NF Composition Synthesis

Pedro Miguel Drogas Moreira
Instituto Superior Técnico
Lisbon, Portugal

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 [26] 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

1 INTRODUCTION

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 NF service chains: typically network traffic traverses a sequence of NFs. For example, a packet may go through a firewall, then through an 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.

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.

The rest of the paper is structured as follows: after presenting an overview of the current state of the art of the Network Function Composition field (§2), we explain how our tool automatically generates combined implementations of NFs (§3), evaluate the process of the combined implementations' acquisition and benchmark them (§4), and finally share the conclusions taken from our work (§5).

2 RELATED WORK

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. Whereas traditional networks use dedicated hardware devices (i.e., routers and switches) to control network traffic, SDNs (Software-defined networks) [21] control the network via software that runs in an external (to the devices) entity.

Many large carriers [18] initiated an effort, called Network Function Virtualization (NFV), to replace hardware middleboxes with software implementations running in VMs. This approach enabled middlebox functionality (called Network Functions or NFs) to be

run on commodity servers and was supposed to bring several advantages such as simpler deployment and management, fast development, and reduced cost of deploying several NFs on a single machine. The move from hardware middleboxes to software network functions has proven more challenging than expected. Developing new NFs remains a tedious process, requiring that developers repeatedly rediscover and reapply the same set of optimizations, while current techniques for providing isolation between NFs (using VMs or containers) incur high-performance overheads.

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. Verification can be based on two main approaches. First, theorem proving [7, 9, 14–16] - 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 [4, 23, 24, 26, 27]. It is used in verification tools, 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.

Program Synthesis[8, 12, 13] 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).

Chipmunk[11] 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. Sketches encode human insight into the shape of synthesized programs. SKETCH then fills in all holes with integers so that the completed sketch meets the specification, assuming it is possible to meet the specification or says that it is impossible to do so.

Facon [5] 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.

Frenetic [10], 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, came to address those challenges and 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 that handles all of the details of installing and uninstalling rules on switches.

Pyretic [22] is a new 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.

Openbox[3] 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. One of the best features 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. With OpenBox it is possible to merge the two graphs into one, such that the logic of the former is first executed, followed by the execution of the latter. Additionally, it is possible to reduce the total delay incurred on packets by reducing the number of blocks each packet traverses.

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 [17] 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 [26]. 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.

3 ARCHITECTURE

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.

One of the advantages of using Vigor's [26] networks functions is that they are formally verified. Their correctness is ensured by generating a complete and sound representation of the NF's functionality using symbolic execution: BDD. These representations are crucial to overcome the program composition problem, as they represent all possible outcomes to every type of incoming traffic, to

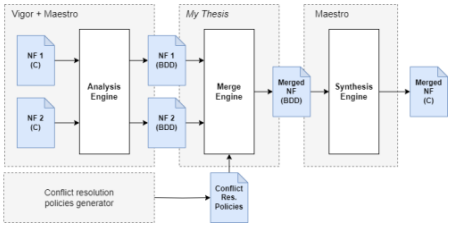

Figure 1: Architecture of our solution

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). 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 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. Our tool pairs all compatible outcomes from both NFs and uses merge mechanisms defined by the user to produce a new and correct output.

3.1 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. 1.

The Analysis Engine is based on Maestro [20], which extends the Vigor [26] symbolic analysis, which in turn builds on KLEE [4]. 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 [20]) is used to output the final implementation.

3.2 Analysis engine

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 DPDK, putting the persistent state in data structures from 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.

3.2.1 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 one can use several data structures provided by libVig. Since the correct behavior of a stateful NF depends directly on its structures, their initialization is encapsulated in another important method: *nf_init*.

The *nf_init* 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 *maps* (used to store integers associated with a key of arbitrary type), *vectors* (used to store arbitrary type data associated with an integer index), and *dchains* (double linked-list of integers used to allocate indexes, verify if a specific index is already allocated, or rejuvenate a previously allocated index).

All packet process logic is encapsulated in *nf_process* method. 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, IPv4, TCP and UDP headers. Different NFs may change the same fields of the packet, or access some fields that the other does not: 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: 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.

3.2.2 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. Each node in the tree operates in its own symbolic context, without changing the actual values of other nodes.

The main types of nodes are call, branch and return nodes.

Call nodes can 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. 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. 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.

Besides packet management functions, they represent data structure related methods. 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. 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 return nodes represent the action returned by the *nf_process* and *nf_init* methods. Regarding the return of the *nf_process* method, it stores 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. When combining different initialization methods, is necessary to ensure that each node ID is unique in the context of both NFs. Furthermore, 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.

3.3 Merge engine: Automated merge process

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 this section we present our proposal to fully automate the merge of two NF representations, in the absence of conflicts.

3.3.1 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. 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 $N+1$. The process of updating the node IDs in a BDD is based on a 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.

3.3.2 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 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.

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 source device, and the time of arrival.

3.3.3 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.

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.

The next group of nodes being added to the new BDD 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.

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. 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.

The last node from each path represents the return of the *nf_process* method. In the absence of conflicts, our tool chooses arbitrarily one of the nodes to be inserted in the final BDD.

3.3.4 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.

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 a conditional branch. Here, our tool resorts to the Z3 solver to choose which path the node should take: two conjunction expressions are made between the node constraints and the branch condition. The algorithm will continue its execution on one of the branches, or in cases where the new node can coexist in both scenarios, it will continue its execution in each branch.

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, we will obtain a new BDD representing the combination of functionalities.

3.3.5 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. 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.

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: this can lead to paths that, for example, when the initialization of a map fails, it proceeds to the initialization of the remaining data structures.

An NF can only function if all 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 return of the function.

3.4 User-defined conflict resolution policies

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 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.

	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 1: Conflict resolution matrix

- *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 1) 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.

Our tool receives as input this configuration in JSON format. The JSON object contains the conflict matrix, the boolean 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.

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.

4 EVALUATION

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?

4.1 Evaluated Network Functions

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. 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.

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.

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

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

address and TCP destination port of the stored flow, translating and redirecting it to the LAN if this check holds.

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.

4.2 Microbenchmarks

The comparison between the time spent on the manual merge versus the one spent by our tool is presented in the Table 2. 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.

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. In addition to the time spent on code combination, the time taken to familiarize with Vigor's framework was also included. The second version of this combination, *fw+polup*, took considerably less time since the code for both NFs

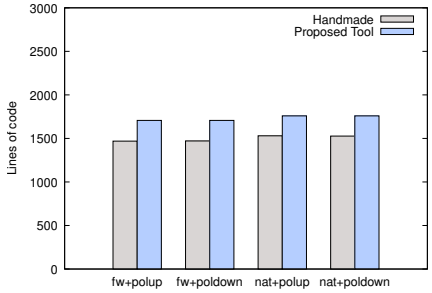

Figure 2: Comparison between the number of lines of code of merged NFs with and without the use of the tool

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. 2. 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* [6], 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. 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.

4.3 Benchmarking methodology

The used testbed as per FRC 2544 [2] follows the same methodology used by Pereira *et al.* in Maestro [20]. 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 [25] to generate and replay traffic, and measure packet loss and latency.

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

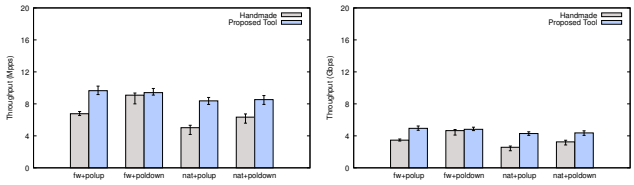

Figure 3: Throughput comparison using Zipfian distribution of traffic

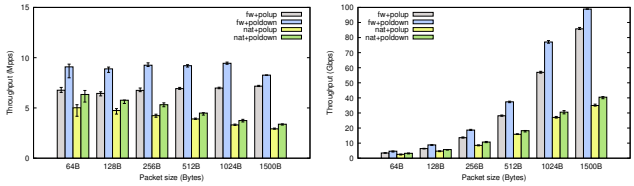

Figure 4: Throughput variation on handmade combinations for different packet sizes using Zipfian distribution of traffic

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

4.4 Performance benchmarking

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. [19], which were found by analyzing real-world traffic samples from a University network [1].

In Figure 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. 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.While this improves code readability and maintainability, it can introduce a small performance overhead due to the extra computational cost of function calls.

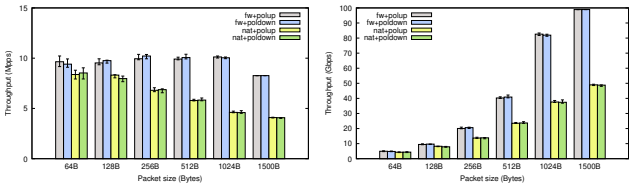

Figure 5: Throughput variation on tool combinations for different packet sizes using Zipfian distribution of traffic

In Figures 4 and 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 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. 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

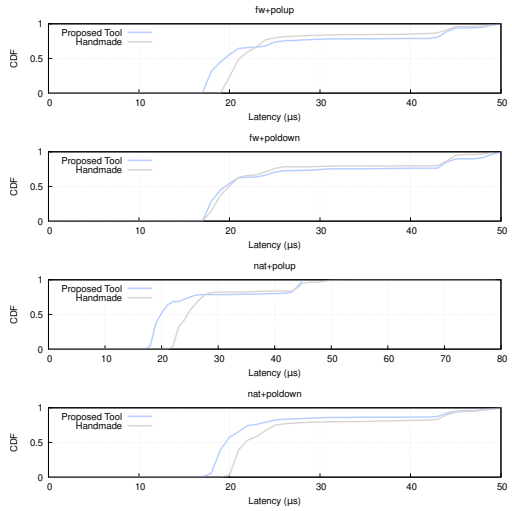

Figure 6: Comparison of latency cumulative distribution functions using Zipfian distribution

of the manual combination beyond this threshold outperform the ones from those created with our tool.

5 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.

5.1 System Limitations and Future Work

5.1.1 Formal verification of the generated implementations. Although our tool uses formally verified implementations of NFs built in Vigor, its generated 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.

5.1.2 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.

5.1.3 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.

6 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).

REFERENCES

[1] Theophilus Benson, Aditya Akella, and David A Maltz. 2010. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. 267–280.

[2] Scott Bradner and Jim McQuaid. 1999. RFC2544: Benchmarking Methodology for Network Interconnect Devices.

[3] Anat Bremler-Barr, Yotam Harchol, and David Hay. 2016. OpenBox: A Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proceedings of the 2016 ACM SIGCOMM Conference* (Florianopolis, Brazil) *(SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 511–524. https://doi.org/10.1145/2934872.2934875

[4] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *OSDI*, Vol. 8. 209–224.

[5] Haoxian Chen, Anduo Wang, and Boon Thau Loo. 2018. Towards Example-Guided Network Synthesis. In *Proceedings of the 2nd Asia-Pacific Workshop on Networking* (Beijing, China) *(APNet '18)*. Association for Computing Machinery, New York, NY, USA, 65–71. https://doi.org/10.1145/3232565.3234462

[6] Al Danial. 2023. cloc: Count Lines of Code. https://github.com/AlDanial/cloc. Accessed: October 2023.

[7] Seyed K Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd Millstein, Vyas Sekar, and George Varghese. 2016. Efficient network reachability analysis using a succinct control plane representation. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*. 217–232.

[8] Yu Feng, Ruben Martins, Osbert Bastani, and Isil Dillig. 2018. Program Synthesis Using Conflict-Driven Learning. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. Association for Computing Machinery, New York, NY, USA, 420–435. https://doi.org/10.1145/3192366.3192382

[9] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A general approach to network configuration analysis. In *12th {USENIX} symposium on networked systems design and implementation ({NSDI} 15)*. 469–483.

[10] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. 2011. Frenetic: A network programming language. *ACM Sigplan Notices* 46, 9 (2011), 279–291.

[11] Xiangyu Gao, Taegyun Kim, Michael D. Wong, Divya Raghunathan, Aatish Kishan Varma, Pravein Govindan Kannan, Anirudh Sivaraman, Srinivas Narayana, and Aarti Gupta. 2020. 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* (Virtual Event, USA) *(SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 44–61. https://doi.org/10.1145/3387514.3405852

[12] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019. Automated program repair. *Commun. ACM* 62, 12 (2019), 56–65.

[13] Sumit Gulwani, Alex Polozov, and Rishabh Singh. 2017. *Program Synthesis*. Vol. 4. NOW. 1–119 pages. https://www.microsoft.com/en-us/research/publication/program-synthesis/

[14] Bart Jacobs and Frank Piessens. 2008. *The VeriFast program verifier*. Technical Report. Technical Report CW-520, Department of Computer Science, Katholieke ....

[15] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. 2013. Veriflow: Verifying network-wide invariants in real time. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*. 15–27.

[16] Jed Liu, William Hallahan, Cole Schlesinger, Milad Sharif, Jeongkeun Lee, Robert Soulé, Han Wang, Călin Caşcaval, Nick McKeown, and Nate Foster. 2018. P4v: Practical Verification for Programmable Data Planes. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 490–503. https://doi.org/10.1145/3230543.3230582

[17] Robert Morris, Eddie Kohler, John Jannotti, and M. Frans Kaashoek. 1999. The Click Modular Router. In *Proceedings of the Seventeenth ACM Symposium on Operating Systems Principles (SOSP '99)*.

[18] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. 2016. NetBricks: Taking the V out of NFV. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 203–216. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/panda

[19] Lucas Pedrosa, Rishikanth Iyer, Alexey Zaostrovnykh, Jonas Fietz, and Katerina Argyraki. 2018. Automated synthesis of adversarial workloads for network functions. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. 372–385.

[20] Francisco Manuel Chamiça Pereira. [n.d.]. NF Parallel Synthesis. https://fenix.tecnico.ulisboa.pt/cursos/meic-a/dissertacao/1128253548922590

[21] L Peterson, C Cascone, B O'Connor, T Vachuska, and B Davie. 2020. Software-Defined Networks: A Systems Approach. *Systems Approach LLC* (2020).

[22] Joshua Reich, Christopher Monsanto, Nate Foster, Jennifer Rexford, and David Walker. 2013. Modular sdn programming with pyretic. *Technical Reprot of USENIX* (2013), 30.

[23] Radu Stoenescu, Dragos Dumitrescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2018. Debugging P4 Programs with Vera. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (Budapest, Hungary) *(SIGCOMM '18)*. Association for Computing Machinery, New York, NY, USA, 518–532. https://doi.org/10.1145/3230543.3230548

[24] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2016. Symnet: Scalable symbolic execution for modern networks. In *Proceedings of the 2016 ACM SIGCOMM Conference*. 314–327.

[25] Daniel Turull, Peter Sjödin, and Robert Olsson. 2016. Pktgen: Measuring performance on high speed networks. *Computer Communications* 82 (2016), 39–48. https://doi.org/10.1016/j.comcom.2016.03.003

[26] Arseniy Zaostrovnykh, Solal Pirelli, Rishabh Iyer, Matteo Rizzo, Luis Pedrosa, Katerina Argyraki, and George Candea. 2019. Verifying software network functions with no verification expertise. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*. 275–290.

[27] Arseniy Zaostrovnykh, Solal Pirelli, Luis Pedrosa, Katerina Argyraki, and George Candea. 2017. A Formally Verified NAT. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication* (Los Angeles, CA, USA) *(SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 141–154. https://doi.org/10.1145/3098822.3098833