett, A. Gupta, R. Mahajan, and D. Walker, “A general approach to network configuration verification,” in *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017, pp. 155–168.

[18] S. Gulwani, A. Polozov, and R. Singh, *Program Synthesis*. NOW, August 2017, vol. 4. [Online]. Available: https://www.microsoft.com/en-us/research/publication/program-synthesis/

[19] C. L. Goues, M. Pradel, and A. Roychoudhury, “Automated program repair,” *Communications of the ACM*, vol. 62, no. 12, pp. 56–65, 2019.

[20] X. Gao, T. Kim, M. D. Wong, D. Raghunathan, A. K. Varma, P. G. Kannan, A. Sivaraman, S. Narayana, and A. Gupta, “Switch code generation using program synthesis,” 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*, ser. SIGCOMM ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 44–61. [Online]. Available: https://doi.org/10.1145/3387514.3405852

[21] K. Zhang, D. Zhuo, and A. Krishnamurthy, “Gallium: Automated software middlebox offloading to programmable switches,” 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*, ser. SIGCOMM ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 283–295. [Online]. Available: https://doi.org/10.1145/3387514.3405869

[22] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker, “Frenetic: A network programming language,” *ACM Sigplan Notices*, vol. 46, no. 9, pp. 279–291, 2011.

[23] J. Reich, C. Monsanto, N. Foster, J. Rexford, and D. Walker, “Modular sdn programming with pyretic,” *Technical Reprot of USENIX*, p. 30, 2013.

[24] R. Morris, E. Kohler, J. Jannotti, and M. F. Kaashoek, “The click modular router,” in *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’99, 1999.

[25] F. M. C. Pereira, “Nf parallel synthesis.” [Online]. Available: https://fenix.tecnico.ulisboa.pt/cursos/meic-a/dissertacao/1128253548922590

[26] A. Danial, “cloc: Count lines of code,” https://github.com/AlDanial/cloc, 2023, accessed: October 2023.

[27] S. Bradner and J. McQuaid, “Rfc2544: Benchmarking methodology for network interconnect devices,” 1999.

[28] D. Turull, P. Sjödin, and R. Olsson, “Pktgen: Measuring performance on high speed networks,” *Computer Communications*, vol. 82, pp. 39–48, 2016. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0140366416300615

[29] L. Pedrosa, R. Iyer, A. Zaostrovnykh, J. Fietz, and K. Argyraki, “Automated synthesis of adversarial workloads for network functions,” in *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, 2018, pp. 372–385.

[30] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, 2010, pp. 267–280.

Appendix A

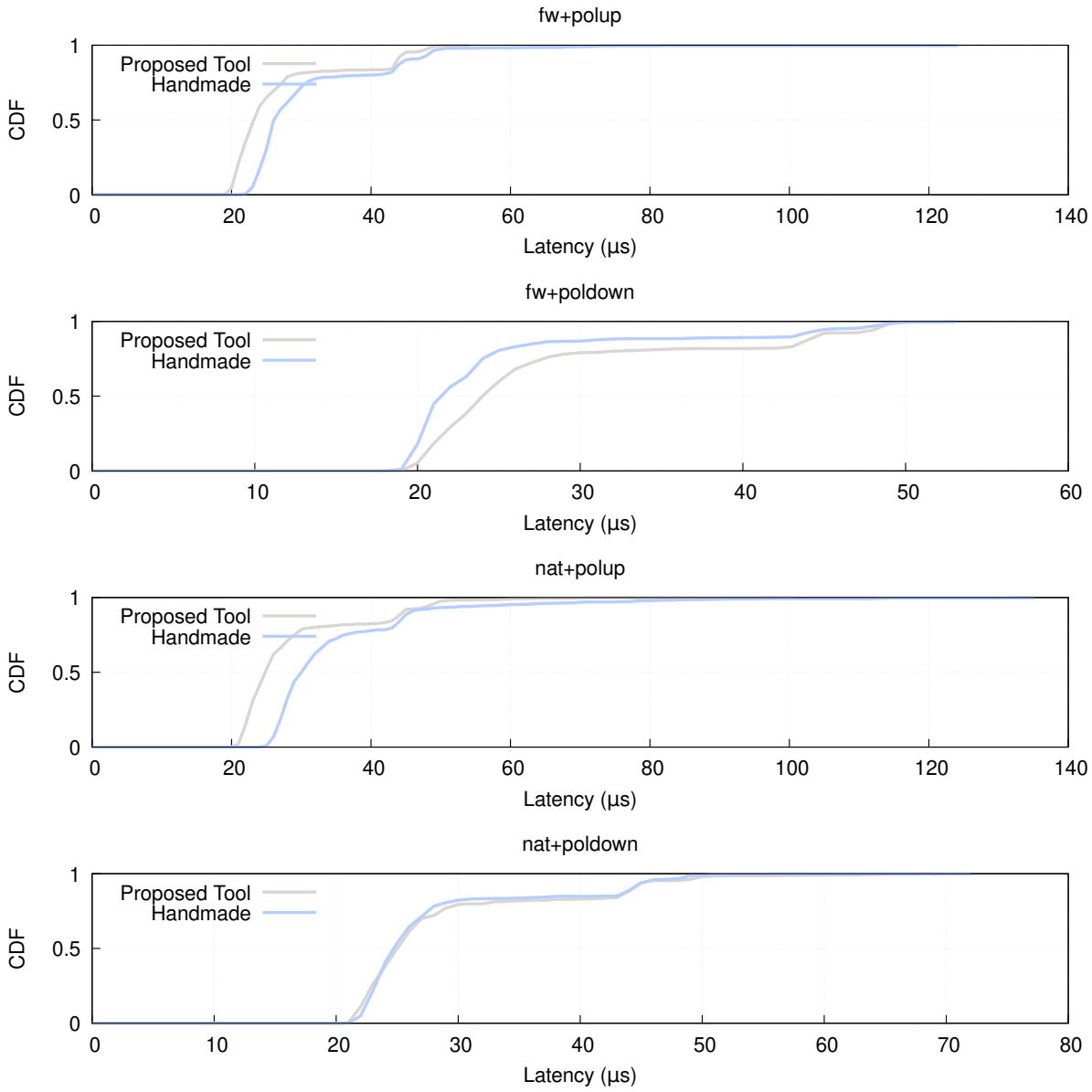

Figure A.1: Latency comparison using uniform traffic

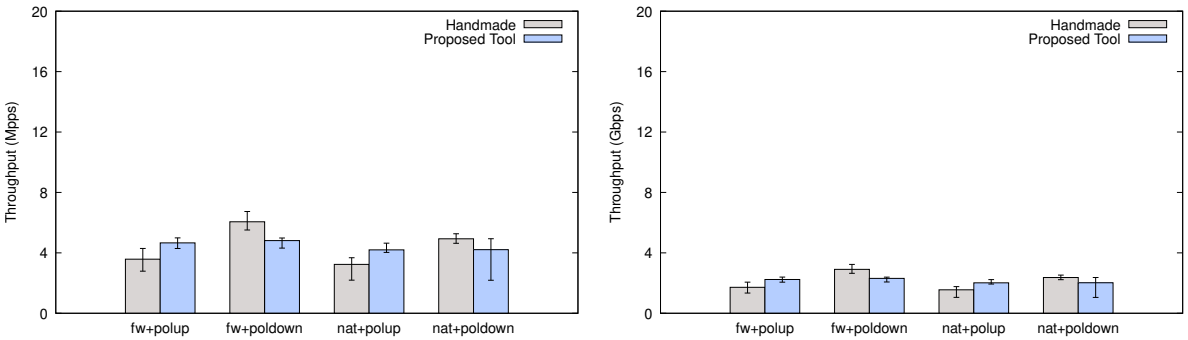

Figure A.2: Throughput comparison using uniform traffic

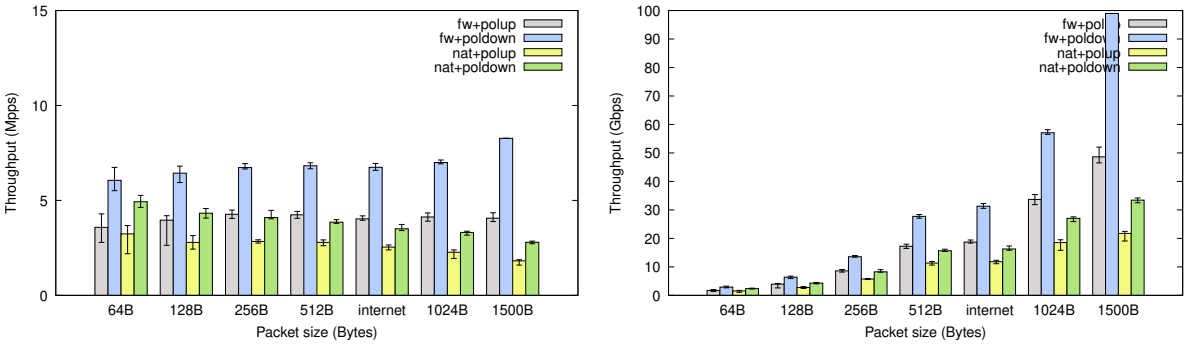

Figure A.3: Packet size on handmade combinations using uniform distribution

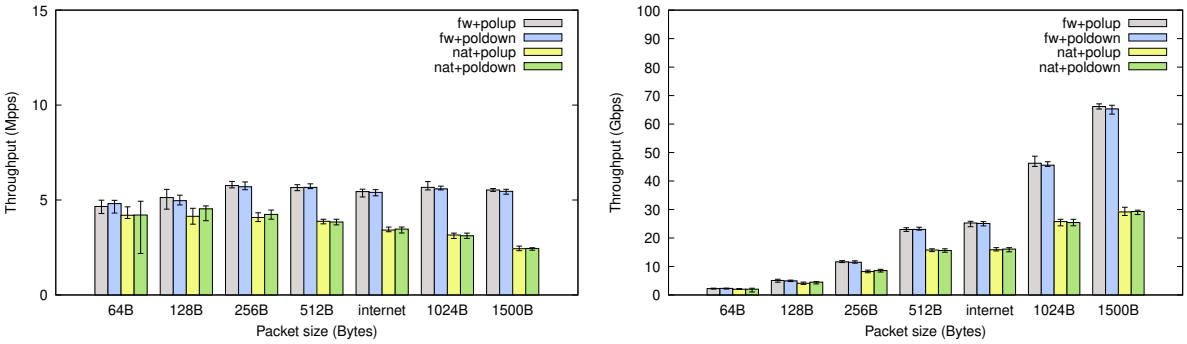

Figure A.4: Packet size on tool combinations using uniform distribution

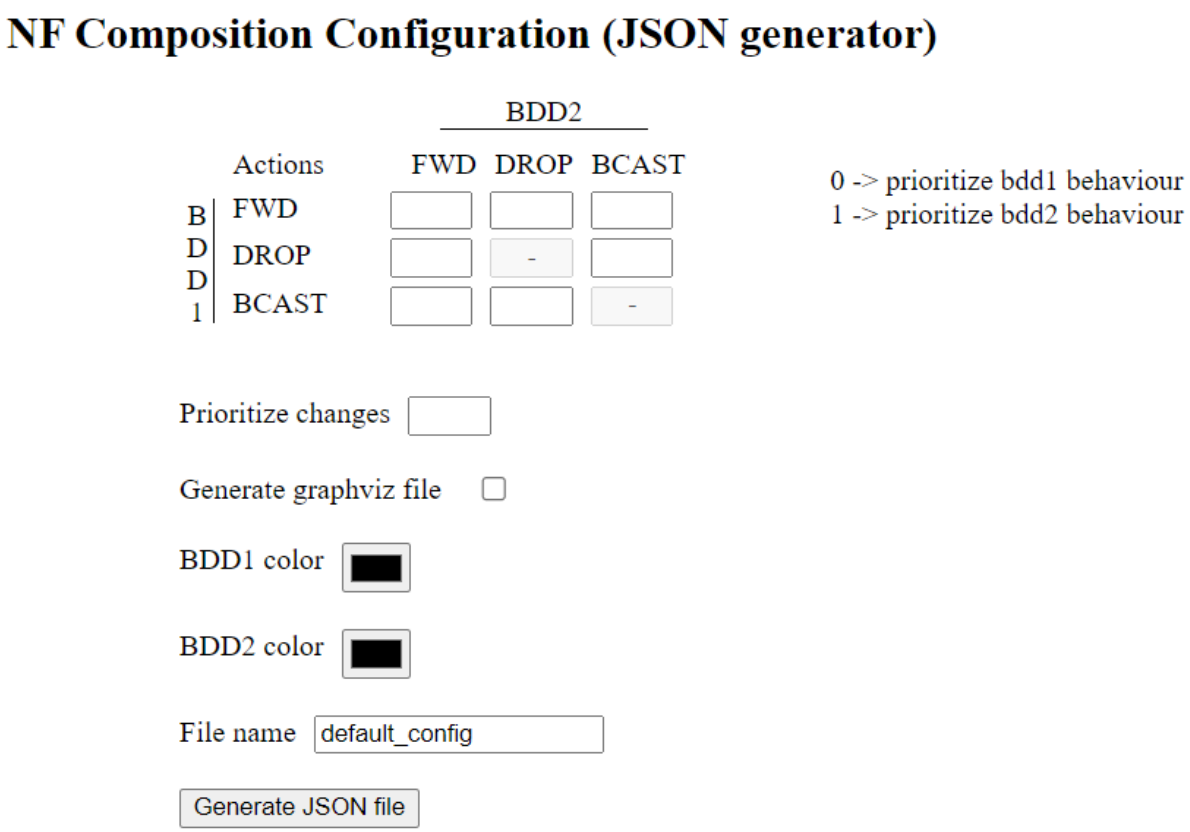

Figure A.5: UI to generate JSON files containing the conflict resolution policies