Computational Verification of Network Programs in Coq

Gordon Stewart [⋆]

Princeton University

Abstract. We report on the design of the first fully automatic, machine-checked tool suite for verification of high-level network programs. The tool suite targets programs written in NetCore, a new declarative network programming language. Our work builds on a recent effort by Guha, Reitblatt, and Foster to build a machine-verified compiler from NetCore to OpenFlow, a new protocol for software-defined networking.

1 Introduction

The past few years have witnessed a groundswell of interest in *software-defined networks (SDNs)*, as evidenced by the popularity of new standards for programmable networking such as OpenFlow [9]. In an SDN, rules for packet processing still live on network switches with dedicated hardware, as in traditional networks (data plane). But unlike in traditional networks, SDNs allow decisions about when and how to update network policies in response to network events (control plane) to be handled by a dedicated *controller* program running on one or more general-purpose computers. The controller machine(s) and switches interoperate via an open standard, such as OpenFlow, that allows on-the-fly switch re-programming via special *configuration* or *control* messages.

Several recent research efforts have capitalized on the modular structure of SDNs to build new high-level programming languages for networks, such as Nettle [13], Frenetic [2], and NetCore [10]. These high-level languages, which are typically compiled to low-level OpenFlow forwarding rules, are characterized by a focus on *declarative* and *modular* programming of network policies: The programmer defines *what* a particular policy is, not *how* it is implemented; and programs are constructed by composing small, reusable components. These two features of the new breed of network languages make them an ideal target for program verification. Yet there have been few, if any, efforts to-date to build verification tools for network programs written in these languages.

As an initial foray, this paper presents the first machine-certified toolset for verifying network programs written in a high-level network programming language. We build on recent work by Monsanto *et al.* [10], which defined the syntax and semantics of the network programming language NetCore, and on work by Guha, Reitblatt, and Foster [3], which presented a Coq formalization of NetCore and of a lightweight version of the OpenFlow protocol called Featherweight

[⋆] Supported in part by the National Science Foundation under grant CNS-0910448.

OpenFlow. Drawing on the NetCore compilation algorithm of Monsanto *et al.*, Guha *et al.* formalized these models in Coq in order to verify the correctness of a compiler and runtime for NetCore targeting Featherweight OpenFlow. The result of their work was a fully machine-verified network programming platform that targeted actual switch hardware.

In this paper we start where Guha *et al.* left off, by building a suite of tools for verifying properties of the NetCore programs that are the input to their verified compiler. For many concrete network specifications—for example, reachability or security specifications targeting a particular network topology with known port identifiers—our verification tools are fully automatic: In order to prove the specification $\{P\}\ pg\ \{Q\}$ of NetCore program pg, we calculate the weakest precondition $wp(pg, Q)$ of Q given pg. Then we verify that P implies $wp(pg, Q)$ by checking the implication $P \implies wp(pg, Q)$ in a special-purpose resolution theorem prover coded in Gallina, the functional programming language embedded within Coq. Because all of our tools are proved sound in Coq with respect to an extension of the NetCore semantics presented by Guha *et al.* [3], we end up with a fully automatic verification toolset that when connected to Guha *et al.*'s verified compiler will provide strong guarantees on the correctness of generated OpenFlow programs. To demonstrate our tool suite, we use it to verify a multiplexing network address translation module (§5).

Contributions. The novel contributions of this paper are the following.

1. We develop the first suite of machine-checked tools for verifying correctness and security properties of network programs written in a high-level programming language (NetCore) targeting an open SDN platform (OpenFlow).
2. To fully automate proofs of NetCore specifications within our system, we develop (§3) two related weakest precondition calculi for NetCore, and build a special-purpose resolution theorem prover, in Coq's Gallina language, for checking entailments of NetCore program specifications. We prove the resolution prover sound in Coq, and the weakest precondition calculi both sound and complete, with respect to the NetCore semantics.
3. Because our tool suite targets an extension of the NetCore semantics of Guha *et al.* [3], it can be connected to Guha *et al.*'s verified NetCore compiler to provide strong guarantees on generated OpenFlow programs. However, we have not yet fully integrated the Coq proofs of our NetCore verifier with those of Guha *et al.*'s NetCore compiler due to engineering concerns.
4. We use the tool suite to verify correctness and security properties of a network address translation module (§5). Section 8 describes additional applications.

An alternative to checking entailments within Gallina using a custom theorem prover is to send verification conditions to an external first-order prover or SMT solver, and then to check proof certificates *post hoc*. We prefer the Gallina approach for two reasons. First, implementing the entailment checker in Gallina means we can prove it sound once and for all. In the certificate checking approach, the potential for soundness bugs in an external tool means that program verification may fail unnecessarily when a bad certificate is detected. Second, building a custom entailment checker means that we can apply domain-specific reasoning in ways that may not be directly exploited by an external tool. The final few paragraphs of Section 3 provide one such example, in which we exploit Coq's theory of inductive datatypes in order to reason constructively by inversion, without explicit first-order inversion laws. There are of course disadvantages to using a special-purpose entailment checker as well: our tool is necessarily much less sophisticated than state-of-the-art provers such as Z3 [1]. However, it is still sufficient to discharge the verification conditions that arise when verifying a range of network programs, as later sections of this paper will demonstrate.

The Coq Development. This paper is closely tied to a mechanized proof development in Coq, which can be downloaded from the address below.[1] In code listings, line numbers refer to the corresponding files in the mechanized development.

2 Software-defined Networking

In a traditional network, control logic is distributed among a number of physically distinct routers and switches, each with its own *flow table*. The flow tables define how packets are processed and forwarded through the router or switch, and are typically implemented using dedicated hardware such as ternary content addressable memories (TCAMs), in order to process packets at line rate.

Software-defined networks differ from traditional networks by splitting the control plane from the data plane, providing logically centralized control in the form of a general-purpose *controller*. The controller, which is connected to the switches over a secure, special-purpose link, decides when and how to update the flow rules installed on the switches in response to network packets and other network events.

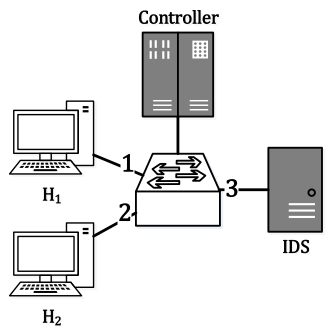


Fig. 1. A software-defined network with a single switch (center), two endhosts (H_1, H_2), a middlebox that performs intrusion detection (*IDS*), and a general-purpose controller machine.

High-level network programming languages such as NetCore build another layer of abstraction above software-defined networking platforms such as OpenFlow. Network programs in NetCore are built from small, reusable packet-processing actions that are composed both in sequence—in order to apply multiple modifications, in order, to a single packet—and in parallel—in order to apply multiple forwarding or modification rules to multiple copies of a single packet. In order to scope the applicability of a network program constructed in

[1] `http://www.cs.princeton.edu/~jsseven/papers/netcorewp`

this manner, NetCore programs can be restricted by *predicates* on the location and header fields of network packets. For example, it is good style in NetCore to write, *e.g.*, a forwarding policy for HTTP traffic first as if it applied to all packets, and then to restrict the resulting policy just to those TCP packets that have port $\mathbf{tcpDst} = 80$.

Example. To make this concrete, consider the simple network topology depicted in Figure 1. The network, which is adapted from Guha *et al.* [3], comprises a single switch S_1 connected to two endhosts H_1 and H_2, a middlebox *IDS* that performs intrusion detection, and a controller. The endhosts are connected to the switch on ports 1 and 2 respectively, while *IDS* is connected on port 3. The controller machine is connected to the switch via a special-purpose link.

Now imagine we want to impose the following traffic policy, also adapted from Guha *et al.* [3]: HTTP traffic on TCP port 80 gets forwarded to *IDS* (as well as to its original destination), SSH traffic is dropped, and all other traffic gets forwarded to the appropriate endhost. In an SDN platform like OpenFlow, this policy would be defined as a set of *flow tables*, or packet forwarding rules—one set of rules for each switch. For example, the following rule expressed in Guha *et al.*'s notation causes SSH traffic to be dropped: $\mathbf{Add}\ 10\ \{\mathbf{tcpDst} = 22\}\ \{\!|\ |\!\}$. The 10 after **Add** is the rule's priority. On the switch, any rules with lower priority will be applied only if this rule fails to fire. The expression $\mathbf{tcpDst} = 22$ is a *pattern*: it limits the applicability of the rule to packets with TCP destination port equal to 22. The expression $\{\!|\ |\!\}$ denotes the empty multiset of ports. It specifies that packets matching the pattern should be forwarded along *no* ports (that is, they should be dropped). After configuring this rule on the switch, an OpenFlow implementation of the high-level policy would add lower-priority rules for forwarding HTTP traffic to *IDS* and to the appropriate endhost, in addition to even lower priority rules for forwarding the remaining traffic.

NetCore. While SDN platforms such as OpenFlow give programmers a great deal of flexibility when configuring networks, writing network controllers in OpenFlow is still quite painful. In addition to the actual forwarding logic of the network application, the programmer must keep track of numerous low-level details such as dependencies between rule priorities. He also must determine (manually) on which switch to install each forwarding rule. In essence, the programmer is writing a low-level distributed program by hand. This can become quite a difficult task, especially as networks scale to tens or hundreds of switches.

High-level SDN programming languages such as NetCore mitigate many of these challenges by providing a programming model that is at once more declarative and more modular than those provided by traditional SDN platforms. NetCore, for example, provides as one of its key abstractions the notion of *whole-network programmability*: instead of defining forwarding rules for particular switches, a NetCore program defines the behavior of the entire network all at once. The NetCore compiler and runtime system determine on which switch to install each rule, and with what priority.

```
(* basic actions *)
Inductive action :=
| Id: action
| UpdIpSrc: Word32.t → action | UpdTcpSrc: Word16.t → action
| UpdIpDst: Word32.t → action | UpdTcpDst: Word16.t → action
| Fwd: Word16.t → action | Drop: action.
(* selected packet patterns *)
Inductive ppat :=
| DlSrc: Word48.t → ppat (*MAC src*)
| DlDst: Word48.t → ppat (*MAC dst*)
| TpSrc: Word16.t → ppat (*TCP src*)
| TpDst: Word16.t → ppat (*TCP dst*)
(* atomic predicates *)
Inductive atom :=
| Wild: atom | Location: lpat → atom | Packet: ppat → atom.
(* Boolean predicates *)
Inductive pred :=
| Atom: atom → pred
| And: pred → pred → pred | Or: pred → pred → pred
| Not: pred → pred.
(* NetCore programs *)
Inductive prog :=
| Act: pred → action → prog | Restrict: prog → pred → prog
| Par: prog → prog → prog | Seq: prog → prog → prog.
```

Listing 1. Excerpts from the syntax of NetCore (`src/NetCoreSyntax.v`)

To illustrate, consider the following implementation in NetCore, the syntax of which is given in Listing 1, of the high-level policy for the network in Figure 1.[2] First, we define the rules that establish point-to-point connectivity in the network.

```
Definition pg1 := DLDST=H1 ⇒ FWD 1.
```

For example, program `pg1` defines a basic guarded command in NetCore that forwards to port 1 (`FWD 1`) any packets satisfying the predicate `DLDST=H1`, that is, with destination MAC address equal to `H1`. Here `DLDST=H1` is syntactic sugar for the atomic predicate `Atom (Packet (DlDst (Val H1)))` (*cf.* Listing 1).

In NetCore, we can compose this first program with a second program that defines the routing policy for host 2 as follows.

```
Definition pg2 := pg1 'PAR' DLDST=H2 ⇒ FWD 2.
```

The combinator `'PAR'`, which is infix notation for the `Par` constructor of Listing 1, defines the parallel composition of two NetCore programs. Semantically, it duplicates its input packets, applying the program on the left (`pg1`) to one of the duplicated input packets and the program on the right (`DLDST=H2 ⇒ FWD 2`)

[2] The code that follows can be found in file `src/examples/Guha.v` in the code distribution that accompanies this paper.

to the other. The result is a set of packets that will be transferred across links and further processed by other switches in the network, if any.

The resulting program pg2 can be further composed with the routing policy for the intrusion detection system, resulting in the following program.

```
Definition pg3 := pg2 'PAR' TPDST=80 ⇒ FWD 3.
```

Program pg3 forwards HTTP packets (TPDST=80) to the IDS middlebox on port 3, packets destined for MAC address H1 to host 1, and those destined for MAC address H2 to host 2.

Finally, in order to satisfy the high-level policy described above we need to ensure that SSH traffic on port 22 is dropped. In NetCore, this is accomplished by restricting program pg3 by a predicate that scopes the resulting program. Packets that do not satisfy the predicate are implicitly dropped.

```
Definition routing := RESTRICT pg3 BY (NOT (TPDST=22)).
```

Here, RESTRICT pg3 BY (NOT (TPDST=22)) is syntactic sugar for an application of the Restrict constructor of Listing 1. This has the effect of applying pg3 to any packet satisfying the predicate NOT (TPDST=22) (that is, with TCP destination port not equal to 22) and dropping all other packets (*i.e.*, those on port 22), which is the behavior we intended.

3 Verification

Now that we have defined the routing policy for the network topology in Figure 1, we can begin proving properties of the resulting network. For example, imagine we would like to prove that the routing network defined in Section 2 *actually does* drop all SSH traffic. For this particular network, the property is of course trivial: the network program is guarded by a RESTRICT that filters packets satisfying exactly this predicate! However, for more complicated network programs, security properties such as this one can be significantly less obvious. In any case, it will be instructive to present our verification methodology in the context of this simple example; we consider more interesting networks and verification problems in Sections 5 and 8.

In order to state the theorem described informally above, we first briefly describe the semantics of NetCore programs. In our Coq development, NetCore programs are interpreted as inductively defined relations on *located packets*, where a located packet is a pair of a packet, including its header fields and payload, and a location, which is a pair of a switch identifier and a port number.[3]

```
Inductive progInterp: prog → lp → lp → Prop :=
(* ··· *)
| InterpUpdSrcIp: ∀x x' ip cond,
    (predInterp cond x)=true →
    upd_ip_src x ip = Some x' →
```

[3] The semantics of NetCore is defined in file src/NetCoreSemantics.v.

```
    progInterp (Act cond (UpdIpSrc ip)) x x'
(* ··· *)
```

For example, the InterpUpdSrcIp constructor of the relation states that packet x is related to packet x' by program Act cond (UpdIpSrc ip)), which in sugared form is cond => UpdIpSrc ip, if (1) the predicate cond is satisfied by x (predInterp cond x=true) and (2) updating the IP address of packet x to ip succeeds, resulting in x' (our semantics must handle situations in which x is not a valid IP packet, in which case the upd_ip_src operation will fail).

A bit more formally now, the security property we would like to prove is: for all packets x, if (predInterp (TPDST=22) x)=true then progInterp routing x x' is false. That is, no input packet with TCP destination port equal to 22 is ever routed as output packet x'. We could state (and prove) this theorem directly, but instead we will encapsulate the general kind of specification as a Hoare triple, with the following definition.

```
Definition triple (P: pred) (pg: prog) (Q: pred) :=
  ∀x y, (predInterp P x)=true →
  progInterp pg x y →
  (predInterp Q y)=true.
```

That is, a program pg satisfies triple P pg Q when it takes packets x satisfying precondition P (predInterp P x=true) to packets satisfying postcondition Q (predInterp Q y=true).

Now, with the help of some syntactic sugar for triple P pg Q, we can restate the theorem as follows. Using our NetCore tool suite, the proof is a single line.

```
Lemma ssh_traffic_dropped: |- [TPDST=22] routing [NOT WILD].
Proof. Time checker. (*0. secs (0.0156001u,0.s)*) Qed.
```

Here NOT WILD is the representation of False in the NetCore predicate language. Thus |- [TPDST=22] routing [NOT WILD] states that packets satisfying TPDST =22 are never routed (*i.e.*, they are always dropped).

To prove this theorem, one could reason from the definitions of the triple |- [P] pg [Q] and of the interpretation relation progInterp, perhaps proving a few general-purpose Hoare rules along the way. Indeed, this would be the conventional way to proceed in an interactive proof assistant such as Coq. However, we would like to automate this proof, and others like it. In general, we will avoid making use of the semantic meaning of the Hoare triple defined above whenever possible, instead relying on the computational verification procedure given in Listing 2.[4]

The function check takes as arguments a bound n on the number of iterations of the procedure, a background theory th, the program pg to be verified, and its specification spec. The main steps of the procedure are the following.

1. Calculate wp pg Q, the weakest precondition of the postcondition Q with respect to program pg. By soundness of the weakest precondition calculus,

[4] The code that follows is found in file `src/Checker.v` in our source distribution.

```
Definition check (n: nat) (th: pred) (pg: prog) (spec: pred*pred) :=
  let P := fst spec in
  let Q := snd spec in
  let vc := th 'AND' P 'AND' (NOT (wp pg Q)) in
    go n nil (preprocess (clausify (normalize n vc) nil nil) nil).
```

Listing 2. Top-level Verification Procedure

`|- [wp pg Q] pg [Q]`. Thus by the rule of consequence for Hoare triples, `|- [P] pg [Q]` if P $\implies$ wp(pg, Q).

2. Prove that P $\implies$ wp pg Q. This entails: Encoding the *negation*[5] of the implication P $\implies$ wp pg Q as a formula in clausal normal form; Simplifying the resulting formula by removing tautological and subsumed conjuncts; and Proving that the resulting simplified formula is unsatisfiable. In the code in Listing 2, these steps correspond to the calculation of vc and the call to go, the top-level loop of the resolution prover. In the definition of vc, the negation of P $\implies$ wp pg Q is implicitly simplified to And P (Not (wp pg Q)).

Weakest Preconditions. Of these two steps, the calculation of the weakest precondition of Q with respect to program pg is the most straightforward. Because NetCore contains no looping constructs, and therefore no loop invariants are required, we can calculate wp pg Q using the recursive function defined in Listing 3.

```
Fixpoint wp (pg: prog) (R: pred): pred :=
  match pg with
    | Act cond Id => cond ⟹ R
    | Act cond (Fwd pt) => cond ⟹ subst_port pt R
    | Act cond (UpdIpSrc ip) =>
        cond ⟹ Atom (Packet IsIp) ⟹ subst_ip_src ip R
    (* ··· *)
    | Act cond Drop => Atom Wild
    | Restrict pg' cond => cond ⟹ wp pg' R
    | Par pg1 pg2 => wp pg1 R 'AND' wp pg2 R
    | Seq pg1 pg2 => wp pg1 (wp pg2 R)
  end.
```

Listing 3. Weakest Precondition Calculus for NetCore (excerpt)

For example, the weakest precondition of postcondition R and the the guarded identity action Act cond Id (in sugared form cond $\Rightarrow$ Id) is just R, under the assumption that cond evaluates to true (cond $\implies$ R). Likewise, the weakest precondition of the parallel composition of two programs pg1 and pg2 (Par pg1 pg2) is just the conjunction of the weakest preconditions of the component programs (wp pg1 R 'AND' wp pg2 R), while the weakest precondition of the sequential composition of pg1 with pg2 is the weakest precondition of pg1 given postcondition wp pg2 R.

[5] Although this procedure follows the usual proof-by-contradiction approach of automated tools for propositional and first-order logic, it can be done without classical axioms in Coq because the language of NetCore predicates is decidable.

The weakest preconditions of commands that update packet headers or locations (Fwd, UpdIpSrc, UpdTcpSrc) are calculated as one would calculate the weakest precondition of an assignment statement in a typical imperative language. That is, the weakest precondition of $x := e$ for R is $R[e/x]$. However, instead of substituting an expression e for variable x in R, we substitute *true* for occurrences of location or packet predicates that are consistent with a packet modification, and *false* for any such atomic predicates that are inconsistent. For example, the following code excerpt (file src/WP.v) performs the substitution that is required for forwarding actions.

```
Fixpoint subst_port (x: Word16.t) (p: pred) {struct p} :=
  match p with
    | Atom Wild => Atom Wild
    | Atom (Location (Switch _)) => p
    | Atom (Location (Port y)) =>
        if Word16.eq x y then WILD else NOT WILD
    | Atom (Packet _) => p
    (* ··· *)
  end.
```

In our Coq development, we have proved that wp as defined above is both sound and complete.

```
Lemma wp_sound: ∀pg R, |- [wp pg R] pg [R].
```

```
Lemma wp_complete:
  ∀P pg Q,
  |- [P] pg [Q] →
  ∀l, (predInterp P l)=true → (predInterp (wp pg Q) l)=true.
```

The proof of soundness is straightforward by induction on the program pg. Completeness requires that the language of predicates be full-featured enough to express equality on located packets. That is, we must define a predicate Eq x such that predInterp (Eq x) x' if, and only if, x=x'. The equality predicate is used in the Seq case of the proof to constrain intermediate packets.

Resolution. After we have calculated the weakest precondition wp pg Q of postcondition Q and program pg, we next must check that P entails wp pg Q (in fact, this implication *must* be provable in order for |- [P] pg [Q] to hold since wp is proved complete). Here we resort to *resolution* [12], a standard method from automated theorem proving, in order to check the implication automatically within Gallina.

To do so, we first encode the negation of the implication as a set of *clauses*, or disjunctions of logical literals. Literals are, in turn, either positive or negative (*i.e.*, negated) atomic predicates.

```
Inductive lit := Neg: atom → lit | Pos: atom → lit.
```

Clauses are defined in the code as lists of literals, and are interpreted as the following disjunction of their elements.

```
Definition clauseInterp (cl: clause) (l: lp) :=
  foldInterp (fun p => litInterp p l) orb false cl.
```

Here the function foldInterp folds an interpretation function (litInterp) and a combinator (orb) over the constituent elements of the list (cl), with unit false.

Encoding a formula as a set of clauses entails: (1) Converting the formula to *negation normal form (NNF)*, by moving negations inwards using De Morgan equalities; (2) Distributing disjunctions over conjunctions; and (3) Rewriting the resulting formula as a set of (implicitly conjoined) clauses. Once we have encoded the negation of the initial implication $\mathsf{P} \implies \mathsf{wp\ pg\ Q}$, together with the background theory th, as a set of clauses (its so-called *clausal normal form*) we simplify the resulting clause set to remove tautologous and otherwise redundant clauses, then begin searching for a contradiction by iterating the following procedure.

```
Definition step (act pas: list clause): result :=
  match pas with
    | nil => if invert act then Unsat else Sat act
    | nil :: pas' => Unsat
    | given :: pas' =>
        let act' := given::act in
        let resolvents := map condense (resolve given act' nil) in
        let resolvents' := filter (negb ∘ subsumedBy pas) resolvents in
          Later act' (pas' ++ resolvents')
  end.
```

The step function implements a variation of what is known as the *given clause algorithm* for saturating a search space by resolution, which was popularized by the OTTER theorem prover [8]. It operates on two sets of clauses: act, the set of active or usable clauses, and pas, the set of passive clauses that have not yet taken part in resolution inferences. Initially, all clauses are in pas.

At each iteration of step, we do a case analysis on pas, resulting in a three-way branch: Either (1) pas is empty, in which case the search space is saturated (traditional first-order provers would return Sat at this point); or (2) the head of pas is the "always false" clause nil[6] (pas = nil :: pas'); or (3) the given clause at the head of pas (pas = given :: pas') contains at least one literal.

Case (3) is the most interesting. Here we add given to act, resulting in the new clause set act', then attempt to resolve given with each clause in act' (resolve given act' nil), including itself. The resulting set of resolvents is then *condensed* to remove unnecessary duplicate literals (map condense ⋯). Finally, newly resolved clauses that are subsumed by clauses already in pas are filtered away as redundant (resolvents' := ⋯) and the resulting set is appended to pas'. The Later constructor is used to communicate the updated clause set to the main loop of the prover, which is not shown in the code above.

In case (2), pas contains the nil, or always false, clause. Thus the prover immediately returns Unsat: nil is unsatisfiable.

[6] Recall that clauses are interpreted as the disjunction of their component literals, with unit false; thus the nil clause is never satisfiable.

In case (1), a traditional resolution prover would return Sat: Because resolution is refutation complete, the procedure is guaranteed to derive the nil clause when given an unsatisfiable initial clause set. But since all clauses have been processed (pas is empty) without the nil clause being discovered, it must be the case that the initial input clause set has a model.

We could stop here. Indeed, standard resolution provers would stop at this point. Instead, we use the fact that we are constructing a custom prover to build in an additional level of inference by inversion on inductive types (invert act).

To see why this is useful, consider a clause set act that contains a pair of singleton clauses asserting TPSRC=22 and TPSRC=80 respectively. Both of these assertions cannot be true simultaneously. Yet a traditional first-order prover would not be able to derive a contradiction at this point; the standard inversion principles we get when reasoning about the inductive packet and nat types in Coq must first be explicitly added to the prover's background theory. This can be done. For example, in this particular case, we can safely add the clause

```
Neg (TPSRC=22) :: Neg (TPSRC=80) :: nil
```

which asserts that TPSRC=22 and TPSRC=80 cannot be true simultaneously. However, a set of more general inversion principles would clutter the search space with many (usually unnecessary) clauses. It is quite convenient, instead, to be able to do a domain-specific check, at the point at which all other first-order inferences have been exhausted, for clauses that are incompatible by inversion.

4 Reachability

In Section 3, we described a general methodology for proving theorems of the form

```
Lemma ssh_traffic_dropped: |- [TPDST=22] routing [NOT WILD].
```

in which the triple |- [P] pg [Q] made a claim about *all* packets that may result after routing packets satisfying P but did not ensure that at least one such packet *existed.* This form of Hoare triple was useful for writing specifications of security properties such as "all packets with TCP destination port equal to 22 are dropped." We will see in Section 5 that this triple is also useful for specifying the security properties of a network address translation module.

However, it is also quite useful when describing high-level properties of a network to be able to specify *reachability*, in addition to security properties. That is, we would like to be able to prove that, given a packet x satisfying some predicate P, *there exists* a second packet y such that y satisfies the predicate Q. Furthermore, it should be the case that progInterp pg x y for the NetCore program pg in question, *i.e.*, x is actually routed to y by pg. For example, if P is specialized to PORT=1 and Q is specialized PORT=2, then a reachability specification for P and Q states that a host located on port 2 is reachable by a host on port 1.

In order to specify and prove reachability queries of this form, we have adapted the weakest precondition calculus of Section 3 to the following variation of the Hoare triple of that section.

```
Definition triple' (P: pred) (pg: prog) (Q: pred) :=
  ∀x, (predInterp P x)=true →
  ∃y, progInterp pg x y ∧ (predInterp Q y)=true.
```

This Hoare triple states that there *exists* a y for which progInterp pg x y holds, and such that predInterp Q y evaluates to true. In what follows, we will use the notation |-r [P] pg [Q] to denote reachability specifications of this form.

Adapting the weakest precondition calculus of Section 3 to reachability specifications is reasonably straightforward. For example, here are the weakest precondition rules for Restrict, Par, and Seq.

```
Fixpoint wp' (pg: prog) (R: pred) :=
  match pg with
    (* ··· *)
    | Restrict pg' cond => cond 'AND' wp' pg' R
    | Par pg1 pg2 => wp' pg1 R 'OR' wp' pg2 R
    | Seq pg1 pg2 => wp' pg1 (wp' pg2 R)
  end.
```

They are essentially dual to those given by wp of Section 3.

With these definitions in place, we can easily adapt the verification procedures of Section 3 to prove reachability theorems such as the following.

```
Lemma http_reaches_ids: |-r [TPDST=80] routing [PORT=3].
Proof. Time checker'. (*0. secs (0.u,0.s)*) Qed.
```

This theorem states that all packets with TCP destination port equal to 80 are forwarded to port 3, the network location of the intrusion detection middlebox.

5 Network Address Translation

In *Network Address Translation (NAT)*, IP packet headers are modified on the fly as packets are routed through a network, typically to implement IP sharing. For example, in private networks, the source IP addresses of packets routed from internal hosts to hosts outside the private network will be modified to the IP address of an externally visible router. The result is that only the router need be assigned a globally unique IP; internal hosts are not directly visible to the outside world. The technique can be extended to handle multiple internal hosts by storing information about the sender in an auxiliary field of the packet header. For example, for TCP traffic, the source port of the sender might be stored as the TCP source port.

As a concrete example, consider the network topology depicted in Figure 2. It consists of a single switch (center), three endhosts (H_1-H_3), and a general-purpose controller. The endhosts are connected on ports 1 through 3, while port 4 maintains connectivity with the Internet.

The network policy we would like to implement is: HTTP packets on port 80 destined for external hosts are forwarded to port 4, but only after their source IP address has been overwritten to ExternalIp, the IP address of the gateway switch. In order to correctly multiplex HTTP packets, the network program must also update the source TCP port of outgoing HTTP packets to equal the switch location of the sending host. Accordingly, incoming external HTTP packets should be sent to the switch port given by the packet's TCP destination, but only after the packet's TCP and IP fields have been restored. We implement this policy by combining a number of small NetCore programs, as follows.

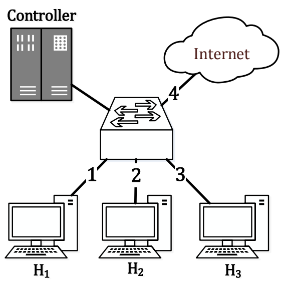


Fig. 2. *Network Address Translation.* HTTP packets sent from internal endhosts to external hosts (*Internet*) are multiplexed over the single external IP address assigned to a gateway switch (center).

First, we define[7] a NetCore program fragment that overwrites a packet's source IP address, then forwards the packet to port 4.

```
Definition pg1 :=
  WILD ⇒ UPD_IP_SRC ExternalIp 'SEQ' WILD ⇒ FWD 4.
```

Program pg1 can be combined with a parameterized NetCore program that updates a packet's TCP source field to equal the sender's switch port, as follows.

```
Definition pg2 (n: Z) :=
  PORT=n ⇒ UPD_TCP_SRC n 'SEQ' pg1.
```

All together, the rule for outgoing packets is the restriction of pg2, for hosts 1 through 3, to internal packets that are both *not* located on port 4 and have TCP destination port equal to 80.

```
Definition outgoing :=
  RESTRICT (pg2 1 'PAR' pg2 2 'PAR' pg2 3)
  BY (NOT PORT=4 'AND' TPDST=80).
```

The rule for incoming packets on port 4 is defined in a similar way, by first restoring the packet's IP and TCP destination fields, then forwarding the packet to the internal port given by the packet's initial TCP destination field.

Now that we have defined a NetCore program that implements NAT for the topology given above, we can verify that the program behaves as we expect. For example, the following lemma states that packets sent by hosts H_1 through H_3 are forwarded to port 4, with source IP address modified to equal ExternalIp and TCP source port set to the host port number h.

```
Lemma nat_ok: ∀h, List.In h hosts →
  |- [PORT=h] incoming 'PAR' outgoing
```

[7] The code that follows is found in `src/examples/NAT.v`.

```
    [NWSRC=ExternalIp 'AND' TPSRC=h 'AND' PORT=4].
Proof. Time unfold outgoing; check_all hosts. Qed.
(*Finished transaction in 0. secs (0.3125u,0.s)*)
```

Here `hosts = [1; 2; 3]`. The proof of this theorem relies on an additional automation tactic provided by our Coq library (check_all). The check_all tactic proves correctness theorems of the form ∀x, List.In x all → |- [P x] pg [Q x], where all is a finite multiset of x's, and P and Q are predicates on x's. The tactic works by breaking the theorem down into a finite set of verification conditions, which are all then proved automatically using the checker tactic of previous sections.

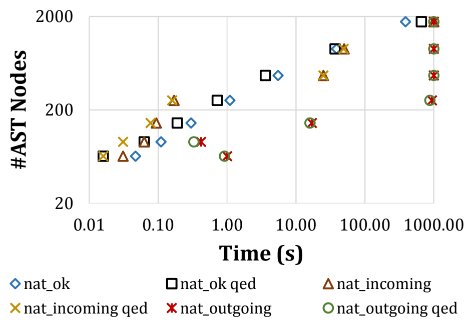
Using a variation of the check_all tactic for reachability queries, we can quite easily prove that TCP packets destined for external hosts are forwarded to port 4 of the gateway.

```
Lemma nat_outgoing: ∀h, List.In h hosts →
  |-r [TPDST=80 'AND' PORT=h 'AND' IS_IP 'AND' IS_TCP]
      incoming 'PAR' outgoing
     [PORT=4].
Proof. Time check_all' hosts. Qed.
(*Finished transaction in 0. secs (0.40625u,0.s)*)
```

Recall that the outgoing NetCore program updates both the the TCP source port and the source IP of outgoing packets. Thus the theorem holds only for packets that are indeed valid TCP/IP packets. Our Coq development proves an analogous reachability theorem for incoming traffic.

6 Measurements

Fig. 3. NAT timings for 2 to 65 endhosts.



To better understand the performance profile of our tool suite, we extended the network address translation example of the previous section to scale from 2 to 65 endhosts.[8] The plot in Figure 3 presents timing results, on a log-log scale, for the NAT security and reachability theorems we stated and proved in the previous section. The y-axis gives the size of the verified NAT programs in number of AST nodes. We measured both the time to execute proof scripts in Coq, and the time to typecheck proof terms at Qed (x-axis).

While these experiments are still quite preliminary, they seem to indicate that our tool suite is quite suitable for interactive use, at least for moderately

[8] File `src/examples/NATBench.v` in our development.

sized programs. Verifications of programs of up to approximately 250 AST nodes usually took no more than a second or two. On the other hand, there is still room for improvement. We would like to increase the efficiency of our resolution backend, by using more efficient data structures and also term orderings. We also plan to experiment with certificate-producing backends, in order to better understand the concomitant tradeoffs. For example, it is not immediately clear which matters more: the time to check proof certificates versus the potential gains from using a highly tuned external prover.

7 Related Work

There has been a great deal of work on verification of low-level network configurations in recent years [5–7, 11]. VeriFlow [6] uses an incremental analysis of OpenFlow rule updates to check network invariants such as reachability in real time. Anteater [7], an earlier effort, reduces verification of data plane invariants to SAT. Header space analysis [5] does static analysis of low-level network configurations using a geometric abstract domain. Reitblatt *et al.* [11] apply techniques from model checking to verify invariants of OpenFlow configurations. The VeriFlow paper [6] provides a good summary of additional related work.

The techniques described above all operate directly on switch and router configurations, in the form of unstructured flow tables. They therefore incorporate very little of the high-level structure present in the NetCore programs we analyze in this paper. In addition, our NetCore weakest precondition calculi are *proved complete* (and sound) in Coq. The analyses cited above provide no such formal guarantees. On the other hand, techniques such as header space analysis, which operates on a geometric abstraction of headers as uninterpreted bit vectors, make fewer assumptions about the underlying network protocols, and therefore are more general than the analyses we describe in this paper.

8 Conclusions

We have only scratched the surface of potential applications of the verification techniques we describe in this paper. In our Coq development, we do example verifications[9] of the conditions that arise when proving disjointness of virtual networks, or VLANs, using the network *slice* abstraction recently proposed by Gutz *et al.* [4]. We have also begun to explore the use of our tool to detect, and prove the absence of, loops in multi-switch networks, using a static analysis that depends heavily on our weakest precondition calculus for reachability. Although this work is still in progress, completeness of the weakest precondition calculus should allow us to *prove* the absence of network loops, given a network program and topology, rather than just detect them, as is done using existing techniques such as Header Space Analysis [5].

At the same time, our Coq library is still in its early stages, and is therefore limited in some ways. For example, at the moment we only target static NetCore

[9] File `src/examples/Slice.v`.

programs running on concrete network topologies (that is, in which the number of switches, ports, and hosts are all known in advance). We would like to experiment with using the library as a subcomponent of a larger tool suite, in order to do verification of controller programs that generate streams of NetCore programs in response to a stream of network input events. In addition, our specification language currently only targets expressions in the NetCore predicate language. In the future, we plan to extend it, and the accompanying tool support, to enable verification of richer properties. Finally, because our NetCore semantics was defined before Guha *et al.*'s verified compiler was publically available, it differs in some details from the Guha *et al.* implementation. For example, in order to do proof by reflection in Coq, we specify packets using a fixed-width machine integer library that supports computable equality, whereas Guha *et al.* use an axiomatization of machine words. We also support sequential composition, whereas Guha *et al.*'s compiler does not. It is details like these that have so far prevented complete convergence of our mechanized development with theirs.

Acknowledgments. I am indebted to the members of the Princeton programming languages group for reading and commenting on early versions of this paper, and to the anonymous reviewers for their insightful comments.

References

1. L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*. 2008.
2. N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *ICFP*, 2011.
3. A. Guha, M. Reitblatt, and N. Foster. Machine-verified network controllers. In *PLDI*, 2013.
4. S. Gutz, A. Story, C. Schlesinger, and N. Foster. Splendid isolation: A slice abstraction for software-defined networks. In *Hot Topics in SDNs*. ACM, 2012.
5. P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. *NSDI*, 2012.
6. A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *Hot Topics in SDNs*. ACM, 2012.
7. H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with Anteater. *ACM SIGCOMM CCR*, 41(4), 2011.
8. W. McCune and L. Wos. Otter: The CADE-13 competition incarnations. *JAR*, 18:211–220, 1997.
9. N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling innovation in campus networks. *ACM SIGCOMM CCR*, 38(2):69–74, 2008.
10. C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. In *POPL*, 2012.
11. M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, 2012.
12. J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of The ACM*, 12:23–41, 1965.
13. A. Voellmy and P. Hudak. Nettle: Taking the sting out of programming network routers. In *PADL*. 2011.