Formal Reasoning in Software-defined Networks

Mark Reitblatt
Cornell University
reitblatt@cs.cornell.edu

February 24, 2014

Abstract

I propose an approach to the verification of network systems using high-level languages. I reduce the problem of end-to-end verification to three different steps: verifying that the source program has the desired property, using a compiler and runtime to generate network configurations that are provably equivalent to the source program, and using abstractions that guarantee the preservation of properties when moving between different configurations. In this document, I outline work that has been carried out on the latter two steps, and detail a plan for building a tool to directly verify source programs written in a high-level language.

Introduction

Software-defined networking (SDN) is a clean-slate networking architecture that replaces the messy world of distributed network control protocols and proprietary interfaces with a centralized control abstraction built on an open protocol. Network programming languages (*e.g.* Frenetic [Foster et al., 2011], PANE [Ferguson et al., 2013], Maple [Voellmy et al., 2013], and NetKAT [Anderson et al., 2014]), in turn simplify the task of SDN programming by providing programmers with intuitive, high-level abstractions on top of the raw SDN control primitives. However, by interposing software layers between the programmer and the underlying hardware, these languages can complicate the task of reasoning about the end-to-end behavior of the actual system.

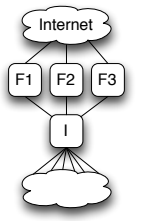
My thesis will show how to reason about SDN systems built with high-level network programming languages, and gain full behavioral guarantees about the resulting system. I reduce the problem of reasoning about network programs to 3 steps:

1. Verifying the source program has the desired property
2. Using a verified language compiler and runtime that provably converts source programs into equivalent network configurations
3. Using update abstractions that preserve invariants when transitioning between network configurations

In this document, I will describe my prior work on (2) and (3), and conclude by outlining a plan to build a verification tool for the NetKAT language, solving (1).

Example

To illustrate the challenges of end-to-end reasoning in SDN systems built with network programming languages, consider an example network with one ingress switch I and three "filtering" switches F_1, F_2, and F_3, each sitting between I and the rest of the Internet, as shown on the left side of Figure 1. Several classes of traffic are connected to I: untrustworthy packets from Unknown and Guest hosts, and trustworthy packets from Student and Faculty hosts. At all times, the network should enforce a security policy that denies SSH traffic from untrustworthy hosts, but allows all



Configuration I

	Type	Action
I	U, G	Forward F_1
	S	Forward F_2
	F	Forward F_3
F_1	*SSH*	Monitor
	$*$	Allow
F_2	$*$	Allow
F_3	$*$	Allow

→

Configuration II

	Type	Action
I	U	Forward F_1
	G	Forward F_2
	S, F	Forward F_3
F_1	*SSH*	Monitor
	$*$	Allow
F_2	*SSH*	Monitor
	$*$	Allow
F_3	$*$	Allow

Figure 1: Access control example.

other traffic to pass through the network unmodified. We assume that any of the filtering switches have the capability to perform the requisite monitoring, blocking, and forwarding.

There are several ways to implement this policy, and depending on the traffic load, one may be better than another. Suppose that initially we configure the switches as shown in the leftmost table in Figure 1: switch I sends traffic from U and G hosts to F_1, from S hosts to F_2, and from F hosts to F_3. Switch F_1 monitors (and denies) SSH packets and allows all other packets to pass through, while F_2 and F_3 simply let all packets pass through.

Now, suppose the load shifts, and we need more resources to monitor the untrustworthy traffic. We might reconfigure the network as shown in the table on the right of Figure 1, where the task of monitoring traffic from untrustworthy hosts is divided between F_1 and F_2, and all traffic from trustworthy hosts is forwarded to F_3.

Now, suppose that we have built an SDN program that implements this policy, exactly as outlined above. But, after traffic load shifts and the network is reconfigured, we get a security alert from an internal server that our access policy was violated and a guest host sent unauthorized traffic through the network.

If the program was written directly in OpenFlow (the leading SDN configuration protocol), the programmer could carefully inspect their program and, after great effort, discover the bug in the program[1]. But, if the program was written in a network programming language, the bug could come from any one of four places:

- The source network program
- The compiler incorrectly generating ruletables from the source program
- The runtime incorrectly installing ruletables in the network configuration
- The runtime incorrectly transitioning between network configurations

Only one of these four is directly under the control of the programmer. Worse, if the code the programmer actually writes is abstract and removed from the underlying implementation (as it should in a good language!), it can be difficult to connect the observed behaviors with the original program while debugging.

Verified Compilation

This section describes an approach that rules out the possibility of the compiler incorrectly generating ruletables, and the runtime incorrectly installing them in the network. By building a formal model of OpenFlow and the network programming language, we can build *verified* compilers and runtimes that provably correctly implement source programs in the network.

Concretely, I have developed a verified SDN runtime in the Coq proof assistant and proved it correct against a formal specification and a detailed operational model of OpenFlow. With this runtime, programmers specify the behavior of the network using the NetCore programming language [Monsanto et al., 2012], which abstracts away from the details of the underlying switch hardware and distributed system, and allows programmers to reason in terms of simple hop-by-hop packet-processing steps. The NetCore compiler and run-time system translates programs written in this language down to low-level packet-processing rules. Because its behavior is verified in Coq, we establish the correctness of our runtime once and for all, obviating the need for run-time or post hoc verification.

[1]This, of course, assumes that the erroneous behavior is not due to a bug in the hardware, or the operating system the program is running on.

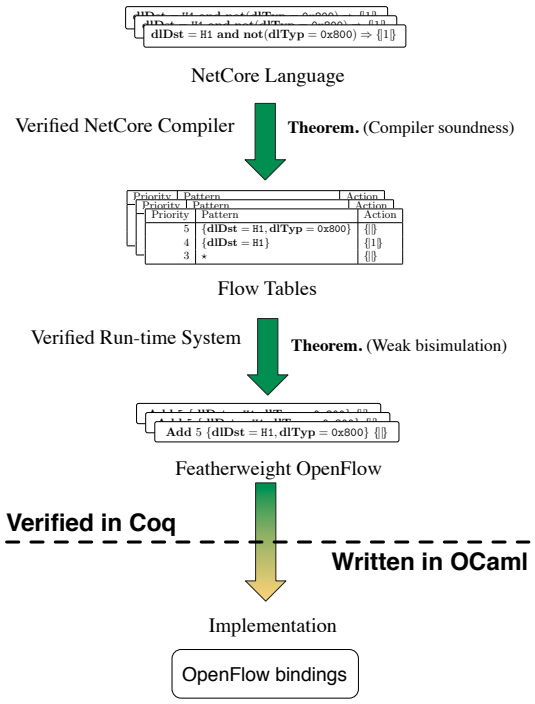


Figure 2: System architecture.

Architecturally, the system is organized as a verified software stack that translates through the following levels of abstraction, depicted in Figure 2:

- **NetCore.** The highest level of abstraction is the NetCore language, proposed in prior work by Monsanto *et al.* [Monsanto et al., 2012]. NetCore is a declarative language that allows programmers to describe *what* network behavior they want, without specifying *how* it should be implemented. It offers a collection of intuitive constructs for matching, filtering, and transforming packets, as well as natural logical operators for combining smaller programs into bigger ones such as union and domain restriction. Although NetCore programs are ultimately executed in a distributed system—the network—they have a simple semantics that models their behavior as functions from packets to packets.

- **Flow tables.** The intermediate level of abstraction is *flow tables*, a representation that sits between NetCore programs and switch-level configurations. There are two main differences between NetCore programs and flow tables. First, NetCore programs describe the forwarding behavior of a whole network, while flow tables describe the behavior of a single switch. Second, flow tables process packets using a linear scan through a list of prioritized rules. Hence, to translate operators such as union and negation, the NetCore compiler must generate a sequence of rules that encodes the same semantics. However, because flow table matching uses a lower-level packet representation (as nested frames of Ethernet, IP, TCP, etc. packets), flow tables must satisfy a well-formedness condition to rule out invalid patterns that are inconsistent with this representation.

- **Featherweight OpenFlow.** The lowest level of abstraction is *Featherweight OpenFlow*, a new foundational model we have designed that captures the essential features of SDNs. Featherweight OpenFlow models switches, the runtime, the network topology, as well as their internal transitions and interactions in a small-step operational semantics. This semantics is non-deterministic, modeling the asynchrony inherent in networks. To implement a flow table in a Featherweight OpenFlow network, the runtime instructs switches to install or uninstall rules as appropriate while dealing with two important issues: First, switches process instructions concurrently with packets flowing through the network, so it must ensure that at all times the rules installed on switches are

consistent with the flow table. Second, switches are allowed to buffer instructions and apply them in any order, so it must ensure that the behavior is correct no matter how instructions are reordered through careful use of synchronization primitives.

Using this Coq development, we prove two theorems:

Theorem 1 (Compiler Soundness) *For all NetCore programs pg, switches sw, $[\![Compile(sw, pg)]\!] = [\![pg]\!]\ sw$.*

This theorem states that the flowtable of the compiled NetCore program pg at an arbitrary switch sw has precisely the same denotation (semantics) as pg at that same switch. This is the strongest equivalence possible (equality) between a source program and its compiled form.

We also built a FeatherWeight OpenFlow runtime that takes flowtables and installs them into the network. The correctness theorem for this runtime is:

Theorem 2 (Runtime System Correctness) *The NetCore runtime is weakly bisimilar to the semantics of the source flowtable.*

A bisimulation between two systems says that they have identical traces, *i.e.* no external observer is able to distinguish between them by looking at their observable events. The "weakness" of the bisimulation is a technical detail, arising from the fact tha FeatherWeight OpenFlow models OpenFlow in great detail, including such elements as buffers. Weakly bisimilar systems are also indistinguishable by observable events.

Taken together, these two theorems state that when a NetCore program is compiled and run using this system, the resulting OpenFlow network behaves exactly like the original program.

This was joint work with Arjun Guha and Nate Foster, and is described in full detail in [Guha et al., 2013].

Network Transitions

The approach outlined in the previous section ensures that a single network program is correctly installed in the network. But what happens in a dynamic network, where policies change over time, and the network configuration must change with them?

Returning to the example, consider what happens when the switches are updated to the new configuration after re-balancing. Because we cannot update the network all at once, the individual switches need to be reconfigured one-by-one. However, if we are not careful, making incremental updates to the individual switches can lead to intermediate configurations that violate the intended security policy. For instance, if we start by updating F_2 to deny SSH traffic, we interfere with traffic sent by trustworthy hosts. If, on the other hand, we start by updating switch I to forward traffic according to the new configuration (sending U traffic to F_1, G traffic to F_2, and S and F traffic to F_3), then SSH packets from untrustworthy hosts will incorrectly be allowed to pass through the network. There is one valid transition plan:

1. Update I to forward S traffic to F_3, while continuing to forward U and G traffic to F_1 and F traffic to F_3.
2. Wait until in-flight packets have been processed by F_2.
3. Update F_2 to deny SSH packets.
4. Update I to forward G traffic to F_2, while continuing to forward U traffic to F_1 and S and F traffic to F_3.

In general, finding a transition plan that preserves a given property is difficult, and that assumes that the programmer provided the correctness property to the underlying runtime!

Instead, this section describes a technique (*per-packet consistent updates*) for guaranteeing that whenever the old and the new configurations satisfy a (correctness) property, the network satisfies the property during the transition.

Per-packet consistency guarantees that each packet flowing through the network will be processed according to a single network configuration—either the old configuration prior to the update, or the new one after the update, but not a mixture of the two. In the example in Figure 1, per-packet consistency rules out situations in which a packet is processed by the new configuration on ingress switch S and the old configuration on the filtering switch F_2, thereby circumventing the access control policy.

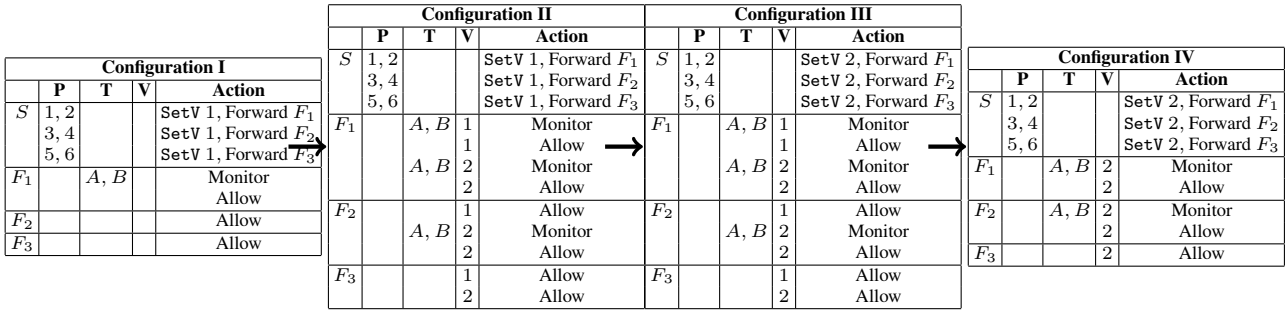


Figure 3: Configurations for per-packet consistent update example. The V column matches on version number.

To implement per-packet consistency, we propose a simple mechanism that stamps packets with their configuration version at ingress switches and tests for the version number in all other rules. This can be implemented in OpenFlow using a header field to encode version numbers (*e.g.*, VLAN tags or MPLS labels). To update to a new configuration, the runtime first pre-processes the rules in the new configuration, augmenting the pattern of each rule to match the new version number in the header. Next, it installs these rules on all of the switches, leaving the rules for the old configuration (whose rules match the previous version number) in place. At this point, every switch can process packets with either the old or new policy, depending on the version number on the packet.

The runtime then starts updating the ingress switches, replacing their old rules with new rules that stamp incoming packets with the new version number. Because the ingress switches cannot all be updated atomically, packets entering the network are processed with a mixture of the old and new policies for a time, but *any individual packet* is handled by just one policy throughout the network. Finally, once all packets following the "old" policy have left the network, the runtime deletes the old configuration rules from all switches, completing the update.[2] Figure 3 shows the intermediate configurations generated by this approach. Note that in going from one configuration to the next, the individual switches can be updated in any order.

We argued that a per-packet consistent update would enforce the access control policy in our example, but, in general, how can developers know if their invariants will be preserved? Formally, per-packet consistent updates preserve all path properties. A *path property* captures behaviors that can be expressed in terms of packets p and the list l of links the packet traverses as it is forwarded through the network. Many useful properties can be expressed as path properties including basic connectivity, loop-freeness, and security properties such as "all packets from host h must be dropped" or "all Web traffic must waypoint via middlebox m." When using a per-packet consistent update, the programmer is guaranteed that if a path property P is true of all valid packet-path pairs in both the initial and final configurations, it will be true of all paths taken by any packet at run time before, during, and after the update. In other words, per-packet consistent update *preserves all path properties*.

With this property preservation, a programmer can be sure that any errors seen in the network during an update do not arise from the update itself. The work described in this section was joint work with Nate Foster, Jennifer Rexford, Cole Schlesinger, and David Walker, and is described in full detail in [Reitblatt et al., 2011].

NetKAT Verifier

Taken together, the techniques of the previous sections ensure that the network correctly implements the source network program. Therefore, any bugs in the network must reflect bugs in the actual program. This section outlines a proposal to build a network programming language verification tool that can decide whether source programs satisfy a given specification.

[2]The runtime can safely delete the old rules after some maximum transmission delay (*i.e.*, the sum of propagation and queuing delay, maximized over all paths) has elapsed. Since our mechanisms never introduce loops, we do not need to account for increases in delay due to transient forwarding loops. In practice, the runtime can be quite conservative in estimating the delays and simply wait for several seconds (or even minutes) before removing the old rules.

Concretely, I propose to build a verification engine for the NetKAT programming language ([Anderson et al., 2014]). This tool will take a source program, written in NetKAT, and a correctness specification, written in the Pathetic language[3], and decide if the source program satisfies the specification.

The precise semantics for both languages can be found in their respective papers, but they are both based upon regular expressions, and have a denotational semantics given by a function from packets to paths traversing the network. Using this denotational semantics, one can view a Pathetic formula ϕ as specifying the legal set of paths for a packet to take across the network, and a NetKAT program P as assigning packets to specific paths. Thus, verifying that P satisfies ϕ amounts to checking that every path in P is "legal", *i.e.* in ϕ.

Because the language have regular expression based interpretations, we can use the fact that any regular expression can be represented by an automaton to perform verification. Viewed this way, an automaton, corresponding to a specific NetKAT program P or Pathetic formula ϕ, accepts as a language precisely the paths given by the denotation of P or ϕ. Therefore, checking that every path in P is in ϕ (*i.e.* that P satisfies specification ϕ) is the same as checking that the language of the automaton A_P (the automaton representing P) is a subset of the language of the automaton A_ϕ (the automaton representing ϕ).

Conveniently, the NetKAT compiler already needs to construct an automaton A_P for the source program P as part of its internal stages. The plan is to use this automata representation to perform model checking. Concretely, to verify that program P has property ϕ, the verifier will first negate ϕ and construct an automaton $A_{\neg\phi}$ equivalent to $\neg\phi$. It will then construct the product automaton $A_P \times A_{\neg\phi}$ of A_P and $A_{\neg\phi}$, and check that the language of this automaton is empty. Recalling automata theory, the language of the product automaton of A_P and $A_{\neg\phi}$ is the intersection of the languages of A_p and $A_{\neg\phi}$. Therefore, if the language of $A_P \times A_{\neg\phi}$ is empty, it means that every path in P is not in $\neg\phi$, *i.e.* every path in P *is* in ϕ (*i.e.* is legal).

This general approach of automata-theoretic verification was introduced by [Vardi and Wolper, 1986], and has been tremendously successful in domains ranging from concurrent systems to low-level systems code to hardware designs to automotive control software.

References

[Anderson et al., 2014] Anderson, C. J., Foster, N., Guha, A., Jeannin, J.-B., Kozen, D., Schlesinger, C., and Walker, D. (2014). NetKAT: Semantic foundations for networks. In *POPL*.

[Ferguson et al., 2013] Ferguson, A. D., Guha, A., Liang, C., Fonseca, R., and Krishnamurthi, S. (2013). Participatory Networking: An API for application control of SDNs. In *SIGCOMM*.

[Foster et al., 2011] Foster, N., Harrison, R., Freedman, M. J., Monsanto, C., Rexford, J., Story, A., and Walker, D. (2011). Frenetic: A network programming language. In *ICFP*.

[Guha et al., 2013] Guha, A., Reitblatt, M., and Foster, N. (2013). Machine-Verified Network Controllers . In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), Seattle, WA*.

[Monsanto et al., 2012] Monsanto, C., Foster, N., Harrison, R., and Walker, D. (2012). A compiler and run-time system for network programming languages. In *POPL*.

[Reitblatt et al., 2013] Reitblatt, M., Canini, M., Foster, N., and Guha, A. (2013). Fattire: Declarative fault tolerance for software defined networks. In *ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (HotSDN), Hong Kong, HK*.

[Reitblatt et al., 2011] Reitblatt, M., Foster, N., Rexford, J., and Walker, D. (2011). Consistent updates for software-defined networks: Change you can believe in! In *HotNets*.

[Vardi and Wolper, 1986] Vardi, M. Y. and Wolper, P. (1986). An automata-theoretic approach to automatic program verification. In *Proceedings of the First Symposium on Logic in Computer Science*. IEEE Computer Society.

[3]Pathetic is exactly the FatTire language from [Reitblatt et al., 2013], minus the fault tolerance annotations

[Voellmy et al., 2013] Voellmy, A., Wang, J., Yang, Y. R., Ford, B., and Hudak, P. (2013). Maple: Simplifying SDN programming using algorithmic policies. In *SIGCOMM*.