Bagpipe: Verified BGP Configuration Checking

Konstantin Weitz
University of Washington
weitzkon@cs.uw.edu

Doug Woos
University of Washington
dwoos@cs.uw.edu

Emina Torlak
University of Washington
emina@cs.uw.edu

Michael D. Ernst
University of Washington
mernst@cs.uw.edu

Arvind Krishnamurthy
University of Washington
arvind@cs.uw.edu

Zachary Tatlock
University of Washington
ztatlock@cs.uw.edu

Abstract

To reliably and securely route traffic across the Internet, Internet Service Providers (ISPs) must configure their Border Gateway Protocol (BGP) routers to implement policies restricting how routing information can be exchanged with other ISPs. Correctly implementing these policies in low-level router configuration languages, with configuration code distributed across all of an ISP's routers, has proven challenging in practice, and misconfiguration has led to extended worldwide outages and traffic hijacks.

We present Bagpipe, a system that enables ISPs to concisely express their policies and automatically check that router configurations adhere to these policies. To check policies efficiently, Bagpipe introduces the initial network reduction, exploits modern satisfiability solvers by building on the Rosette framework for solver-aided tools, and parallelizes configuration checking across many nodes. To ensure Bagpipe correctly checks configurations, we verified its implementation in Coq, which required developing both a new framework for verifying solver-aided tools and also the first formal semantics for BGP based on RFC 4271.

To validate the effectiveness of our verified checker, we ran it on the router configurations of Internet2, a nationwide ISP. Bagpipe revealed 19 violations of standard BGP router policies without issuing any false positives. To validate our BGP semantics, we performed random differential testing against C-BGP, a popular BGP simulator. We found no bugs in our semantics and one bug in C-BGP.

1. Introduction

Over 3 billion people are connected to the Internet through university and corporate networks, regional ISPs, and nationwide ISPs [1]. These networks, collectively known as Autonomous Systems (ASes), exchange routing information—the paths traffic can take across the Internet—via the Border Gateway Protocol (BGP). To reliably and securely route traffic, ASes must configure all their BGP routers to implement *BGP policies* restricting how routing information can be exchanged. For example, some nation-operated ASes use BGP to censor websites with political content by announcing fake routing information; if such bogus announcements leak outside the censoring nation, they can cause widespread outages. In 2009, YouTube was inaccessible worldwide for several hours due to a misconfiguration in Pakistan [4]. To protect against this problem, an AS could implement a BGP policy to ignore such censorious neighbors when they announce routes to addresses (e.g., for YouTube) that are outside their control.

Correctly implementing BGP policies in low-level configuration languages has proven challenging. Large ASes maintain millions of lines of frequently changing configurations that run distributed across hundreds of routers [17, 36]. Router misconfigurations are common and have led to highly visible failures affecting ASes and their billions of users. In addition to the YouTube outage mentioned above, in 2010 and 2014, China Telecom hijacked significant but unknown fractions of international traffic for extended periods [8, 25, 26, 32]. Goldberg surveys several additional major outages and their causes [15], some of which could have been prevented by correctly implementing appropriate BGP policies. Less visible is the high cost ASes pay every day to develop and maintain configurations with little to no tool support.

This paper presents Bagpipe[1], a system that enables ASes to concisely express BGP policies and automatically check that router configurations correctly implement them. At a high level, Bagpipe checks that a policy will hold in all reachable router states, for every possible announcement, along every path through an AS. Bagpipe is efficient because it exploits the insight that a router's behavior is maximal in the initial network: if a policy holds for all possible announcements along every path through the AS in the empty router state, then it will also hold for all reachable router states. While this *initial network reduction* makes the BGP policy checking problem finite, the number of possible announcements is still far too large to permit brute-force enumeration. To efficiently check all possible announcements, Bagpipe is implemented in Rosette [34], a framework for building solver-aided tools. Rosette provides expressive verification facilities, which are implemented efficiently on modern satisfiability modulo theories solvers like Z3. Finally, because checking the policy is independent along each path through the AS, Bagpipe checks paths in parallel to scale up to large configurations.

[1] Bagpipe is open-source. Link redacted for review. Source attached as Supplemental Material.

To ensure that the Bagpipe checker is itself correct, we verified its implementation in Coq. Since Bagpipe is implemented as a solver-aided language in Rosette, verifying it required formalizing the semantics of Rosette. We designed SEARCHSPACE, a new framework for verifying solver-aided tools in Coq. Verifying Bagpipe also requires reasoning about BGP, so we also developed the first formal semantics for BGP based on RFC 4271 [30].

Bagpipe supports policies found in the literature, such as the Gao-Rexford model [14], prefix-based filtering [27], and policies inferred from real AS configurations. Bagpipe works out of the box with existing Juniper router configurations.

This paper's contributions include:

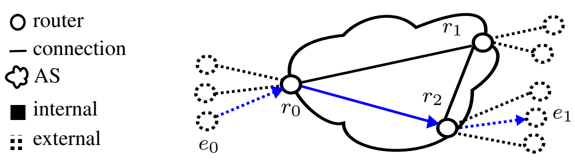
- The first formal semantics of BGP (based on RFC 4271 [30]), and a specification language for BGP policies.
- SEARCHSPACE, a framework for verifying scalable solver-aided tools in Coq. We describe the design of SEARCHSPACE and how we applied it to formally verify the Bagpipe checker that implements BGPV.
- BGPV, an algorithm to check that router configurations correctly implement BGP policies. BGPV is based on the new *initial network reduction*, is designed to use powerful verification operators provided by Rosette, and is parallelized to scale up to checking large configurations.

We performed case studies to evaluate the above contributions. To show that our semantics is accurate and useful, we performed random differential testing against C-BGP [29], a popular BGP simulator. The only situations in which our semantics differed were due to a previously-unknown bug in C-BGP, which was acknowledged by C-BGP's maintainer. We also discuss two bugs uncovered in an unverified prototype of Bagpipe during the process of verifying BGPV, thus highlighting the value of using SEARCHSPACE to formally verify solver-aided tools. We evaluated Bagpipe on Internet2, a nationwide ISP with over 100,000 lines of BGP configuration. We expressed standard BGP policies for Internet2, and Bagpipe found 19 violations without issuing any false positives.

The rest of this paper is organized as follows: Section 2 provides an overview of Bagpipe. Section 3 formalizes the semantics of BGP. Section 4 discusses and formalizes AS policies. Section 5 formalizes SEARCHSPACE and describes the translation to Rosette. Section 6 presents BGPV, Bagpipe's core algorithm for checking restricted policies, and the initial network reduction. Section 7 discusses the implementation of Bagpipe. Section 8 evaluates Bagpipe. Section 9 reviews related work, and Section 10 concludes.

2. Overview

This section shows how network operators can use Bagpipe to express and check a specification for an AS. To illustrate this workflow, we consider the Internet2 AS and the $blockToExternal$ specification, which ensures that private routes are not leaked to other ASes.



$$blockToExternal(r, \_, o, \_, \_, \_, a) : spec :=$$
$$\texttt{BTE} \in communities(original(a)) \to o \in R_e(r) \to a = na$$

Figure 1. The $blockToExternal$ specification. A BGP network consists of routers internal and external to an AS. The $blockToExternal$ specification states that any update message whose attributes includes the BTE community when it entered the AS must not be broadcast to any external neighbor o. $original$ returns the original attributes of an update message before it has been propagated through the AS.

Each device on the Internet is uniquely identified by an *Internet Protocol* (IP) address. To transmit a packet through the Internet, a device addresses the packet with the IP address of its destination and sends the packets to an Internet router. The router either delivers the packet (if it is connected to the destination) or forwards the packet to one of its neighboring routers that is closer to a router that can deliver the packet, i.e., the router must know a *path* or sequence of routers that will deliver the packet to its destination. Routers use BGP to share paths to IP addresses so that the routers can deliver packets across the Internet.

A device d's IP address does not describe the path along which a packet must be transmitted to reach d. Routers use BGP to inform each other of the destination prefixes they can reach, including the prefixes they own as well as the prefixes they have learned about from previous BGP messages. These BGP *update messages* include a prefix and an *AS path*, which indicates the ASes through which the message has propagated. Each router stores received update messages in a *routing table*, which is then used to determine how data packets should be routed.

Figure 1 shows an AS's network and the $blockToExternal$ specification. An AS consists of internal routers that it controls, connected to external routers that it does not. The $blockToExternal$ specification holds if it is impossible for any of the AS's routers r to send any update message a to an external router o when a was tagged with the BTE community when it entered Internet2's AS. For example, if internal router r_0 receives an update message tagged with the BTE community from external router e_0, then r_0 may send that message to router r_2, but r_2 may not send it to external router e_1.

Bagpipe can automatically check the $blockToExternal$ specification. At the core of Bagpipe is the BGPV algorithm summarized in Fig. 2. BGPV either returns a proof that the policy being checked holds or a counterexample showing a violation of the policy. To check the policy, BGPV has to verify that it holds in all reachable states of the AS's BGP network. This is made tractable by the initial network reduction (INR), which shows that it is sufficient to only consider the search

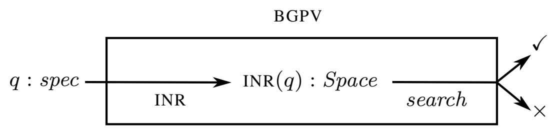


Figure 2. BGPV Workflow. To check the specification q (e.g. *blockToExternal*), BGPV first applies the initial network reduction (INR) to convert q into a search $Space$. Rosette then searches this space for a counterexample, returning it if it finds one (×). If no counterexample is found, the AS's router configuration adheres to its policy (✓).

```
group OTHER {                                    -- group of neighbors
  import REJECT-ALL;                 -- use REJECT-ALL as an import policy
  neighbor 128.223.51.102 {        -- configuration for a specific neighbor
    import REJECT-ALL; ...}}         -- use REJECT-ALL as an import policy
```

Figure 3. Broken Juniper Configuration. This shows Internet2's configuration for a specific neighbor, `128.223.51.102` in the `OTHER` group. As shown, Internet2 uses `REJECT-ALL`, which rejects all incoming routes, as its import policy for the whole group. The configuration for neighbor `128.223.51.102` redundantly also uses `REJECT-ALL` as its import policy. We believe that Internet2 intended to use `REJECT-ALL` as its export policy for this neighbor.

space ($Space$) of messages sent in the initial network — i.e. no other messages have yet been sent in the network. Bagpipe efficiently explores this search space with Rosette ($search$) which uses Z3 instead of brute-force search. Because each path can be checked independently, Bagpipe also parallelizes the operations so they can be efficiently executed by a cluster.

It is difficult for network administrators to correctly implement the *blockToExternal* specification because router configurations 1) are distributed: r_0 could, unknown to r_1, clear the BTE community of certain update messages, 2) interfere: rules intended to block update messages with BTE could be disabled by other rules, and 3) are written in languages with little static checking: a misspelling of BTE would not be detected. One of the violations of *blockToExternal* found in Internet2 is caused by a rule with a redundant import statement, but no statement limiting the export of update messages, as shown in Fig. 3.

Bagpipe is implemented and verified in Coq against the SEARCHSPACE framework. We expect that this methodology — implementing and verifying solver-aided tools in Coq against the SEARCHSPACE framework — can be used to develop and verify the end-to-end soundness of other automated and scalable solver-aided checkers, such as data-race detectors [20, 23, 31], memory-model checkers [35], and compiler optimization validators [24, 33].

3. BGP Semantics

This section presents our formal semantics of BGP. It is the first formal semantics based on the BGP specification RFC 4271 [30]. The semantics enables rigorous reasoning about the correctness of router configurations and can be used in various applications outside of BGP configuration verification, e.g. verifying tools that check data plane properties as in Batfish [12], or testing simulators as described in Section 8.

$R : Set$ — routers
$C \subseteq R \times R$ — connections between routers
$P := [0,256) \times [0,256) \times [0,256) \times [0,256) \times [0,33)$ — prefixes
$A := \mathcal{A} \cup na$ — attributes are routing information $\mathcal{A}$ or not available na
$M := P \times A$ — update messages
$in(r) := \{s \mid (s,d) \in C, r = d\} \cup \{injected\}$
$out(r) := \{d \mid (s,d) \in C, r = s\}$ — r's outgoing neighbors
$imp_r : in(r) \to M \to A$ — configures message import
$exp_r : in(r) \to out(r) \to M \to A$ — configures message export
$dec_r : (in(r) \to A) \to in(r)$ — configures message selection
$S_r := (in(r) \times P \to A) \times (P \to A) \times (out(r) \times P \to A)$ — state
$empty := (\lambda(\_,\_).na, \lambda\_.na, \lambda(\_,\_).na)$ — empty router state

Figure 4. Semantics Definitions. The semantics are parametric over the routing attributes $\mathcal{A}$ (which includes communities such as BTE) . $in(r)$ and $out(r)$ are a router r's incoming and outgoing connections. $injected$ is a fake incoming connection used to inject messages into a router. na is used to withdraw a route. A router r's configuration consists of an imp_r, exp_r, and dec_r rule. S_r is the state of router r.

The presentation of our BGP semantics is split into three parts. Section 3.1 describes how BGP sends messages between routers. Section 3.2 describes how BGP routers process messages. Section 3.3 describes the restrictions BGP places on its configuration.

3.1 Network Semantics

We say an AS *owns* an IP address ip if it can deliver packets addressed to ip without the packets traversing through another AS. Sets of IP addresses are commonly written in Classless Interdomain Routing (CIDR) notation: $ip/size$, where ip is an IP address and $size$ is a number between 0 and 32. All addresses whose initial $size$ bits are the same as those of ip are in the set $ip/size$; for example, `192.168.1.0` and `192.168.1.42` are in `192.168.1.0/24` but `192.168.2.0` is not. CIDR notation specifies a set of IP addresses that start with the same prefix, so a set of IP addresses is referred to as a *prefix* P. This and other important definitions are summarized in Fig. 4.

The BGP configuration at each router controls how update messages are processed, modified, and forwarded to other routers. A simple configuration might accept all messages and then forward them all unmodified. Configured in this simple way, the semantics of BGP reduce to the following two rules:

INJ Each router r in AS A sends each neighbor r' a message containing each prefix owned by r and AS path $[A]$.
FWD Each router r in AS A forwards to each neighbor r' each message m it received, appending A to m's path.

With more complex configurations, the rules are somewhat more complicated, but the basic structure is the same: messages can be injected for prefixes owned by routers, and messages can be forwarded through the network. These rules are formalized in Figure 5.

$$\Gamma : (C \rightarrow \mathit{list}(M))$$
$$\Sigma : (R \rightarrow S)$$

$$\boxed{(C \rightarrow \mathit{list}(M)) \times (R \rightarrow S) \rightsquigarrow (C \rightarrow \mathit{list}(M)) \times (R \rightarrow S)}$$

$$\frac{m \qquad \sigma', \Gamma' = \mathit{handle}_r(\mathit{injected}, m, \Sigma[r])}{\Gamma, \Sigma \rightsquigarrow (\lambda c.\mathit{append}(\Gamma(c), \Gamma'(c))), \Sigma[r := \sigma']} \text{ INJ}$$

$$\frac{(s, r) \in C \qquad \sigma', \Gamma' = \mathit{handle}_r(s, m, \Sigma[d])}{\Gamma[(s, r) := m :: \Gamma(s, r)], \Sigma \rightsquigarrow (\lambda c.\mathit{append}(\Gamma(c), \Gamma'(c))), \Sigma[r := \sigma']} \text{ FWD}$$

$$\boxed{\mathit{reachable}((C \rightarrow \mathit{list}(M)), (R \rightarrow S))}$$

$$\frac{}{\mathit{reachable}(\lambda(\_, \_).[], \lambda\_.\mathit{empty})} \text{ INIT}$$

$$\frac{\mathit{reachable}(\Gamma, \Sigma) \qquad \Gamma, \Sigma \rightsquigarrow \Gamma', \Sigma'}{\mathit{reachable}(\Gamma', \Sigma')} \text{ STEP}$$

Figure 5. Network Semantics. The INJ and FWD rules define a step relation $\rightsquigarrow$ on AS states (where an AS state consists of the state at each router as well as the in-flight messages on each connection). The INJ rule injects an arbitrary update message into a router for processing. The FWD rule removes an update message from the network and injects it into a router for processing. The *reachable* relation is the reflexive transitive closure of these rules. $m[k := v]$ is a notation for map/dictionary updates and is defined as $(\lambda k'.\text{if } k = k' \text{ then } v \text{ else } m(k'))$.

We model the BGP protocol as repeated application of these rules. In principle, this protocol might never converge—for instance, update messages might be sent in an infinite loop around a network. In practice, however, router configurations and the topology of the Internet guard against this possibility (the Bagpipe checker described in the following sections avoids the convergence issue entirely since it only checks a single AS network in a full-mesh configuration, which cannot contain update loops). When the protocol converges, it reaches a steady state in which every router's routing table contains exactly one path to the owner of every prefix. This means that each router knows along which path to transmit any packet.

An AS is described by a set of routers R and a relation describing the connections between routers $C \subseteq R \times R$. The AS state consists of a list of in-flight messages for each connection $C \rightarrow \mathit{list}(M)$, and a state for each router $R \rightarrow S$ that contains the routing table.

An update message consists of a prefix p and attributes a. An update message's attributes are either na (not available), if the prefix is being withdrawn, or some value of type $\mathcal{A}$. Our BGP semantics are parametric over $\mathcal{A}$, but requires some projections from $\mathcal{A}$, including a projection path to extract the attributes' AS path.

The handle_r function returns a router r's response to an incoming message. Specifically, it returns r's new state and the messages that r sends to its neighbors. handle is defined in the next subsection.

The INJ rule injects an arbitrary message m into router r. The INJ rule invokes handle_r, which returns r's new state σ' and a list of messages for each of r's neighbors Γ'. Intuitively, this rule represents r announcing a prefix it owns; handle_r should return messages for its neighbors if and only if r owns m's prefix p.

The FWD rule picks an arbitrary connection (s, r) and the first in-flight message m on that connection (the BGP specification requires that the messages delivery on each connection are ordered; this is accomplished using TCP). The FWD rule then invokes handle_r, which can modify r's local state and return messages for its neighbors just as in INJ.

The INJ and FWD rules define a step relation $\rightsquigarrow$ on AS states (where an AS state consists of the state at each router as well as the in-flight messages on each connection). The *reachable* relation is the reflexive transitive closure of this step relation; an AS state is *reachable* if there is a finite number of $\rightsquigarrow$ steps from the initial AS state to it. In the initial AS state, there are no in-flight messages and every router is in the *empty* state.

Note that these semantics only describe the network's BGP behavior, not its packet-forwarding behavior. For Bagpipe, it was only necessary to formalize the operation of the *control plane* on which BGP messages are propagated, rather than the *data plane* where ordinary packets are forwarded between routers. The network's data plane behavior is controlled by the state σ at each router.

3.2 Router Semantics

Many ASes act in self-interest and are in direct competition. For example, an AS might not want to forward update messages to its competitors. The BGP specification gives ASes freedom to configure their routers. This means that BGP provides very few general guarantees — to ensure desirable properties, they have to be proven for a particular topology and set of router configurations.

The BGP specification requires a router to execute several steps in response to an incoming message m which consists of a prefix p and attributes a. These steps are implemented by the *handle* function defined in Fig. 6. The behavior of each step can be customized in the router's BGP configuration by specifying an *imp*, *dec*, and *exp* rule. The steps are:

1. The `import` step modifies the attributes a of the incoming message. The step can be configured using the *imp* rule which defines how to modify the attributes. This step can, for example, be useful to filter incoming messages with invalid prefixes.
2. The `decision` step selects a single value $a*$ from the most recently received and modified attributes by the router for each neighbor and prefix p. The step can be configured using the *dec* rule which defines which attributes to select. This step can, for example, be useful to prefer messages from paying neighbors over messages from neighbors that expect to be paid.

```
$handle_r : in(r) \to M \to S(r) \to ((C \to list(M)) \times S(r))$
$handle_r(i, (p, a), (\sigma_i, \sigma_l, \sigma_o)) :=$
    let $\sigma'_i := \sigma_i[i, p := a]$
        $i^* := dec(r, \lambda i.imp(r, i, p, \sigma'_i[i, p]))$
        $a^* := imp(r, i^*, p, \sigma'_i[i^*, p])$
        $\sigma'_l := \sigma_l[p := a^*]$
        $\sigma'_o := \lambda(o, p')$.if $p' \neq p$ then $\sigma_o[o, p']$
                        else $exp(r, i^*, o, p, a^*)$
        $\Gamma := \lambda(s, d)$.if $s = r \wedge \sigma_o[d, p] \neq \sigma'_o[d, p]$
                     then $[(p, \sigma'_o[d, p])]$ else $[]$
    in  $(\Gamma, (\sigma'_i, \sigma'_l, \sigma'_o))$
```

Figure 6. Router Semantics. *handle* defines how a router processes an update message (p, a) received from connection i. First, r stores (p, a) in the `adjRIBsIn` σ_i. Second, r imports all attributes in its `adjRIBsIn`, choosing the best attribute a^* from neighbor i^* and storing it in the `locRIB` σ_l. Third, r exports a^* to all its neighbors, storing the result in its `adjRIBsOut`.

3. The `export` step modifies and forwards $a*$ to each of the router's neighbors, if the modified attributes differ from the attributes previously sent to the neighbor for prefix p. The step can be configured using the exp rule which defines how to modify the attributes. This step can, for example, be useful to block certain messages to some neighbors.

The `import` and `export` rules can discard a message by modifying the message's attributes to na. For example, to discard all messages with prefix p' that are about to be sent to a neighbor o, an AS could provide the following exp rule:

$$\lambda r\ i\ o\ p\ a.\ \text{if } p = p' \text{ then } na \text{ else } a$$

Depending on r's state, either one of the following two action are executed by $handle$: if `adjRIBsOut`(o, p) is already na (e.g. in the $empty$ network), no messages will be sent because `adjRIBsOut`(o, p) does not change; otherwise na is sent to o, withdrawing any previous update messages.

$handle$ also updates a router r's state σ. σ consists of three Routing Information Bases (RIBs) $\sigma_i, \sigma_l, \sigma_o$. The `adjRIBsIn`, σ_i, contains the most recently received update message for each neighbor and prefix. The `locRIB`, σ_l, contains the most recently selected update message for each prefix; σ_l is the routing table used to forward actual packets in the data plane. The `adjRIBsOut`, σ_o, contains the most recently sent update message for each neighbor and prefix. A router's initial state is $empty$.

$in(r)$ is the set of all neighbors from which r can receive update messages, unioned with a dummy neighbor called $injected$ that is used to inject routes, as described in the INJ rule in Fig. 5. $out(r)$ is the set of all neighbors to which r can send update messages.

3.3 Rule Restrictions

The BGP specification places some restrictions on import, decision, and export rules. Our formalizations of these restrictions are shown in Fig. 7.

Restriction 1 and 2 state that imp and exp rules may not create attributes "out of thin air" for update messages that are

```
1. $\forall r\ i\ p.\ imp(r, i, p, na) = na$
2. $\forall r\ i\ o\ p.\ exp(r, i, o, p, na) = na$
3. $\forall r\ i\ p\ a.\ md(i, r) = md(r, o) = ibgp \to exp(r, i, o, p, a) = na$
4. $\forall r\ i\ p\ a.\ asn(r) \in path(a) \to imp(r, i, p, a) = na$
5. $\forall r\ i\ o\ p\ a.\ md(r, o) = ebgp \to$
       $path(exp(r, i, o, p, a)) = asn(r) :: path(a)$
6. $\forall r\ \sigma\ i.\ pref(\sigma[i]) \leq pref(\sigma[dec(r, \sigma)])$
```

Figure 7. Rule Restrictions. BGP requires that the imp and exp rules cannot create attributes "out of thin air" (1,2), avoid forwarding loops (3,4), and extend paths appropriately (5). BGP also restricts dec rules; the $pref$ associated with the connection chosen by dec must be maximal (6).

na. Restriction 3 states that exp rules only forward update messages once within an AS. This avoids routing loops. Restriction 4 states that imp rules drop update messages with loops in their path. Restriction 5 requires that exp rules extend an update message's AS path whenever a message crosses an AS border.[2] Restriction 6 states that dec rules must select a connection whose associated update message has maximal $pref$. $pref$ is a projection, like $path$, that extracts a number from attributes. The fact that the imp rule can modify update messages can be used to change the $pref$ and thus influence the choice of the dec rule. The BGP specification places additional tie-breaking restrictions on the dec rule. To model this, our semantics is parameterized over an arbitrary dec function.

The function $asn(r)$ returns the AS number (a unique AS identifier) of r's AS. The function md assigns a mode to every connection. Connections between routers owned by the same AS are in $ibgp$ (internal BGP) mode, and connections between routers owned by different ASes are in $ebgp$ (external BGP) mode, i.e.

$$md(s, d) = \text{if } asn(s) = asn(d) \text{ then } ibgp \text{ else } ebgp$$

Because real-world BGP configuration languages often enforce a subset/superset of the above restrictions, our semantics was developed to make it easy to enforce only a subset/superset of the restrictions. For example, the router configuration language used by Juniper-manufactured routers does not enforce restriction 5, and allows arbitrary manipulations of the AS path. On the other hand, the C-BGP router configuration language, unlike the specification, ensures that update messages may not be sent back along the connection that they came from.

3.4 Comparison to RFC 4271

Our semantics models the full BGP specification (RFC 4271) except for low-level details (bit representation of update messages and TCP). It does not model all optional features, such as route aggregation. It models some extensions such as the communities attribute, but not others such as route reflectors. We believe that our semantics could be extended with these additional features.

[2] The BGP spec allows $asn(r)$ to be repeated multiple times to influence tie-breaking.

```
$spec := (R \times R \times R \times P \times A \times A \times A \to bool)$

$specHolds : spec \to Prop$
$specHolds(\tau) :=$
  $\forall \Gamma\ \Sigma\ (t : reachable(\Gamma, \Sigma))\ r\ p\ (i \in in(r))\ (o \in out(r)),$
      let $\sigma_i, \sigma_l, \sigma_o := \Sigma[r]$
          $a_i := \sigma_i[i, p], a_l := \sigma_l[p], a_o := \sigma_o[o, p]$
      in  $\tau(r, i, o, p, a_i, a_l, a_o) = true)$
```

Figure 8. Specification Definition. A specification τ is an invariant over BGP behavior. τ is a boolean predicate over the attributes a_i, a_l, and a_o which represent values in the `adjRIBsIn`, `locRIB`, and `adjRIBsOut` of a reachable router state Σ for a particular router r, incoming neighbor i, outgoing neighbor o, and prefix p. τ holds if it is true for all traces t to reachable AS states (Γ, Σ).

4. Specifications

To enable efficient checking, Bagpipe restricts specifications of BGP policies to predicates of the following form, where τ is a boolean predicate of type *spec* as shown in Fig. 8:

> For router r with an incoming neighbor i and outgoing neighbor o, and triple (a_i, a_l, a_o) of attributes received, selected, and sent (respectively) for prefix p from i through r to o, $\tau(r, i, o, p, a_i, a_l, a_o) = true$.

Note that attributes a_i, a_l, and a_o represent values in the `adjRIBsIn`, `locRIB`, and `adjRIBsOut` of r for incoming neighbor i and outgoing neighbor o at prefix p.

Specifications of this form provide sufficient expressiveness in practice as they capture the behaviors of configuration rules *imp*, *dec*, and *exp* that network operators write to implement their policies. Furthermore, our restriction on specifications capture restrictions RFC 4271 places on BGP policies:

> [The *imp*/*exp* rule] SHALL NOT use any of the following as its inputs: the existence of other routes, the non-existence of other routes, or the path attributes of other routes.

For example, a predicate can constrain which attributes a_l are selected depending on the received attributes a_i, but cannot constrain which attributes a_o are sent depending on some other attributes that have previously been sent.

As described in the previous section, the behavior of an AS is captured by *reachable*, the reflexive transitive closure of the step relation. For a given AS state (Γ, Σ), we call a derivation t of $reachable(\Gamma, \Sigma)$ a *trace* as it contains the sequence of states from the initial network to (Γ, Σ). A specification S holds if it is true for all reachable network states, as detailed in Fig. 8.

Bagpipe verifies the configurations of a single AS against a given specification. The topology of the AS under consideration is given by the set of routers internal to the AS R_i and the set of neighboring external routers $\mathcal{N}_r$ that each internal router r is connected to. As required by the BGP specification, the AS is assumed to be in a full-mesh configuration: each internal router is directly connected to all other internal routers.[3]

The topology and configuration of routers outside the AS are treated as "havoc": a specification must hold for all possible topologies and configurations of external routers. Note that this means specifications verified by Bagpipe hold even for neighbors which do not adhere to restrictions imposed by RFC 4271. We believe this conservative approach will be useful in practice, since common router configuration languages often do not strictly enforce the restrictions on *imp*, *dec*, and *exp* rules required by the RFC 4271.

Figure 9 defines three example specifications: *noMartian*, *blockToExternal* and *goodPreference*.

The *noMartian* specification ensures that routers drop update messages with "martian" prefixes, i.e. invalid prefixes such as the private prefix `10.0.0.0/8` or the loop-back prefix `127.0.0.0/8` which should not be used to transmit packets over the Internet. *noMartian* ensures that routes for such bogus prefixes are never stored in an AS's routing tables, and are thus never used to transmit packets.

The *blockToExternal* specification ensures that certain update messages are not exported. RFC 1997 [5] extends the attributes $\mathcal{A}$ of an update message with a projection *communities* to extract the attributes' set of communities. A community is a named flag that is either set or unset. For example, the RFC defines the `NO_EXPORT` community: when `NO_EXPORT` is set on a message m, the receiver r of m should not forward m to any router external to r's AS. *blockToExternal* ensures that a message m that entered the network with the community C set is never exported to an external router $d \in R_e$. The attributes of an update message m as it entered the current AS can be accessed with the projection *original* described later in Section 6.

The *goodPreference* specification ensures that an AS chooses the "best" update messages. In the Gao-Rexford model [14], a widely-used description of AS behavior, ASes partition their neighbors into three *relation*s: *customer*s, *peer*s, and *provider*s. Customers pay the AS to forward packets, peers neither charge nor pay money to forward packets, and providers charge money to forward packets. The Gao-Rexford model states that in order to maximize profit, a router should always prefer an update message from a customer over an update message from a peer or provider and should always prefer an update message from a peer over an update message from a provider. This is captured by the $\sqsubseteq$ relation. The *goodPreference* specification ensures that for every router r, the update message installed in the r's routing table σ_l for a certain prefix p is always better than or equal to ($\sqsubseteq$) all update messages received by r for prefix p. The external router from which an update message m entered

[3] Some large ASes avoid the performance penalty of a full-mesh configuration by using *route reflectors*, routers that exist to propagate messages between multiple connected components of an AS's topology. Bagpipe does not currently model this optional extension of the BGP specification.

$noMartian(\_,\_,\_,p,\_,a_l,\_) : spec :=$
$\quad p \in martian \rightarrow a_l = na$

$blockToExternal_C(\_,\_,o,\_,\_,\_,a_o) : spec :=$
$\quad C \in communities(original(a_o)) \rightarrow o \in R_e \rightarrow a_o = na$

$goodPreference_{\sqsubseteq}(r,i,\_,p,a_i,a_l,\_) : spec :=$
$\quad imp(r,i,p,a_i) \sqsubseteq a_l$

$a \sqsubseteq a' = value(a) \leq value(a')$

```
value(a) :=
  if $a = na$ then 0
  else let $r = relation(origin(a))$ in
    if $r = provider$  then 1 else
    if $r = peer$      then 2 else
    if $r = customer$ then 3 else undefined
```

Figure 9. Example Specifications. These are standard BGP specifications, based on policies common to many ASes. The *noMartian* specification ensures that routers drop update messages with "martian" prefixes, like `10.0.0.0/8`. The *blockToExternal* specification ensures that certain update messages are not exported. The *goodPreference* specification ensures that an AS chooses the "best" update messages, according to the Gao-Rexford model that partitions an AS's external neighbors into *customer*s, *peer*s, and *provider*s.

the current AS can be accessed with the projection *origin* described later in Section 6.

5. SearchSpace

This section describes SEARCHSPACE, a framework for verifying solver-aided tools in Coq. Solver-aided tools written against SEARCHSPACE can be translated to a solver-aided host language (e.g., Rosette [34] and Smten [37]), which uses SMT solvers for efficient verification of (finite) programs.

We do not expose a full solver-aided host language directly to Coq. Instead, we expose a monadic domain-specific language (DSL) called SEARCHSPACE. SEARCHSPACE is inspired by Smten, and is based on constructing search spaces of values to be explored by the underlying SMT solver. SEARCHSPACE is defined in Fig. 10, and includes a number of functions for building spaces and a search function for finding a member of a space if one exists.

In Coq, the SEARCHSPACE functions are denoted in terms of sets. The $enumerate_A$ set, which must be implemented for every type A separately (using the provided space-building functions), contains every member of a finite type A. This set cannot be implemented for infinite types like $\mathbb{N}$.

SEARCHSPACE can run programs written against this interface by extracting the Coq terms to Rosette. Rosette implements symbolic boolean and integer values, assertions, and a `search` function, which takes as input an expression and tries to find a concrete assignment to any symbolic values in that expression that does not violate any assertions. The `search` function works by translating the input expression into an SMT formula and solving it with an off-the-shelf SMT solver. Coq's built-in extraction facility translates user-defined functions directly to Racket syntax (which is also

$Space : Type \rightarrow Type$	— sets whose members are of type A
$enumerate_A : Space(A)$	— all terms inhabiting A
$empty_A : Space(A)$	— an empty space
$single_A : A \rightarrow Space(A)$	— create a singleton space
$union_A : Space(A) \rightarrow Space(A) \rightarrow Space(A)$	— merge two spaces
$bind_{A,B} : Space(A) \rightarrow (A \rightarrow Space(B)) \rightarrow Space(B)$	
$search_A : Space(A) \rightarrow A \cup \bot$	— returns an element in A; $\bot$ if empty

$[\![Space(A)]\!] = Set(A)$
$[\![enumerate_A]\!] = \{a \mid a : A\}$
$[\![empty_A]\!] = \emptyset$
$[\![single_A(a)]\!] = \{a\}$
$[\![union_A(s,t)]\!] = [\![s]\!] \cup [\![t]\!]$
$[\![bind_{A,B}(s,f)]\!] = \bigcup_{a \in [\![s]\!]} [\![f(a)]\!]$
$[\![search_A(s)]\!] =$ `if` $[\![s]\!] = \emptyset$ `then` $\bot$ `else` $choose([\![s]\!])$

```
$empty_A() \triangleq$ (lambda (v) (assert false))
$single_A(a) \triangleq$ (lambda (v) a)
$union_A(s,t) \triangleq$ (lambda (v)
  (if (symbolic-bool) (s v) (t v)))
$bind_{A,B}(s,f) \triangleq$ (lambda (v) (f (s v) v))
$search_A(s) \triangleq$ (search s) ;; calls Rosette solver
```

Figure 10. SEARCHSPACE DSL. *Space* forms a monad with *single* and *bind*. $bind(s,f)$ maps the function f over every element in s and unions the results. For Rosette to execute Coq terms written against this interface, the Coq terms must be extracted to Racket as shown at the bottom of the figure.

Rosette's syntax), but the extractions for particular functions can be overridden. SEARCHSPACE does this for the functions listed in Fig. 10; each set-construction function is translated to an equivalent term that uses Rosette's assertions and symbolic booleans. The *search* function, meanwhile, is extracted to a call to Rosette's `search` function.

SEARCHSPACE supports extraction of search spaces to both *concrete* and *symbolic* expressions in Rosette. When the SEARCHSPACE functions are extracted to concrete expressions, spaces are enumerated explicitly; for instance, $bind(s,f)$ is extracted to `flatten (map` f s`)`. When they are extracted to symbolic expressions, their enumeration is performed symbolically, by the underlying SMT solver. Concrete extraction is best suited for small to medium-sized search spaces s, such as all routers in a network, and complex functions f, such as checking whether a policy holds for a router, enabling trivial parallelization by distributing the application of f with elements in s to nodes in a cluster. Symbolic extraction, in contrast, works best for simple functions f (such as $\lambda n.n \neq 892412$) applied to large search spaces s (such as the 32 bit integers). By extracting a program's outer binds with medium-sized search spaces to concrete expressions, and extracting an algorithm's inner binds with large search spaces to symbolic expressions, users of SEARCHSPACE can get the best of both worlds: algorithms that explore large search spaces using both parallelism and solver technology.

Fig. 11 shows an example of a simple program in the SEARCHSPACE DSL. The program verifies the idempotence of the XOR operation on boolean values. The *bind* operations can be extracted to either concrete or symbolic expressions.

```
$enumerate_{bool} : Space(bool)$
$enumerate_{bool} := union(single(true), single(false))$

$idempotent : Space(bool \times bool)$
$idempotent :=$
  $x \leftarrow enumerate_{bool}$; $y \leftarrow enumerate_{bool}$;
  if $(x \oplus y \oplus y = x)$ then $empty$ else $single(x, y)$

$counterExample : (bool \times bool) \cup \perp$
$counterExample := search(idempotent)$
```

Figure 11. SEARCHSPACE Example. *idempotent* is the *Space* which contains all examples for which the XOR operation on boolean values is *not* idempotent. When the space is passed to Rosette via *search*, the result is $\perp$ because there is no counterexample for the idempotence of XOR. The notation $x \leftarrow s; f$ means $bind(s, \lambda x.f)$.

6. BGPV and the Initial Network Reduction

This section describes BGPV, the algorithm at the core of Bagpipe that checks whether a specification τ holds ($specHolds(\tau)$). Intuitively, BGPV works as follows: it enumerates network paths, enumerates the set of update messages that could have been forwarded along those paths in the AS in the *initial network* in which no update messages have been delivered, and ensures that the policy holds for all of these update messages. This defines an enumerable superset of reachable AS states which, as shown in Section 8, does not lead to false positives in practice. We describe BGPV and the initial network reduction in detail below.

To verify that a specification τ holds, it is necessary to show that τ holds in all reachable router states. SEARCHSPACE cannot construct the space of all traces and their associated router states because the set of all traces is infinite. Instead, we require a specification to hold for a specific set of router states which can be enumerated and for which, if τ holds on these states, τ holds on all reachable router states.

We define this specific set of router states based on the insight that each update message in the RIBs of all reachable router states must have been forwarded through the network, being accepted by all import and export policies along the way. In fact, each update message in the RIBs of all reachable router states must be forwardable in the *initial network*—the network where no update messages have been forwarded before. We define the property *trackingOk*, which captures this insight. Instead of searching the space of all reachable router states, BGPV searches the space of all router states consisting entirely of update messages with *trackingOk*. Because we can decide *trackingOk*, we can enumerate this space.

To decide this property, we extend the type of update messages with "ghost state" which is tracked by our semantics but which would not be tracked in an actual BGP implementation. The ghost state consists of the router from which the message originated, a copy of the original message before any modifications, and a record of each connection the message traversed on its way. The ghost state is used to ensure

```
$A := (\mathcal{A} \times \mathcal{A} \times list(C)) \cup na$

INR($\tau$) : $Space(R \times R \times R \times P \times A \times A \times A \times A) :=$
  $r \leftarrow enumerate_{R_i}$;  $o \leftarrow enumerate_{out(r)}$;  $p \leftarrow enumerate_P$;
  $(i, a_i) \leftarrow enumerateTrackingOk(r, p)$;
  $(i^*, a_i^*) \leftarrow enumerateTrackingOk(r, p)$;
  let  $a_l^* := imp(r, i^*, p, a_i^*), a_l := imp(r, i, p, a_i)$,
       $a_o := exp(r, i^*, o, p, a_l)$  in
  if  $pref(a_l) \leq pref(a_l^*) \wedge \neg\tau(r, i, o, p, a_i, a_l^*, a_o))$
  then  $single(r, i, o, p, a_i, a_i^*, a_l, a_o)$  else  $empty()$

$enumerateTrackingOk(r, p) : Space(in(r) \times A)$  :=
  $union(bind(enumerate_{in(r)}, \lambda i.single(i, na))$,
    $a_0 \leftarrow enumerate_{\mathcal{A}}$;
    $r_i \leftarrow enumerate_{R_i}$;  $r_o \leftarrow enumerate_{\mathcal{N}_{r_i}}$;
    let  $P$  :=  if  $r = r_i$  then  $[(injected, r_o), (r_o, r)]$
                  else  $[(injected, r_o), (r_o, r_i), (r_i, r)]$  in
    let  $a_c$  :=  $transmit(P, p, a_0)$  in
      $single($if  $a_c = na$  then  $na$
             else  $(last(P), (a_c, a_0, P))))$
```

Figure 12. Initial Network Reduction. BGPV extends attributes A with "ghost state" consisting of the router from which the message originated, a copy of the original attributes before any modifications, and the path the message traversed on its way through the network. If INR(τ) is the empty *Space*, then $specHolds(\tau)$. $search$(BGPV(τ)) is thus a semi-decision procedure for *specHolds*. *enumerateTrackingOk* is a *Space* that contains all attributes with prefix p and correct tracking to the router r.

that only messages with valid paths through the network are considered.

Besides simplifying the decision of *trackingOk*, adding the ghost state to update messages has the additional advantage that specifications can reference it. For instance, specifications can constrain the ghost state to ensure that no update message from a particular neighbor is exported from the AS. BGPV exposes the following methods to specifications: $original((\_, a_0, \_)) := a_0$ (the original attributes before any modifications), and $origin((\_, \_, (injected, r) :: \_)) := r$ (the router that originated the update message).

The *Initial Network Reduction* (INR(τ) in Fig. 12) is a function from policies to a space of counterexamples. $search$(INR(τ)) is thus a semi-decision procedure for $specHolds(\tau)$. Several sets that are universally quantified in *specHolds* are finite: r, p, i, and o. Since the ghost state includes a list of hops, however, the attributes a_i, a_l, and a_o might initially appear to be infinite. However, there are only a finite number of paths with correct tracking, since the AS is guaranteed to be in a full-mesh configuration and therefore cannot introduce loops in the path. The function *enumerateTrackingOk*, shown in Fig. 12, enumerates the attributes with correct tracking. Below, we prove that this function correctly enumerates the attributes with correct tracking:

Theorem 1. *enumerateTrackingOk enumerates all attributes a_i with correct tracking.*

Proof. An attribute a_i with correct tracking is either na or (a_c, a_0, P). *enumerateTrackingOk* enumerates na in the left hand side of *union*. *enumerateTrackingOk* enumerates

(a_c, a_0, P) in the right hand side of the *union*, by enumerating all attributes a_0 and all correct paths P. a_c can be computed by transmitting a_0 along P. All attributes a_0 can be enumerated because they are finite. All correct paths P can be enumerated, since they are at most four hops long. Paths longer than four hops must contain a forwarding loop because the AS under consideration is in a full-mesh configuration. Forwarding loops in an AS, however, are impossible: an internal router does not forward update messages received from an internal router to an internal router (see restriction 3 in Section 3.1). Therefore, *enumerateTrackingOk* will eventually enumerate all attributes a_i with correct tracking. □

Because the INR enumerates a superset of all reachable router states, we can lift the result of $\mathit{search}(\text{INR}(\tau))$ to $\mathit{specHolds}(\tau)$.

Corollary 2. *(soundness) For all policies τ, if* BGPV(τ) *is empty then* $\mathit{specHolds}(\tau)$*.*

7. Bagpipe Checker Implementation

This section describes the implementation of the Bagpipe checker. At its core is an extraction of BGPV to Racket. Rosette is used to implement the SEARCHSPACE interface, and thus to execute BGPV efficiently. Users of Bagpipe can choose which variables to keep symbolic and which ones to enumerate for parallelism. Predicate abstraction on attributes reduces the search space size.

In the semantics discussed in Section 3, routers are configured with rules governing their behavior on incoming and outgoing announcements (*imp* and *exp* rules). In practice, these rules are written in router configuration languages; most real-world routers use languages developed Juniper and Cisco. To enable verification of these real-world configurations, the Bagpipe checker includes a parser for Juniper-formatted configurations. When verifying a policy against a configuration, Bagpipe runs the parser on the given configuration, returning *imp* and *exp* rules that can be used by the BGPV algorithm. Note that the Juniper parser is unverified and therefore trusted; verifying this component would be a useful avenue for future work.

8. Evaluation

In this section, we evaluate several components of Bagpipe. Our goal is to answer the following questions: 1) Are our BGP semantics correct, and are they useful in applications other than Bagpipe? 2) What were the benefits of verifying Bagpipe? 3) Can the Bagpipe checker efficiently check AS specifications, and does it return false positives?

8.1 Evaluating the BGP Semantics

We evaluated our BGP semantics by randomized differential testing [10] against C-BGP [29], a popular open-source BGP simulator. In randomized differential testing, randomly generated test cases are run in multiple versions of a piece of software and their output is compared; differences indicate possible bugs. Our application of randomized differential testing both increases confidence in the semantics' correctness and demonstrates the semantics' broader utility by identifying a bug in C-BGP.

To test our semantics against C-BGP, we developed a program in Coq, based on our semantics, that checks sequences of BGP events (deliveries of update or withdraw messages) to ensure that they are permitted by our semantics of the BGP specification. We then used Coq's extraction facility to extract this program to Haskell. We wrote a Haskell test harness, using the QuickCheck random testing tool [7], which generates random configurations and topologies and runs C-BGP to generate a trace. The test harness then calls the extracted Coq program to ensure that the trace is permitted under our semantics.

We ran this differential testing tool 8,401 times, on randomly generated topologies of moderate size and configurations of moderate complexity. 8317 (98.95 %) test cases resulted in agreement between C-BGP and the semantics. 83 test cases (1.00 %) were parse errors, indicating issues with our parser for C-BGP traces. 4 test cases (0.05 %) resulted in different BGP behavior between C-BGP and our semantics. We manually inspected these test cases, and found that all four resulted from a bug in C-BGP: routers sometimes send update messages even when the routes they are advertising have not changed, in violation of Section 9.2 of the BGP specification. We reported this bug, and it was acknowledged by the C-BGP maintainer.

The fact that C-BGP and our semantics agree on most test cases provides evidence that our semantics correctly reflect the real world. The fact that our semantics were used to find a bug in C-BGP indicates that they can be used for applications outside Bagpipe.

8.2 Evaluating SearchSpace

We implemented a prototype of the Bagpipe checker before attempting to verify it. This implementation implicitly relied on the initial network reduction, but without a rigorous definition we were not confident in its correctness. During the verification process, we identified several bugs. Specifically, our prototype checker did not verify specifications for na announcements in the `adjRIBsIn`, and duplicated the router r in the path of update messages that enter the AS at router r and then exit the AS without being forwarded within the AS.

As another benefit of the verification effort, we were able to formalize our intuitions about BGP behavior. For example, formalizing "maximal behavior in the initial network" as the initial network reduction, and proving its soundness, provided us with an understanding of the conditions under which the initial network reduction is sound but not complete: false positives are possible when a specification depends on relationships between a router's `adjRIBsIn` and `locRIB`.

Policy	Time	Searches	Issues
$noMartian$	1,178s (20min)	3,114	0
$noMartian$ (no checks)	1,194s (20min)	3,114	N/A
$blockToExternal_C$	28,594s (8h)	115,330	5
$goodPreference_{\sqsubseteq}$	260,790s (72h)	971,680	14

Figure 13. Overview of the Internet2 case studies. "Searches" is the number of calls to Rosette's search function (i.e., the number of paths under consideration). "Issues" is the number of specification violations found. Bagpipe does not issue false positives on any experiment we ran.

The BGP specification consists of 362 lines of Coq code. The implementation and verification of BGPV consists of 3781 lines of code.

8.3 Bagpipe Policy Verification

To evaluate Bagpipe, we verified specifications about Internet2. We ran Bagpipe on Amazon EC2 with 2 instances of type `c3.8xlarge`, each with 32 virtual-cores and 60 GB of memory. Figure 13 summarizes the results of our evaluation. The experiments ran for a total of 81h, the cost for which is about $30 using EC2 spot instances. For each specification, Figure 13 summarizes the time required to verify the specification, the number of searches that were performed by Rosette (these can be performed in parallel), and the number of import and export rules that violate the specification. Note that Bagpipe did not produce any false positives.

Internet2 is a not-for-profit AS that connects educational, research, and government institutions. Internet2 consists of 10 BGP routers spread throughout the US. Internet2 operates a full mesh network; each internal router is connected to every other internal router. Internet2 is connected to 274 neighboring ASes. Internet2's router configurations total 100,651 lines of Juniper configuration code [17], which amounts to 76,448 Juniper commands, of which we support 44,474 (58%). Most of the unsupported commands are not related to BGP. At this scale, it is not feasible to guarantee correctness by manually auditing configurations.

We identified and verified three specifications for Internet2, described in the following paragraphs. These are specializations of the specifications from Fig. 9.

noMartian It takes Bagpipe 1,178s (20min) to verify the $noMartian$ specification for Internet2. Internet2 never imports announcements with a $martian$ prefix. Internet2 implements this specification with a dedicated sanity check in 237 out of the 274 imp rules. This sanity check returns na for all update messages with a $martian$ prefix.

After removing these sanity checks from the configurations, it takes Bagpipe 1,194s (20min) to verify the $noMartian$ specification for Internet2. Because Internet2 performs a large variety of other checks on update messages, the specification continues to hold for 189 neighbors. This result implies that Bagpipe's ability to verify specifications is not only useful to increase confidence in the correctness of router configurations, but also to safely remove unnecessary sanity checks — 152 in this case.

blockToExternal From Internet2's configurations, it appears to be Internet2's policy not to forward an update message to external routers if the update message has the `BLOCK_TO_EXTERNAL` community set. It takes Bagpipe 28,594s (8h) to check the $blockToExternal$ specification for `BLOCK_TO_EXTERNAL`. The configurations of 5 neighbors do not adhere to the $blockToExternal$ specification.

It is unclear whether $blockToExternal$ should hold for these neighbors. One of the neighbors is RouteViews, which contains two identical imp rules, but no exp rule to block update messages with the `BLOCK_TO_EXTERNAL` community. This might be a real bug, caused by a network operator accidentally specifying an imp instead of an exp (which would explain redundant import statements for this router's configuration). It is also possible that this is the intended behavior of Internet2, as RouteViews is an AS that aggregates routing information. We have contacted Internet2 about these violations, but have not yet received a response.

goodPreference From Internet2's pricing structure [18] and configurations, Internet2 appears to operate according to a refined version of the Gao-Rexford model. Internet2 categorizes the relationship with its neighbors either as $customer$ or $peer$. None of Internet2's neighbors is a $provider$. Internet2 refines the usual $goodPreference$ specification by allowing both customers and peers to influence Internet 2's preference of an update message by setting certain communities. For example, a peer can set the `HIGH_PEERS` community to increase an update message's preference. This is formalized by appropriately adapting the $value$ function defined in Fig. 9.

It takes Bagpipe 260,790s (72h) to verify the $goodPreference$ specification. The configurations of 14 neighbors do not adhere to the $goodPreference$ specification. We have contacted Internet2 about these violations, but have not yet received a response.

9. Related Work

We have built on previous work in building and ensuring the correctness of solver-aided tools, as well as in modeling, analysis, and simulation of BGP configurations. In this section, we address related work in those fields. We also briefly discuss software-defined networking.

Solver-aided tools Advances in solver technology, including SMT, SAT, and model-finding, have made solver-aided tools a compelling option in many domains; here, we briefly mention a handful of representative examples. Boogie [21] and related tools [20, 22] enable general purpose verification by compiling verification conditions to SMT queries. Alive [24] and PEC [19] verify compiler optimizations using a solver back end. Batfish [12] verifies data plane properties using a Datalog solver; Vericon [3] verifies policies for software-defined networking controllers using an SMT solver. All of these tools reduce queries in some application domain to queries answerable by some automated solver. Unfortunately, none of these tools come with mechanically-checked proofs that this reduction is sound. Although some solver-aided tools have been formally verified [33], to our knowledge, SEARCHSPACE is the first general framework for verifying these tools.

BGP Modeling The Stable Paths Problem (SPP) is a simplified model of BGP for which many theoretical results have been proven, including that solving SPPs is PSPACE hard [6]. In contrast, our semantics models BGP according to RFC 4271 and is more expressive than SPP; thus, deciding policies for our semantics is at least PSPACE-hard. While many problems related to BGP are PSPACE-hard, semi- deciding specification for a full-mesh single AS (as described in RFC 4271) is not. Bagpipe efficiently checks specifications in this restricted scenario, which are still expressive enough for important policies of a nationwide AS.

Andreas Voellmy [38] used Isabelle/HOL to formalize a simplified model of BGP's operation at the AS level, without modeling the behavior of individual routers or communication within an AS. This model was used to verify one policy for one textbook example configuration.

The Gao-Rexford model [14] has become the standard model for describing relationships between ASes. In this model, ASes have peers, providers, and customers. Peer relationships do not involve compensation; traffic can flow both directions without either AS having to pay the other. ASes pay their providers and are paid by their customers. This influences BGP route announcements: in general, ASes want to route traffic to and from their customers but not through other providers. It also influences routing decisions: ASes would prefer to send traffic through their customers, their peers, and their providers, in that order. Later work by Gao [13] shows that most ASes display behavior consistent with the Gao-Rexford model. Bagpipe's specification language enables verification of Gao-Rexford based policies.

BGP Analysis ***rcc*** [11] is a tool to find bugs in BGP configurations. It attempts to find violations of route validity and path visibility by inferring inter-AS relationships from the configuration itself (the input to the tool is a set of configurations from all of the routers in an AS). Unlike Bagpipe, ***rcc*** does not provide strong guarantees about the checked configurations; there are both false positives and false negatives in the configuration errors flagged by the tool.

Batfish [12] is a Datalog-based network configuration analysis tool. Router configurations and topology descriptions are translated into Datalog facts and routing-table generation is formalized as a set of Datalog rules. Z3 is then used to verify properties of the generated routing-table given a particular set of received BGP announcements. Bagpipe and Batfish make different design decisions, and are thus able to verify different properties. Bagpipe operates entirely at the level of BGP announcements, and verifies its configurations with respect to *any* set of received announcements, whereas Batfish verifies routing-table properties with respect to a particular set of BGP announcements.

The Formally Verifiable Routing (FVR) project [39–41] developed a formal algebra for describing and reasoning about routing policies, and used it to prove convergence or divergence for a variety of intra- and inter-domain BGP topologies and configurations. Rather than focusing on convergence when multiple AS policies interact, Bagpipe allows a single AS to ensure that its interests are expressed by its BGP configuration. Bagpipe and FVR therefore represent complementary research directions.

BGP Simulation C-BGP [29] is a BGP simulator. Given a topology and a set of configurations, it determines how traffic will be routed. Network administrators can use it both for debugging existing problems and for testing potential new configurations. C-BGP and Bagpipe are potentially complementary; a network administrator could test configurations using C-BGP and then verify them using Bagpipe to guarantee that the network is configured to correctly handle any set of received path announcements.

SDN Software defined networking is a new paradigm for local networks in which router configuration is controlled by a single program running on a master router. There has been a large amount of work on verifying the behavior of software-defined networks, including language support [28], model-checking [3, 9], and full formal verification [2, 16]. SDN has thus far not been used to control BGP-speaking border routers, but even if current BGP configuration languages are supplanted by SDN, tools like Bagpipe will still be useful to ensure that configurations respect AS policies.

10. Conclusion

This paper described: the first formal semantics of BGP and a specification language for BGP policies; SEARCHSPACE, a framework for verifying scalable solver-aided tools in Coq; BGPV, an algorithm to check that router configurations correctly implement BGP policies; and the Bagpipe checker, a BGP configuration checking tool built on BGPV.

To validate our BGP semantics, we performed random differential testing against C-BGP, a popular BGP simulator. We found no bugs in our semantics and one bug in C-BGP. To validate the effectiveness of our verified checker, we ran it on the router configurations of Internet2, a nationwide ISP. Bagpipe revealed 19 violations of standard BGP router policies without issuing any false positives.

In future work, we hope to extend Bagpipe to: support other router configuration languages (e.g. Cisco), support additional BGP features (e.g. route reflectors), incrementalize the verification for configuration updates, and use Rosette's synthesis features to automatically generate BGP configurations that implement particular policies. More broadly, we hope to apply the SEARCHSPACE framework to other solver-aided tools.

References

[1] International Telecommunication Union Statistics, 2014.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *POPL 2014*, pages 113–126, New York, NY, USA, 2014. ACM.

[3] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. VeriCon: Towards verifying controller programs in software-defined networks. In *PLDI 2014*, pages 282–293, New York, NY, USA, 2014. ACM. doi: 10.1145/2594291.2594317.

[4] M. Brown. Pakistan hijacks YouTube. `http://research.dyn.com/2008/02/pakistan-hijacks-youtube-1/`, 2008. Accessed: 2015-02-20.

[5] R. Chandra, P. Traina, and T. Li. BGP communities attribute. RFC 1996, Network Working Group, Aug. 1996.

[6] M. Chiesa, L. Cittadini, L. Vanbever, S. Vissicchio, and G. Di Battista. Using routers to build logic circuits: How powerful is bgp? In *Proc. International Conference on Network Protocols (IEEE ICNP 2013)*, 2013.

[7] K. Claessen and J. Hughes. QuickCheck: A lightweight tool for random testing of Haskell programs. In *ICFP 2000*, pages 268–279, Montreal, Canada, Sept. 2000.

[8] J. Cowie. China's 18-minute mystery. `http://research.dyn.com/2010/11/chinas-18-minute-mystery/`, 2010. Accessed: 2015-03-21.

[9] M. Dobrescu and K. Argyraki. Software dataplane verification. In *NSDI 2014*, pages 101–114, Berkeley, CA, USA, 2014. USENIX Association.

[10] R. B. Evans and A. Savoia. Differential testing: A new approach to change detection. In *ESEC-FSE 2007*, pages 549–552, New York, NY, USA, 2007. ACM. doi: 10.1145/1295014.1295038.

[11] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI 2005*, pages 43–56, Berkeley, CA, USA, 2005. USENIX Association.

[12] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *NSDI 2015*, Oakland, CA, May 2015. USENIX Association.

[13] L. Gao. On inferring autonomous system relationships in the Internet. *IEEE/ACM Trans. Netw.*, Dec. 2001.

[14] L. Gao and J. Rexford. Stable Internet routing without global coordination. In *SIGMETRICS 2000*, SIGMETRICS '00, pages 307–317, New York, NY, USA, 2000. ACM. ISBN 1-58113-194-1. doi: 10.1145/339331.339426. URL `http://doi.acm.org/10.1145/339331.339426`.

[15] S. Goldberg. Why is it taking so long to secure Internet routing? *Queue*, 12(8):20:20–20:33, Aug. 2014. ISSN 1542-7730. doi: 10.1145/2668152.2668966. URL `http://doi.acm.org/10.1145/2668152.2668966`.

[16] A. Guha, M. Reitblatt, and N. Foster. Machine-verified network controllers. In *PLDI 2013*, PLDI '13, pages 483–494, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2014-6. doi: 10.1145/2491956.2462178. URL `http://doi.acm.org/10.1145/2491956.2462178`.

[17] Internet2 Configurations. Internet2 configurations. `http://vn.grnoc.iu.edu/Internet2/configs/configs.html`.

[18] Internet2 Fees. Internet2 fees. `http://www.internet2.edu/about-us/membership/#membership-dues`.

[19] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 327–337, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-392-1. doi: 10.1145/1542476.1542513. URL `http://doi.acm.org/10.1145/1542476.1542513`.

[20] S. K. Lahiri, S. Qadeer, and Z. Rakamaric. Static and precise detection of concurrency errors in systems code using smt solvers. In *Computer Aided Verification (CAV '09)*. Springer Verlag, February 2009. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=80360`.

[21] K. R. M. Leino. This is Boogie 2. Technical report, Microsoft Research, June 2008. URL `http://research.microsoft.com/apps/pubs/default.aspx?id=147643`.

[22] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *Proceedings of the 16th International Conference on Logic for Programming, Artificial Intelligence, and Reasoning*, LPAR'10, pages 348–370, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-17510-4, 978-3-642-17510-7. URL `http://dl.acm.org/citation.cfm?id=1939141.1939161`.

[23] G. Li and G. Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 187–196, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-791-2. doi: 10.1145/1882291.1882320. URL `http://doi.acm.org/10.1145/1882291.1882320`.

[24] N. P. Lopes, D. Menendez, S. Nagarakatte, and J. Regehr. Provably correct peephole optimizations with Alive. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 22–32, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737965. URL `http://doi.acm.org/10.1145/2737924.2737965`.

[25] D. Madory. Chinese routing errors redirect Russian traffic. `http://research.dyn.com/2014/11/chinese-routing-errors-redirect-russian-traffic/`, 2014. Accessed: 2015-02-20.

[26] D. McConnell. Chinese company 'hijacked' U.S. web traffic. `http://www.cnn.com/2010/US/11/17/websites.chinese.servers/`, 2010. Accessed: 2015-03-21.

[27] D. Meyer, J. Schmitz, and C. Alaettinoglu. Application of routing policy specification language (RPSL) on the Internet, 1997.

[28] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. In *POPL 2012*, pages 217–230, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.

2103685. URL `http://doi.acm.org/10.1145/2103656.2103685`.

[29] B. Quoitin and S. Uhlig. Modeling the routing of an autonomous system with C-BGP. *Netwrk. Mag. of Global Internetwkg.*, 19(6):12–19, Nov. 2005. ISSN 0890-8044. doi: 10.1109/MNET.2005.1541716. URL `http://dx.doi.org/10.1109/MNET.2005.1541716`.

[30] Y. Rekhter, T. Li, and S. Hares. A Border Gateway Protocol 4 (BGP-4). RFC 4271, Network Working Group, Jan. 2006.

[31] M. Said, C. Wang, Z. Yang, and K. Sakallah. Generating data race witnesses by an SMT-based analysis. In *Proceedings of the Third International Conference on NASA Formal Methods*, NFM'11, pages 313–327, Berlin, Heidelberg, 2011. Springer-Verlag. ISBN 978-3-642-20397-8. URL `http://dl.acm.org/citation.cfm?id=1986308.1986334`.

[32] D. Slane. 2010 report to Congress of the U.S.–China Economic and Security Review Commission, Sept. 2010.

[33] Z. Tatlock and S. Lerner. Bringing extensibility to verified compilers. In *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Toronto, Canada, June 2010.

[34] E. Torlak and R. Bodik. Growing solver-aided languages with Rosette. In *Onward!*, 2013.

[35] E. Torlak, M. Vaziri, and J. Dolby. MemSAT: Checking axiomatic specifications of memory models. In *Proceedings of the 31st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '10, pages 341–350, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0019-3. doi: 10.1145/1806596.1806635. URL `http://doi.acm.org/10.1145/1806596.1806635`.

[36] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California fault lines: Understanding the causes and impact of network failures. In *Proceedings of the ACM SIGCOMM 2010 Conference*, SIGCOMM '10, pages 315–326, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0201-2. doi: 10.1145/1851182.1851220. URL `http://doi.acm.org/10.1145/1851182.1851220`.

[37] R. Uhler and N. Dave. Smten with satisfiability-based search. *SIGPLAN Not.*, 49(10):157–176, Oct. 2014. ISSN 0362-1340. doi: 10.1145/2714064.2660208. URL `http://doi.acm.org/10.1145/2714064.2660208`.

[38] A. R. Voellmy. Proof of an interdomain policy: a load-balancing multi-homed network. In *Proceedings of the 2nd ACM workshop on Assurable and usable security configuration*, SafeConfig '09, pages 37–44, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-778-3. doi: http://doi.acm.org/10.1145/1655062.1655071. URL `http://doi.acm.org/10.1145/1655062.1655071`.

[39] A. Wang, L. Jia, C. Liu, B. T. Loo, O. Sokolsky, and P. Basu. Formally verifiable networking. In *Proceedings of the 8th ACM Workshop on Hot Topics in Networks (HotNets-VIII)*, New York City, NY, Oct. 2009.

[40] A. Wang, L. Jia, W. Zhou, Y. Ren, B. T. Loo, J. Rexford, V. Nigam, A. Scedrov, and C. Talcott. FSR: Formal analysis and implementation toolkit for safe inter-domain routing, 2011.

[41] A. Wang, C. Talcott, L. Jia, B. Loo, and A. Scedrov. Analyzing BGP instances in Maude. In R. Bruni and J. Dingel, editors, *Formal Techniques for Distributed Systems*, volume 6722 of *Lecture Notes in Computer Science*, pages 334–348. Springer Berlin Heidelberg, 2011. ISBN 978-3-642-21460-8. doi: 10.1007/978-3-642-21461-5_22. URL `http://dx.doi.org/10.1007/978-3-642-21461-5_22`.