University POLITEHNICA of Bucharest

Faculty of Automatic Control and Computers,
Computer Science and Engineering Department

BACHELOR THESIS

Formal Analysis of iptables Configurations

Scientific Adviser:
Conf.dr.ing. Costin Raiciu

Author:
Călin-Cristian Cruceru

Bucharest, 2017

I am extremely grateful to my adviser, Costin Raiciu, for giving me the opportunity to pursue my bachelor thesis as part of the NETSYS group, and for investing his time to regularly follow up with the progress. His vision on the importance of verification in modern networks has been truly inspiring.

I thank Radu Stoenescu, my direct supervisor, for tirelessly helping me to get started with this novel, fascinating field, and for his continous feedback based on his extensive work on network verification.

Abstract

As modern networks become increasingly complex and more and more operators move towards Network Function Virtualization as a way of reducing costs for purchasing and upgrading middleboxes, verifying certain properties of networks built around iptables-enabled devices becomes one of the top priorities in network verification today. Thus, we create a model of an iptables-enhanced router which can be fed to SymNet, a fast symbolic execution tool for stateful networks, and show how it can be used in practice.

We thoroughly evaluate the model on two dimensions: correctness and performance. The former refers to the equivalence between the semantics of our model and the real implementation of iptables, while the latter quantifies the quality of the model in terms of the running time of SymNet when run against it.

Contents

List of Figures

List of Tables

Notations and Abbreviations

ACL – Access Control List
API – Application Programming Interface
AST – Abstract Syntax Tree
CTL – Computation tree logic
DMZ – Demilitarized Zone
DSL – Domain Specific Language
FIB – Forwarding Information Base
IVD – Iptables Virtual Devices
MAC – Media Access Control
NAT – Network Address Translation
NFV – Network Function Virtualization
SDN – Software-Defined Networking
SEFL – Symbolic Execution Friendly Language
TCP – Transmission Control Protocol
VLAN – Virtual Local Area Network
VRF – Virtual Routing and Forwarding

Chapter 1

Introduction

Modern networks have grown to the point where understanding the way their components interact with each other is beyond most administrators' ability. In fact, studies have shown that more than 62% of network failures today are due to network misconfiguration [3]. In addition to that, hardware-based network appliances, while essential in today's operational networks, bring about even more complexity: they are expensive, complex to manage and introduce new failure modes [15].

At the same time, more and more x86 infrastructure is being deployed for networking purposes. One particular direction which has gained popularity lately is Network Function Virtualization (NFV), a network architecture concept that unifies virtualization and networking by running the functionality provided by middleboxes as software on commodity hardware [12, 17]. The idea has been taken even further by operators who have spotted the potential cost benefits and are planning to extend their services to become cloud providers specialized for in-network processing [17], which is bound to intensify this trend.

When speaking about running network functions on x86 hardware, we immediately have to turn our attention to **netfilter**, the customizable framework provided by the Linux kernel that offers various functions and operations for packet filtering, packet mangling and Network Address Translation (NAT), thus enabling most of the functions featured by dedicated firewalls. For IPv4 traffic, these functions are administrated through **iptables**, its better-known user-space counterpart, allowing us to refer to them almost interchangeably.

To understand the increased interest in iptables we have to look once again at the cloud computing world. OpenStack, an open-source software platform for cloud computing, relies heavily for its networking component, Neutron, on iptables which is used for filtering, NAT and implementing security groups, to name a few, both in its core infrastructure and, on top of that, by its users [6]. One common problem that arises here is that tool-generated rules do not always compose as expected with the ones managed by users.

In this context, it is clear that applying formal verification methods to networking problems is motivated not just by the theoretical challenges it poses, but also by practical needs. Driven by this insight, we propose a way to model an iptables-enabled device (a software router built around netfilter) using SEFL, a network description language that is **symbolic execution** friendly, and SymNet, a tool for static network checking [19, 18]. Together, they assemble into a state-of-the-art framework that enables scalable verification for stateful networks.

The resulting tool, naturally called *iptables-to-sefl*, compiles iptables deployments to SEFL models, and sums up to approximately 8k lines of Scala code (including unit tests). These models, in turn, can then be plugged into large networks and reveal configuration errors or, more generally put, policy inconsistencies. Therefore, it is a step forward towards the goal of verifying networks which are increasingly complex.

Chapter 2

Background

This chapter introduces the concepts which form the building blocks of our iptables model, making use of the extensive work conducted by Stoenescu et al. [18, 19]

We first give a short overview of the broad field of network verification, with an emphasis on the approach that we use: static verification powered by symbolic execution. We then show how it is implemented in SymNet, and how SymNet can be used to verify the models we build. Finally, we cover iptables in detail; we discuss its internal organization, as well as its most common features.

2.1 Network verification

Formal verification is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics [14]. Network verification is merely formal verification tailored for network-related questions.

Correctness. But in order to apply it, we first need to provide a rigorous definition for network correctness. It turns out that this is by itself a very involved task. Being abstract is what one might try in order to get around this formalism:

Definition 2.1.1 (Network correctness) *A network behaves correctly as long as it complies to operator's policy.*

At first, this definition does not seem to accomplish much, as what it essentially does is to delegate the requirement by introducing the need to formally define a *policy*, or, more precisely, its composing rules. Therefore, the follow-up question that arises is:

> What is a (policy) rule and how is it specified?

In fact, there is ongoing work that focuses on bringing policy specification closer to the natural language and using the resulting format to automate policy proving. This novel technique, called **policy-driven network verification**, could significantly reduce verification run-times, similar to the way *informed search* algorithms compare to the uninformed ones on large search spaces. Another analogy stems from the field of automatic theorem proving: policy-based verification resembles the *forward chaining* inference method, by propagating packets through the modelled network (a procedure further detailed in Section 2.1.2) until either the policy is proved or a contradiction is reached.

A complementary approach, rather than an alternative, which proves more practical by circumventing the need for a correctness definition that covers all desired properties is to specialize the verification to specific goals. So, instead of a *proper* correctness definition, we could define multiple *partial* ones, such as:

Definition 2.1.2 (Partial network correctness - Reachability) *If the network behaves correctly, then nodes A and B should be reachable from one another.*

Definition 2.1.3 (Partial network correctness - Encrypted traffic) *If the network behaves correctly, then TCP (Transmission Control Protocol) traffic between Alice and Bob should be encrypted.*

Notice the $correctness \implies property$ form, instead of $correctness \iff policy$, in Definition 2.1.1. Still, using simple logic rules, if we admit that the policy is a conjunction of rules (i.e. properties), $P = r_1 \wedge r_2 \wedge ... \wedge r_n$, and we prove r_i, for $i = 1, ..., n$, then $correctness \implies P$. Since the reverse implication is implicit, this yields the same equivalence. Therefore, by decomposing the policy into its defining rules not only do we get an easier to implement verification system, but we also reach the same theoretical result, provided that all rules are known. In addition to that, policy specification is an iterative process in practice, making this approach even more desirable.

Modelling and verification procedure. The other two essential ingredients needed to apply formal verification to networking problems are a model of the network and a specific procedure for proving or disproving its correctness. The model is an abstract (i.e. mathematical) representation of the network that can be easily handled by the proof procedure.

The most well-established proof procedure is **model checking** [5, 13] which does an exhaustive exploration of the states a system can be in to verify that a given property holds. Another technique which has been thoroughly explored is **reduction to SAT** (AntEater [10]). However, for large models that result even from moderately sized networks the aforementioned approaches render the verification problem intractable. Other modelling/verification techniques which have been considered over time are:

- Modelling the network as a **distributed system**, where each network element (i.e. computing node) has its own corresponding *C code*, and then running symbolic execution (discussed in Section 2.1.2) on it. This approach has two major downsides:
  (i) Currently, symbolic execution rapidly reaches its worst-case exponential complexity when run on large C programs, and
  (ii) Having to consider all possible interleavings of different threads, as in any other parallel model, makes it unsuitable even for small C programs.
- Creating a simplified model of both the **control plane** and the **data plane** of each network element and simulate their interactions (Figure 2.1). However, modelling the control plane is hard to achieve in practice, because:
  (i) Many processes that are part of it are asynchronous, which means that they inherit all the downsides of modelling distributed systems; examples include dynamic routing protocols updates and SDN (Software-defined networking) controller updates.
  (ii) Building on the previous one, control planes are usually very complex, especially since more and more middleboxes become fundamental in today's networks. One way to simplify the process of modelling it is to specialize the model for one particular process [20, 7], but this looses generality.
- Model only the data plane of the network in such a way that processing logic which does not affect packet flows is ignored (Figure 2.1). When combined with **symbolic**

execution, the result is a scalable **static data plane verification** approach that features one of the best trade-offs between model accuracy and verification complexity. Therefore, it is further discussed in the next two sections.

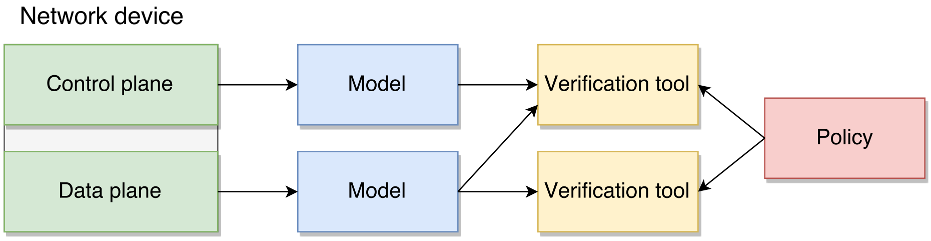


Figure 2.1: Data plane verification vs. control plane *and* data plane verification.

It is also worth briefly introducing one of the principal alternatives to model-based static network verification, **Dynamic Network Testing** (also known as *packet injection*). As a testing technique, rather than a formal verification one, it runs on the actual network infrastructure in a specially configured, isolated environment. Some network hosts, called *test agents*, generate traffic according to some test cases and evaluate the results of each network element that is subject to testing. However, its downsides greatly overweight its advantages:

- The search space is implicitly defined by the test cases, which means that a good coverage corresponds to a(n) (exponentially) large number of tests.
- Adding to the previous point, there is an inherent bias introduced by having the test cases cover commonly observed behaviours, without exploring unexpected ones, which might lead to failures.
- Overall, it is more expensive to perform and requires more resources than static analysis.

2.1.1 Static data plane verification

Starting from the observation that including control planes in our models with the end goal of verifying multiple properties of large (i.e. enterprise-size) networks does not scale, we are left with the smaller part of each network device: its data plane (Figure 2.2).

Data planes (often **forwarding planes**) include only the *rules* needed by devices to expose the functionality they are designed for. For example, the data plane of a router is represented by its routing table, and, possibly, packet filtering ACLs (Access Control Lists). A switch takes its decisions based on the MAC table. Both of these tables are specific instances of the more general Forwarding Information Base (FIB), a name that refers to the specialized hardware components responsible for forwarding in various network devices.

Limiting the models to network data planes can be interpreted in two contradictory ways:

(i) As a downside, for ignoring the other half that makes up a network device: its rule-making processes, the control plane.

(ii) As an approach that enables scalable verification, based on the insight that, even if short-lived, the data plane of a network is what dictates its operation at an instant.

The second point constitutes the motivation behind the framework described in Section 2.2. Another way of looking at it is that we simply ignore any process that leads to the current network *configuration* and establish whether it is a correct one or not. If it is, then an incremental analysis can be applied from that point forward: this implies analyzing only the updates that reach the data plane, a technique which has already been studied in the parent field of software

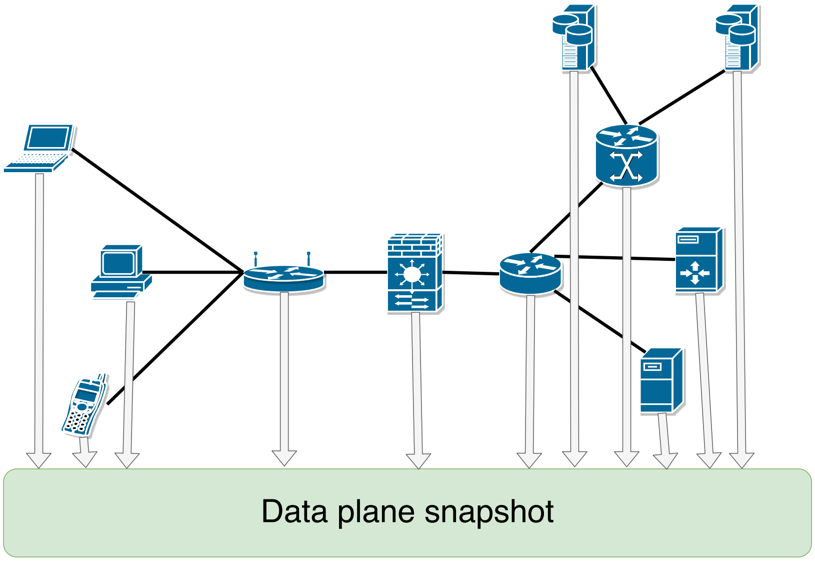


Figure 2.2: Data plane/control plane decoupling.

verification [11]. If our initial verification reports failure, human intervention is required in order to detect the control plane problem that caused the unexpected rule in the data plane.

2.1.2 Symbolic execution

Symbolic execution is a means of statically analyzing a program in order to find the inputs that cause certain parts of it to be executed. It results in a tree with each node representing a possible state in its execution and edges corresponding to state transitions caused by executing the next instruction on that specific path. Therefore, execution paths in the analyzed program correspond to state chains starting from the root of the symbolic execution tree. A state, in this context, is a collection of variables together with their constraints expressed in terms of symbolic inputs of the program and/or other variables.

To clarify things, let us step through a concrete example and highlight the construction of the symbolic execution tree (Figure 2.3) for a simple C function (Listing 1):

- We assume that the `max3` function can be called with any integer arguments. Thus, its formal parameters `a`, `b` and `c` are the input of this symbolic execution; they are assigned pure[1] *symbolic values* denoted by Greek letters α, β and γ, and together constitute the initial state (i.e. the root node in the symbolic execution tree). Implementation-wise, symbolic values can be represented as a full range of their corresponding types on the machine model that is targeted (e.g. $[-2.147.483.648; +2.147.483.647]$ for a 4-bytes signed integer).
- At **line 3**, a new variable, `max` is defined. This is where language-specific behaviour comes into play, as we use the fact that the value of an uninitialized variable allocated on the stack is unspecified[2] and we accordingly assign it a symbolic value.

  Note that in Figure 2.3 variable `max` is added to the root node due to lack of space; formally, there should be a transition between the initial state (described above) and the

[1] *Pure* in this context is synonymous with *unconstrained.*

[2] From the ISO/IEC 9899:2011 standard, informally **C11**, paragraph **3.19.3 unspecified value** says: *valid value of the relevant type where this International Standard imposes no requirements on which value is chosen in any instance.*

state where it gets introduced.

```
1   int max3(int a, int b, int c)
2   {
3           int max;
4
5           if (a < b) {
6                   if (c < b) {
7                           max = b;
8                   } else {
9                           max = c;
10                  }
11          } else {
12                  if (c < a) {
13                          max = a;
14                  } else {
15                          max = c;
16                  }
17          }
18          return max;
19  }
```

Listing 1: Simple C function used to visualize symbolic execution.

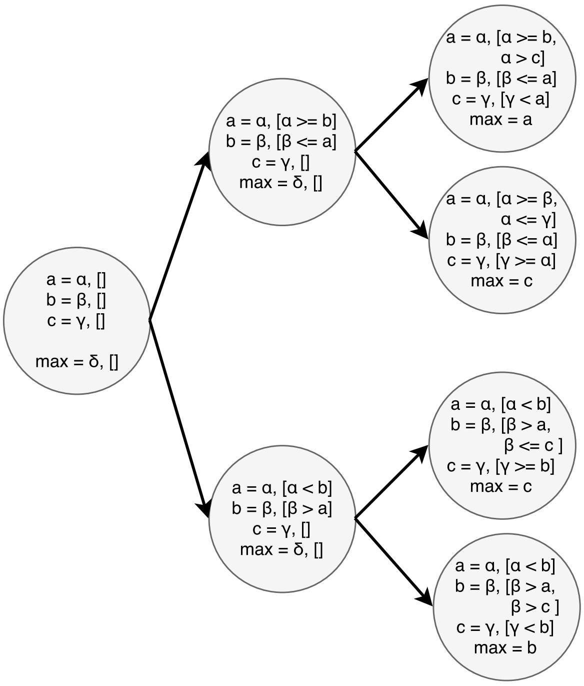


Figure 2.3: Symbolic execution tree for the C function in Listing 1.

- At **line 5**, the first `if/then/else` statement is reached causing the current state to *fork*; this creates two child nodes that correspond to the two possible scenarios: following the `then` part, which means that the condition is true, and following the `else` part, corresponding to its negation being true. Notice that constraints are added to both variables `a` and `b` in each of the new states, using antisymmetry.
- Similar to the previous step, at **lines 7 and 12** the inner `if/then/else` statements are symbolically executed yielding four more states, as well as two more constraints in each one of them.

  The interesting bit here is that variable `max` loses its previously assigned symbolic value, δ, together with its **constraints** (which happens to be an empty list in this example) once it is assigned one of `a`, `b` or `c`. It also inherits all their constraints, which is not explicitly highlighted in Figure 2.3.

We can derive a couple of properties of symbolic execution even from such a small example:

a) On the **upside**, it enables *invariant checking* by ensuring that certain properties hold in all explored states, and, thus, are true irrespective of program's input. Furthermore, it can help increase code coverage[1] by generating unit tests that explore previously untouched parts of the program.

b) On the **downside**, it commonly leads to **path explosion**, which is arguably its single most limiting factor. Path explosion refers to the exponential increase in the number of states (and, implicitly, paths) in the symbolic execution tree as the number of traversed *conditional branches* in the program accumulate. Even though heuristic methods to reduce time and/or space have been devised (e.g. parallelizing independent paths [16], merging similar paths [8], etc), it still remains a concern.

[1] Code coverage is a measure used to described the degree to which the source code of a program is executed when a particular test suite runs.

That being said, it is worth noting that it is not the *length* of the analyzed program that causes path explosion, but certain types of instructions (i.e. conditional branches). This is an important remark which becomes meaningful when used to guide the design of a DSL (Domain Specific Language) that aims to be *symbolic execution friendly*, introduced in the following section.

2.2 SymNet and SEFL

SymNet and SEFL establish together the framework we use to model and verify large, complex, stateful networks. SymNet is a network analysis tool that accepts as input a (SEFL) model of the network and uses symbolic execution to find all possible flows that might traverse it. SEFL is a network processing DSL which describes network functions as flow transformations. Even though they serve conceptually different purposes, they have been designed and optimized to be used together. Therefore, it is common to refer to the system as a whole as **SymNet**.

The **key novelty** of this network verification system is making the verification procedure (that is, symbolic execution) a core part of the design behind the network description language it uses (and even name it). Thus, careful crafting of SEFL models can keep *path explosion* (Section 2.1.2) under control and achieve performances unreached before: it has been used to verify backbone routers with hundreds of thousands of prefixes in seconds [19], and has been shown to scale linearly with the size of the analyzed network [18].

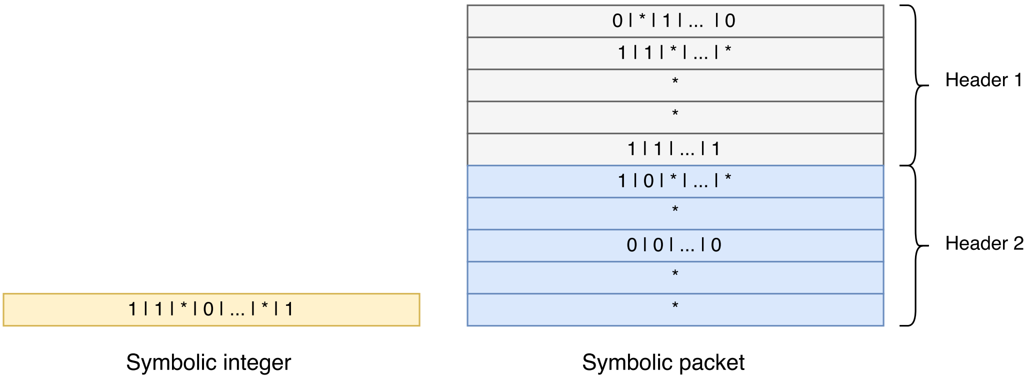


Figure 2.4: Visualization of a symbolic integer value and a symbolic packet.

Modelling packets. Packets are to SymNet how variables are to regular symbolic execution tools (e.g. KLEE [4]). The main difference is that instead of a 4-bytes signed integer, for instance, a packet is a collection of *headers*, and each header is a collection of *header fields*, similar to real implementations (Figure 2.4). The common symbolic execution terminology is also inherited: *symbolic* packet, *symbolic* header field, *constrained* header field, *concrete* packet, etc.

To continue the analogy, SEFL instructions are similar to usual instructions in a C program. However, in SEFL, instructions implicitly operate on the header fields of a single packet; that is, the memory space is a packet (or, more precisely, a *packet flow*, as we will shortly see). Therefore, another defining design aspect of SymNet is that a packet is tied to a symbolic execution path.

Another novel capability of SymNet is modelling **stateful** network devices. The way this is achieved is by adding per-flow state as additional dimensions in the packet header. It includes all state that would normally be stored in each stateful device it traverses.

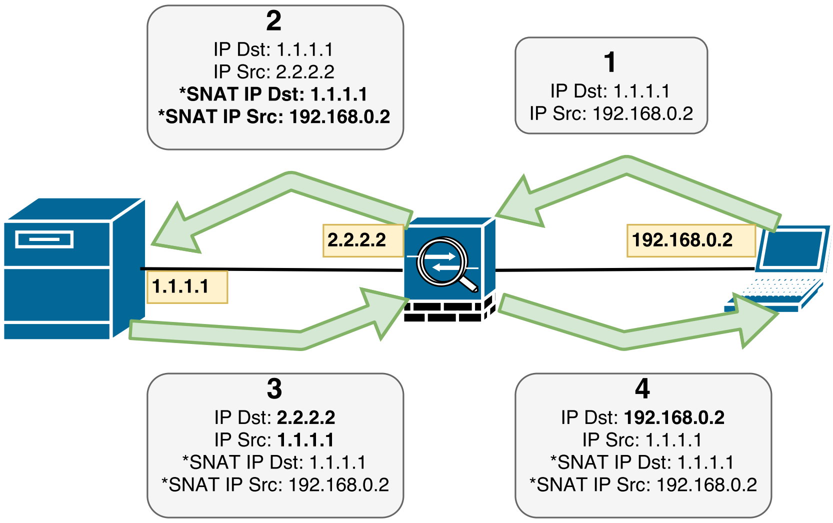


Figure 2.5: Packet flow during connection initiation and reply while behind a NAT appliance.

An example of how this works is presented in Figure 2.5. The user on the right initiates a connection with the server on the left **(1)**. He is behind a NAT appliance which is configured to SNAT (Source NAT) outgoing connections. Since the 5-tuples identifying address translations are stored by the NAT box, we model its logic in SEFL by adding this information as part of the flow. Due to lack of space, in this example we only include the source and the destination IP addresses; the star in front of these new fields helps differentiate regular packet headers and state-related fields **(2)**. The reply packet is part of the same flow, so we make sure it keeps these fields unchanged **(3)**. When the reply arrives at the NAT box, it uses the state fields to apply the reverse translation; finally, it forwards the packet to its original source **(4)**.

It is worth reiterating that SymNet fits into the static data plane verification category since the models built using SEFL are solely based on the data plane of each network element. As already argued in Section 2.1.1, this technique proves to be a good trade-off between model accuracy and verification complexity. In many formal verification systems the equivalence between the model and the target system is enforced by construction. However, in order to reach a tractable problem, besides ignoring the control plane in the modelling process, SymNet makes a few additional assumptions:

- **Independence between different flows.** A consequence of storing the state of middleboxes as part of the flow and having SEFL focus on one flow only means that per-device global state is ignored. This includes internal buffer sizes, seeds used for randomization, and various other statistics. There is also no notion of flow ordering, which might make a difference in practice.
- **The network is operationally correct.** Scenarios such as memory corruption and devices failing are not considered.

2.2.1 Building network models with SEFL

A **network model** described using SEFL and processed by SymNet is a directed graph, $G = (V, E)$, with nodes $u \in V$, often called **ports**, that have SEFL instructions associated with them, and edges $(u, v) \in E$ referred to as **links**. Additionally, the following property holds: $\forall u \in V,\ deg^{+}(u) <= 1$. In other words, each node should have no more than one outgoing edge (i.e. one-to-many relationships are not allowed). Otherwise, a form of **implicit**

Instruction	**Description**
Constrain(var, condition)	Adds *condition* to the list of constraints for variable *var*. If it is unsatisfiable, the current execution path fails. Note that *condition* is a predicate (i.e. boolean-valued function), or a conjunction/disjunction of predicates, generally expressed as a partially applied two-parameters boolean function in Polish notation. Example: Constrain("a", $== 10$), Constrain("b", $\wedge(>10, <12)$)
If (condition, then, else)	Forks the execution path, yielding two new packet flows: one continues on the *then* branch iff. *condition* is satisfiable, and the other continues on the *else* branch iff. its negation is satisfiable. Note that conditions are expressed in terms of *Constrain*.
Fail(message)	Explicitly stop symbolic execution for the current flow. Note that this is one of the instructions that are part of the language just to ease the interaction with the symbolic execution engine.
Forward(port)	Forwards this packet to *port*. Execution will continue starting with the instructions associated with it.
Fork(i1, i2, ...)	Explicitly forks the current execution path into a number of new flows and for each new flow j, continues execution with instruction i_j.
InstructionBlock(i1, i2, ...)	Acts as the *composite* type in a composite design pattern; simply aggregates the given instructions and executes them in the specified order.
NoOp	This instruction is simply skipped by the symbolic execution engine.

Table 2.1: Common SEFL instructions.

non-determinism would result (i.e. *unconditional fork*) which does not make sense in this context. **Explicit** forks are supported, as highlighted in Table 2.1 which is an overview of the most common SEFL instructions.

Note that many-to-one relationships are still possible. Therefore, while it is natural to think of a link as a physical wire connecting two ports, they are not limited to that, as it is shown in the example below.

So far we have seen **what** SEFL is capable of doing: model network functions as packet flow transformations. To illustrate **how** this is accomplished by putting together all the pieces discussed above, let us introduce the model of a simple router that only forwards packets based on its routing table (Figure 2.6).

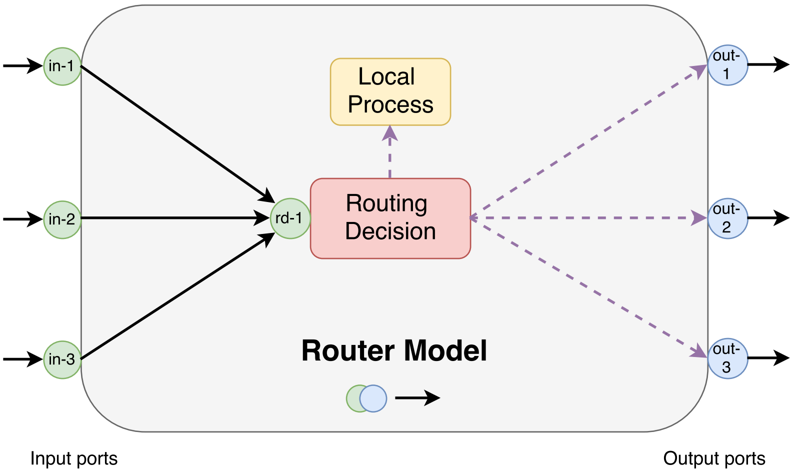


Figure 2.6: Modelling a simple router that only forwards packets based on its routing table. The model is given by $G = (V, E)$, with $V = \{in_1, in_2, in_3, rd_1, out_1, out_2, out_3\}$, $E = \{(in_1, rd_1), (in_2, rd_1), (in_3, rd_1)\}$, and the **port** rd_1 having assigned the routing decision logic.

Firstly, we notice that the edges are directed and, thus, our model has separate input and output ports. Secondly, its only job is to route any incoming packet irrespective of the input port it arrives on, so we connect all input ports to an internal, hidden port, which is the input port of a *virtual device* called **routing decision**. It is worth mentioning that the same behaviour could have been achieved by assigning the instruction *Forward(rd_1)* to all input ports instead of adding new edges to our model.

The routing decision has an input port and forwards (dotted lines) the packet to exactly one output port or to the local process. One important thing to notice is that the logic to iterate through an ordered list of IP prefixes can be naively modelled as a sequence of `if/then/else` SEFL statements, which means that its input port would suffice to express all this functionality. However, we model it as a standalone entity (through the red square) to represent the idea of model encapsulation and components decoupling by committing to well-defined **interfaces**. For instance, if we decide to use another model for routing decisions, which needs some intermmediate ports, we can do so in isolation, as long as the *contract* (input port, output ports it forwards too, etc) is respected.

Finally, the `if/then/else` approach mentioned above is the most straightforward one, but also the most inefficient way of implementing routing table lookup. That being said, the more advanced and better performing ones do not make the subject of this chapter.

2.2.2 Running SymNet and interpreting the results

Running SymNet is as simple as providing a) the graph G defined in the previous section that models the analyzed network, and b) an initial packet and a port to bootstrap symbolic execution with.

Owing to the one-to-many restriction, the graph G is usually given as a **Map** data structure (as opposed to a **Multimap**), with *from* ports as keys (therefore, each one will appear at most once) and *to* ports as values. The set V of nodes is simply the union of the set of keys and the set of values in this Map. The logic expressed using SEFL is given as a separate data structure, mapping ports to their assigned SEFL instructions.

What should the initial packet be? The *policy rule* (Section 2.1) that is being verified should drive this choice. For instance, if we want to ensure that an intranet is not reachable from the outside, we use pure symbolic packets to ensure that all possible packets are verified. On the other hand, for more specific rules, such as verifying TCP reachability between two nodes, we use concrete values for source and destination IP addresses, as well as for protocol number, and leave all the rest unconstrained.

As previously mentioned, SymNet outputs an exhaustive list of flows together with their constraints. If the list is not a very large one, manually looking through the resulting execution paths to ensure that the policy is respected is a viable option. This is often the case when a very constrained bootstrapping packet is used. However, more often than not this list is too large to be inspected by hand. For these cases, we have devised Scala matchers to use in our automated testing suites. In addition to that, as already discussed in Section 2.1, there are ongoing efforts to integrate policy validation as part of the symbolic execution engine, by means of various computation logics, such as CTL (Computation tree logic).

2.3 iptables

Before delving into how we build a model of an iptables-enabled device using SEFL and have it verified with SymNet, we must give an overview of what iptables is and how it works.

We start with **netfilter** which is a framework provided by the Linux kernel that allows various networking-related operations (such as packet filtering, network address translation and other packet mangling) to be implemented in the form of customized handlers. Historically, the project *netfilter/iptables* was started by Rusty Russel in 1998 with the intent to re-design and improve software packages that were part of previous versions of the Linux kernel, such as ipchains and ipfwadm.

iptables is the user-space command line tool used to configure the *tables* defined by netfilter, for IPv4 traffic (hence the name). Given that it is the main application used by system administrators to define rules for packet filtering/mangling, and having been an essential part of the initial project that eventually evolved into today's framework, it is common to refer through *iptables* to the entire software stack that enables this, and use *iptables* and *netfilter* interchangeably. In fact, by *modelling iptables* we actually mean modelling the netfilter logic that it triggers.

2.3.1 Organization

netfilter attains a modular internal organization by splitting rules into a hierarchy of predefined **tables** and **chains**. The latter can be further extended by users, as depicted in Figure 2.7.

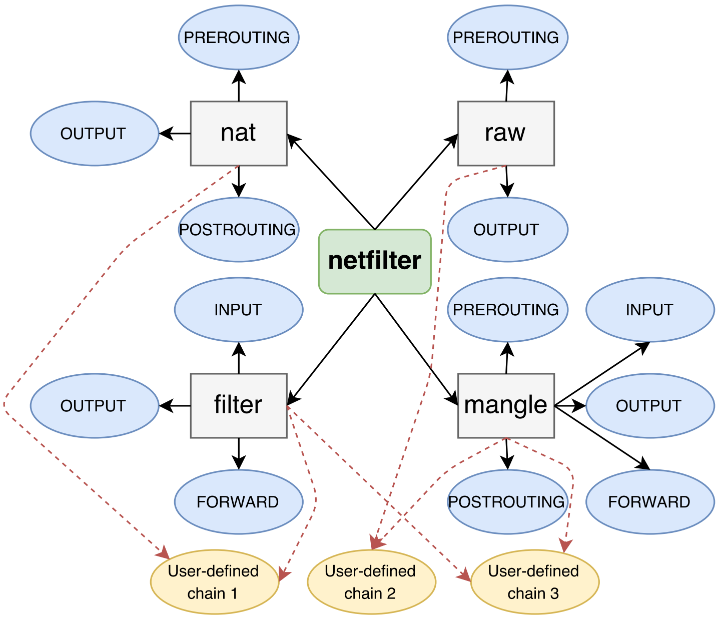


Figure 2.7: A visualization of the tables/chains hierarchy in netfilter. Note that there is at least one more table, *security*, that is not included here. Also, while it is possible to have rules from different tables *jump* to the same user-defined chain (as depicted here), it is not a good practice.

Tables. Informally, tables are used to classify the rules based on their purpose. Thus, each table is introduced to serve a specific function. The most commonly used ones are:

- **filter** - for packet filtering only
- **nat** - for one-to-many NAT, port forwarding, masquerading, packet redirecting
- **mangle** - for specialized packet alteration; one feature that comes up often in practice is *packet marking*, which allows customized traffic handling by chains following it in the processing stack.

The lowercase spelling of table names is part of the convention adopted by iptables/netfilter. In terms of the netfilter implementation, each table corresponds to a new hook in the hook system used to allow seamless extension and hot-plugging. Therefore, tables are traversed in a pre-established, built-in **order**: raw, mangle, nat, filter.

Chains. As far as the chains are concerned, they specify the point in the **processing stack** where the *chain of rules* they represent will be applied (Figure 2.8). There are 5 **built-in chains**:

- PREROUTING - right after a packet is received and before any routing decision is made
- INPUT - for locally delivered packets
- FORWARD - all packets that transit this machine will be matched against the rules in this chain
- OUTPUT - packets sent from the machine itself will be visiting this chain
- POSTROUTING - packets enter this chain after the last routing decision is made and just before handing them off to the hardware

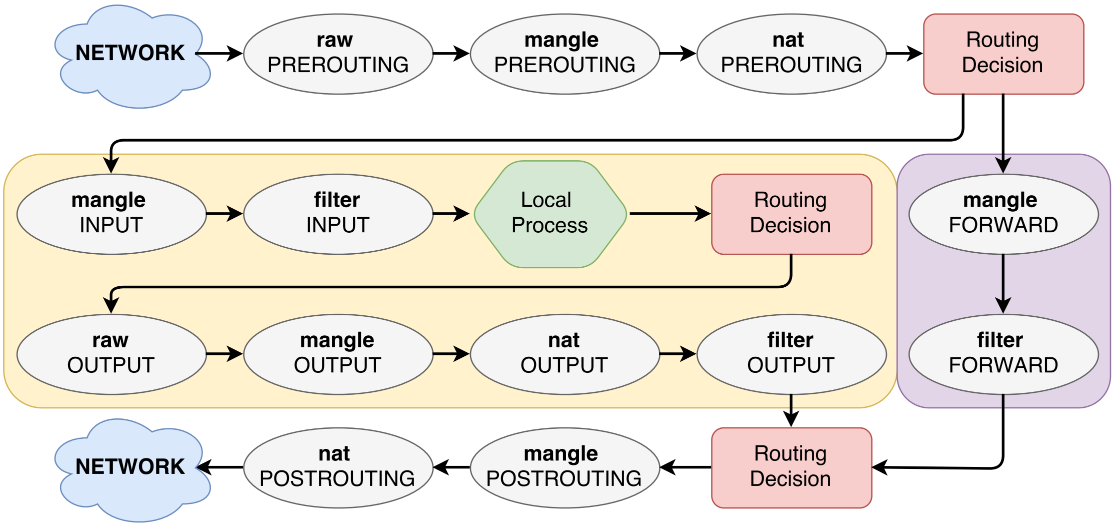


Figure 2.8: A flowgraph showing the usual processing stack in netfilter/iptables. The part with a yellow background is meaningful only for locally delivered or locally generated packets. The one with a purple background is traversed for all packets that only transit (are routed by) this machine.

Additional chains can be defined; in netfilter/iptables terminology they are called **user-defined chains**, and are featured in Figure 2.7. The motivation behind their existence is twofold:

- To **share** logic between two built-in chains from the same table and avoid rule duplication. For instance, we might want to apply the same *filtering* rules both in the FORWARD chain and in the INPUT chain.
- To achieve some sort of **separation of concerns** when organizing the rules in a large chain. An example is a chain which only does SNAT, and the last rule of the POSTROUTING chain simply jumps to it. In this way, we can short-circuit the traversal of this chain (i.e. `-j ACCEPT`) for traffic that need not be SNAT-ed.

Rules. Tables and chains are only meant to organize the rules. A rule is essentially a **(match, action)** pair. The first part, the **match**, is a conjunction of zero or more *predicates* that refer either to header fields of the packet or to some local state associated with it. The second part, the **action**, specifies *what* should happen with the packets that match. It can be followed by zero or more *target parameters*. The grammar that summarizes this is shown in Listing 2.

```
rule-specification = [matches...] [target]
match = -m matchname [per-match-options]
target = -j targetname [per-target-options]
```

Listing 2: Grammar of an iptables rule, taken from the Linux manual pages.

Rules in iptables feature an extensions-based design with separate extensions for matches and targets. There are over 60 match extensions and around 40 target extensions. Match extensions are *activated* with the `-m/--match` flag followed by the extension name. Appendix A summarizes some of the extensions often used in practice.

2.3.2 Relation to *conntrack*

Connection tracking (also known as *the state machine*) is the mechanism that allows a Linux box to relate incoming packets to tracked connections. **conntrack** is the name of the kernel module that implements this functionality. Firewalls that include it are generally called *stateful firewalls*.

conntrack is part of netfilter and it is embedded in the processing stack described in Figure 2.8. More precisely, it is called right after the raw table in the PREROUTING and OUTPUT chains. In fact, one of the purposes of the **raw** table is to allow **skipping** connection tracking logic for certain traffic (i.e. `-j CT --notrack`).

Internally, for each connection, it holds two 5-tuples, one for each direction. A 5-tuple contains protocol number, source IP, source port, destination IP and destination port. It recognizes four states:

- **`NEW`**. It is the state of a connection when the first packet (within that connection) is seen.
- **`ESTABLISHED`**. In this state, traffic in both directions has been seen. A connection in state **`NEW`** is promoted to state **`ESTABLISHED`** once a reply packet is received.
- **`RELATED`**. A connection is tagged as **`RELATED`** when it is related to another already **`ESTABLISHED`** connection. An example is a FTP data connection which is related to an established FTP control connection.
- **`INVALID`**. A connection switches to the **`INVALID`** state if the packet cannot be identified, which could be caused by various system errors (going out of memory, etc).

By itself, conntrack is simply an aggregator of connections data. It becomes particularly powerful when the information it holds is used by iptables. To do so, the **conntrack** match extension must be loaded (i.e. `-m conntrack`). The most common use-case is matching against the connection state itself (i.e. `-m conntrack --ctstate NEW`). In addition to the four states which are defined by the conntrack module, netfilter adds three more **virtual** states to be used by iptables:

- **`UNTRACKED`**. A state explicitly set in the raw table for untracked traffic, as shown above.
- **`SNAT`**. A state set for connections that have their packets Source NAT-ed by this machine.
- **`DNAT`**. A state set for connections for which this machine performs Destination NAT (e.g. port forwarding).

Besides the connection state, it is also possible to match against any of the 10 fields which form the two 5-tuples that are stored for every connection.

2.4 Summary

We started this chapter by introducing the theoretical concept of *network verification* together with the most explored techniques to achieve it. Once we argued why static verification based on models built from network data planes is the most promising approach, we switched our attention to symbolic execution as a concrete verification procedure.

In the next section, we introduced SymNet and SEFL: the former is an implementation of the previously discussed symbolic execution approach, while the latter is a modelling language tailored for network processing. We showed how models are built using SEFL and how the output of SymNet can be interpreted.

Finally, in the last section of this chapter we gave a detailed presentation of iptables, its internal organization as well as its relation with connection tracking.

In the next chapter we will make use of all the concepts accumulated so far to build a model of an iptables-enabled device.

Chapter 3

Towards a model

In this chapter we derive a model for an iptables-enabled device using SEFL. We start with a simple one, and increase its complexity and the amount of details as we gradually introduce features that allow us to express more and more complex iptables rules. We sum up with an overview of the limitations encountered along the way.

3.1 Making the first steps

Building a SEFL model which can then be verified by SymNet is as simple as providing two Map data structures, as discussed in Section 2.2:

a) the first one, from *source* ports to *destination* ports; it implicitly defines a directed graph which is our formal way of defining networks.

b) the other one, from *ports* (i.e. nodes in this graph) to SEFL instructions; this one captures the actual behaviour of each network element.

The goal of this chapter is to show how we provide these Maps starting from a deployment of iptables rules, in such a way that once fed into SymNet, the verified behaviour is that of an iptables-enabled device. To do so, in this section we start by describing the high-level idea of our model as well as the underlying algorithms that we use.

To ease the bootstrapping of our modelling process, we notice that packet processing in netfilter is built around the routing decision. To be more precise, in Figure 2.8 from the previous chapter there are three points in the processing stack where the routing table is consulted. Moreover, a routing decision was the only feature of the simple router model we introduced in Section 2.2.1. Well-known software engineering practices tell us that we should reuse existing functionality as long as it makes sense to do so. In our scenario, it does.

Thus, in the first iteration towards our end goal of reaching an iptables model from a router model we separate the only routing decision featured in Figure 2.6 into three different routing decisions, as shown in Figure 3.1.

Note that we *specialized* each one of the three new routing decisions:

- **ingress** routing decision: for traffic that **enters** this device
- **egress** routing decision: for traffic that **exits** this device
- **output** (sometimes *local*) routing decision: for traffic that is **generated by** this device

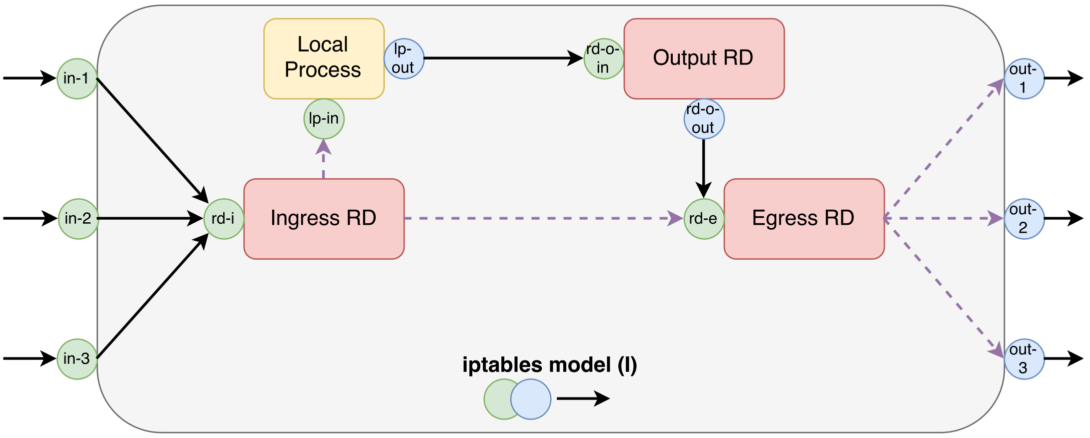


Figure 3.1: iptables model (I): Separated routing decisions. Straight arrows correspond to directed edges in our network model, while dotted-arrows correspond to possible *Forward* instructions directing traffic on those paths. Note that it is important to assign unique port names because there are no name scopes; implementation-wise, this is achieved by prefixing the *role* of the port (e.g. *in*, *out*) by the unique name of the *virtual device* it belongs to.

Another thing that might not be obvious at first is that the behaviour of two out of three of them has changed. It is easier to see this if we focus on the one for locally generated packets. We notice that its single output port is connected to the input port of the *egress* one, which means that all packets that enter it will follow that path. Thus, it does not capture the usual routing decision behaviour which simply forwards packets to output ports based on some logic (i.e. destination IP and routing table). The same applies to *ingress* which only partitions packets based on whether they should be delivered locally or not. It turns out that this new behaviour is essential to iptables, to enable matching against the output interface in the FORWARD/POSTROUTING chains, or to allow redirecting traffic in nat/OUTPUT. It is usually implemented by storing the decision (output port) instead of forwarding packets based on it.

As simple as this step might be, it still corresponds to a valid iptables configuration: the *void* one. This is a valuable remark because it allows us to validate future, more complex versions of the model by comparing their output when all chains/tables are empty to that of this *featureless* one (a form of **regression testing**).

To advance one more step in our attempt to reach a model of an iptables-enabled device, let us observe the similarity between Figure 3.1 and Figure 2.8 from Chapter 2. It seems that all we need to do in order to make them be truly alike is to add some *virtual devices* to model chain traversal. The resulting schema is shown in Figure 3.2.

The new virtual devices are called **Iptables Virtual Devices** (IVD) and have the distinctive feature that always have one input port and two output ports: an output port corresponding to *accepting* packets, whether modified or not, and another one for *dropping* packets, which is especially useful when modelling the filter table. In Figure 3.2, the former corresponds to the output ports (blue circles) of IVDs (purple squares) which are linked to the next virtual device in the processing stack. On the other hand, the *dropping* ports are not connected to any other ports and have a SEFL `Fail` instruction associated with them.

So what does this new augmented model buy us? Firstly, as already mentioned, it is closer (at least from a high-level perspective for now) to the internal organization of iptables. Secondly, it further separates the actual iptables logic to be modelled on a per-chain basis. In fact, this can be further divided by acknowledging that each chain IVD, as it represented here, is a sequence

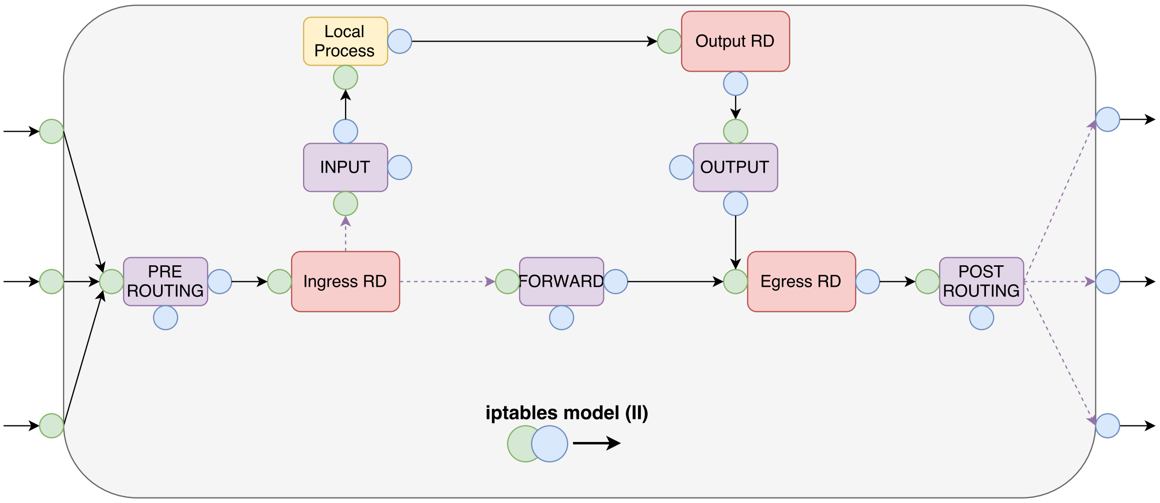


Figure 3.2: iptables model (II): Incorporated chain virtual devices. The purple squares are Iptables Virtual Devices (IVDs) and they abstract chain traversal logic.

of concrete (per table) chain traversals; for instance, there are 3 tables that might define rules in the PREROUTING chain: raw, mangle, nat. Therefore, we can improve our model by splitting each chain IVD as shown in Figure 3.3.

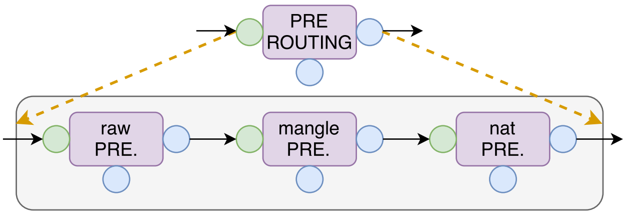


Figure 3.3: Concrete (per-table) chains constitute a processing step.

Thus, we claim that we reduced the problem of modelling an iptables-enabled device to that of implementing (in SEFL) the traversal of a chain of rules. Algorithm 1 captures the naive but straightforward way of doing this.

Algorithm 1: Traversing a chain. A chain aggregates a list of *rules* and a *default target* (i.e. policy). A rule contains a list of *matches* and a *target*.

```
TraverseChain $(C, P)$
    inputs : A chain $C$ and a packet $P$
    output: The target $T$ to jump to
    foreach $rule\ r \in C.rules$ do
        if $m(p)$ is true $\forall m \in r.matches$ then
            return $r.target$
    return $C.policy$
```

It is possible to express this algorithm in SEFL. However, there are two problems with it:

- It **ignores** a couple of essential features in iptables (e.g. jumping to user-defined chains). Thus, Figure 3.2, even though correct in terms of the abstractions it makes, hides a couple of rather involved functionality.
- It could be inefficient due to many `if/then/else` statements that result. It is worth

reiterating here that SEFL and SymNet do not give outstanding performance results by default; they require well crafted models.

We tackle the former in the next couple of sections.

3.2 User-defined chains

User-defined chains (also known as user-specified chains, or simply user chains) differ from built-in chains in that they are not automatically traversed at specific points in the processing stack of a packet. Instead, they are traversed for certain traffic that matches rules they are a target of. In other words, built-in chains are unconditionally matched against, while user-specified chains are conditionally consulted at adjustable points.

One might observe that Algorithm 1 is still valid even in the context of rules that jump to user-defined chains, as we can simply forward packets to another port that applies the same logic using the rules in that chain. What actually makes our model a lot more complex is the special **RETURN** target which is essential in advanced iptables configurations that make great use of user chains.

When a matching rule jumps to **RETURN**, the next rule that is matched against is the successor of the rule that previously caused a jump to the current user chain, as shown in Figure 3.4. Therefore, when combined with **RETURN**, user-specified chains enable chain traversal orders that resemble function calls in common procedural languages; the *callee* corresponds to the user chain, the call itself is a jump action, and the caller is analogous to the original chain where the jump to the the user chain has been made.

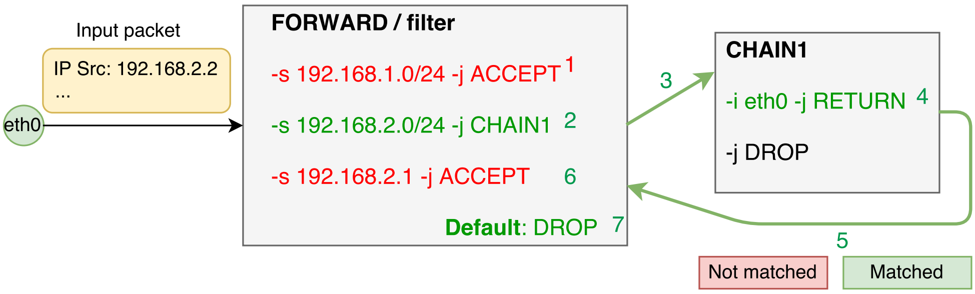


Figure 3.4: Control flow of user-defined chains in iptables, featuring the **RETURN** target. Following the function analogy, `CHAIN1` here can be interpreted as a function that encapsulates the logic for filtering out traffic arrived on input interfaces other than *eth0*.

In order to include this behaviour in our chain SEFL models (i.e. chain IVDs), let us first notice that as long as there is no rule in a chain that might jump to a user-specified one, Algorithm 1 is correct. Thus, driven by the motivation to reduce our more complicated problem to one that we know how to solve, we observe that we could obtain **RETURN**-free lists of rules by splitting our original one on boundaries given by rules that might jump to user-defined chains (called hereafter **boundary rules**).

An (abstract) example of this idea is presented in Figure 3.5. Rectangles correspond to rules in a chain. The ones that might jump to user chains are highlighted in red. Following the boundary rule-splitting idea, three new ordered sublists of rules result, as pointed out by the dashed brackets.

How do we model this new internal structure of a chain IVD using SEFL? Firstly, by decomposing its previously unified logic into multiple smaller IVDs called **contiguous IVDs**. Each

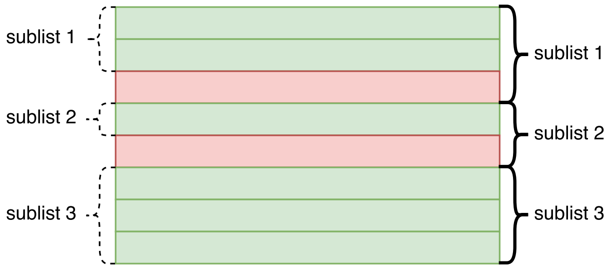


Figure 3.5: Splitting rules in a chain on boundary rules. A boundary rule is a rule that might jump to a user-defined chain.

one of these new virtual devices that are part of up a chain IVD implements the simple traversal algorithm presented in the previous section for a sublist of rules. Furthermore, to do this, it turns out that we do not need to separate boundary rules from the sublist that **precede** them. This is indicated via bold brackets Figure 3.5. This separation is not needed because our splitting guarantees that the last rule in the previous sublist does not jump to a user chain, and, thus, can lead to two actions only: a) matching the rule and jumping to a built-in target (e.g. `NAT, ACCEPT, RETURN, MARK`), and b) not matching the rule and *continuing* to the next one, none of which could lead to a **`RETURN`** to the succeeding one. In other words, we split only **after** boundary rules because that is the point we might **need to return** to.

Besides organizing rules on contiguous IVDs, we also need to link them together, such that if no rule matches in a contiguous IVD, packets are forwarded to the next one. Finally, we need to dispatch as part of our SEFL model both input and returned packets, as follows:

- **Input** packets to one of the (possibly) many contiguous IVDs that are part of a chain IVD. That is, newly arrived packets should always be forwarded to the first one, while *returning* packets should be forwarded to the one that follows the *caller* IVD (using the function analogy from a previous paragraph). To do this, a metadata field is allocated and assigned when performing jumps, and deallocated when returning. This is essentially a SEFL stack. The input dispatch is performed by a new virtual device component called **InputTagDispatcher**.
- **Returned** packets to the caller IVD. Since this is a dynamic decision too (i.e. multiple chains could jump to the same user chain), another tag is stored as a flow metadata which determines the **backlink** to forward the packet to. The virtual device that does this is called **OutputTagDispatcher**.

The resulting internal organization of a virtual device that models a chain is shown in Figure 3.6.

To summarize this section, we started with a model suitable for iptables configurations that do not make use of the **`RETURN`** target (and, thus, make little use of user-defined chains) and have extended it to support the control flow it adds. We showed how we split a chain of rules into multiple sublists that can then be modelled as separate contiguous IVDs, a new virtual device we introduced that applies the naive algorithm presented in the previous section (Algorithm 1). Finally, we illustrated how all these new features play out together in a comprehensive diagram (Section 3.6).

3.3 The *nat* table

As mentioned at the beginning of this chapter, our goal is to add an increasing number of features to an iptables-enabled device and see how they influence the model we devised so far.

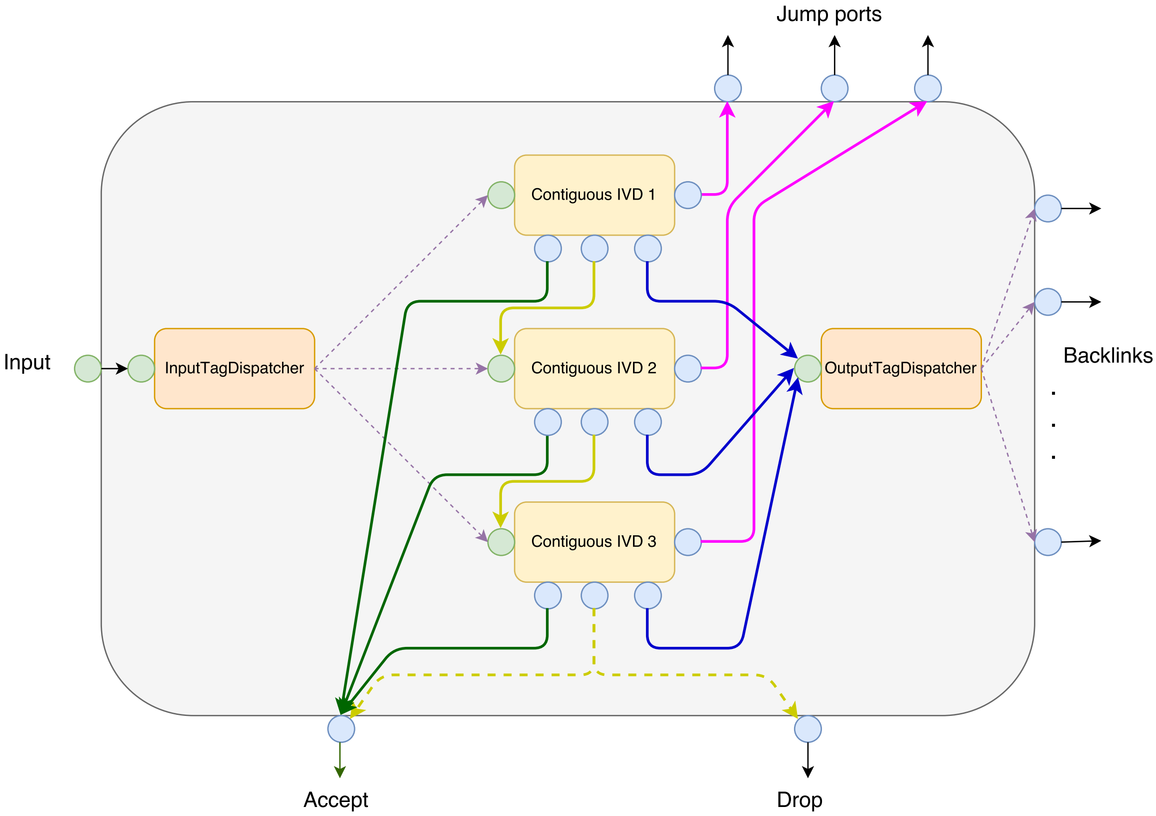


Figure 3.6: Internal organization of a chain IVD. It has one input port, one accept port, one drop port, a number of jump ports equal to the number of rules that might jump to user chains, and a variable (but still static) number of backlinks, that corresponds to the chains that might jump to this one. Each **contiguous IVD** has 1 input and 5 outputs: one for `ACCEPT` (green), one for `DROP` (skipped due to lack of space), one for forwarding packets to the next contiguous IVD (yellow; for the last one it is given by the chain policy), one for jumping to a user chain (magenta; only in the last rule of that sublist of rules), and, finally, one for returning to the caller chain (blue; managed by **OutputTagDispatcher**).

In this section we show why the **nat** table needs special handling and adapt our model to it.

Network Address Translation (NAT) is a generic way of referring to one of the many types of address translation that have been conceived over time. In addition to them, iptables introduces its own terminology including the following important features (also listed in Appendix A):

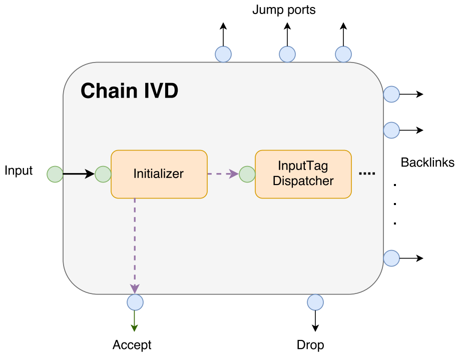
- **`SNAT`**: rewrites source IP/port with other IP/port (possibly from a given range); alternative names for this technique include *many-to-one NAT*, *port address translation*, *NAT overload*.
- **`DNAT`**: similar to the previous one but applied to destination IP/port; its most common use-case is *port forwarding*.
- **`REDIRECT`**: redirects packets to the machine itself; it is similar to **`DNAT`**-ing to one of this machine's IP addresses.
- **`MASQUERADE`**: this is the true *many-to-one NAT* technique; in addition to the capabilities provided by regular **`SNAT`**, it is resilient to interface-level changes (reset, IP change, etc).

In the given context, what does our current model fail to capture? It is true that all targets listed above have a well-defined effect on the flow they are applied to, as shown in an example in Chapter 2 (Figure 2.5). It turns out that the problem is not related to a specific target, but instead it has to do with the semantics of the entire table as a whole, as the name of this section suggests. To be more precise, the following should be ensured:

- Each chain in this table (i.e. PREROUTING, POSTROUTING and OUTPUT) is traversed **one time only** for each *flow*. The *decision* (i.e. target) taken is then automatically applied for each packet belonging to that flow.
- The **reverse translation** should be performed on *reply* packets automatically.

To be more explicit, Listing 3 shows how the latter can be achieved in SEFL for a **SNAT**-ed packet. It is a simplified version as it assumes TCP traffic only; to conform to the real implementation, an additional `If` statement must be included to check that the L4 protocol is TCP.

```
If(Constrain(IPDst, =="nat-new-src-ip"),
   // then..
   If(Constrain(TcpDst, =="snat-new-src-port"),
      // then..
      InstructionBlock(
        Assign(IPDst, "snat-orig-src-ip"),
        Assign(TcpDst, "snat-orig-src-port")),
      // else..
      NoOp),
   // else..
   NoOp)
```



Listing 3: Sample SEFL code showing how reply packets are automatically rewritten in NAT-ed TCP connections.

Figure 3.7: Updated chain IVD model to accomodate the Initializer component.

The logic for automatically reapplying previous decisions is similar. The only question that remains is how do we **skip** the *nat* table for these connections? In order to do so, we need to ensure that the corresponding SEFL code is executed prior to dispatching packets to one of the contiguous IVDs, which takes place in the InputTagDispatcher, as shown in Figure 3.6.

Therefore, we introduce a new *virtual device* that sits in front of the InputTagDispatcher that can rewrite packets either as a reapplication of a previous *nat* decision, or as part of reply packets, and skip the whole internal logic for the former. It is called **Initializer**. Figure 3.7 shows how our previous model of a chain IVD changes to include it. Note that this new virtual device is only really needed for chains which are part of the *nat* table.

3.4 Connection tracking

In Section 2.3.2 of the previous chapter we discussed the relation between iptables/netfilter and **conntrack**. We saw that while its business is kept separately from the other netfilter components, it is embedded in the processing stack after the *raw* table in the PREROUTING and OUTPUT chains.

We also described the four states a connection can be in (**NEW**, **ESTABLISHED**, **RELATED**, **INVALID**), as well as three additional virtual states that are used in iptables rules (**UNTRACKED**, **SNAT**, **DNAT**). However, we did not mention how to model the logic behind conntrack that controls the state of connections associated with incoming packets. In fact, there are certain limitations due to our flow modelling based on SEFL that makes us unable to model some of them (further discussed in Section 3.5).

What we do model are the transition to **`NEW`** and the one between states **`NEW`** and **`ESTABLISHED`**. The former is as simple as marking flows as **`NEW`** when the metadata field that keeps track of the connection state is not set. The latter is analogous: we mark flows as **`ESTABLISHED`** when a *reply* packet that belongs to a **`NEW`**-tagged connection is received.

To augment the model we devised so far with this new tiny, but important logic, we introduce a new *virtual element* right after the chain IVDs corresponding to the *raw* table in the PREROUTING and OUTPUT chains. The result is shown in Figure 3.8.

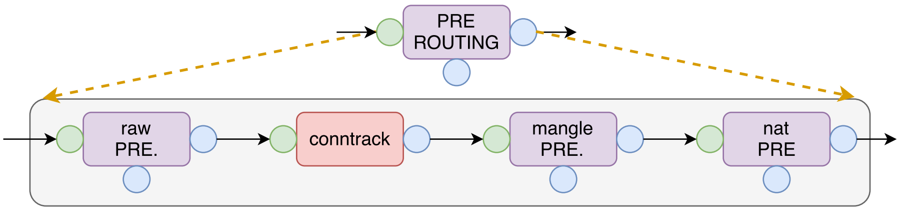


Figure 3.8: Element modelling conntrack transitions in the processing stack. The OUTPUT chain is analogous.

3.5 Limitations

Our model is built to mimic the netfilter implementation when stripping it to the bare minimum that deals with packet flows. This should be of no surprise as it is one of the core design principles behind SEFL and SymNet: focus only on execution paths in the original code that are tied to at least one flow, and ignore boundary cases (i.e. out-of-memory errors, memory corruption, etc).

This also means that we inherit all the modelling limitations of our toolset. The most important ones have already been touched upon in Section 2.2. In the following paragraphs we analyze the way these limitations reflect on our iptables model.

Global NAT state. One of the assumptions that SymNet makes as a consequence of having each execution path tied to a packet flow is that global state is not captured.

One of the network function that makes use of global state is Network Address Translation. Besides the per-flow state, one thing that NAT network appliances ensure is that there are no rewrite clashes across different connections. For instance, for a rule such as `-s 10.0.0.0/8 -j SNAT -to-source 1.1.1.1`, a NAT device guarantees that each source address belonging to the specified network gets a different port number. More than that, it can start rejecting upstream traffic once the total number of simultaneous connections is reached. Note that this might happen due to a threshold imposed by the device rather than consuming the entire port range (implicit in this rule).

However, that is a boundary case, not a rule that a network administrator would be able to enforce nor want to. In other words, if host **A** cannot connect to server **B** because there are no ports available anymore, that is a faulty situation which cannot be deterministically relied on. Thus, the benefit from modelling this behaviour would be negligible even if we were able to.

`RELATED` (connection state). In Section 3.4 we left unmodelled some of the states a connection can be. In particular, the **`RELATED`** state is one which comes up often in iptables rules.

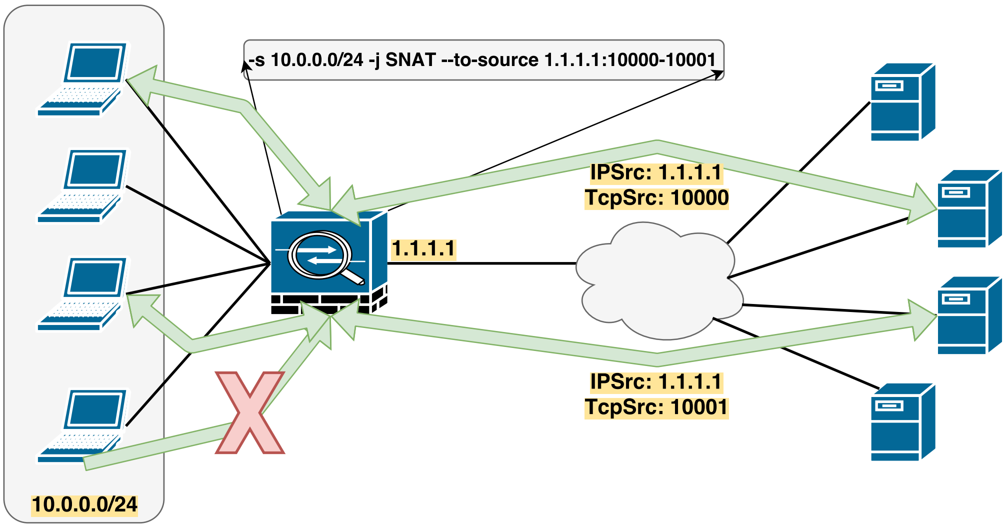


Figure 3.9: Connection is rejected by NAT appliance due to no available ports remaining. The SNAT rule gives a range with only two ports which makes the third connection initiation to be rejected by the NAT box. This can only be accomplished by keeping track of state across flows.

Informally, a connection is considered **`RELATED`** when it is **caused** by another established one. This includes all *data connections* which are started as part of a protocol that have a dedicated *control connection*.

The reason it cannot be modelled stems from the **flows independence** design assumption: the two connections correspond to two different flows which simply cannot be *related* by our network description language.

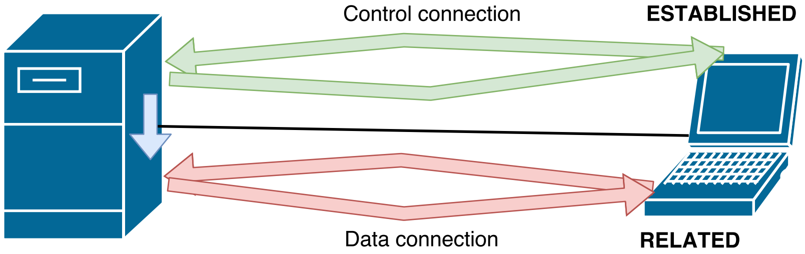


Figure 3.10: A control (established) connection starts a new data (related) connection.

Local process. In the block diagrams showing the internal components of our iptables model (Figure 3.2) we included one that abstracts the local process. This was done to be on par with the internal organization of iptables featured in the previous chapter (Figure 2.8).

A very accurate model of the local process would be the source code itself, which would imply running symbolic execution on the entire kernel and the applications that sit on top of it. This is obviously intractable.

An alternative inspired by how entire networks are modelled with SEFL is to focus only on simplified application-specific behaviours. In fact, our design allows embedding such user-specified custom logic. For instance, a very raw model of a server would only swap source and destination addresses and forward the packet to the OUTPUT chain. Alternatively, the simplest local process would act as a *sink*, not doing anything with incoming packets which causes symbolic execution to consider that flow a success.

Additionally, a design choice we made to allow bootstrapping symbolic execution with traffic originating from an iptables-enabled device is to **expose** the output port of the *virtual device* that abstracts the local process (Figure 3.11). This is an alternative, simplified way of modelling certain applications, as shown above, but only for outbound traffic.

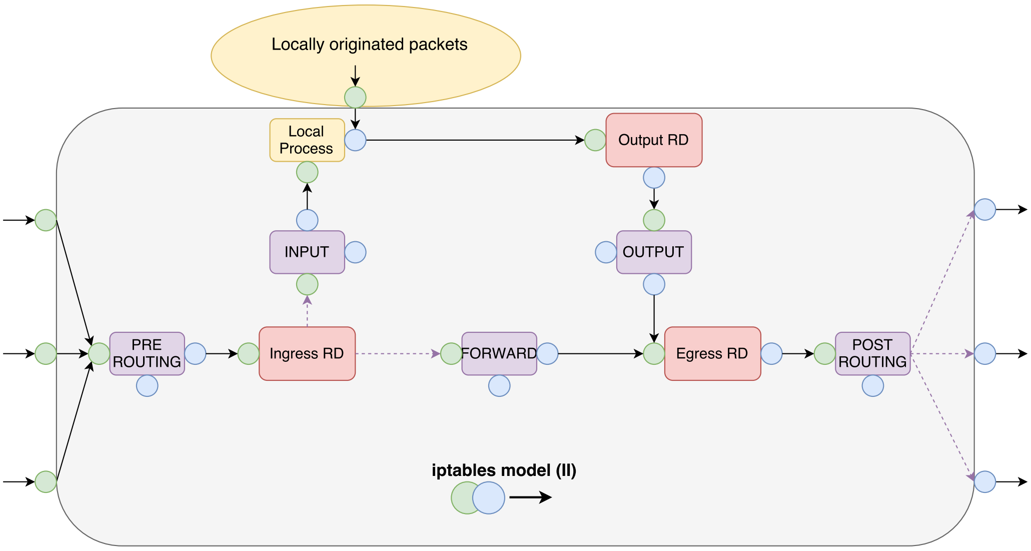


Figure 3.11: Exposing the output port of the local process VD to allow injecting packets on it, and, thus, simulating locally originated traffic.

3.6 Summary

In this chapter we derived a model for an iptables-enabled device starting from a simple router.

We started by building a processing stack in which each chain is modelled as a separate entity that encapsulates the traversal of a list of rules. We continued by adding support for user-defined chains and the **RETURN** target. Together, they introduce a new traversal order that resembles function calls in procedural languages. Next, we dedicated two sections to show that the *nat* table and *connection tracking* need dedicated logic. Once we solved (some of) the issues they introduced, we can finally assert that the resulting model is the final one. We followed that up with a discussion about its limitations.

In the next two chapters we present the internal design and implementation choices behind the *iptables-to-sefl* tool, and evaluate the models we construct from various iptables configurations.

Chapter 4

Design and implementation

In this chapter we delve into the design and the implementation details of *iptables-to-sefl*, the tool we built to generate SEFL models from iptables configurations with the end goal of having them embedded in complex networks which can then be verified using SymNet.

We begin with a high-level outline of the computation steps that take place between dumping iptables configurations to returning a model as expected by SymNet. Following that, we take each of those steps separately and discuss the most important aspects of their implementation.

4.1 Design overview

iptables-to-sefl is essentially a compiler: its input is a file that aggregates per-table rules dumped by the *iptables* command line tool, and it outputs two Map data structures expected by SymNet, one for port connections and the other for port instructions (Section 3.1). Alternatively, it can return a Scala object modelling the whole device if integration with other network components is desired.

Driven by this observation, we designed *iptables-to-sefl* to mimic the internal structure of a compiler too; it features three separate, well-defined phases:

- **Parsing**. The input file is read and an in-memory Abstract Syntax Tree (AST) is built. It materializes the hierarchy shown in Figure 2.7.
- **Validation**. This step resembles *semantic analysis* in usual compilers and its purpose is twofold: i) ensures that the configurations conform to the official specifications, and ii) augments the AST with additional semantic information that either could not be gathered during parsing or it is more robust to do it in a second pass.
- **Code generation**. This phase is essential in any compiler and ours is no exception: based on the now validated in-memory AST we generate SEFL code for every rule as a two step process: first generate the constraints that correspond to rule matches, and then generate the code to follow its target.

  In addition to that, we also consider as part of this step putting all things together according to the model template devised in the previous chapter (Figure 3.2 and Figure 3.6). We call it a *template* because it is only a model once we fill it with real rules. In fact, this step is similar to the *backend* component in a compiler that is tied to a concrete machine architecture instead of an abstract one. Prior to this step, the AST is a simplified and better organized view of the input rules but it still abstract enough to target various table/chain structures.

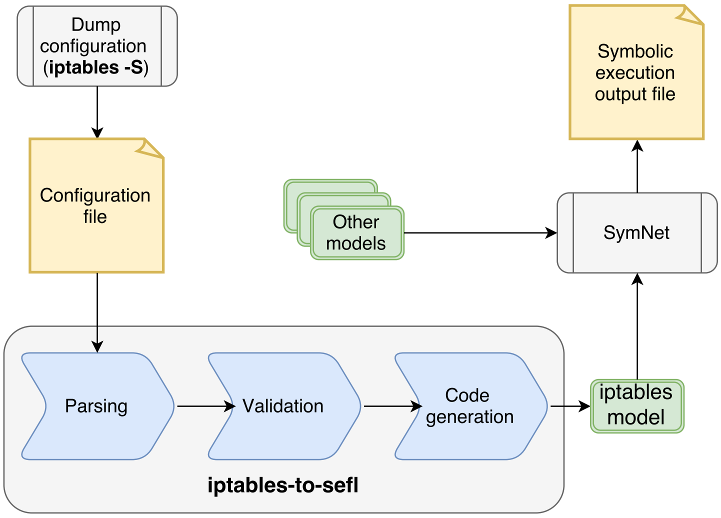


Figure 4.1: High-level design of *iptables-to-sefl*.

Feeding the resulted model to SEFL, possibly alongside models of other network elements, can be regarded as a fourth step and is included in Figure 4.1 which is a high-level overview of how our tool fits in the big picture of network verification using SymNet. Similarly, dumping the configs can be considered step zero.

Nodes in AST are called **IptElement**s. The class hierarchy that makes the core of *iptables-to-sefl* is presented in Figure 4.2. It is a proper use of the composite design pattern in which multiple classes are both *components* and *composites* at the same time (e.g. *Chain*, *Rule*, *NegatedMatch*). ASTs follow the aggregation relationships. As with code generation, the only (concrete) types that can change are the leaves: the matches and the targets.

In addition to the AST types that are used throughout all three computation phases of our tool, it is worth introducing the class hierarchy behind our model template too (Figure Figure 4.3). It is designed to clearly separate each component (i.e. *virtual device*) that we identified in the previous chapter and to make their interconnection as smooth as possible. It also inherits the composite pattern approach from our core AST hierarchy.

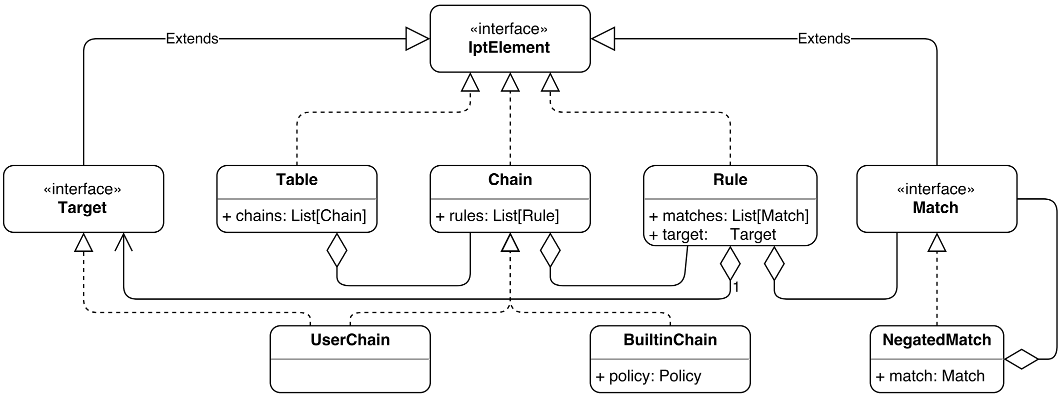


Figure 4.2: The core class hierarchy in *iptables-to-sefl*. Interfaces *Target* and *Match* are the ones that must be subclassed when adding extensions. It is also indicated that a *UserChain* can be jumped to as it implements the *Target* interface. The **NegatedMatch** class is used to conveniently express the negation of another *Match* instance.

On top of the compiler-like internal structure, *iptables-to-sefl* features an **extension-oriented**

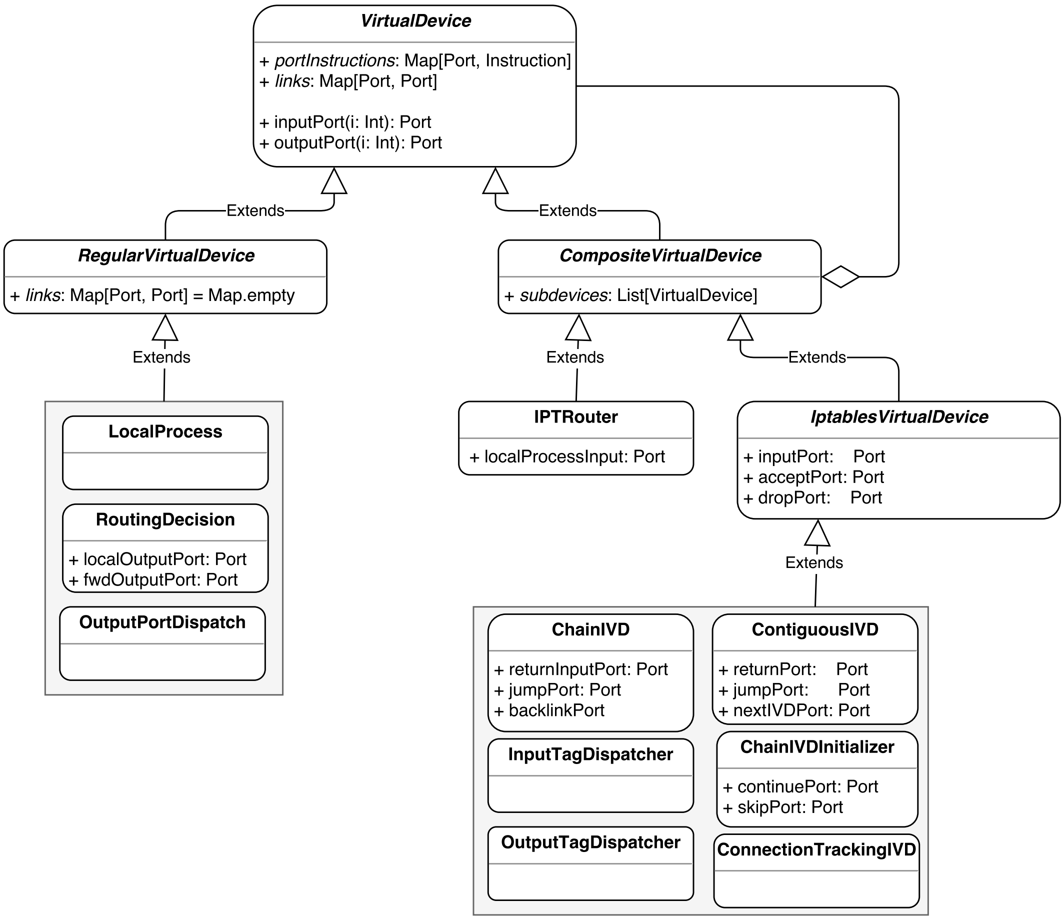


Figure 4.3: The class hierarchy of the model template part in *iptables-to-sefl*. Besides the abstract base class *VirtualDevice*, there are three other classes designed for components with certain expectations: *RegularVirtualDevice* for self-contained ones, *CompositeVirtualDevices* for components that might built upon some other ones, and *IptablesVirtualDevice* for components that implement *"iptables"* logic: receive packets, traverse list of rules and apply target, which might result in an *accept* or a *drop*.

design that aims to ease as much as possible the addition of new extensions. Since iptables supports over 100 official match and target extensions, it is unrealistic to cover all of them upfront, in a inextensible, monolithic design. Even if we could, that is still not advisable, as it is error prone and hard to debug or refactor. To add to that, netfilter itself is built around the idea of seamless extensibility. Therefore, while it does not happen too often, new, fresh extensions can be added anytime.

The building blocks of this design are the interfaces *Target* and *Match* (pedantically, Scala traits), on one hand, and a generic rule parser that can be **injected** user-defined parsers for newly added targets and matches, on the other hand. They are applied in the order given as part of the *ParsingContext* object. This object gets passed and updated during parsing and is further discussed in the section dedicated to the parsing phase below.

4.2 Parsing

Our approach to iptables configuration parsing is inspired by **Parsec** [9], a functional, monadic parser combinator library. The term *combinator* here refers to two different things:

(i) A function with no *free variables* (i.e. self-contained, pure function). It is at the core of **Combinatory logic**, the theory behind the design of functional programming languages.

(ii) A loosely defined functional design pattern. It is centered around the idea of combining other (smaller) functions to reach a more complex functionality.

The parsers built using this well-studied approach have a couple of desirable properties. Firstly, they are able to parse certain *context-sensitive grammars* by employing an LL(*), backtracking-based approach. Secondly, they yield loosely coupled parse rules. Since iptables configurations can be easily generated by a context-free grammar, our choice is mainly motivated by the second property, especially given the extension-oriented design that we are trying to achieve.

Listing 4 shows some of the basic combinators that enable us to express increasingly complex parsers. *many* and *some* are analogous to the unary operators Kleene star and Kleene plus (its extension), respectively: they take a parser that can parse a type **T** and repeatedly apply it, each time consuming some portion of the input. In the end, they return a list of **T**s parsed. The *optional* combinator tries to parse a type **T** and returns `Some[T]` if it succeeds or `None`, otherwise. Note that the `for/yield` syntax is a syntactic sugar for monadic computation which propagates error implicitly, without obfuscating the code we write. It de-sugars into repeated applications of the `>>=` operator (called **bind**) which is one of the defining functions behind monads.

```
1   def optional[T](p: Parser[T]): Parser[Option[T]] =
2     (p >>= (x => pure(Option(x)))) <<|> pure(None)
3
4   def many[T](p: Parser[T]): Parser[List[T]] =
5     some(p) <<|> pure(Nil)
6
7   def some[T](p: Parser[T]): Parser[List[T]] =
8     for {
9       x <- p
10      y <- many(p)
11    } yield (x +: y)
```

Listing 4: Some of the base combinators we use to build parsers. Note that *some* and *many* (line 4) are mutual recursive. For instance, the implementation of *many* can be read as *either parse some **T**s or fail* (return Nil, the empty list). For *some* (line 7) it would be *one **T** needs to be parsed; then, maybe parse some others.*

iptables defines a grammar only for rules because chains and tables are managed through the command line tool. To get everything as part of a single configuration file, we defined a simple format for chain and table specification, starting from (and integrating) the grammar of a rule (Listing 2). To get such a configuration file from an existing deployment we also provide a script that translates from the raw format output (`iptables -S`) to the one defined by us. The complete grammar is shown in Listing 5 below.

```
table  = <<table_name>> [chains...]
chain  = <chain_name[:policy]> [rules...]
rule   = [matches...] [target]
match  = -m match_name [per-match-options]
target = -j target_name [per-target-options]
```

Listing 5: The complete grammar of an iptables configuration file. The policy of a chain is specified only for built-in chains. For user chains it is always **`RETURN`**.

As mentioned in the previous section, the extension-based design is achieved by passing a **ParsingContext** object to the rule parser, allowing us to keep it (besides the table and chain parsers) unchanged irrespective of the extensions we want to turn on. This object contains just that: lists with match and target extensions that should be used when trying to parse a rule. By using the *implicit parameters* Scala feature this allows us to inject various extension configurations at run time.

In addition to that, there is one more subtlety when it comes to match extensions: the module loading part (i.e. the flags `-m/--match`). One might think that providing dummy parsers to *consume* this part of an iptables rule would suffice. However, the real reason they are used is to create a sort of *namespace* for match extension options. This means that different extensions could use the same option and give it a completely different meaning. To illustrate this, let us consider the rules from Listing 6. Both of them use the `--mark` match option, although the first one refers to a metadata field known as *ctmark* (connection tacking mark) while the second one refers to the *nfmark* field (netfilter mark).

```
-m connmark --mark 0x0/0xffff0000 -j CONNMARK --save-mark ...
-m mark ! --mark 0x2/0xffff -j SNAT ...
```

.

Listing 6: Two iptables rules that highlight the extension specific option *mark* which yields different behaviours when activated by match extensions *connmark* and *mark*. They are taken from a real-world OpenStack deployment. For more details about the rules, check out Appendix A.

How do we support this behaviour as early as parsing time? We introduce the notion of *match-enabled extensions*. Thus, to correctly parse (i.e. build a correct AST) for both rules, we change our rule parser to use a *cycle-based parsing* approach: we start with two extensions that recognize flags `-m connmark` and `-m mark`, and leverage our new feature by updating the current activated extensions with the ones enabled by the previously parsed match. Then, the next *cycle* begins.

The real code that accomplishes what we have just described is presented in Listing 7.

```scala
def ruleParser(implicit context: ParsingContext): Parser[Rule] = {
  def matchesParserRec(
      context: ParsingContext,
      accMatches: List[Match]): Parser[List[Match]] = {
    val matchParsers = context.matchExtensions.flatMap(_.matchParsers)

    for {
      newMatch <- optional(oneOf(matchParsers: _*))
      accNext <- newMatch match {
        case Some(m) => matchesParserRec(
          context.addMatchExtensions(m.extensionsEnabled), accMatches :+ m)
        case None => pure(accMatches)
      }
    } yield accNext
  }

  for {
    matches <- matchesParserRec(context, Nil)
    target  <- oneOf(context.targetExtensions.map(_.targetParser): _*)
  } yield Rule(matches, target)
}
```

Listing 7: The implementation of the rule parser. It uses a helper, tail-recursive, accumulator-based function to represent *cycles*. Each recursive call might modify the *ParsingContext* argument by adding new match extensions as dictated by the previously parsed match. An interesting remark is that it is one of the few parsers we defined (including the ones for extensions) that exceeds 15 lines of code, which is an indication of the effectiveness of our parsing framework in simplifying our work.

4.3 Validation

What is validation and why is it important? As mentioned in the first section of this chapter, the validation step in *iptables-to-sefl* resembles semantic analysis in common programming language compilers: it catches errors that escaped the parsing phase and changes or augments the AST with information or structure needed during code generation. We go through each one of the use cases that are derived from this motivation in the following paragraphs.

Configuration checking. The need to ensure that the rules we test against were valid was the first motivation for this separate computation step in *iptables-to-sefl*. Not only did it help us to better understand the semantics of each iptables feature, but it also boosted our productivity by avoiding endless debugging when the root problem was an incorrectly written rule.

One might ask, if this was the only motivation behind the *validation* step, would it be reasonable to disable it once we made sure that our implementation is *correct* and the rules we feed it are taken right from an iptables dump (which we assume is correct)? We believe that the answer is still **no** because (i) knowing for sure that your implementation is correct is a very hard problem (we should know that very well), and (ii) there is an inherent risk to introduce bugs determined by us trying to understand an informal specification. Therefore, there is always a chance to find that our tool reports validation error on a configuration file, which would have us investigate which misunderstanding of ours caused it. That is particularly likely to happen as we started with very tight conditions and have incrementally loosen them once we found that a specific scenario **is** allowed and adapted our code to it.

How does this step reflect in code? The **IptElement** interface provides the method validate(context: ValidationContext): Maybe[Self] that is implemented in subclasses. For instance, to validate a user-chain as a target of a rule (as depicted in Figure 4.2) we do the following check: `chain != this && table.chains.contains(this)`. It says that a user chain we jump to (i) has to be different than the current one (*recursive jumps* are not allowed in iptables), and (ii) should be part of the same table as the caller chain.

Other common configuration checks are (in no particular order):

1. table name validity: only *raw*, *mangle*, *nat*, *filter* are recognized
2. built-in chains can appear only in certain tables (e.g. FORWARD in *mangle* and *filter*)
3. some matches and targets are only valid in certain chains and tables:
   (i) target DNAT is only valid in table *nat*, chains PREROUTING and OUTPUT
   (ii) target DROP is allowed only in table *filter*
   (iii) match `--out-interface` can be used only in chains FORWARD, OUTPUT and POSTROUTING
4. port range validity
5. valid protocol in match `--protocol` (can be a number or a name)

We end this subsection with an example of a scenario in which our validation rules were too tight, reflecting our understanding of the official specification at that moment, and a real-world deployment proved us wrong. Some manual pages mention that target ACCEPT can appear only in the *filter* table. Despite being a trustworthy source, it turns out that it can be used in any other table for **short-circuiting** the remaining traversal of that chain. This is one of the desirable situations in which all we had to do was to drop one check and still have everything work correctly. In other scenarios, code refactoring was needed.

Forward references to user chains. Another problem that came out early in the development process was that a user chain might appear as part of the input configuration file **after** a rule that jumps to it, belonging to a previous chain. The problem is that both a list of *Match*es and a *Target* are required in order to construct an instance of class *Rule*, but at that point in the parsing process, a reference to the *yet to be parsed* chain cannot be obtained.

The way we solved it was by creating a dummy implementation of trait *Target*, called **PlaceholderTarget**, that points to a user chain through its name only. Thus, later, in the *validation* step, we make sure to check that the pointed-to chain does exist and replace the placeholder with a reference to it.

This task is performed by other compilers too, especially for programming languages that do not impose preceding declarations at lexical scope for referenced names.

Interface name wildcards. Lastly, a less well known feature of iptables is that a string followed by a plus can be used to match all interfaces that are prefixed by that string, as part of the `--in-interface` and `--out-interface` matches. For instance, the rule `--out-interface eth+ -j ACCEPT` can be used to accept traffic that leaves this device on Ethernet interfaces.

Modelling string operations in an efficient manner for symbolic execution is an open problem. However, we can avoid doing that by statically determining the interfaces that start with the specified string, followed by modifying and repeating that rule for each one of them. For the previous rule, if our device used interfaces *eth0*, *eth1* and *eth2*, among others, the chain of rules would become:

```
<MY_CHAIN>
  [preceding rules]
  --out-interface eth0 -j ACCEPT
  --out-interface eth1 -j ACCEPT
  --out-interface eth2 -j ACCEPT
  [succeeding rules]
```

Implementation-wise, we handle this similarly to how we handled match extension loading to avoid name clashes between different options: we add a method to class *Rule*, mutate(interfaces: List[String]): List[Rule], that receives a list of interface names and returns a list of generated rules. It is called for all rules during chain validation. A rule that does not contain any interface match will return a list containing just itself. One that does will return a list as shown in the previous example. An interesting thing to note is that it might also return an empty list which means that no interface matched. This can be seen as an optimization. However, we would probably be much better off check-failing on this condition and letting the network administrator know that his deployment contains a redundant rule, which is probably not intended.

4.4 Code generation

Code generation in *iptables-to-sefl* refers mainly to retrieving the SEFL instructions that model rule matches and targets. In essence, rules are the input to our tool.

However, similarly to how a compiler in its backend component might insert architecture specific instructions to ensure program correctness (e.g. memory barriers), we also need to link together rule specific code to form chains, followed by linking chains from multiple tables to form a whole step in the iptables processing stack. This is repeated until we get the model for the entire iptables-enabled device, following the principles thoroughly explained in Chapter 3.

We cover both aspects in this section.

Match/target codegen. The extension-based design allows decoupling extensions from the core implementation. Following the same approach as for validation, we augment the code with two methods:

- seflCode(options: CodegenContext): Instruction in trait *Target*, and
- seflCode(options: CodegenContext): SeflCondition in trait *Match*.

Both of them receive an object of type *CodegenContext* which aggregates both context related information, such as port ids for accepting, dropping, returning packets (e.g. used as *Forward(port)*), and constant data structures such as a Map from interface names to their internal identifiers (e.g. used by matches such as `--in-interface`).

That being said, their return types differ. A target may return any instruction type, and usually it really aggregates multiple ones in a single *InstructionBlock* when complex logic is expressed (e.g. SNAT, MARK). On the other hand, a match returns a custom class, *SeflCondition*. In most cases it contains a single *Constrain* instruction, but there are some matches that express a conjunction of *Constraint*s. The best example is the `--tcp-flags` match which needs to ensure that certain flags are set while others are cleared.

Assembling the rules. Both the arbitrary *Instruction* and the *SeflCondition* objects that model a rule need to be assembled into a larger SEFL program that captures the traversal of a chain of rules.

Due to the complex semantics of certain targets (see below), we decided to use the simple algorithm discussed in Chapter 3, trading off efficiency for correctness. In our code, it is implemented as part of the *ContiguousIVD* class which contains a sequence of rules of which only the last one might jump to a user chain. Scala's functional APIs make this as easy to implement as specifying two **reduce** (also known as *fold*) operations:

- The first one operates on the list of rules and generates a sequence of `if/then/else` instructions. For each rule, the condition is a conjunction of all SEFL instructions generated out of its matches. The *then* part is simply the instruction given by target, while *else* contains the recursive application on the remaining rules, as shown in Listing 8.
- The second one is used to generate the conjunction mentioned above. This time, it is an **imbricated** sequence of `if/then/else` instructions with a depth given by the accumulated *SeflCondition*s (Listing 9).

  Note that this explains why we do not return an arbitrary *Instruction* in the code generation function that is part of the *Match* trait: only a *Constrain* can be used as a condition in an *If* instruction in SEFL.

```
if (condition1)
  target1
else if (condition2)
  target2
else if (condition3)
  target3
...
else
  targetN
```

Listing 8: Pseudocode for traversing a chain of rules, as implemented in SEFL.

```
if (match1)
  if (match2)
    ...
    if (matchN)
      target
  else
    nextRule
else
  nextRule
```

Listing 9: Pseudocode for expressing a conjunction, as implemented in SEFL.

This algorithm is made a lot more complex by the following iptables features:

- **Negating** the `--tcp-flags` option yields a disjunction which needs to be integrated with the rest of the matches that are part of a rule.
- Some matches need to compute a **temporary value** to be used in *Constrain* instructions. This is due to SEFL providing very tight semantics for the *If* instruction. An example of such a match is `--mark` *value[/mask]* that first need to do a bitwise AND between the *mark* field and the specified *mask* before comparing it with *value*.
- Some targets do not imply **stopping traversing** the rest of the chain. An example of `MARK` being used precisely in this way to filter out any traffic other than the one identified as management in a Demilitarized Zone (DMZ) is presented in Listing 10.

  In chain FORWARD packets are redirected to the user chain R-ALLOW-DMZ-MGMT for better modularity and readability. In this chain, it first zeros out the current *nfmark* value and then it sequentially jumps to three other user chains. Each one of them contains rules to match specific traffic and mark it with a unique value. If at the end of this processing the mark value is 0xE (specific to management traffic), it accepts the packet. Otherwise, it is dropped, as specified in the chain policy.

```
<<mangle>>
  <FORWARD:DROP>
    -j R-ALLOW-DMZ-MGMT
  <R-ALLOW-DMZ-MGMT>
    -j MARK --set-xmark 0x0/0x0
    -j S-TRUSTED
    -j D-DMZ
    -j D-SRV-MGMT
    -m mark --mark 0xE -j ACCEPT
  <S-TRUSTED>
    [other rules]
  <D-DMZ>
    [other rules]
  <D-SRV-MGMT>
    [other rules]
```

Listing 10: An example of a *mangle* table that relies on MARK target not short-circuiting the traversal of the chain.

In summary, even though they introduced drastically different behaviour compared to our early versions of *iptables-to-sefl*, we managed to nicely integrate them with the existing algorithm.

4.5 Code structure

Now that we clarified the implementation challenges we encountered in each of the three phases that define our tool and showed how we solved them, let us briefly present the physical structure and organization of this project.

As most Scala projects, the test suites are separated from the source code: the former reside in src/test while the latter fills the src/main directory. Both of these subtrees share the same structure. Each one of them sums up to approximately 4k LOC (so a total of **8k LOC**), while the dominant paradigm used is **functional programming**, as already indicated in the previous sections.

Even if seemingly tightly bound to SymNet, *iptables-to-sefl* has been developed as a standalone project. In fact, even though not so obvious from SymNet's internal structure, we only interact with it through a **public** API that exposes the following components:

- The *Instruction* class hierarchy that allows us to generate SEFL code. Table 2.1 is a summary of the most common instructions we use.
- The *Expression* class hierarchy that allows us to express *Constraints*. It includes equality expressions, logical expressions, etc.
- The *canonical names* module that simplifies the way we refer to fields in packet headers, which is especially useful since one of the features in SEFL is *guaranteed memory safety*.
- The *execution context* module that provides a very simple and intuitive interface to run symbolic execution on a given model.

The root of the actual source directory is shown in Listing 11. The core/ directory contains the **IptElement** class hierarchy (Figure 4.2), the monadic base parsers discussed in Section 4.2, and the generic parsers for tables, chains and rules. The extensions/ directory has one subdirectory for each extension supported. Each one of them defines custom *Target*s and/or *Match*es alongside their corresponding parsers. The virtdev/ directory contains the model template

```
core/
extensions/
virtdev/
Driver.scala
package.scala
```

Listing 11: Contents of the relevant source code subdirectory.

class hierarchy (Figure 4.3) and all the logic discussed in Section 4.4 for putting all chains and tables together. **Driver** is a class that sequentially runs the three phases and outputs all explored execution paths. It makes use of the *execution context* exposed by SymNet. Finally, **package.scala** is a file that defines various package-level utility functions and/or constants.

4.6 Summary

A high-level design overview of *iptables-to-sefl* followed by detailed discussions about each one of its processing phases were covered in this chapter.

In the first part we argued that *iptables-to-sefl* features most traits of a compiler and briefly described each one of its three computation steps: parsing iptables rules, validating them and generating SEFL code. We also presented the main class hierarchies that are part of its core: the **IptElement** one, which defines nodes in the AST that results after first step, and the one with abstract class **VirtualDevice** at its root, for modelling packet processing components.

In the second part we took a more detailed approach at discussing the challenges we faced in each one of the three steps that *iptables-to-sefl* splits its execution in. We started with the monadic combinators we define to build fast and easily customizable parsers. We followed that up with three main problems that are not caught during parsing, but are dealt with in the second pass through the AST, in the validation phase. Finally, we presented the building blocks for code generation and brought into discussion the unusual behaviour featured by some extensions.

Chapter 5

Evaluation

In this chapter we summarize the key results we reached by running SymNet on various models generated by our tool, *iptables-to-sefl*. It is divided into two sections. In the first one we argue that the models we build capture the packet processing logic behind netfilter. In the second one we use models built from synthetic rules to evaluate the quality of our code generation approach discussed in previous chapters.

5.1 Acceptance tests

Acceptance tests help answer the question *"Does our model reflect the semantics of netfilter?"* Certainly we would rather not introduce inconsistencies in a software system that is supposed to find bugs in another one. In some formal verification systems the *model - real system* equivalence is enforced by construction. However, this is not true for SymNet for reasons described in Section 2.2, and, thus, we need to run hand-crafted tests to confirm our expectations.

Driven by this insight from the very beginning, we have performed extensive unit testing, focusing on specific components and/or behaviours, as well as integration testing, when linking different parts together. Our testing suite contains more than **130 total tests** and achieves over **91% code coverage** [2]. This means that we have generated models that cover most of our model generation corner cases. However, this should not be confused with *generated code* coverage, which depends on input packet, input port, as well as the interaction with the network as a whole, considering that some of the networks elements that we model are stateful.

In the following paragraphs we show a small subset of the scenarios that we tested against.

Simple NAT policy. We start with a very simple example to give an intuition of how these tests are performed by including the Scala code too.

Listing 12 shows the entire test. Between **lines 2-7** the nat/POSTROUTING chain is set up with two rules: one to match an entire private network (`-s 192.168.1.0/24`) and the other one to only match a host in the same network (`-s 192.168.1.100`). Between **lines 8-13** we run symbolic execution injecting a packet with the source IP set to `192.168.1.100` (line 12), starting from this chain's input port (lines 10-11). Between **lines 14-19** we define the *rewrite constrain* that we expect to take place. Finally, at line 22 we **express the policy** using the custom Scala matcher *containConstrain* that looks up the given constrain in the list of successful paths output by symbolic execution.

```
test("rl lecture - unreachable example") {
  val postroutingChain = toChain(
    toRule("-s 192.168.1.0/24
              -j SNAT --to-source 141.85.200.2-141.85.200.6"),
    toRule("-s 192.168.1.100
              -j SNAT --to-source 141.85.200.1")
  )
  val (successPaths, _) =
    symExec(
      postroutingChain,
      postroutingChain.inputPort,
      Assign(IPSrc, ConstantValue(Ipv4(192, 168, 1, 100).host))
    )
  val rewriteConstrain =
    Constrain(
      IPSrc,
      :&:(:>=:(ConstantValue(Ipv4(141, 85, 200, 1).host)),
          :<=:(ConstantValue(Ipv4(141, 85, 200, 1).host)))
    )

  // FAILS
  successPaths should containConstrain rewriteConstrain
}
```

Listing 12: An example of a NAT misconfiguration taken from a *Local Area Networks* lecture quiz. Notice that Scala already makes policy specification easy to understand and express owing to its relaxed syntax rules.

This example is intended to show how a network administrator would express policies and catch inconsistencies. Indeed, this test fails because the specified host IP matches the first rule in the table. In fact, the second rule will never be matched, indicating a misconfiguration which would hopefully be found as a result of the policy failing.

NAT-ed connections. To take our previous SNAT configuration from a toy example that is intended to give a glimpse at how simple policies can be expressed in code to a fully-fledged, end-to-end one, we use a network topology as shown in Figure 5.1.

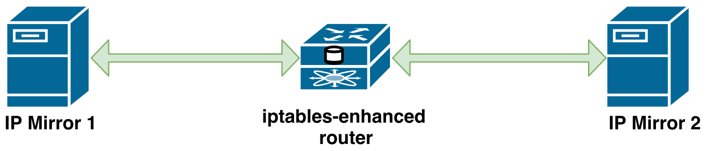


Figure 5.1: Generic network configuration used to test various stateful functions.

The **IP mirror** component is used to model reply packets. All it does is to swap the source and destination addresses (both IP and port). Thus, by connecting two of them through an intermediate device that is our iptables-enhanced router, we obtain bi-directional communication between them. This is in fact a pattern we often use when testing any kind of stateful configuration. It is not tied to NAT-ing in any way.

How do we specialize it for NAT testing? Besides the rules that define our model, the specific scenarios that we test are controlled by input packets and input ports. For instance:

- To test that SNAT rules are **correctly applied**, we inject an initial packet that matches the rule we are interested in on the output port of *IP mirror 1* and check if a packet with the rewritten source IP reaches the input port of *IP mirror 2*.

- To test that **reverse** SNAT rules are correctly applied, we add one more policy rule to the previous example: a packet with the reverse source/destination IPs should reach the input port of *IP mirror 1*.
- To test that the nat/POSTROUTING chain is traversed **only once** for a SNAT-ed connection, we add the following policy rule: the input port of this particular chain is traversed only once. The special logic for skipping it otherwise is subjected to test here.
- To test that the same rule does not apply if the connection is initiated from the **opposite direction** (SNAT rules use source IP matches, in general), we inject a packet on the output port of *IP mirror 2* and check that a packet with the rewritten source IP **does not** reach the input port of *IP mirror 1*.
- To make the previous scenario work, we change our model by adding a DNAT rule in the nat/PREROUTING chain. We make sure that all previous policies are still valid.

This is a non-exhaustive list but gives a good idea of the challenges behind Network Address Translation testing.

Connection state switch. Another key feature that has been thoroughly tested is the transition between connection states.

The network model we built to test it is shown in Figure 5.2. It consists of our regular iptables-enabled device (left) and an IP mirror element that is used to generate reply packets, as previously described (right). We simulate traffic from our device to the server by injecting a concrete packet on the output port of the local process component. As discussed at the end of Chapter 3, we expose this port as part of our model interface precisely for testing purposes like this one (Figure 3.11).

A new metadata field is stored for each packet flow to track its connection state, intuitively called **ctstate**. The first time a packet belonging to a connection is seen, its *ctstate* is set to **NEW** (transition **1 -> 2**). Once a packet in the **reverse** direction is processed, the state becomes **ESTABLISHED** (transition **3 -> 4**). Our policy in this case can be stated such as: *there must be a packet flow that reaches the local process and its ctstate field is ESTABLISHED.*

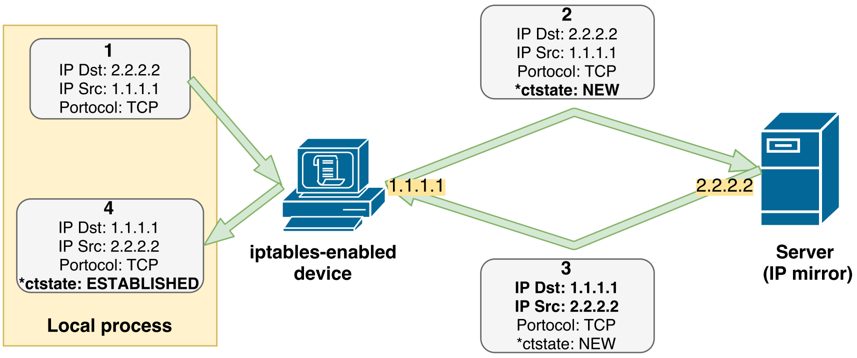


Figure 5.2: Network model built to test connection tracking. Fields in bold font are the ones modified in the last step.

Untracked connection. As a follow-up scenario for the previous one, we also ensured that marking certain traffic as **UNTRACKED** in the *raw* table leads to no state transitions.

An alternative approach to bypassing the *model - real system* equivalence problem that we have yet to experiment with is *black-box testing*. It requires establishing a(n) (usually large) list of input packets and injecting them both in the real system and in its model constructed with *iptables-to-sefl*. Then, monitor output packets in the real iptables-enabled device and try to match them against the exhaustive list of (symbolic) packet flows output by SymNet. If for some input packet there is no symbolic flow that matches the observed output packet, the equivalence is denied. The reversed implication is obviously not necessarily true, but it increases our *belief* that the implementation is correct.

5.2 Performance tests

Things to say: * tests performed: 1. many rules, on a single filter table; 2. many IPTRouters in line. * overall results * tables/graphs

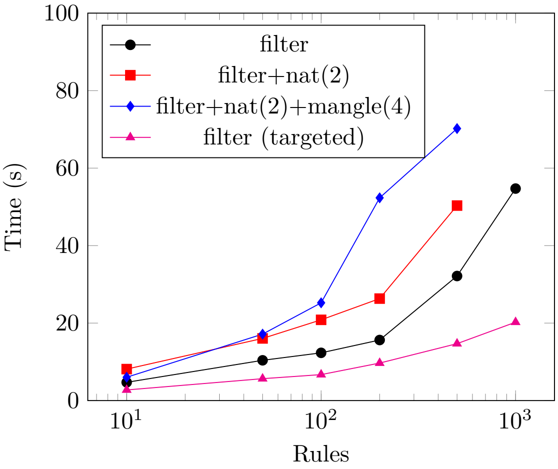


Figure 5.3: Verification time by chain size.

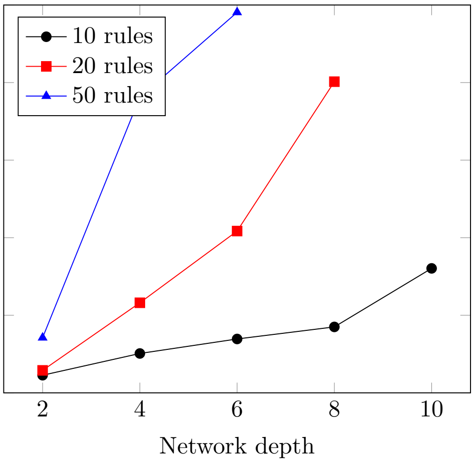


Figure 5.4: Verification time by network depth.

Besides these script-generated configurations, we have also run SymNet on a model of a Neutron L3 agent[1] from a real OpenStack deployment. Its complete iptables dump is included in Appendix B.

Exploring all possible execution paths by injecting a pure symbolic packet on one of its input ports takes around **25s** and **6GB** of RAM. This generates 1345 packet flows. Only 74 of them are successful. Out of all failed ones only 6 are explicitly dropped. The remaining ones are packets caused by internal logic of our model that cannot be materialized. Since there are only two rules that might DROP packets, it becomes manageable to inspect those packets by hand. Moreover, if we constrain just the destination IP address of the injected packet to a concrete value, symbolic execution time drops to **5s**. This shows the importance of having a policy drive symbolic execution rather than exploring all possible execution outcomes.

[1] It is essentially a router that uses the Linux IP stack and iptables in an isolated *network namespace*. This in turn is the Linux terminology for the *Virtual Routing and Forwarding* (VRF) concept that takes the idea of Virtual Local Area Networks (VLANs) from the L2 switching world to the L3 world.

Chapter 6

Conclusion

Summary. TBA

Future work. Because our efforts were mostly targeted at having a feature-full implementation that can take a real world iptables deployment and return a SEFL model, there is still room for optimization in terms of the generated SEFL code. One key observation is that most of the verification time is spent in the underlying satisfiability solver. It is the main source of memory consumption too. Therefore, fine-tuning our code to generate only the needed conditions on each execution path (or as close to that as possible) can dramatically speed up symbolic execution. However, this is by itself a very hard problem considering the involved semantics of some iptables rules.

In addition to that, further integration testing needs to be performed. In this thesis, we limited our tests mostly to synthetic rules which might miss certain practices that are common in real deployments. Nevertheless, this is just a small part of a larger project that aims to build a provably correct model of an OpenStack deployment and have it verified with SymNet. Therefore, integrating it with the other networking components used as part of Neutron (e.g. Open vSwitch) will be the immediate follow-up challenge.

Appendix A

iptables match/target extensions

Match extensions [1]	
`--comment` *comment*	Administrator-friendly way of commenting rules.
`[!] --mark value[/mask]`	Matches the netfilter mark field associated with a packet (which can be set using the *MARK* target below). If a mask is specified, it is ANDed with the mark value before the comparison.
`[!] --mark value[/mask]`	Matches the netfilter mark field associated with a connection (which can then be set using the *CONNMARK* target below). If a mask is specified, it is ANDed with the mark value before the comparison.

`[!] --ctstate statelist`
`[!] --ctproto l4proto`
`[!] --ctorigsrc address[/mask]`
`[!] --ctorigdst address[/mask]`
`[!] --ctreplsrc address[/mask]`
`[!] --ctrepldst address[/mask]`
	This module, when combined with connection tracking, allows access to the connection tracking state for this packet/connection. The first option matches a comma-separated list of connection states. The second one matches the layer-4 protocol. The next four match against original/reply source/destination address.

`[!] --source-port,--sport port[:port]`
`[!] --destination-port,--dport`
`↪ port[:port]`
`[!] --tcp-flags mask comp`
`[!] --syn`
`[!] --tcp-option number`
	These extensions can be used if `--protocol/-p` `↪ tcp` is specified, besides the usual `-m/--match` `↪ tcp`. The first option matches the given source port number or port range. The second option matches the given destination port number or port range. The third option matches when the TCP flags are as specified. The first argument, *mask* is the flags which we should examine, given as a comma-separated list. The second argument, *comp*, is a comma-separated list of flags which must be set. Flags are **SYN ACK FIN RST URG PSH ALL NONE**. The fourth option matches TCP packets with the SYN bit set and the ACK, RST and FIN bits cleared. Such packets are used to request TCP connection initiation. The fifth option matches if the specified TCP option is set.

`[!] --source-port,--sport port[:port]`
`[!] --destination-port,--dport`
`↪ port[:port]`
	These extensions can be used if `--protocol/-p` `↪ udp` is specified, besides the usual `-m/--match` `↪ udp`. The first option matches the given source port number or port range. The second option matches the given destination port number or port range.

Table A.1: Common iptables match extensions.

Target extensions [1]	
`-j CT --notrack`	Allows to set parameters for a packet or its associated connection. The option `--notrack` disables connection tracking for this packet. `-j NOTRACK` is an alias. This target is only valid in table *raw*.
`-j MARK --set-xmark value[/mask]`	It is used to set the Netfilter mark value (32 bits wide) associated with the packet. Option `--set-xmark` zeros out the bits given by *mask* and XORs *value* into the packet mark (nfmark). This target is generally used in table *mangle*, in the PREROUTING.

`-j CONNMARK --set-xmark value[/mask]`
`-j CONNMARK --save-mark [--nfmask m1]`
`↪ [--ctmask m2]`
`-j CONNMARK --restore-mark [--nfmask`
`↪ m1] [--ctmask m2]`
	This module sets the netfilter mark value (32 bits wide) associated with a connection. The first option zeros out the bits given by mask and XORs value into ctmark. The second one copies the packet mark (nfmark) to the connection mark (ctmark) using the given mask. The third one is the opposite of the second one.

`-j SNAT --to-source`
`↪ ip[-ip][:port[-port]]`
	Specifies that the source address of the packet should be modified, as well as all future packets in this connection. A single new source IP address can be specified, or an inclusive range. The same applies for the port number. Where possible, no port alteration will occur.
	This target is only valid in table *nat*, in the POSTROUTING and INPUT chains.

`-j DNAT --to-destination`
`↪ ip[-ip][:port[-port]]`
	It is analogous to *SNAT* but for destination IP and port addresses. Also, if the port range is not specified, then the destination port will never be modified.
	This target is only valid in table *nat*, in the PREROUTING and OUTPUT chains.

`-j MASQUERADE --to-ports port[-port]`	Masquerading is equivalent to specifying a SNAT mapping to the IP address of the interface the packet is going out, but also has the effect that connections are forgotten when the interface goes down. This target is only valid in table *nat*, in the POSTROUTING chain.
`-j REDIRECT --to-ports [port[-port]]`	It redirects the packet to the machine itself by changing the destination IP to the primary address of the incoming interface. Option `--to-ports` specifies a destination port or range of ports to use. Without it, the destination port is never altered. This target is only valid in table *nat*, in the PREROUTING and OUTPUT chains.

Table A.2: Common iptables target extensions.

Appendix B

iptables configuration of a Neutron L3 agent

```
<<raw>>
  <PREROUTING:ACCEPT>
    -j neutron-l3-agent-PREROUTING
  <OUTPUT:ACCEPT>
    -j neutron-l3-agent-OUTPUT
  <neutron-l3-agent-OUTPUT>
  <neutron-l3-agent-PREROUTING>

<<mangle>>
  <PREROUTING:ACCEPT>
    -j neutron-l3-agent-PREROUTING
  <INPUT:ACCEPT>
    -j neutron-l3-agent-INPUT
  <FORWARD:ACCEPT>
    -j neutron-l3-agent-FORWARD
  <OUTPUT:ACCEPT>
    -j neutron-l3-agent-OUTPUT
  <POSTROUTING:ACCEPT>
    -j neutron-l3-agent-POSTROUTING
  <neutron-l3-agent-FORWARD>
  <neutron-l3-agent-INPUT>
  <neutron-l3-agent-OUTPUT>
  <neutron-l3-agent-POSTROUTING>
    -o qg-09d66f0a-46 -m connmark --mark 0x0/0xffff0000 -j CONNMARK --save-mark
        ↪ --nfmask 0xffff0000 --ctmask 0xffff0000
  <neutron-l3-agent-PREROUTING>
    -j neutron-l3-agent-mark
    -j neutron-l3-agent-scope
    -m connmark ! --mark 0x0/0xffff0000 -j CONNMARK --restore-mark --nfmask
        ↪ 0xffff0000 --ctmask 0xffff0000
    -j neutron-l3-agent-floatingip
    -d 169.254.169.254/32 -i qr-+ -p tcp -m tcp --dport 80 -j MARK --set-xmark
        ↪ 0x1/0xffff
  <neutron-l3-agent-float-snat>
    -m connmark --mark 0x0/0xffff0000 -j CONNMARK --save-mark --nfmask 0xffff0000
        ↪ --ctmask 0xffff0000
  <neutron-l3-agent-floatingip>
  <neutron-l3-agent-mark>
    -i qg-09d66f0a-46 -j MARK --set-xmark 0x2/0xffff
  <neutron-l3-agent-scope>
    -i qr-6a98a347-87 -j MARK --set-xmark 0x4000000/0xffff0000
    -i qg-09d66f0a-46 -j MARK --set-xmark 0x4000000/0xffff0000

<<nat>>
```

```
  <PREROUTING:ACCEPT>
    -j neutron-l3-agent-PREROUTING
  <INPUT:ACCEPT>
  <OUTPUT:ACCEPT>
    -j neutron-l3-agent-OUTPUT
  <POSTROUTING:ACCEPT>
    -j neutron-l3-agent-POSTROUTING
    -j neutron-postrouting-bottom
  <neutron-l3-agent-OUTPUT>
    -d 203.0.113.103/32 -j DNAT --to-destination 192.168.13.3
  <neutron-l3-agent-POSTROUTING>
    ! -i qg-09d66f0a-46 ! -o qg-09d66f0a-46 -m conntrack ! --ctstate DNAT -j ACCEPT
  <neutron-l3-agent-PREROUTING>
    -d 203.0.113.103/32 -j DNAT --to-destination 192.168.13.3
    -d 169.254.169.254/32 -i qr-+ -p tcp -m tcp --dport 80 -j REDIRECT --to-ports
        ↪ 9697
  <neutron-l3-agent-float-snat>
    -s 192.168.13.3/32 -j SNAT --to-source 203.0.113.103
  <neutron-l3-agent-snat>
    -j neutron-l3-agent-float-snat
    -o qg-09d66f0a-46 -j SNAT --to-source 203.0.113.100
    -m mark ! --mark 0x2/0xffff -m conntrack --ctstate DNAT -j SNAT --to-source
        ↪ 203.0.113.100
  <neutron-postrouting-bottom>
    -m comment --comment "Perform␣source␣NAT␣on␣outgoing␣traffic." -j
        ↪ neutron-l3-agent-snat

<<filter>>
  <INPUT:ACCEPT>
    -j neutron-l3-agent-INPUT
  <FORWARD:ACCEPT>
    -j neutron-filter-top
    -j neutron-l3-agent-FORWARD
  <OUTPUT:ACCEPT>
    -j neutron-filter-top
    -j neutron-l3-agent-OUTPUT
  <neutron-filter-top>
    -j neutron-l3-agent-local
  <neutron-l3-agent-FORWARD>
    -j neutron-l3-agent-scope
  <neutron-l3-agent-INPUT>
    -m mark --mark 0x1/0xffff -j ACCEPT
    -p tcp -m tcp --dport 9697 -j DROP
  <neutron-l3-agent-OUTPUT>
  <neutron-l3-agent-local>
  <neutron-l3-agent-scope>
    -o qr-6a98a347-87 -m mark ! --mark 0x4000000/0xffff0000 -j DROP
```

Listing B.1: iptables dump of a Neutron L3 agent.

Bibliography

[1] The iptables-extensions website by netfilter. http://ipset.netfilter.org/iptables-extensions.man.html. Accessed: 2017-06-28.

[2] iptables-to-sefl - git repository and issue tracker. https://github.com/calincru/iptables-sefl. Accessed: 2017-06-28.

[3] Richard Alimi, Ye Wang, and Y Richard Yang. Shadow configuration as a network management primitive. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 111–122. ACM, 2008.

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

[5] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[6] James Denton. *Learning OpenStack Networking (Neutron)*. Packt Publishing Ltd, 2014.

[7] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd D Millstein. A general approach to network configuration analysis. In *NSDI*, pages 469–483, 2015.

[8] Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, and George Candea. Efficient state merging in symbolic execution. *Acm Sigplan Notices*, 47(6):193–204, 2012.

[9] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. 2002.

[10] Haohui Mai, Ahmed Khurshid, Rachit Agarwal, Matthew Caesar, P Godfrey, and Samuel Talmadge King. Debugging the data plane with anteater. In *ACM SIGCOMM Computer Communication Review*, volume 41, pages 290–301. ACM, 2011.

[11] Paul Dan Marinescu and Cristian Cadar. Katch: high-coverage testing of software patches. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pages 235–245. ACM, 2013.

[12] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. Clickos and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 459–473. USENIX Association, 2014.

[13] Kenneth L McMillan. *Model checking*. John Wiley and Sons Ltd., 2003.

[14] Alok Sanghavi. What is formal verification? *EE Times_Asia*, 2010.

[15] Justine Sherry, Shaddi Hasan, Colin Scott, Arvind Krishnamurthy, Sylvia Ratnasamy, and Vyas Sekar. Making middleboxes someone else's problem: network processing as a cloud service. *ACM SIGCOMM Computer Communication Review*, 42(4):13–24, 2012.

[16] Matt Staats and Corina Păsăreanu. Parallel symbolic execution for structural test generation. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 183–194. ACM, 2010.

[17] Radu Stoenescu, Vladimir Olteanu, Matei Popovici, Mohamed Ahmed, Joao Martins, Roberto Bifulco, Filipe Manco, Felipe Huici, Georgios Smaragdakis, Mark Handley, et al. In-net: in-network processing for the masses. In *Proceedings of the Tenth European Conference on Computer Systems*, page 23. ACM, 2015.

[18] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. Symnet: Static checking for stateful networks. In *Proceedings of the 2013 workshop on Hot topics in middleboxes and network function virtualization*, pages 31–36. ACM, 2013.

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

[20] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D Ernst, Arvind Krishnamurthy, and Zachary Tatlock. Bagpipe: Verified bgp configuration checking. In *Proc. OOPSLA*, 2016.