Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver

Konstantin Weitz   Doug Woos   Emina Torlak
Michael D. Ernst   Arvind Krishnamurthy   Zachary Tatlock

University of Washington, USA

{weitzkon, dwoos, emina, mernst, arvind, ztatlock}@cs.washington.edu

Abstract

Internet Service Providers (ISPs) use the Border Gateway Protocol (BGP) to announce and exchange routes for delivering packets through the internet. ISPs must carefully configure their BGP routers to ensure traffic is routed reliably and securely. Correctly configuring BGP routers has proven challenging in practice, and misconfiguration has led to worldwide outages and traffic hijacks.

This paper presents Bagpipe, a system that enables ISPs to declaratively express BGP policies and that automatically verifies that router configurations implement such policies. The novel *initial network reduction* soundly reduces policy verification to a search for counterexamples in a finite space. An SMT-based symbolic execution engine performs this search efficiently. Bagpipe reduces the size of its search space using predicate abstraction and parallelizes its search using symbolic variable hoisting.

Bagpipe's policy specification language is expressive: we expressed policies inferred from real AS configurations, policies from the literature, and policies for 10 Juniper TechLibrary configuration scenarios. Bagpipe is efficient: we ran it on three ASes with a total of over 240,000 lines of Cisco and Juniper BGP configuration. Bagpipe is effective: it revealed 19 policy violations without issuing any false positives.

Categories and Subject Descriptors   D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords   Bagpipe, BGP, domain-specific language, solver-aided languages, correctness

1. Introduction

Over 3 billion people are connected to the Internet through university and corporate networks, regional ISPs, and nation-wide ISPs [18]. These networks, collectively known as Autonomous Systems (ASes), use the Border Gateway Protocol (BGP) to exchange route announcements, which describe paths that traffic can take across the Internet. To route traffic reliably and securely, ASes must configure their BGP-speaking border routers to implement policies restricting how route announcements can be used and exchanged.

Router misconfigurations are common and have led to many visible failures [8, 35, 28, 27]. For example, in 2008, in response to a government order, the Pakistan Telecom AS intended to block YouTube by announcing a non-existent YouTube route to ASes within Pakistan. Due to a misconfiguration, this route was also advertised to an AS outside Pakistan, PPCC. PPCC forwarded the route to its neighbors. The non-existent route to YouTube then quickly spread throughout the Internet and was selected for packet forwarding by most ASes. YouTube was then unavailable to most Internet users, as their packets to YouTube were incorrectly forwarded to Pakistan Telecom. About two hours later, PPCC fixed the problem by disconnecting Pakistan Telecom from the Internet [5]. If Pakistan Telecom had correctly implemented its policy to only block YouTube to ASes within Pakistan, or if PPCC had correctly implemented its policy to only import routes that an AS actually owns, this outage could have been avoided.

Other failures could also have been prevented by correctly implementing appropriate BGP policies [15]. However, doing so with little to no tool support is difficult and expensive, particularly since large ASes maintain millions of lines of frequently changing configurations distributed across hundreds of routers [19, 40].

This paper presents Bagpipe[1], which uses automatic verification to prevent router misconfiguration. An AS operator expresses control-plane policies as declarative specifications.

---

[1] Bagpipe is open-source, see http://bagpipe.uwplse.org

Then, Bagpipe verifies that the router configurations satisfy the specifications.

A straightforward implementation of Bagpipe would need to consider infinitely many possible network traces to determine whether a BGP configuration violates a router policy; if no trace is a counterexample to the specification, then the policy holds. The novel *initial network reduction* enables Bagpipe to verify a specification by searching for counterexamples over a finite space of network traces.

While the initial network reduction makes Bagpipe's search space finite, the space is still far too large to permit brute-force enumeration. Bagpipe partitions the search space by case splitting on the ranges of some symbolic variables, which enables parallel symbolic exploration using an SMT-based symbolic execution engine [39] to efficiently exploit range information within each partition. Bagpipe also uses predicate abstraction to further reduce the search space, coalescing announcements with the same control flow in the BGP configuration.

Bagpipe is an expressive, efficient, and effective verification tool. Bagpipe's policy specification language is expressive: specifications are rich enough to express policies inferred from real AS configurations, express BGP policies found in the literature (such as the Gao-Rexford model [14] and prefix-based filtering [29]), and express policies for 10 configuration scenarios from the Juniper TechLibrary [22, 4]. Bagpipe is efficient: we applied it to three ASes with over 240,000 lines of BGP configuration written in the Cisco and Juniper configuration language (which Bagpipe supports out-of-the-box). Bagpipe is effective: it revealed 19 policy violations without issuing any false positives.

This paper's contributions include:

- A means of expressing AS-wide policies as declarative specifications (Section 3).
- The initial network reduction, which enables Bagpipe to verify specifications by searching a finite set of network traces (Section 4).
- An efficient verifier based on this reduction, which employs a novel combination of search space partitioning, unused variable omission, symbolic execution, and predicate abstraction (Section 5).
- An evaluation of Bagpipe on 10 configuration scenarios and on 3 real ASes with over 240,000 lines of Cisco and Juniper BGP configuration (Section 6).

2. Overview

This section provides background on BGP, shows example policies that AS operators may need to guarantee, and illustrates how Bagpipe automatically verifies such policies.

2.1 Background

The core of the Internet is a network of routers that forward data packets toward their destinations. Routers learn of a route — a path through other routers to a destination — via

```
1   # Control plane for router r.
2
3   # The following RIBs contain the announcements
4   # received, selected, and forwarded by the router r.
5   # Initially, no such announcements are available.
6   ∀ n pfx, adjRIBsIn[n][pfx]  = notAvailable
7   ∀ pfx,   locRIB[pfx]        = notAvailable
8   ∀ n pfx, adjRIBsOut[n][pfx] = notAvailable
9
10  # Process incoming announcement ann
11  # from the source src for the prefix pfx.
12  while (src, pfx, ann) = recvUpdateMessage():
13    adjRIBsIn[src][pfx] = ann
14
15    # Import and select announcements.
16    locRIB[pfx] = notAvailable
17    for n in r.neighbors:
18      annImp = IMPORT(r, n, pfx, adjRIBsIn[n][pfx])
19      if better(annImp, locRIB[pfx]):
20        locRIB[pfx] = annImp
21
22    # Export and forward announcements.
23    for n in r.neighbors:
24      annExp = EXPORT(r, n, pfx, locRIB[pfx])
25      if annExp != adjRIBsOut[n][pfx]:
26        adjRIBsOut[n][pfx] = annExp
27        sendAnnBGP(n, pfx, annExp)
```

Figure 1. Simplified BGP Router Implementation. This pseudocode sketches the high-level behavior specified by the BGP standard, RFC 4271 [32]. AS operators can only configure the IMPORT and EXPORT rules. Section 3.1 describes restrictions on the IMPORT and EXPORT programs to ensure conformity with RFC 4271; for example, IMPORT must return notAvailable on announcements with AS routing loops.

the Border Gateway Protocol (BGP). Using BGP, a router selects at most one route ϱ per destination p and, once selected, adds itself to ϱ and forwards the announcement to its neighbors. The router then forwards packets for p to the router from which ϱ was received. The forwarding and selection of routes takes place on the *control plane*. It runs separately and asynchronously from the *data plane*, on which routers forward data packets using the routes selected by the control plane. Bagpipe verifies control plane policies that characterize both (1) which routes may be selected and used by the data plane and (2) which routes a router may forward.

The Internet's routers are owned by *Autonomous Systems* (ASes) such as universities, corporate networks, regional ISPs, and nationwide ISPs; each router is operated by exactly one AS. ASes are identified by globally unique AS numbers. Bagpipe verifies control plane policies for a single AS because AS operators do not control their neighbors' configurations. We call the routers owned and operated by the single AS under consideration *internal*; other routers are *external* and may behave arbitrarily.

Router control planes forward routes via *update messages*, which consist of a *prefix* and an *announcement*.

- A prefix is a set of destination IP addresses. Sets of destinations are represented in Classless Interdomain Routing (CIDR) notation. A set of destinations in CIDR

notation ip/n (e.g., `192.168.1.0/24`) is referred to as a *prefix*, because it contains all IP addresses starting with the same n bits as ip.

- An announcement contains metadata about the route including: *AS-path*, a list of the ASes the announcement has previously traversed which is used for avoiding routing loops as well as serving as a measure of "distance" on the Internet; *communities*, a set of flags which are uninterpreted by the BGP protocol but can be used by ASes to exchange additional information about a route; and *local preference*, a number that influences route selection.

Figure 1 sketches how each Internet router r manages the control plane, as described by the BGP specification, RFC 4271 [32]. r maintains three tables, or *Routing Information Bases*, to track announcements it has received (`adjRIBsIn`), selected for routing packets on the data plane (`locRIB`), and forwarded to its neighbors (`adjRIBsOut`). Initially (lines 6 to 8), all these tables contain only `notAvailable` to indicate that r has not received any announcements. r then enters an infinite loop to process incoming BGP update messages. Each incoming update message (line 12) includes the address *src* of the router that forwarded the update message, the prefix *pfx* that the update message is offering to route to, and the announcement *ann*. Receipt of an update message triggers a three-phase process.

1. r stores the received announcement in its `adjRIBsIn` routing table for announcements for prefix *pfx* from neighbor *src* (line 13).

2. The router applies the configurable `IMPORT` program to each announcement in `adjRIBsIn` for prefix *pfx* and each neighbor n (lines 17 to 18). `IMPORT` may transform or replace the announcement, including returning `notAvailable`.

   The router chooses the best (possibly-transformed) announcement (lines 19 to 20). An announcement with higher local preference is considered `better` than an announcement with lower local preference. If two announcements have the same local preference, other factors like the length of their AS-path are considered. The router stores the best announcement in `locRIB[`pfx`]`, which is used by the data plane to forward packets.

   Since AS operators cannot directly control the `better` test that selects announcements, they influence selection by configuring their `IMPORT` rules to either drop announcements or to modify the local preference so that the `better` test (line 19) will select their desired announcements.

3. For every neighbor, the router applies the configurable `EXPORT` program to the announcement that was selected as best. If the result of `EXPORT` differs from the announcement that was most recently sent to that neighbor, then `EXPORT`'s result is stored in `adjRIBsOut` and forwarded to the neighbor. A router can retract a previously advertised

announcement by announcing `notAvailable` for the route's destination prefix.

To verify an AS configuration, it is not sufficient to check the `IMPORT` and `EXPORT` programs for each router individually. This is because internal routers establish invariants on the announcements they forward to other internal neighbors (e.g., the ISP Internet2 establishes the invariant that its internal routers never send announcements for invalid prefixes). The configurations of those neighbors often rely on such invariants. Thus, the verification task is to establish that the AS configuration correctly implements the policy for all routers in the AS at all times, for all possible sequences of incoming update messages.

Figure 2 illustrates an example network of several ASes. One AS is under consideration with three internal routers r_0, r_1, and r_2. The external router e_0's data-plane is capable of directly delivering packets with destinations in the prefix `128.208.7.0/24`. Note that the data plane delivers packets along a route in the opposite direction from which update messages for that route were forwarded through the control plane. Consider the following example *trace*, i.e.,, sequence of events happening in the AS:

1. The internal router r_0 receives an update message m_0 from the external router e_0, which indicates that e_0 has a route to the prefix `128.208.7.0/24`.
2. r_0 extends m_0's announcement with its own AS number, and selects it to route packets.
3. r_0 forwards the selected announcement in the update message m_1 to its neighbors, including the internal router r_1.
4. r_1 receives the update message m_1 from r_0.[2]
5. r_1 selects the received announcement to route packets.
6. r_1 forwards the selected announcement in the update message m_2 to its neighbors, including the external router e_0.

r_0, r_1, and e_0 can use the selected announcements to forward packets with destinations in the prefix `128.208.7.0/24`: for example, `128.208.7.1` and `128.208.7.42` but not `128.208.8.0`.

2.2 Policies

An AS operator decides on policies that restrict the announcements that the AS's BGP routers' control planes select and exchange. Some policies are used to ensure security properties: for instance, a country's ASes might want to ensure that national traffic is routed within the country [33]. Other policies are used to uphold business contracts, for example, to honor agreements that certain announcements should not be publicly shared [35].

This section uses the *BlockToExternal* policy as a running example. *BlockToExternal* prohibits internal routers from forwarding "classified" announcements to external

---

[2] r_1 does not extend m_1 with its own AS number because it received the announcement from an internal neighbor.
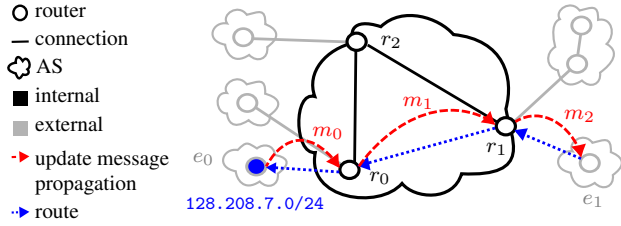
Figure 2. BGP Routing Example. The AS under consideration (black) consists of three internal routers connected to five external routers. These routers use the Border Gateway Protocol (BGP) to establish a route between e_1 and e_0. This route can be used to forward packets.

Figure 3. Bagpipe Workflow. Bagpipe takes a specification and an AS's router configurations as input. Bagpipe verifies that the configuration correctly implements the specification using multiple concurrent SMT solver calls, and then either indicates success or returns a counterexample that AS operators can use to debug their configurations.

routers. An announcement is considered classified if it had the BTE community set *when it was received by the AS*.

Bagpipe's specifications are assertions written in the Racket language that restrict the announcements selected and exchanged by an AS's internal routers. In Bagpipe, the *BlockToExternal* policy is expressed as:

```
(implies
  (external? receiver)
  (not (has-community? 'BTE (original sent))))
```

The policy asserts that if an announcement (*sent*) is forwarded by an internal router to one of its neighboring external routers (*receiver*), then the announcement that gave rise to *sent* (i.e., the original announcement received by the AS before it was modified by import and export filters), must not have contained the BTE community.

AS operators implement policies by configuring their AS's routers. A BGP router is configured with IMPORT and EXPORT programs that modify announcements in order to influence which ones are selected and forwarded (and therefore used to route packets on the data plane). These programs are typically written in either the Juniper or Cisco configuration language, which are loop-free imperative programming languages with domain-specific syntax and semantics.

Consider the following snippet of Juniper configuration code to implement the *BlockToExternal* specification for the neighbor with IP 62.40.125.17.

```
neighbor 62.40.125.17 {             — start neighbor configuration
  export [RULE1 RULE2... BLOCK-BTE...];   — call export rules
  ...}
policy-statement BLOCK-BTE {        — define export rule
  term block-to-external {
    from community BTE;             — match announcement
    then reject; }}                 — reject matched announcement
```

The export statement runs each passed rule (policy-statement) from left to right, stopping once a rule either accepts or rejects the announcement. Each executed rule can also modify the announcement. Note that if the AS operator does not correctly order statements, they may not fire on the announcements they are intended to check or modify. The BLOCK-BTE rule rejects an announcement if it has the BTE community set, and otherwise does nothing.
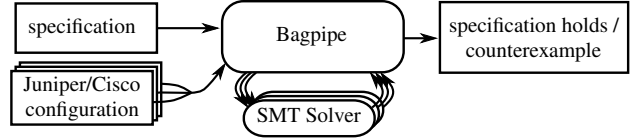
To manually verify that an AS's router configurations implement the *BlockToExternal* specification, an AS operator must check that:

- For every external neighbor, every router has an export rule that drops announcements whose BTE community is set. These rules are similar to the BLOCK-BTE rule in the example above.
- Each of these rules is executed: no preceding rule accepts the announcement. For example, the rules RULE1 and RULE2 in the above example configuration should not accept announcements whose BTE community is set.
- No rule in any router clears the BTE community.

More complex policies, such as the Gao-Rexford policy discussed later in this paper, are even harder to verify manually. Large ASes have millions of lines of configuration, making manual verification of even simple policies expensive and error-prone.

2.3 Bagpipe

As illustrated in Fig. 3, Bagpipe enables AS operators to express their AS-wide policies as declarative specifications, and Bagpipe then *automatically* verifies that their router configurations correctly implement such policies. Since AS operators generally do not have access to their neighboring ASes' configurations, Bagpipe soundly assumes that external routers may exhibit any behavior. Bagpipe verifies control plane policies to ensure that inter-AS guarantees are upheld (e.g., confidential announcements are not leaked) and to prevent control-plane performance issues. For example, ASes often reject update messages whose prefix is too long (corresponding to a too-small set of destinations) in order to avoid filling their routing tables with too many routes and thus degrading performance.[3]

To verify that an AS's routers are safely configured, Bagpipe must ensure that every possible behavior of the AS satisfies the policy. Bagpipe models the behavior of an AS as a *trace* of announcement receipts, route selections,

---

[3] From the data plane perspective, this behavior may seem unreasonable: if an AS would be willing to forward a packet with an IP in the prefix 64.57.29.0/24 to a particular AS, it seems it should also be willing to forward a packet with an IP in the range 64.57.29.0/25 (since the latter represents a strict subset of the former). However, filling RIBs with many announcements for small prefixes can severely degrade performance.

and forwarding events in the network. Bagpipe models the polciy in Racket as a boolean predicate over traces. For example, to verify the *BlockToExternal* specification from Section 2.2, Bagpipe must guarantee that in every trace, no internal router forwards a classified announcement. Note that Bagpipe verifies policies even for oscillating and non-deterministic BGP networks; thus, *the guarantees provided by Bagpipe hold even outside a BGP network's steady state.*

To verify a policy, it is not feasible to brute-force search the space of all possible traces for a counterexample, because the set of all traces is infinite. Not only can external routers send arbitrarily long sequences of announcements, but the BGP protocol itself can oscillate (in which case routers keep announcing and withdrawing routes indefinitely).

To address this challenge, Section 4 introduces the *initial network reduction* (INR). In the *initial network*, all RIBs contain only `notAvailable` (as in Fig. 1 after line 8). Intuitively, the INR exploits the observation that if a router will ever select or forward a particular announcement, it will also do so the initial network. An AS's behavior is thus "maximal" in the initial network. Bagpipe exploits this fact to reduce its search for a counterexample from the infinite set of traces to a finite set of traces that process two arbitrary external announcements in the initial network.

The initial network reduction makes Bagpipe's search space finite, but it is still too large for naive brute force search. Section 5 introduces four optimizations that Bagpipe implements to improve search performance. Instead of using brute-force search, Bagpipe searches the space symbolically using an SMT solver (§5.1). Bagpipe uses predicate abstraction to soundly reduce the number of announcements that must be considered (§5.2). Bagpipe hoists symbolic variables so that search can be parallelized across many nodes in a cluster (§5.3). Bagpipe avoids enumerating hoisted variables whose values are not used (§5.4).

3. Specifications

This section first formalizes the behavior of the BGP control plane in Section 3.1; and then uses the formalization to describe a general framework for expressing BGP policy specifications, and what it means for these specifications to hold in Section 3.2.

3.1 Control Plane Formalization

This section and Figure 4 formalize the BGP control plane discussed in Section 2. A more complete formalization appears in a technical report [41].

The set of routers is represented as a set of IP addresses, connected via bidirectional links over which routers exchange announcements.

A *trace* is a sequence of *event*s within the AS under consideration. There are three kinds of event. The event $recv(r, i, p, a)$ means that a router r received an update message from its neighbor i for prefix p with announcement

$$IP := [0, 255] \times [0, 255] \times [0, 255] \times [0, 255] \quad \text{— ip addresses}$$
$$P := IP \times [0, 32] \quad \text{— prefixes}$$

$$R \subseteq IP \quad \text{— routers}$$
$$R_i \subseteq R \quad \text{— internal routers}$$
$$R_e \subseteq R \quad \text{— external routers}$$
$$in(r) = out(r) = neighbor(r) \subseteq R \quad \text{— router } r\text{'s neighbors}$$

$$event := recv : (r : R) \to in(r) \to P \to A \to event$$
$$\quad | \quad slct : (r : R) \to P \to A \to event$$
$$\quad | \quad frwd : (r : R) \to out(r) \to P \to A \to event$$

$$trace \subseteq list(event) \quad \text{— a trace is a legal sequence of events}$$

$$notAvailable \quad \text{— placeholder used when announcement is not available}$$
$$\mathcal{A} := \{pref : \mathbb{N}, communities : \mathcal{P}(\mathbb{N}), aspath : list(asn)\}$$
$$\quad \text{— BGP announcement; } \mathcal{P} \text{ stands for powerset.}$$
$$A := \{current : \mathcal{A}, original : \mathcal{A}\} \cup \{notAvailable\} \quad \text{— tagged ann}$$

$$imp : (r : R) \to in(r) \to P \to A \to A \quad \text{— IMPORT program of Fig. 1}$$
$$exp : (r : R) \to out(r) \to P \to A \to A \quad \text{— EXPORT program of Fig. 1}$$

$$adjRIBsIn : trace \to (r : R) \to in(r) \to P \to A \quad \text{— active received}$$
$$locRIB : trace \to (r : R) \to P \to A \quad \text{— active selected}$$
$$adjRIBsOut : trace \to (r : R) \to out(r) \to P \to A \quad \text{— active fwded}$$

Figure 4. Control Plane Formalization. A router r is connected via incoming $in(r)$ and outgoing $out(r)$ links. A *trace* is a valid sequence of *event*s within the AS under consideration. imp and exp refer to a router's configurable import and export programs. In any state reachable via some trace t, $adjRIBsIn(t, r, i, p)$ contains the announcement most recently received for a prefix p by a router r from r's neighbor i. $locRIB(t, r, p)$ contains the announcement most recently selected for a prefix p by a router r. $adjRIBsOut(t, r, o, p)$ contains the announcement most recently forwarded for a prefix p by a router r to r's neighbor o.

a. The event $slct(r, p, a)$ means that a router r selected an announcement a for prefix p. The event $frwd(r, o, p, a)$ means that a router r forwarded an update message to its neighbor o for prefix p with announcement a.

A valid trace must have the following 4 properties: 1) A router can receive an update message from an internal neighbor i only if there has previously been a corresponding forwarding event by i. A router can always receive an update message from an external neighbor, because Bagpipe treats external neighbors as "havoc". 2) A router selects an announcement in its $locRIB$ if and only if it is chosen as the result of the import and selection process (shown in Fig. 1 on lines 16 to 20). 3) A router can forward an update message only if its announcement is the result of applying the export process (shown in Fig. 1 on lines 23 to 27) to a selected announcement. 4) The receipt of an update message by a router must be immediately followed by all the resulting select and forward events at that router. The end of Section 2.1 provides an example trace. A network state is *reachable* via some trace t if it is the result of executing every event in the trace t.

$\mathcal{A}$ represents an announcement as specified by the BGP protocol. It is a record consisting of an AS path $aspath$ (the AS numbers of every AS traversed by the announcement), local preference $pref$ (the `better` routine of Fig. 1 prefers announcements with higher local preference), and a set of communities $communities$ (used by ASes to exchange additional information about a route; a community is a number).

Bagpipe can be easily extended to track additional information in announcements, e.g., to model BGP extensions.

For reasoning, Bagpipe uses *tagged announcements* which consist of the *current* value of the announcement, plus the *original* value of the announcement at the time when it entered the AS under consideration. Tracking this additional provenance information enables AS operators to write policy specifications such as *BlockToExternal*.

A router r imports an announcement a_i for prefix p received from neighbor i using the import program $imp(r, i, p, a_i)$. imp is a loop-free imperative program that either modifies or drops (by returning $notAvailable$) the passed announcement. The export program exp is modeled similarly. These programs are called `IMPORT` and `EXPORT` in the BGP specification and in Section 2.

For compliance with RFC 4271, there are some restrictions on how the `IMPORT` and `EXPORT` programs can operate. For example, both programs must map `notAvailable` inputs to `notAvailable` outputs, `IMPORT` must mark announcements with AS routing loops as `notAvailable`, and `EXPORT` must prevent announcements received from internal neighbors from being exported to other internal neighbors.

Each router r stores the most recently received, selected, and forwarded announcements. These are the *active* announcements that can be used to forward packets. Specifically:

- $adjRIBsIn$ contains the active received announcements: the most recently received announcement for each prefix and neighbor of r. A new announcement for some prefix from a neighbor always implicitly withdraws (and thus deactivates) any previously-received announcements for that prefix and neighbor.
- $locRIB$ contains the active selected announcements: the most recently selected announcement for each prefix.
- $adjRIBsOut$ contains the active forwarded announcements: the most recently forwarded announcement for each prefix and neighbor of r. Older routes are implicitly withdrawn by BGP.

Given a trace t, a router r, a neighbor i of r, and a prefix p, the function $adjRIBsIn$ computes the network state resulting from the execution of every event in the trace t, i.e., the state reachable via the trace t, and returns the most recently received announcement by r from i with prefix p in that state, or $notAvailable$ if no announcement has been received. Note that the $adjRIBsIn$ formalism takes more arguments than `adjRIBsIn` of Fig. 1. When applied to a trace t and a router r, it corresponds to `adjRIBsIn`. The $locRIB$ and $adjRIBsOut$ are modeled similarly.

3.2 Policy Specifications

This section describes a general framework for expressing BGP policies specifications, and what it means for these specifications to hold. Later sections show how Bagpipe automat-

$$V := \{router \in R; prefix \in P; sender \in R; received \in A;$$
$$\quad selected \in A; bestSender \in R; bestReceived \in A;$$
$$\quad receiver \in R; sent \in A\}$$

$$spec : Type := V \rightarrow bool$$

$$specHolds(\tau : spec) :=$$
$$\forall (t : trace) (r \in R_i) (p \in P) (i \in in(r)) (o \in out(r)),$$
$$\texttt{let}\ i^* := inLocRIB(t, r, p)$$
$$\quad a_i := adjRIBsIn(t, r, p, i)$$
$$\quad a_i^* := adjRIBsIn(t, r, p, i^*)$$
$$\quad a_l^* := locRIB(t, r, p)$$
$$\quad a_o := adjRIBsOut(t, r, p, o)$$
$$\texttt{in}\ \tau(\{router := r; prefix := p; sender := i; received := a_i;$$
$$\quad selected := a_l^*; bestSender := i^*; bestReceived := a_i^*;$$
$$\quad receiver := o; sent := a_o\}) = true$$

Figure 5. Policy Specification Definition. A policy specification $\tau : spec$ is a predicate over a record of variables V, representing certain active announcements. $specHolds(\tau)$ defines what it means for a specification to hold.

ically verifies that specifications in this general framework hold.

A specification is an invariant over an AS's active announcements, i.e., an invariant over an AS's routing information bases. Formally, a specification τ is a predicate over a record of variables V. V represents certain values in $router$'s routing information bases, namely an announcement $received$ from a $sender$ for $prefix$, an announcement $selected$ for $prefix$ which was received as $bestReceived$ from $bestSender$, and an announcement $sent$ to a $receiver$ for $prefix$.

The definition of a specification $spec$, and what it means for a specification τ to hold $specHolds(\tau)$, is given in Figure 5. A specification τ holds (i.e., an AS's routers correctly implement a specification), if and only if the invariant expressed by τ is true for every router state reachable via any trace. Formally, τ holds if and only if for any network trace t, router r, prefix p, neighbor i, and neighbor o, τ returns $true$ when invoked with the most recently received announcement a_i for prefix p and neighbor i, the most recently selected announcement a_l^* for prefix p which was received as a_i^* from i^* (starred variables are associated with the selected announcement), and the most recently forwarded announcement a_o for prefix p and neighbor o, along with r, p, i, and o.

The function $inLocRIB(t, r, p)$ computes the neighbor i^*. $inLocRIB(t, r, p)$ first passes all announcements in the $adjRIBsIn(t, r, i, p)$ to the imp program, and then determines the neighbor with the "best" imported announcement.

Expressiveness Specifications are not arbitrary invariants over all active announcements — specifications can only quantify over variables in V. In particular, a specification cannot depend on routers other than r, announcements for prefixes other than p, announcements received from neighbors other than i and i^*, and announcements forwarded to neighbors other than o. For example:

- A specification *cannot* require routers to forward a received announcement for prefix p, if and only if an announcement for some other prefix q has been selected.
- A specification *cannot* require routers to select a received announcement from neighbor i, if and only if exactly k (where $k \geq 3$) announcements from other neighbors have been received.
- A specification *can* require routers to select a received announcement with either prefix p or q, if and only if that announcement has the IMPORTANT community set.
- A specification *can* require routers to forward an announcement a_o, if and only if a_o is equal to the selected announcement a_l^*.

As shown in the evaluation, even with these restrictions, Bagpipe still provides sufficient expressiveness for many interesting policies. This is due to two properties of BGP:

1) A BGP router r selects and forwards announcements for a certain prefix p, completely independent of any announcements for any other prefix p' or any other router r' (see Fig. 1). For example, the first disallowed policy above cannot be implemented, because the decision process for announcements of prefix p cannot inspect announcements for prefix q.

2) The *imp* and *exp* programs can only consider one announcement at a time. This restriction is defined by the BGP specification:

> [A router's *imp*/*exp* programs] 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. [32]

For example, the second disallowed policy above cannot be implemented, because an import rule can only consider a single received announcement at a time.

Instead of interpreting a specification as a network state invariant, some (but not all) specifications can also be interpreted as a the composition of an *import specification*, an *export specification*, and a *selection ranking*.

An import specification π_i restricts how routers can import received announcements. Formally, $\pi_i(r, p, i, a_i, a_l) : bool$ holds if and only if for any network trace, π_i returns *true* for any announcement a_i which was received by router r from neighbor i for prefix p, and which was imported as announcement a_l (a_l does not have to be selected). Note that an import specification quantifies only over a single announcement, i.e., an import specification restricts all announcements independently. An export specification π_e restricts how routers can export selected announcements, and is formalized similarly to an import specification. Import and export specifications resemble the domain-specific languages employed by Frenetic [13], NetCore [30], and NetKat [1].

A selection ranking $\leq$ restricts which announcements routers can select. Formally, $\leq (r, p, i, a_i, i^*, a_i^*) : bool$ holds if and only if for any network trace, router r, prefix p, and neighbor i, the announcement a_i received from neighbor i

is ranked lower than the announcement a_i^* received from neighbor i^* (i^* is the neighbor from which r has selected the announcement). In contrast to much related work, the selection ranking is unique because it considers multiple announcements simultaneously.

Many specifications τ can be decomposed into an import specification π_i, an export specification π_e, and a selection ranking $\leq$ as follows:

$$
\begin{aligned}
\tau(v) : spec := \\
\quad \pi_i(router(v), prefix(v), bestSender(v), \\
\qquad bestReceived(v), selected(v)) \land \\
\quad \pi_e(router(v), prefix(v), receiver(v), \\
\qquad selected(v), sent(v)) \land \\
\quad \leq (router(v), prefix(v), sender(v), received(v), \\
\qquad bestSender(v), bestReceived(v))
\end{aligned}
$$

While some specifications cannot be decomposed into this form, e.g., if the specification relates $received(v)$ and $sent(v)$, all specifications that we expressed for the evaluation of Bagpipe can.

The rest of this section describes examples of useful policies, and shows how they are expressed as specifications.

Block To External Specification Section 2 described the *BlockToExternal* specification, which prohibits internal routers of the AS under consideration from forwarding classified announcements to any external routers. An announcement is considered classified if it had the BTE (block to external) community set at the time that it was received by the AS. We can express this specification as:

$$
\begin{aligned}
BlockToExternal(v) := receiver(v) \in R_e \rightarrow \\
BTE \in communities(original(sent(v)))
\end{aligned}
$$

An AS's router configurations correctly implement this specification if and only if $specHolds(BlockToExternal)$. Inlining the definitions of $specHolds$ and $BlockToExternal$, as well as removing unused variables, leads to the formula below, which states that an AS's router configurations correctly implement $BlockToExternal$ if and only if the most recently forwarded announcement a_o in any reachable router state of any internal router r is not classified.

$$
\begin{aligned}
\forall \, (t : trace) \, (r \in R_i) \, (p \in P) \, (o \in out(r)), \\
\quad \textbf{let } a_o := adjRIBsOut(t, r, p, o) \\
\quad \textbf{in } \ o \in R_e \rightarrow BTE \in communities(original(a_o))
\end{aligned}
$$

Note that the *BlockToExternal* specification is an invariant on an AS's *adjRIBsOut*. Further, *BlockToExternal* can be decomposed into an export specification $\pi_e(r, p, o, a_l, a_o) = o \in R_e \rightarrow BTE \in communities(original(a_o))$, and an import specification and selection ranking that always return *true*. We say that *BlockToExternal* is composed of only an export specification.

No Martian Specification The *BlockToExternal* specification restricts which announcements an AS can forward. The *NoMartian* specification restricts which announcements an AS can select.

The *NoMartian* specification prohibits internal routers from selecting a route announcement *selected* for martian

prefixes $prefix$, 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 forward packets over the Internet. Formally:

$$NoMartian(v) := \text{martian}(prefix(v)) \rightarrow selected(v) = notAvailable$$

The specification holds iff $specHolds(NoMartian)$, i.e.,:

$\forall\,(t: trace)\,(r \in R_i)\,(p \in P),$
   `let` $a_l := locRIB(t, r, p)$
   `in` $\text{martian}(p) \rightarrow a_l = notAvailable$

Note that the $NoMartian$ specification is an invariant on an AS's $locRIB$, and is composed of only an import specification.

Gao-Rexford Specification   According to the Gao-Rexford model [14], a widely-used description of AS behavior, there are three kinds of relationship that an AS can have with any of its neighbors: $customer$, $peer$, or $provider$. Customers pay the AS to forward packets, peers neither charge nor pay money to forward packets, and providers charge money to forward packets. To maximize profit, an AS's routers should thus prefer an announcement from (i.e., a route through) a customer over the announcement from a peer or provider, and should prefer the announcement from a peer over the announcement from a provider. This preference can be captured with a relation $<$, where $peer < customer$, $provider < customer$, and $provider < peer$. The $GaoRexford$ specification prohibits a router from selecting an announcement a_i^* that is "worse" than any announcement a_i received from some neighbor i. Given a function $relationship(a)$ that returns the relationship of the neighbor from which a was received, $GaoRexford$ can be defined as follows:

$$GaoRexford(v) := relationship(received(v)) \leq$$
$$relationship(bestReceived(v))$$

The specification holds iff $specHolds(GaoRexford)$, i.e.,:

$\forall\,(t: trace)\,(r \in R_i)\,(p \in P)(i \in in(r)),$
   `let` $i^* := inLocRIB(t, r, p)$
        $a_i := adjRIBsIn(t, r, p, i)$
        $a_i^* := adjRIBsIn(t, r, p, i^*)$
   `in` $relationship(a_i) \leq relationship(a_i^*)$

Note that the $GaoRexford$ specification is composed of only a selection ranking.

4. The Initial Network Reduction

As defined in Section 3, a specification holds if it evaluates to true for all network states reachable by any trace in the infinite set of possible traces. However, to verify $specHolds$ using current automated solvers, it is necessary to reduce the problem to one without universal quantification over the infinite set of traces. This section describes the *initial network reduction*, which proves that a specification holds if it evaluates to true for all network states reachable by any trace in the *finite* set of traces that arise in the *initial network*. At a high level, the initial network of an AS corresponds to the network state where all routers have initialized their

$specHolds(\tau : spec)$

$\Longleftrightarrow$

$\forall\,(t : trace)\,(r \in R_i)\,(p \in P)\,(i \in in(r))\,(o \in out(r)),$
   `let` $i^* := inLocRIB(t, r, p)$
        $a_i := adjRIBsIn(t, r, p, i)$
        $a_i^* := adjRIBsIn(t, r, p, i^*)$
        $a_l^* := locRIB(t, r, p)$
        $a_o := adjRIBsOut(t, r, p, o)$
   `in` $\tau(\{router := r; prefix := p; sender := i; received := a_i;$
           $selected := a_l^*; bestSender := i^*; bestReceived := a_i^*;$
           $receiver := o; sent := a_o\}) = true$

$\Longleftrightarrow$ — 1) rewrite RIBs

$\forall\,(t : trace)\,(r \in R_i)\,(p \in P)\,(i \in in(r))\,(o \in out(r)),$
   `let` $i^* := inLocRIB(t, r, p)$
        $a_i := adjRIBsIn(t, r, p, i)$
        $a_i^* := adjRIBsIn(t, r, p, i^*)$
        $\color{blue}{a_l^* := imp(r, i^*, p, a_i^*)}$
        $\color{blue}{a_o := exp(r, o, p, a_l^*)}$
   `in` $\tau(\{router := r; prefix := p; sender := i; received := a_i;$
           $selected := a_l^*; bestSender := i^*; bestReceived := a_i^*;$
           $receiver := o; sent := a_o\}) = true$

$\Longleftarrow$ — 2) generalize best neighbor

$\forall\,(t : trace)\,(r \in R_i)\,(p \in P)\,(i, i^* \in in(r))\,(o \in out(r)),$
   `let` $a_i := adjRIBsIn(t, r, p, i)$
        $a_i^* := adjRIBsIn(t, r, p, i^*)$
        $\color{blue}{a_l := imp(r, i, p, a_i)}$
        $a_l^* := imp(r, i^*, p, a_i^*)$
        $a_o := exp(r, o, p, a_l^*)$
   `in` $\color{blue}{pref(a_l) \leq pref(a_l^*)} \rightarrow$
        $\tau(\{router := r; prefix := p; sender := i; received := a_i;$
           $selected := a_l^*; bestSender := i^*; bestReceived := a_i^*;$
           $receiver := o; sent := a_o\}) = true$

$\Longleftarrow$ — 3) generalize received announcements

$\forall\,(r \in R_i)\,(p \in P)\,(i, i^* \in in(r))\,(o \in out(r))\,\color{blue}{(a_i, a_i^* \in A)},$
   `let` $a_l := imp(r, i, p, a_i)$
        $a_l^* := imp(r, i^*, p, a_i^*)$
        $a_o := exp(r, o, p, a_l^*)$
   `in` $\color{blue}{transmittable(r, p, i, a_i) \rightarrow transmittable(r, p, i^*, a_i^*)} \rightarrow$
        $pref(a_l) \leq pref(a_l^*) \rightarrow$
        $\tau(\{router := r; prefix := p; sender := i; received := a_i;$
           $selected := a_l^*; bestSender := i^*; bestReceived := a_i^*;$
           $receiver := o; sent := a_o\}) = true$

$\Longleftrightarrow$

`INR`(τ)

`where`
   $transmittable(r, p, i, a) :=$
     $a = notAvailable \vee$
     $\exists\,(a_0 \in A)\,(\xi \in path(i, r)), a = transmit(\xi, p, a_0)$

Figure 6. The Initial Network Reduction. This reduction soundly removes the universal quantification over the infinite set of traces from $specHolds$. The reduction proceeds in three consecutive steps. 1) *Rewrite RIBs* rewrites a_l^* and a_o in terms of a_i^*. 2) *Generalize best neighbor* strengthens $specHolds$ with facts about selected announcements and then generalizes i^*. 3) *Generalize received announcements* strengthens $specHolds$ with facts about received announcements and then generalizes a_i and a_i^*. Terms added in each step are blue.
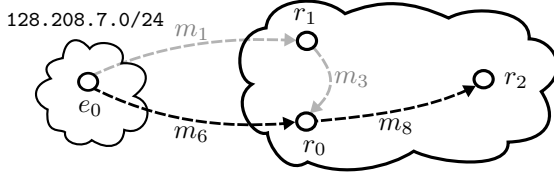
Figure 7. Initial Network Reduction Example.

RIBs to $notAvailable$ (corresponding to the state after line 8 in Fig. 1). We have proven this reduction in Coq. The full formalization of the BGP semantics and the reduction are out of scope for this paper, but can be found in a technical report [41]. Here we focus on the reduction and its intuition.

To understand the intuition behind the initial network reduction, consider an announcement a received by a router r via some trace t. To be received by r, announcement a had to be transmitted along some path (a sequence of connected routers) through the network. The *initial trace* t' of t with respect to a contains only those events in t that transmit a, but not those events that transmit any other announcements (i.e., t' transmits a in the initial network without any other announcements that could interfere).

For an announcement to be transmitted along a path, all the routers along the path have to select and forward the announcement. This means that in trace t: 1) for every router along the path the announcement was selected because it was better than all other announcements, and 2) the announcement was forwarded because it was different from the previously forwarded announcement.

If an announcement was transmitted in t along the path ξ, it will also be transmitted in t' along ξ because: 1) for every router along the path the announcement is selected because there are no other announcements that could be better, and 2) the announcement is forwarded because it is different from the initial announcement $notAvailable$ stored in all RIBs.

This implies that if Bagpipe verifies that a specification holds for those announcements that can be transmitted via the *initial trace*, then the specification also holds for those announcements that can be transmitted via any trace. It thus suffices to only consider the finite set of initial traces, instead of the infinite set of all traces. Thus, an AS's behavior in the initial network is "maximal" in a sense: if an announcement will ever be selected or forwarded by a router in any network state, it will also be selected or forwarded in the initial network.

To illustrate, consider the full trace t corresponding to Fig. 7, where router e_0's data-plane can directly deliver packets for destinations in `128.208.7.0/24`:

1. r_1 receives m_1 from e_0.
2. r_1 imports m_1, resulting in a_2, and selects a_2.
3. r_1 exports a_2, resulting in m_3, and forwards m_3 to r_0.
4. r_0 receives m_3 from r_1.
5. r_0 imports m_3, resulting in a_5, and selects a_5.

6. r_0 receives m_6 from e_0.
7. r_0 imports m_6, resulting in a_7, and selects a_7.
8. r_0 exports a_7, resulting in m_8, and forwards m_8 to r_2.
9. r_2 receives m_8 (containing a_8).

Black events are in the initial trace t' of t with respect to the announcement a_8. Gray events are not in the initial trace t' (these events are also gray in the figure). Subscripted variables r, e, m, and a correspond to internal routers, external routers, update messages, and announcements respectively.

We focus on the announcement a_8 received by router r_2, i.e., $adjRIBsIn(t, r_2, \text{128.208.7.0/24}, r_0)$. The announcement a_8 is the result of transmitting the original update message m_6 along the path $\xi = [e_0, r_0, r_2]$ via the network trace t.

It follows that a_8 is also the result of transmitting m_6 along ξ via the initial trace t' consisting only of the black events in trace t. t' is still legal despite the fact that r_o does not receive m_3 because there is no better announcement than a_7 for r_0 to select, and the value of r_0's RIBs is $notAvailable$ and thus different from a_8

The following paragraphs and Fig. 6 use the above insight to precisely explain the *initial network reduction* which eliminates $specHolds$'s quantification over all traces. As mentioned earlier, we have proven this reduction in Coq, but the full formalization of the BGP semantics and the reduction are out of scope for this paper. The initial network reduction proceeds in three steps.

1. Rewrite RIBs By the aforementioned insight, it suffices to only consider the initial traces for received announcements in the $adjRIBsIn$, but to verify a specification we have to also consider the announcements in the $adjRIBsOut$ and $locRIB$. This step eliminates $specHolds$'s dependence on $adjRIBsOut$ and $locRIB$. This is achieved by rewriting a) the forwarded announcements in the $adjRIBsOut$ in terms of $locRIB$ and b) the selected announcements in $locRIB$ in terms of $adjRIBsIn$. These rewrites are possible because the $adjRIBsOut$ is computed from the $locRIB$, and the $locRIB$ in turn is computed from the $adjRIBsIn$, by the algorithm described in Fig. 1.

a) The announcement $adjRIBsOut(t, r, p, o)$ forwarded by router r to some neighbor o is computed by applying the export program exp to the announcement $locRIB(t, r, p)$ selected by r. $adjRIBsOut(t, r, p, o)$ is therefore equal to $exp(r, o, p, locRIB(t, r, p))$.

b) The announcement $locRIB(t, r, p)$ selected by router r is computed by first applying the import program imp to each announcement $adjRIBsIn(t, r, i, p)$ received by a neighbor i of r, and then selecting the "best" one (as described in Fig. 1). Given the neighbor i^* from which the "best" announcement was selected, $locRIB(t, r, p)$ is therefore equal to $imp(r, i^*, p, adjRIBsIn(t, r, i^*, p))$.

The function $inLocRIB(t, r, p)$ computes the neighbor i^*. $inLocRIB(t, r, p)$ first passes all announcements in the

$adjRIBsIn(t, r, i, p)$ to the imp program, and then determines the neighbor with the "best" imported announcement.

2. Generalize Best Neighbor   Note that the $inLocRIB$ function still depends on the trace t. This step eliminates this dependence.

Consider the announcement a_l^* selected by the router r. A router always chooses the announcement a_l^* which has $pref$ greater or equal to the $pref$ of any other received and imported announcement a_l.

Therefore, if Bagpipe verifies that a specification holds for all received and imported announcements that have greater or equal $pref$ than a_l, no matter from which neighbor they were received, the specification also holds for the announcement a_l^* received from neighbor i^*.

This step applies this insight by strengthening $specHolds$ with the fact $pref(imp(r, i, p, a_i)) \leq pref(a_l^*)$, and then generalizing i^* to eliminate the dependence on t.

For those readers familiar with the BGP specification, note that Bagpipe fully supports tie-breaking on announcements with equal $pref$, e.g., tie-breaking on MED, and OSPF cost. This support stems from the fact that a specification must hold for all announcements with equal $pref$. This means that Bagpipe may falsely claim that a specification does *not* hold for a selected announcement a_l^* which can actually never be selected because a_l^*'s OSPF cost is higher than that of all other announcements, but will never falsely claim that a policy holds.

3. Generalize Received Announcements   We are finally ready to apply the insight mentioned in the beginning of this section. This step replaces $specHolds$'s use of received announcements in the $adjRIBsIn$, which requires quantification over all traces, with any announcements that can be transmitted in the empty network. Note that there are two received announcements a_i and a_i^* that need consideration.

An announcement a received by router r from neighbor i for some prefix p is transmittable in the empty network $transmittable(r, p, i, a)$ in two ways. Either a is the initial value $notAvailable$ stored in r's $adjRIBsIn$, or a_i is the result of transmitting some original announcement a_0 through the initial network along some path $\xi : path(i, r)$ that ends in router i followed by router r.

$transmit(\xi, p, a_0)$ computes the announcement that results from forwarding an announcement a_0 for prefix p along some path ξ in the initial network absent of any other announcements, i.e., it applies the appropriate imp and exp programs of every router along ξ to a_0.

Bagpipe exhibited no false positives in our evaluation. One reason is that modulo $pref$ tie-breaking, the initial network reduction is *complete* for specifications that only depend on either a_i or a_i^* (but not both), because if either a_i or a_i^* can be received by r via the initial trace, then by definition, there exists a trace via which r receives either a_i or a_i^*. Examples of specifications that depend on only a_i or a_l are $NoMartian$ and $BlockToExternal$. If a specification depends on both a_i

and a_i^*, it is possible that no trace exists that forwards both a_i and a_i^* to r, but we have not observed such cases in practice.

The goal of the initial network reduction was to eliminate $specHolds$'s quantification over any infinite sets, specifically the set of all traces. The initial network reduction achieves this goal, because INR only quantifies over finite sets. The set of prefixes P is large but finite. The set of announcements A is large but finite, as the BGP specification restricts the maximal announcement size to 4096 bytes. The set of routers R_i, $in(r)$, and $out(r)$ are also finite.

For ASes in a full-mesh configuration, meaning that each internal router is directly connected to all other internal routers[4], the set of paths $path(i, r)$ is also finite. This follows from the fact that an internal router r either received announcement a_i from some external router i — in which case the path is $[i, r]$ — or from an internal router i (other than r) which in turn received the announcement from an external router r_e — in which case the path is $[r_e, i, r]$. The set of paths is thus:

$$path(i, r) := \{[i, r] \mid i \in R_e\} \cup \\ \{[r_e, i, r] \mid i \in R_i \setminus \{r\} \wedge r_e \in R_e \wedge r_e \in neighbor(i)\}$$

Routing loops inside the AS are impossible, as routers do not forward announcements received from internal neighbors to internal neighbors.

We refer to the formula resulting from the initial network reduction as $INR(\tau)$. Because $INR(\tau)$ only quantifies over finite sets, it enables automatic verification of $specHolds$ using current solvers.

5.   Bagpipe Implementation

Bagpipe verifies a specification τ by searching for counterexamples to $INR(\tau)$. While this search space is finite, it is still far too large to permit naive brute-force search. This section describes how Bagpipe searches this space efficiently.

5.1   Symbolic Search using Rosette

Bagpipe uses Rosette [39] to symbolically search for counterexamples. Rosette extends the Racket language with symbolic values, assertions, and a `verify` function. (`verify e`) attempts to assign a concrete value to every symbolic value in e such that an assertion in e is violated. Rosette implements `verify` by reducing the search for a failure-inducing assignment to a satisfiability query, which is then discharged by an off-the-shelf SAT or SMT solver. Once the solver returns, its output is automatically lifted to a concrete value for each symbolic value. These concrete values are then used to provide counterexamples.

Figure 8 shows the implementation of Bagpipe in Rosette. The core of Bagpipe is a translation of $INR(\tau)$ to a Rosette program. The universally-quantified variables in $INR(\tau)$ are

---

[4] 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 support this optional extension of the BGP specification.

```
(define (bagpipe τ) (verify (begin
  (define r (symbolic R_i))
  (define p (symbolic P))
  (define i i* (symbolic (in r)))
  (define o (symbolic (out r)))
  (define a_i a_i* (symbolic A))
  (define a_l (imp r i p a_i))
  (define a_l* (imp r i* p a_i*))
  (define a_o (exp r o p a_l*))
  (assert
    (implies
        (transmittable r p i a_i) (transmittable r p i* a_i*)
        (≤ (pref a_l) (pref a_l*))
     (τ {router := r; prefix := p; sender := i; received := a_i;
        selected := a_l*; bestSender := i*; bestReceived := a_i*;
        receiver := o; sent := a_o}))))))))
```

Figure 8. Bagpipe implementation in Rosette. The core of Bagpipe is a translation of INR(τ) to a program (`begin ...`). This program uses symbolic variables instead of universal quantification. Rosette's `verify` function attempts to assign a concrete value to every symbolic value in the program such that an assertion in the program is violated. If no such assignment is found, INR(τ) is valid, and the specification τ holds.

translated to symbolic values, which are used as inputs to the assertion that τ holds over all valid pairs of announcements.[5]

Users of Bagpipe express specifications in Racket as boolean predicates over active announcements. For example, a user would express the *NoMartian* specification from Section 3 as follows:

```
(define NoMartian (v)
  (implies (martian (prefix v))
           (= (selected v) notAvailable)))
```

In Bagpipe, routers are configured using the *imp* and *exp* programs. In practice, routers are configured using router configuration languages; most real-world routers use languages developed by Juniper and Cisco. An *interpreter* bridges the gap between Bagpipe and real-world configurations; it is a program that takes a router configuration and inputs (e.g., router, prefix, announcement) and returns the result (an announcement) of running the router configuration. Bagpipe includes interpreters for Juniper and Cisco configurations. These interpreters consist of a parser that generates an AST, and an execution engine that can run an AST given some inputs. Rosette lifts the execution engine to a symbolic execution engine that can run an AST symbolically on all inputs. Bagpipe also infers the network topology of the AS from router configurations.

Bagpipe's interpreters skip commands unrelated to BGP (e.g., configuration commands for other protocols including IGMP, MPLS, and ISIS), and ignore low-level BGP configuration details (e.g., maximum update message TCP packet size). This could introduce unsoundness (e.g., if an AS operator accidentally configured maximum TCP packet size to be 0, then all update messages would be dropped). The

interpreters also currently do not handle some BGP-related commands, such as IP broadcasting.

In our experiments, we found that Bagpipe's current interpreters are sufficient to handle hundreds of thousands of lines of industrial BGP configurations. Extending Bagpipe's interpreters to support additional BGP-conforming features or configuration languages requires no changes in the main algorithm, because Bagpipe models import and export rules as arbitrary functions. Such extensions may however require substantial engineering efforts in the interpreters, as existing configuration languages are often proprietary and contain a vast number of features.

5.2 Predicate Abstraction

The set of possible announcements is finite, since BGP restricts the maximal size of announcements to 4096 bytes. Even when represented symbolically, this is still a large search space. Therefore, Bagpipe also implements a form of predicate abstraction [16] by coalescing *aspath* and *communities* values that induce the same control flow in the BGP configuration.

Bagpipe exploits the fact that the Juniper and Cisco configuration languages use regular-expression predicates to branch on announcement attributes. The following example shows two such predicates contained in the Internet2 configurations:

```
as-path PRIVATE ".* (64512-65535) .*";
community LHCONE-DO-NOT-ANNOUNCE-AS members 65010:*;
```

`PRIVATE` matches all AS-paths that contain at least one AS with an AS number in the range 64512 – 65535. `LHCONE-DO-NOT-ANNOUNCE-AS` matches all communities whose first 16 bits encode the number 65010. The set of predicates in a configuration is finite and usually fairly small. All configurations of Internet2 combined, for example, contain only 73 community predicates.

Bagpipe implements predicate abstraction by automatically discovering all regular-expression predicates used in router configurations, and replacing the *aspath* and *communities* data contained in an announcement with a bit-vector that contains a bit for every predicate over *aspath* or *communities*. These bit-vectors are represented symbolically during Bagpipe's search for counterexamples.

This approach is sound, but incomplete. Consider for example a configuration with two predicates: predicate Φ matches every *aspath*, and predicate ϕ matches exactly one specific *aspath*. Bagpipe would explore a branch on $\Phi(a) \wedge \neg\phi(a)$ which cannot be executed in reality because for every a, $\Phi(a)$ implies $\phi(a)$. We did not see such false positives in our evaluation.

5.3 Parallelization through Hoisting

Bagpipe hoists certain symbolic values out of the program passed to Rosette's symbolic search, and enumerates them instead of representing them symbolically. For example, Bagpipe hoists (`symbolic R_i`) out of the symbolic execution,

---

[5] The translation of *transmittable* into Rosette is omitted due to space reasons, but it is straightforward.

manually enumerating the set R_i instead. Rosette is then invoked for each enumerated value.

```
(define (bagpipe τ) (for/each (r R_i)
  (verify (begin ...r...))))
```

Bagpipe parallelizes the resulting loop across multiple nodes in a cluster. Bagpipe hoists all variables except for announcements and prefixes, i.e., r, o, i, i^*, and some variables in $transmittable$. Bagpipe hoists these sets because their relatively small size and minimal structure are not exploitable by the symbolic execution engine.

The number of calls made by Bagpipe to Rosette can be computed by considering the magnitude of all of the sets that Bagpipe has to enumerate. To verify a policy, Bagpipe has to enumerate every internal router r, all the neighbors to which r can forward an announcement a_o, and all the paths along which the two received announcements a_i and a_i^* could have been transmitted to r.

The number of neighbors to which r can forward announcement a_o is $|out(r)|$. There are three kinds of paths along which an announcement could have been transmitted to router r. (1) The announcement was directly transmitted from one of r's external neighbors e to r. There are $n(r) = |\{e \in R_e | e \in neighbor(r)\}|$ such paths. (2) The announcement was transmitted from an external neighbor through an internal neighbor i to r. There are $\sum_{i \in R_i \setminus \{r\}} n(i)$ such paths. (3) The announcement is the value $notAvailable$ with which the $adjRIBsIn$ for an incoming neighbor was initialized. There are $|in(r)|$ such cases. The total number of calls Bagpipe makes to Rosette are thus:

$$\sum_{r \in R_i} |out(r)| \left(|in(r)| + n(r) + \sum_{i \in R_i \setminus \{r\}} n(i) \right)^2$$

Assuming that an AS has at least one external neighbor, the asymptotic number of Rosette calls made by Bagpipe is $O(|R_i|^4 |R_e|^3)$, where R_i is the set of internal routers and R_e is the set of all external neighbors. This stems from the fact that $O(|out(r)|) = O(|in(r)|) = O(|R_i| + |R_e|)$, and $O(n(r)) = O(|R_e|)$.

5.4 Omitting Unused Specification Arguments

There are specifications that do not use all of their arguments; $BlockToExternal$, for example, uses neither the $sender$ argument nor the $received$ argument (whose value depends on $sender$). Enumerating these unused arguments would result in many calls to Rosette which are guaranteed to be equivalent. To avoid this duplication, Bagpipe does not enumerate certain combinations of unused arguments, thereby reducing the number of calls to Rosette. In practice, we found that useful specifications use one of the following three combinations of unused variables (all supported by Bagpipe):

- Bagpipe does not enumerate $sender$ and $receiver$ if arguments $sender$, $received$, $receiver$, and $sent$ are unused.

This is the case for all specifications composed of only an import specification (e.g., the $NoMartian$ specification). In this case, Bagpipe calls Rosette $O(|R_i|^2|R_e|)$ times.

- Bagpipe does not enumerate $sender$ if arguments $sender$ and $received$ are unused. This is the case for all specifications composed of only an export specification (e.g., the $BlockToExternal$ specification). Bagpipe calls Rosette $O(|R_i|^3|R_e|^2)$ times.

- Bagpipe does not enumerate $receiver$ if arguments $receiver$ and $sent$ are unused. This is the case for all specifications composed of only a selection ranking (e.g., the $GaoRexford$ specification). Bagpipe calls Rosette $O(|R_i|^3|R_e|^2)$ times.

6. Evaluation

This section evaluates Bagpipe by answering the following questions. *Expressiveness:* Can Bagpipe specify policies for common configuration scenarios? *Efficiency:* How long does Bagpipe take to verify specifications? *Effectiveness:* How many bugs does Bagpipe find, and how many false positive does Bagpipe produce?

6.1 Juniper TechLibrary Scenarios

We evaluated Bagpipe on 10 configuration scenarios from the Juniper TechLibrary. Specifically, we evaluated Bagpipe on 10 scenarios described in the *Basic BGP Configuration* [4] and *Configuring Routing Policies* [22] sections of the Juniper TechLibrary, which is the official technical documentation for Juniper products. We used all scenarios from the *Basic BGP Configuration*, but did not use 25 scenarios from *Configuring Routing Policies* because: 5 scenarios require extensions or optional features of the BGP specification RFC 4271 that are inconsistent with Bagpipe's model of BGP (specifically exporting routes that are not selected, and delaying selection of announcements to reduce oscillation), 5 scenarios are unrelated to BGP verification (e.g., logging the number of forwarded announcements), and 15 scenarios require currently unsupported Juniper features (e.g., policy subroutines) that we believe could be implemented in Bagpipe without changing Bagpipe's model of BGP.

Each scenario describes a set of AS operator objectives, and it provides router configurations for an entire example AS that achieves these objectives. For each scenario, we expressed a specification and verified it against the scenario's configurations. Bagpipe verified each specification in less than one minute.

The following list provides the name of each scenario, along with a description and code size of the specification that we expressed and verified (no line is longer than 80 characters):

1. *Configuring Internal BGP Peering.* All internal routers share their route announcements with all other internal routers (5 lines).

2. *Using AS Path Regular Expressions.* Routers block announcements whose AS path matches a given set of regular expressions (4 lines).

3. *Disabling Suppression of Route Advertisements.* Route announcements are sent back to the neighbor from which they were received (5 lines).

4. *Configuring Policy Chains and Route Filters.* The prefix of exported announcements matches the given chained route filters (10 lines).

5. *Configuring a Conditional Default Route Policy.* A default route is exported (3 lines).

6. *Configuring Communities in a Routing Policy.* Routers set appropriate local preferences according to an announcement's communities (8 lines).

7. *Rejecting Known Invalid Routes.* Known invalid routes are rejected (3 lines).

8. *Configuring External BGP Peering.* A router is connected only to external routers in a given set (4 lines).

9. *Configuring Routing Policy Prefix Lists.* The prefix of exported announcements matches the given prefix lists (12 lines).

10. *Using Routing Policy to Set a Preference Value for BGP Routes.* Any local preference set by external neighbors is replaced with a default value (3 lines).

To ensure Bagpipe detects specification violations, we also modified each scenario's configurations to violate its objectives. In each case, Bagpipe detected the violation in under a minute.

Section 3 showed that Bagpipe supports policies found in the literature, such as the Gao-Rexford model [14] and prefix-based filtering [29]. Section 6.2 shows that Bagpipe can also express policies inferred from real AS configurations.

6.2 Real AS Configurations

We inferred and verified specifications from the configurations of three ASes: the nation-wide ISP Internet2, the regional ISP BelWü, and the local ISP Selfnet. These configurations total over 240,000 lines of Cisco and Juniper code.

For each inferred specification, Figure 9 summarizes the time required to verify the specification, the number of searches that were performed by Rosette (which are performed in parallel), the arguments passed to the policy that are not used (see Section 5.4), and the number of import and export programs that violate the specification. *Bagpipe did not produce false positives on any benchmark.*

Timings are on Amazon EC2 with 2 instances of type `c3.8xlarge`, each with 32 virtual-cores and 60 GB of memory. The experiments ran for a total of 82h, the cost for which is about $30 using EC2 spot instances.

Nationwide ISP: Internet2 The Internet2 AS connects educational, research, and government institutions spread throughout the US. We have access to the full configuration of Internet2's 10 BGP routers [19]. These routers are connected to 274 external neighbors. The configurations to-

tal 100,651 lines of Juniper code. We verified four policy specifications for Internet2, described below.

Internet2's configurations contain checks to block the import of announcements with martian prefixes. We thus inferred that it is Internet2's policy to never import martian prefixes. It takes Bagpipe 1,178s (20min) to verify that Internet2 correctly implements the *NoMartian* specification.

Internet2 contains dedicated sanity checks in 237 out of the 274 *imp* programs. After removing these sanity checks from the configurations, it takes Bagpipe 1,194s (20min) to check the *NoMartian* specification for Internet2 (indicated by *no checks* in Fig. 9). Because Internet2 performs a large variety of other checks on announcements, 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.

From Internet2's configurations, it appears to be Internet2's policy not to forward an announcement to external routers if the announcement has the BTE community set. It takes Bagpipe 28,594s (8h) to check the *BlockToExternal* specification. The configurations of 5 neighbors do not adhere to the *BlockToExternal* specification. We notified Internet2 of these violations, but have not yet received a response.

Internet2 appears to operate according to a refined version of the Gao-Rexford model discussed in Section 3. We manually classified neighbors either as *customer* or *peer* using Internet2's pricing structure [20] and comments found in the configurations. None of Internet2's neighbors is a *provider*. We refined the usual *GaoRexford* specification to support Internet2's advanced policies, such as blocking announcements for invalid prefixes and allowing both customers and peers to influence Internet 2's preference of an announcement by setting certain communities. For example, a peer can set the HIGH_PEERS community to increase an announcement's preference. It takes Bagpipe 260,790s (72h) to check the refined *GaoRexford* specification. The configurations of 14 neighbors do not adhere to the refined *GaoRexford* specification. We have contacted Internet2 about these violations, but have not yet received a response.

Regional ISP: BelWü We have access to the BGP related configurations for three BGP routers[6] of the regional ISP BelWü [3]. These routers are connected to 300 external neighbors. The configurations total 143,657 lines of Cisco code.

At the time of our experiments, it was BelWü's policy to monitor all announcements for martian prefixes by tagging them with a particular community, and to use this monitoring information to evaluate the impact of martian filtering on BelWü's existing routes. Because of good results, BelWü is planning to enable strict martian filtering in the near future.

---

[6] Our experiments for these configurations are to demonstrate scaling, and assume that only these three routers are operating in the AS.

AS	Policy	Time	Solver Calls	Unused Arguments	Violations	False Positives
Internet2 (nation-wide)	*NoMartian*	1,178s (20min)	3,114	*sender* and *receiver*	0	0
Internet2 (nation-wide)	*NoMartian* (no checks)	1,194s (20min)	3,114	*sender* and *receiver*	N/A	0
Internet2 (nation-wide)	*BlockToExternal*	28,594s (8h)	115,330	*sender*	5	0
Internet2 (nation-wide)	*GaoRexford*	260,790s (72h)	971,680	*receiver*	14	0
BelWü (regional)	*NoMartian*	2,106s (35min)	1516	*sender* and *receiver*	N/A	0
BelWü (regional)	*TagMartian*	2,165s (36min)	1516	*sender* and *receiver*	0	0
BelWü (regional)	*RemoveOwnASN*	1,838s (31min)	1516	*sender* and *receiver*	0	0
Selfnet (local)	*StaticExport*	2s	9	none	0	0

Figure 9. Real AS Configuration Case Study Results. *Solver Calls* is the number of calls to Rosette's solve function. *Unused Arguments* indicates the arguments passed to the policy that are not used. *Violations* is the number of specification violations found. Bagpipe did not issue false positives in any experiment.

Bagpipe took 2,106s (35min) to check the *NoMartian* specification and, as expected, revealed that BelWü imports martian prefixes (from 268 neighbors). We also expressed the specification *TagMartian*, which requires BelWü to tag all announcements for martian prefixes. Bagpipe took 2,165s (36min) to verify that BelWü correctly implements *TagMartian*.

It is BelWü's policy to remove, from all received announcements, communities that contain the ISP's own AS number. We call this policy *RemoveOwnASN*. It takes Bagpipe 1,838s (31min) to verify that BelWü correctly implements *RemoveOwnASN*.

The configurations did not indicate BelWü's Gao-Rexford relationships (*customer*s, *peer*s, and *provider*s), and we did thus not verify a *GaoRexford* policy. Verifying such a policy would require 619,258 solver calls.

Local ISP: Selfnet We also analyzed the 66 lines of Juniper configuration for the sole BGP router of the local ISP Selfnet [34]. This router has only a single BGP neighbor, and the ISP's policy is to only export announcements with the prefixes that it actually owns. We call this policy *StaticExport*. Bagpipe took 2s to verify that Selfnet correctly implements *StaticExport*.

6.3 Potential False Positives

Bagpipe has three potential sources of false positives:

1) When two announcements are tied on local preference, BGP uses complex rules to choose between them. Bagpipe conservatively and soundly models tie breaking as non-deterministic choice, rather than completely modeling the details of lower-level protocols like OSPF that these rules use (Section 4).

2) Bagpipe uses predicate abstraction to represent announcements. Removing this source of incompleteness would dramatically increase the size of Bagpipe's search space (Section 5.2).

3) The initial network reduction forwards announcements in the initial network, which is sound but could lead to false positives (e.g., when two announcements can be individually forwarded in the initial network but interfere with each other in reality). We do not know how to eliminate this source

of incompleteness while keeping the search space finite (Section 4).

We found, via manual inspection, that *none* of the counter examples returned by Bagpipe during our evaluation were due to false positives.

7. Related Work

In this section, we address related work in network configuration checking. We also briefly discuss software-defined networking and prior work on SMT-based tools.

Network Analysis *rcc* [11] is a tool to find bugs in BGP configurations, which has been adopted by many AS administrators. 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, topology descriptions, and a particular set of received BGP announcements are translated into Datalog facts which are processed by Datalog rules to generate routing-tables. Z3 is then used to verify first-order properties over the generated routing-tables. Bagpipe and Batfish make different design decisions, and are thus able to verify different properties. Bagpipe verifies a restricted set of routing-table invariants (described in Section 3.2) with respect to *any* set of received announcements, whereas Batfish verifies arbitrary first-order logic formulas over routing-tables with respect to a particular set of BGP announcements. Great care has been taken in Bagpipe that all invariants can be translated to SAT formulas which can be decided in exponential time, whereas Batfish's formulas are generally undecidable.

Header Space Analysis (HSA) [23] verifies a data plane's packet forwarding behavior in a similar way to Bagpipe (which verifies a control plane's announcement forwarding behavior). Contrary to Bagpipe, HSA uses a custom symbolic search algorithm instead of an SMT solver; and HSA verifies specifications that restrict the forwarding of packets independent of any other packets in the network (like Bagpipe's

import/export specifications), but cannot verify specifications that restrict the forwarding of packets depending on other packets in the network (like Bagpipe's selection rankings).

BGP Simulation   C-BGP [31] is a BGP simulator. Given a topology and a set of configurations, it determines how traffic will be routed. AS operators can use it both for debugging existing problems and for testing potential new configurations. C-BGP and Bagpipe are potentially complementary; an AS operator 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 [30, 1], model-checking [2, 9], and full formal verification [17]. 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.

SMT-Based Tools   SAT and SMT solvers have been applied in a wide range of automated tools for bug finding [7, 6, 10], verification [25, 26, 37], program synthesis [36, 24], and fault localization [21]. Bagpipe builds on Rosette [38], a programming language designed for easy creation of such tools. In particular, Rosette is equipped with a symbolic compiler that can efficiently reduce a verification, synthesis, or fault localization query about all bounded executions of a program to an SMT formula. Because the initial network reduction enables Bagpipe to treat BGP configurations as finite programs, we can use Rosette's bounded reasoning facilities for sound and scalable verification of BGP policies.

8.   Conclusion

We presented Bagpipe, a tool to automatically verify that router configurations correctly implement AS operators' BGP policies. To make this verification possible, we introduced the *initial network reduction*, which reduces verification of BGP policies from checking an infinite set of traces to checking a finite set of initial traces, thus enabling Bagpipe to use current constraint solvers effectively. Building on this reduction, the Bagpipe implementation additionally employs a novel combination of search space partitioning, unused variable omission, symbolic execution, and predicate abstraction. We evaluated Bagpipe on 10 configuration scenarios from the Juniper TechLibrary and three ASes with a total of over 240,000 lines of configuration, finding that Bagpipe will scale to the complexity and scope of real-world AS configurations.

  Future work will extend Bagpipe to support additional BGP features (e.g., route reflectors), incrementalize verification for configuration updates, and use Rosette's synthesis

features to automatically generate configurations that correctly implement policies.

References

[1] C. J. Anderson et al. "NetKAT: Semantic Foundations for Networks". In: *POPL*. 2014.

[2] T. Ball et al. "VeriCon: Towards Verifying Controller Programs in Software-defined Networks". In: *PLDI*. 2014.

[3] *BelWü*. https://www.belwue.de/.

[4] *BGP Feature Guide for the OCX Series*. 2015.

[5] M. Brown. *Pakistan hijacks YouTube*. http://research. dyn.com/2008/02/pakistan-hijacks-youtube-1/. 2008.

[6] C. Cadar, D. Dunbar, and D. Engler. "KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs". In: *OSDI*. 2008.

[7] E. Clarke, D. Kroening, and F. Lerda. "A Tool for Checking ANSI-C Programs". In: *TACAS*. 2004.

[8] J. Cowie. *China's 18-Minute Mystery*. http://research. dyn.com/2010/11/chinas-18-minute-mystery/. 2010.

[9] M. Dobrescu and K. Argyraki. "Software Dataplane Verification". In: *NSDI*. 2014.

[10] J. Dolby, M. Vaziri, and F. Tip. "Finding bugs efficiently with a SAT solver". In: *FSE*. 2007.

[11] N. Feamster and H. Balakrishnan. "Detecting BGP Configuration Faults with Static Analysis". In: *NSDI*. 2005.

[12] A. Fogel et al. "A General Approach to Network Configuration Analysis". In: *NSDI*. 2015.

[13] N. Foster et al. "Frenetic: A Network Programming Language". In: *ICFP*. 2011.

[14] L. Gao and J. Rexford. "Stable Internet Routing Without Global Coordination". In: *SIGMETRICS*. 2000.

[15] S. Goldberg. "Why Is It Taking So Long to Secure Internet Routing?" In: *Queue* (2014).

[16] S. Graf and H. Saïdi. "Construction of Abstract State Graphs with PVS". In: *CAV*. 1997.

[17] A. Guha, M. Reitblatt, and N. Foster. "Machine-verified Network Controllers". In: *PLDI*. 2013.

[18] *International Telecommunication Union Statistics*. 2014.

[19] *Internet2 Configurations*. http://vn.grnoc.iu.edu/Internet2/configs/configs.html.

[20] *Internet2 Fees*. http://www.internet2.edu/about-us/membership/.

[21] M. Jose and R. Majumdar. "Bug-Assist: assisting fault localization in ANSI-C programs". In: *CAV*. 2011.

[22] *Junos OS: Routing Policies, Firewall Filters, and Traffic Policers Feature Guide for Routing Devices*. 2016.

[23] P. Kazemian, G. Varghese, and N. McKeown. "Header Space Analysis: Static Checking for Networks". In: *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*. 2012.

[24] A. S. Koksal et al. "Synthesis of Biological Models from Mutation Experiments". In: *POPL*. 2013.

[25] K. R. M. Leino. "Dafny: An Automatic Program Verifier for Functional Correctness". In: *LPAR*. 2010.

[26] K. R. M. Leino. *This is Boogie 2*. Tech. rep. 2008.

[27] D. Madory. *Chinese Routing Errors Redirect Russian Traffic*. http://research.dyn.com/2014/11/chinese-routing-errors-redirect-russian-traffic/. 2014.

[28] D. McConnell. *Chinese company 'hijacked' U.S. web traffic*. http://www.cnn.com/2010/US/11/17/websites.chinese.servers/. 2010.

[29] D. Meyer, J. Schmitz, and C. Alaettinoglu. *Application of Routing Policy Specification Language (RPSL) on the Internet*. 1997.

[30] C. Monsanto et al. "A Compiler and Run-time System for Network Programming Languages". In: *POPL*. 2012.

[31] B. Quoitin and S. Uhlig. "Modeling the Routing of an Autonomous System with C-BGP". In: *IEEE Network* (2005).

[32] Y. Rekhter, T. Li, and S. Hares. *A Border Gateway Protocol 4 (BGP-4)*. RFC 4271. 2006.

[33] L. Schaefer. *Deutsche Telekom: 'Internet data made in Germany should stay in Germany'*. http://www.dw.com/en/deutsche-telekom-internet-data-made-in-germany-should-stay-in-germany/a-17165891. 2013.

[34] *Selfnet*. https://selfnet.de/.

[35] D. Slane. *2010 Report to Congress of the U.S.–China Economic and Security Review Commission*. 2010.

[36] A. Solar-Lezama et al. "Combinatorial Sketching for Finite Programs". In: *ASPLOS*. 2006.

[37] P. Suter, A. S. Köksal, and V. Kuncak. "Satisfiability modulo recursive programs". In: *SAS*. 2011.

[38] E. Torlak and R. Bodik. "A Lightweight Symbolic Virtual Machine for Solver-aided Host Languages". In: *PLDI*. 2014.

[39] E. Torlak and R. Bodik. "Growing Solver-aided Languages with Rosette". In: *Onward!* 2013.

[40] D. Turner et al. "California Fault Lines: Understanding the Causes and Impact of Network Failures". In: *SIGCOMM*. 2010.

[41] K. Weitz et al. *Bagpipe: Verified BGP Configuration Checking*. Tech. rep. 2016.