Verifying networks with symbolic execution and temporal logic

Matei Popovici
University POLITEHNICA of Bucharest
matei.popovici@cs.pub.ro

Radu Stoenescu
University POLITEHNICA of Bucharest
radu.stoenescu@cs.pub.ro

Lorina Negreanu
University POLITEHNICA of Bucharest
lorina.negreanu@cs.pub.ro

Costin Raiciu
University POLITEHNICA of Bucharest
costin.raiciu@cs.pub.ro

ACM Reference format:
Matei Popovici, Radu Stoenescu, Lorina Negreanu, and Costin Raiciu. 2017. Verifying networks with symbolic execution and temporal logic. In *Proceedings of NetPl, , 2017 (NETPL17),* 2 pages.
DOI:

1 VERIFYING NETWORKS WITH SYMBOLIC EXECUTION

Symbolic execution is a promising approach to network verification [5, 6]. Inspired from software verification where it is mainly used to generate test-cases (e.g. [1]), *symbolic execution* is a technique for exploring all viable execution paths of a program. Symbolic execution runs programs with *symbolic inputs* instead of concrete ones. Such an input models *all possible values* in it's range. When executing conditional instructions, program execution is *branched*. In the case of an `if` statement, both the `then` and the `else` branches of the program will be explored, and the condition (resp. its complement) will be added as a *constraint* on each execution path. Adding constraints to a symbolic variable will restrict the values in its range. Constraints are added during branching as well as when executing other non-branching instructions (e.g. assignment). If constraints are unsatisfiable on a program branch, execution stops on that branch. The output of symbolic execution consists of all satisfied execution branches and for each branch — the set of constraints on each variable.

To deploy symbolic execution for verifying networks, the topology is interpreted as a single program whose input is a symbolic packet (i.e. a packet having possibly symbolic header fields). The execution paths of such a program correspond to the set of all possible paths the packet may take through the network.

Symnet [6] takes on this approach. Symnet is a *symbolic execution engine* which runs on SEFL (Symbolic Execution Friendly Language) programs. SEFL is a minimalist imperative language specifically designed for: (i) modelling network processing and (ii) fast symbolic execution. To verify a network topology, each of its components (and the topology itself) are translated to SEFL code. Symnet is fast and can check large-scale networks (e.g. the Stanford backbone) in seconds.

2 THE NEED FOR A POLICY LANGUAGE

Network verification is a powerful tool for checking reachability, invariance or the absence of loops. Such information is delivered by symbolic execution, but not necessarily in a human-readable format. For large-scale networks, manually inspecting the symbolic execution output is not viable. To fully benefit from symbolic execution, network administrators need a *policy language* in order to express network correctness properties. The language must: (i) allow for fast verification (i.e. a *single* pass through the symbolic execution output should be necessary), (ii) be *compositional*: more complicated policies should be expressed via combinations of simpler ones, (iii) be infrastructure independent, (iv) easy-to-use by admins via syntactic sugars or other visual tools.

A wide selection of policy languages which partly satisfy our constraints have been developed for SDN (Software-Defined Networking), more specifically, for OpenFlow-enabled networks. Policy specifications are written by administrators; subsequently, an (executable) OpenFlow configuration instance is generated and deployed. This configuration *guarantees* policy compliance.

Unlike the SDN approach, our goal is to verify *arbitrary* network topologies, which have already been instantiated. Other existing policy languages suffer from expressivity constraints mainly related to limited compositionality. For instance, in NetPlumber [4], a policy is of the form:

```
Q {f | f ~ filter} : {f ~ test}
```

where `Q` is a path quantifier (*on some path* or *on all paths*) while `filter` and `test` are flow expressions which can express constraints on a packet header as well as its location in the network.

NetPlumber cannot express *conditional path quantification* as found in policies such as:

(★) *All packets having destination IP 9.9.9.9, can eventually reach the Internet*

The tentative NetPlumber policy:

```
forall {f | f ~ destIP == 9.9.9.9} : {f ~ * Internet *}
```

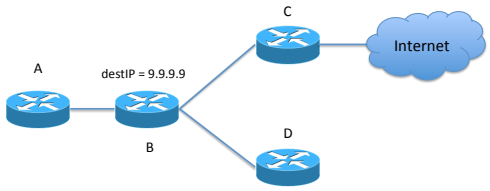
is actually more restrictive. Consider the topology from Figure 1. The policy (★) is satisfied in our example, however the NetPlumber policy is not: the paths which satisfy the filter constraint are: A-B-C and A-B-D, but on the latter path Internet is not reachable.

Finally, while there exist languages with more expressive power (e.g. FML [3]), they rely on a very tight coupling with the network model. The model and the policy need to be developed at the same time, thus making the whole verification effort accessible only to the expert modeller.

3 THE POLICY LANGUAGE NETCTL

We introduce NetCTL- an extension of CTL (Computation Tree Logic) [2] which also relies on SEFL in order to describe state-based properties. The main ingredients of NetCTL are: (i) *temporal*

NETPL17,
2017. . . . $15.00
DOI:



operators: **Future** (i.e. *at some hop on a network path*) and **Globally** (i.e. *at all hops on a network path*) and (ii) *path operators*: ∃ (i.e. *on some path*) and ∀ (i.e. *on all paths*). In CTL, each path operator must be directly preceded by a temporal operator. This restriction ensures that CTL verification can be achieved in *linear time* w.r.t. the size of the formula and that of the model. NetCTL naturally inherits the same property. NetCTL can naturally model reachability properties (e.g. *Internet is reachable*):

$$\exists\textbf{Future}\ \texttt{Internet}$$

as well as invariance properties such as *the destination IP header field is never changed*. For such a property, we first add a variable `v` to our network model, and insert the assignment `v = destIP` before symbolic execution. Finally, our NetCTL policy is:

$$\forall\textbf{Globally}\ (\texttt{destIP == v})$$

Note that `destIP == v` is a SEFL instruction which is used to express our state-(or hop-)based property. The property (★) introduced in the previous section is expressed as:

$$\forall\textbf{Globally}(\texttt{destIP == 9.9.9.9} \rightarrow (\exists\textbf{Future}\ \texttt{port == Internet}))$$

It can be intuitively read as: *on all paths, at each hop, if the destination IP is 9.9.9.9, there exists a path on which, eventually,* `port` *becomes Internet.*

We have successfully used NetCTL to specify a variety of operator policies such as: *end-to-end TCP connectivity*, *tunnel invariance*, arbitrary *path-dependent constraints*, *asymmetric connectivity* (A can initiate a TCP connection to B, but B can only respond), *isolation* (VLAN X can only be reached from machine Y).

4 NETCTL IMPLEMENTATION

NetCTL can be used as a verification procedure performed *after* symbolic execution, on it's output. This approach has the advantage of neatly separating symbolic execution from policy verification, however it is less efficient. Symbolic execution on the complete network model is costly and may be un-necessary for proving policy compliance or policy violation.

Consider a policy of the form $\exists\textbf{Future}\varphi$. It is sufficient to find a program branch satisfying $\textbf{Future}\varphi$ in order to validate the policy.

Starting from this observation, we have implemented an extension of the Symnet engine which performs *policy-driven symbolic execution*. The extension is efficient, and will only execute model components which are required to prove/disprove the policy at hand. The extension supports: (i) *instruction-level* and (ii) *topology-based* granularity. Under (i), we verify our state-based property after each SEFL instruction. This is useful for e.g. checking that a middlebox never touches a header field. Under (ii), we verify state-based properties after each port change in the network, which speeds up the verification process.

We have implemented a **verification algorithm** which performs a depth-first exploration of the network model.

Unlike standard CTL model checking, where the state space of the model has to be built in advance, during NetCTL verification, we build and explore states *on the fly*, by symbolically executing each instruction using Symnet. In effect, NetCTL models are *trees*, and it is sufficient to look at each Symnet state once. We exploit this when verifying a policy φ. On a *sequence of instructions*, we iteratively execute each subprogram from the sequence. If $\varphi \equiv \textbf{Future}\psi$, verification stops when ψ is true in the current state. Conversely, if $\varphi \equiv \textbf{Globally}\psi$ we stop (and report a violation) when ψ is false. If ψ is false (resp. true) in the current state, then $\textbf{Future}\psi$ (resp. $\textbf{Globally}\psi$) cannot be proved or disproved. Verification continues in the next state. The key observation is that we can eventually prove or disprove each (sub-)formula φ by looking at subsequent states only. On a branching instruction (e.g. `Fork` or `if`), we check each program branch and: (i) stop with success when $\varphi \equiv \exists\psi$ and ψ is true on some branch or (ii) stop with failure when $\varphi \equiv \forall\psi$ and ψ is false on some branch. Whenever the truth-value of a policy cannot (yet) be determined, verification continues. The algorithm maintains and updates the state of each sub-formula of our policy, which can be: *true*, *false* or *unknown*.

Open issues Policies such as TCP end-to-end connectivity from point A to B of a network require augmenting the network model at B, such that B behaves like a TCP responder. If B is the port of a middle box (instead of an *end-host*), then the middlebox model needs to be modified. Similarly, when verifying policies such as *no packet with a local source IP address can reach the internet*, a symbolic packet needs to be injected on all ports generating (instead of just forwarding) traffic. We are investigating automatic means for policy-dependent packet injection and model modifications. We are also testing our NetCTL verifier on small to medium-sized networks such as our CS department network. and also plan to deploy our verifier in order to check the correctness of the SEFL models which we currently generate.

REFERENCES

[1] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, Dec 8-10, 2008, California, USA, Proceedings*, pages 209–224, 2008.

[2] Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, Cambridge, MA, USA, 1999.

[3] Timothy L. Hinrichs, Natasha S. Gude, Martin Casado, John C. Mitchell, and Scott Shenker. Practical declarative network management. In *Proc. of the 1st ACM Workshop on Research on Enterprise Networking*, WREN '09, pages 1–10, 2009.

[4] Peyman Kazemian, Michael Chang, Hongyi Zeng, George Varghese, Nick McKeown, and Scott Whyte. Real time network policy checking using header space analysis. In *Proc. of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 99–112, 2013.

[5] Radu Stoenescu, Vladimir Andrei Olteanu, Matei Popovici, Mohamed Ahmed, João Martins, Roberto Bifulco, Filipe Manco, Felipe Huici, Georgios Smaragdakis, Mark Handley, and Costin Raiciu. In-net: in-network processing for the masses. In *Proceedings of the Tenth European Conference on Computer Systems, EuroSys 2015, Bordeaux, France, April 21-24, 2015*, pages 23:1–23:15, 2015.

[6] 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, Florianopolis, Brazil, August 22-26, 2016*, pages 314–327, 2016.