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 configuration languages 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 \textsc{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 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.
- \textsc{SearchSpace}, a framework for verifying scalable solver-aided tools in Coq. We describe the design of \textsc{SearchSpace} and how we applied it to formally verify the Bagpipe checker that implements \textsc{BGPV}.
- \textsc{BGPV}, an algorithm to check that router configurations correctly implement BGP policies. \textsc{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 \textsc{BGPV}, thus highlighting the value of using \textsc{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 \textsc{SearchSpace} and describes the translation to Rosette. Section 6 presents \textsc{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 \textsc{blockToExternal} specification, which ensures that private routes are not leaked to other ASes.

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 \textsc{blockToExternal} specification. An AS consists of internal routers that it controls, connected to external routers that it does not. The \textsc{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 \textsc{BTE} community when it entered Internet2’s AS. For example, if internal router \( r_0 \) receives an update message tagged with the \textsc{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 \textsc{blockToExternal} specification. At the core of Bagpipe is the \textsc{BGPV} algorithm summarized in Fig. 2. \textsc{BGPV} either returns a proof that the policy being checked holds or a counterexample showing a violation of the policy. To check the policy, \textsc{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
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 verifica-

### 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 verifica-

#### 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, while a more complex configuration might allow only certain messages to be processed, modified, and forwarded. 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.
The INJ rule injects an arbitrary message $m$ into router $r$. The INJ rule invokes handle$_r$, which returns $r$’s new state $\sigma'$ and a list of messages for each of $r$’s neighbors $\Gamma'$. 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 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.

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 $C \rightarrow \text{list}(M)$, 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 $A$. Our BGP semantics are parametric over $A$, but requires some projections from $A$, 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.

![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 $(\lambda k'. if k = k' then v else m(k'))$.](image)
handle_r : in(r) → M → S(r) → ((C → list(M)) × S(r))
handle_r(i, (p, a), (σ, σ₁, σ₀)) :=
  let σ' := σ₁[i, p := a]
  i' := dec(r, i, imp(r, i, p, σ'[i, p]))
  a' := imp(r, i', p, σ'[i', p])
  σ'₀ := σ₀[p := a']
  σ'ₙ := σ[n := a']
  exp := e.exp(r, i', a, p, a')
  Γ := λ(s, d). if s = r ∧ σ₀[d, p] = σ₀'[d, p]
       then [(p, σ₀'[d, p])] else []
in (Γ, σ', σ₀', σ₀')

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 adjRIBIn σ. Second, r imports all attributes in its adjRIBIn, choosing the best attribute a' from neighbor i' and storing it in the locRIB σ. Third, r exports a" to all its neighbors, storing the result in its adjRIBOut.

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 used 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 actions are executed by handle: if adjRIBOut(o, p) is already na (e.g. in the empty network), no messages will be sent because adjRIBOut(o, p) does not change; otherwise na is sent to o, rewriting any previous update messages.

handle also updates a router r’s state σ. σ consists of three Routing Information Bases (RIBs) σ₀, σ₁, σ₀. The adjRIBIn, σ₁, contains the most recently received update message for each neighbor and prefix. The locRIB, σ₀, contains the most recently selected update message for each prefix; σ₁ is the routing table used to forward actual packets in the data plane. The adjRIBOut, σ₀, 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. ∀r i p. imp(r, i, p, na) = na
2. ∀r i o p. exp(r, i, o, p, na) = na
3. ∀r i p. a. md(r, i, p) = md(r, o) = ibgp → exp(r, i, o, p, a) = na
4. ∀r i o p. a. asn(r) ∈ path(a) → imp(r, i, p, a) = na
5. ∀r i o p. a. md(r, o) = ebgp → path(exp(r, i, o, p, a)) = asn(r) :: path(a)
6. ∀r i. a. pref[σ[i]] = pref[σ[dec(r, σ[i])]]

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. 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 × R × R × P × A × A × A → bool)

specHolds : spec → Prop

specHolds(τ) :=
 ∀Γ (t : reachable(Γ, Σ)) r p (i ∈ in(r)) (o ∈ out(r)),
 1 let σr, σi, σo := Σ[r]
 2 ai := σi[p], ao := σo[0, p]
 3 in τ(r, i, o, p, a, ai, ao) = true

Figure 8. Specification Definition. A specification τ is an invariant
over BGP behavior. τ is a boolean predicate over the attributes ai,
ai, and ao 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 (ai, ao) of attributes received,
selected, and sent (respectively) for prefix p from i
through r to o, τ(r, i, o, p, ai, ao) = true.

Note that attributes ai, ai, and ao 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 expressiv-
ness in practice as they capture the behaviors of configuration
rules imp, dec, and exp that network operators write to imple-
ment their policies. Furthermore, our restriction on specifi-
cations 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 ai are
selected depending on the received attributes ai, but cannot
constrain which attributes ao 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(Γ, Σ) 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 consider-
ation is given by the set of routers internal to the AS Ri and
the set of neighboring external routers Ni that each internal
router r is connected to. As required by the BGP specifi-
cation, 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 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 ∈ R. 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 relations: customers, peers,
and providers. 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
∈ relation. The goodPreference specification ensures that
for every router r, the update message installed in the r’s
routing table σr for a certain prefix p is always better than or
equal to (∈) all update messages received by r for prefix p.

3 Some large ASes avoid the performance penalty of a full-mesh configu-
ration 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.
This section describes verifying solver-aided tools in Coq. Solver-aided tools written in Coq’s built-in extraction facility translates user-provided specifications 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 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 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 as takes an 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

\[
\text{noMartian}(\_ \rightarrow p \rightarrow a_1, \_) : \text{spec} := \\
p \in \text{martian} \rightarrow a_1 = na
\]

\[
\text{blockToExternal}(\_ \rightarrow a_0, \_ \rightarrow a_0) : \text{spec} := \\
C \in \text{communities}(\text{original}(a_0)) \rightarrow a \in R_e \rightarrow a_0 = na
\]

\[
\text{goodPreference}(r, i, p, a, a_1): \text{spec} := \\
\text{imp}(r, i, p, a) \subseteq a_1
\]

\[
a \subseteq a' = \text{value}(a) \leq \text{value}(a')
\]

\[
\text{value}(a) := \\
\quad \text{if } a = na \text{ then } 0 \\
\quad \text{else let } r = \text{relation}(\text{origin}(a)) \text{ in } \\
\quad \quad \text{if } r = \text{provider} \text{ then } 1 \text{ else } \\
\quad \quad \text{if } r = \text{peer} \text{ then } 2 \text{ else } \\
\quad \quad \text{if } r = \text{customer} \text{ then } 3 \text{ else } \text{undefined}
\]

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 customers, peers, and providers.

The 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.
enumenrate\textsubscript{bool} : \textit{Space\{bool\)}
enumenrate\textsubscript{bool} := \textit{union\{single\{true\}, single\{false\)}

idempotent : \textit{Space\{bool \times bool\)}
idempotent :=
x := enumenrate\textsubscript{bool}; y := enumenrate\textsubscript{bool};
if \((x \oplus y \oplus y = x)\) then empty else single\{(x, y)\}

counterExample : \{(bool \times bool) \cup \bot\}
counterExample := search(idempotent)

6. BGPV and the Initial Network Reduction

This section describes BGPV, the algorithm at the core of Bagpipe that checks whether a specification \(\tau\) holds (\textit{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 \(\tau\) holds, it is necessary to show that \(\tau\) holds in all reachable router states. \textit{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 \(\tau\) holds on these states, \(\tau\) 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 \textit{initial network}—the network where no update messages have been forwarded before. We define the property \textit{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 \textit{trackingOk}. Because we can decide \textit{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

\[ A := (A \times A \times \text{list}(C)) \cup na \]

\(\text{inr}(\tau) : \textit{Space}\{(R \times R \times R \times P \times A \times A \times A \times A) := \}
\tau \leftarrow \text{enumerate}_R; \quad o \leftarrow \text{enumerate}_{\text{out}(\tau)}; \quad p \leftarrow \text{enumerate}_P;\]
\((i, a_i) \leftarrow \text{enumerate}_{\text{TrackingOk}}(r, p);\)
\((i^*, a^*_i) \leftarrow \text{enumerate}_{\text{TrackingOk}}(r, p);\)
\(\text{let } a^*_i := \text{imp}(r, i^*, p, a^*_i); a_i := \text{imp}(r, i, p, a_i);\)
\(a_o := \exp(r, i^*, o, p, a_i) \text{ in}\)
\(\text{if } \text{pref}(a_i) \not\leq \text{pref}(a^*_i) \land \tau(r, i, o, p, a_i, a^*_i) \text{ then}\)
\(\text{single}(r, i, o, p, a_i, a^*_i, a_o, a_i) \text{ else empty()};\)
\[
\text{enumerate}_{\text{TrackingOk}}(r, p) := \text{Space}\{(in(r) \times A) := \}
\text{union}\{(\text{bind}(\text{enumerate}_A(r), \lambda x. \text{single}(i, na)),\)
\quad a_0 \leftarrow \text{enumerate}_A;\)
\quad r_i \leftarrow \text{enumerate}_R; \quad r_o \leftarrow \text{enumerate}_N;\)
\quad \text{let } P := \text{if } r = r_i \text{ then } [(\text{injected}, r_o), (r_0, r)]\)
\quad \text{else } [(\text{injected}, r_o), (r_0, r), (r_i, r)] \text{ in}\)
\(\text{let } a_o := \text{transmit}(P, p, a_o) \text{ in}\)
\(\text{single}(\text{if } a_o = na \text{ then } na \text{ else } (\text{last}(P, (a_c, 0, 0)))),\)

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

Besides simplifying the decision of \textit{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(\((\lambda, a_0, \_)) := a_0\) (the original attributes before any modifications), and
original(\((\lambda, \_)(\text{injected}, r) :: \_)) := r (the router that originated the update message).

The \textit{Initial Network Reduction} (\textit{INR} in Fig. 12) is a function from policies to a space of counterexamples. \textit{SearchINR} is thus a semi-decision procedure for \textit{specHolds}. Several sets that are universally quantified in \textit{specHolds} are finite: \(r, p, i,\) and \(o.\) Since the ghost state includes a list of hops, however, the attributes \(a_i, a_o,\) 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 \textit{enumerateTrackingOk}, shown in Fig. 12, enumerates the attributes with correct tracking. Below, we prove that this function correctly enumerates the attributes with correct tracking:

\textbf{Theorem 1.} \textit{enumerate\textsubscript{TrackingOk}} enumerates all attributes \(a_i\) with correct tracking.

\textbf{Proof.} An attribute \(a_i\) with correct tracking is either \(na\) or \((a_c, 0, 0).\) \textit{enumerate\textsubscript{TrackingOk}} enumerates \(na\) in the left hand side of \textit{union.} \textit{enumerate\textsubscript{TrackingOk}} enumerates
We evaluated our BGP semantics by randomized differential testing. 8,401 times, on randomly generated topologies of moderate size and configurations of moderate complexity. 83 test cases (98.95 %) 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 adjRIBin, 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 adjRIBin and locRIB.
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 import rules, but no export rule to block update messages with the blockToExternal community set. This might be a real bug, caused by a network operator accidentally specifying an import instead of an export (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 highPeers 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 SAT, SMT, 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


