FSR: Formal Analysis and Implementation Toolkit for Safe Inter-domain Routing

Anduo Wang* Limin Jia† Wenchao Zhou* Yiqing Ren* Boon Thau Loo*
Jennifer Rexford‡ Vivek Nigam* Andre Scedrov* Carolyn Talcott¶
University of Pennsylvania* Carnegie Mellon University† Princeton University‡ SRI International¶
{anduo,wenchaoz,yiqingr,boonloo}@seas.upenn.edu, {vnigam,scedrov}@math.upenn.edu liminjia@cmu.edu
jrex@cs.princeton.edu clt@csl.sri.com

***Abstract*—Inter-domain routing stitches the disparate parts of the Internet together, making protocol stability a critical issue to both researchers and practitioners. Yet, researchers create safety proofs and counter-examples by hand, and build simulators and prototypes to explore protocol dynamics. Similarly, network operators analyze their router configurations manually, or using home-grown tools. In this paper, we present a comprehensive toolkit for analyzing and implementing routing policies, ranging from high-level guidelines to specific router configurations. Our Formally Safe Routing (*FSR*) toolkit performs all of these functions from the same algebraic representation of routing policy. We show that routing algebra has a natural translation to both *integer constraints* (to perform safety analysis with SMT solvers) and *declarative programs* (to generate distributed implementations). Our extensive experiments with realistic topologies and policies show how *FSR* can detect problems in an AS's iBGP configuration, prove sufficient conditions for BGP safety, and empirically evaluate convergence time.**

I. Introduction

The Internet's global routing system does not necessarily converge, depending on how the Border Gateway Protocol (BGP) policies of individual networks are configured. Since protocol oscillations cause serious performance disruptions and router overhead, researchers devote significant attention to BGP stability (or "safety"). Abstract formal models of BGP [15], [12], [14], [13], [36] allow researchers to explore how local policies affect BGP stability and identify policy guidelines that, if universally adopted by ISPs, ensure global safety [9], [8], [11], [4], [10], [33]. While our understanding of BGP safety has improved dramatically in the past decade, each research study still proceeds independently—manually creating proofs and counter-examples, and sometimes building simulators or prototypes to study protocol overhead and transient behavior during convergence.

To aid the design, analysis, and evaluation of safe interdomain routing, we propose the *Formally Safe Routing* (*FSR*) toolkit. *FSR* serves two important communities. For researchers, *FSR* automates important parts of the design process and provides a common framework for describing, evaluating, and comparing new safety guidelines. For network operators, *FSR* automates the analysis of internal router (iBGP) and border gateway (eBGP) configurations for safety violations. For both communities, *FSR* automatically generates realistic protocol implementations to evaluate real network configurations (e.g., to study convergence time) prior to actual deployment. The ideas underlying *FSR* also unify research in *routing algebras* [13], [36] with recent advances in *declarative networking* [22] to produce provably-correct implementations of safe interdomain routing.

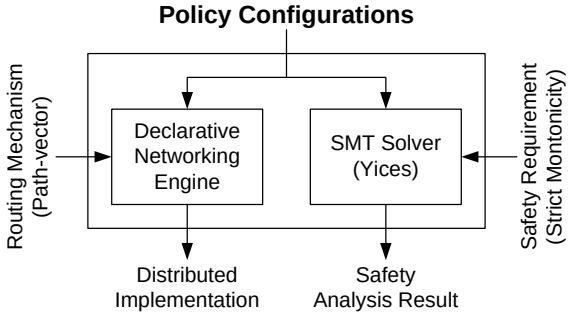

Fig. 1. *FSR* Architecture.

Given policy configurations as input, *FSR* produces an analysis of safety properties and a distributed protocol implementation, as shown in Figure 1. *FSR* has three main underlying technologies:

Policy configuration as algebra: Our extensions to routing algebra [13], [36] allow researchers and network operators to express policy configurations in an abstract algebraic form. These configurations can be anything from high-level *policy guidelines* (e.g., proposed constraints that a researcher wants to study) or a completely specified *policy instance* (e.g., an iBGP configuration or a multi-AS network that an operator wants to analyze). Router configuration files can be automatically translated into the algebraic representation, easing the adoption of *FSR*.

Safety analysis: To automatically analyze the policy configuration, *FSR* reduces the convergence proof to a *constraint satisfaction* problem, solved using the Yices SMT (Satisfiability Modulo Theories) solver [42]. The solver determines whether it is possible to jointly satisfy the policy configuration and the safety requirement of "strict monotonicity" (the rightmost input in Figure 1, drawn from previous work [36]). If all constraints can be satisfied, the routing system is provably safe; otherwise, the solver outputs a smallest subset of the

constraints that are not satisfiable to aid in identifying the problem and fine-tuning the configuration.

Safe implementation: To enable an evaluation of protocol dynamics and convergence time, *FSR* uses our extended routing algebra to automatically generate a distributed routing-protocol implementation that matches the policy configuration—avoiding the time-consuming and error-prone task of manually creating an implementation. Given the policy configuration and a formal description of the path-vector mechanism (the leftmost input in Figure 1), *FSR* generates a correct translation to a *Network Datalog* (*NDlog*) specification, which is then executed using the RapidNet declarative networking engine [31], [27]. *NDlog* enables a direct translation from routing algebra to *NDlog* programs.

In practice, *FSR*'s safe implementation can be used as an emulation platform for studying BGP performance. By changing the left input in Figure 1, researchers can also experiment with alternative routing mechanisms, such as HLP [38]. Researchers and network operators can use *FSR* to evaluate a variety of policy configurations prior to actual deployment on a legacy routing platform. In a more radical form, we envision that the *FSR*-generated protocol implementation could run in the operational network.

FSR toolkit is built on our insights into how to combine the use of routing algebra with an existing SMT solver and declarative networking engine. *FSR* bridges the design, analysis and implementation of routing protocols within one unified framework. Our main research contributions are:

- **Unified policy specifications:** We extend routing algebra to enable automated translation from a policy configuration to a distributed protocol implementation. We propose an automatic way to translate SPP (Stable Paths Problem) [12] instances to an algebraic representation. This allows *FSR* to apply safety analysis based on routing algebras to SPP instances, in addition to policy guidelines.
- **Automated safety analysis:** We formulate the safety analysis as a constraint solving problem. Given a policy configuration, *FSR* automatically generates the constraints as inputs to a standard SMT solver, relieving users from manually constructing safety proofs. The analysis leverages the natural mapping from input algebra and safety requirements to constraints.
- **Generating *NDlog* implementations:** We show how to automatically translate from routing algebra to a provably-correct *NDlog* program, bringing together two major techniques for modeling routing protocols. The translation bridges the gap between formal analysis of abstract routing configurations and actual implementation of routing protocols.
- **Experiments on Internet topologies:** We demonstrate that *FSR* is effective through case studies, including detecting iBGP configuration problems, proving sufficient conditions for BGP safety, evaluating how convergence time scales with network size, convergence behavior of eBGP instances, and impact of alternative routing mechanisms on convergence.

Paper roadmap: After a brief overview of routing algebra in Section II, we describe in Section III how we apply and extend routing algebra to express policy configurations suitable for both analysis and implementation. Section IV defines safety analysis as a constraint satisfiability problem, and describes how *FSR* fully automates the analysis using an SMT solver. In Section V, we describe how *FSR* automatically generates a distributed *NDlog* implementation. We present our experimental evaluation in Section VI, discuss related work in Section VII, and conclude in Section VIII.

II. Routing Algebra: Background

The main input to *FSR* is the policy configuration, specified in our extended version of routing algebra. In addition to concisely specifying routing policy, the routing algebra enables us to use existing results [36] to analyze safety. This section presents a brief overview of the routing algebra and an example that encodes the Gao-Rexford guideline. We also discuss the shortcomings of the existing representation for encoding export filters, to set the stage for our extended algebra in Section III.

A. Abstract Routing Algebra

Routing algebra is an abstract structure that describes how network nodes calculate routes, and the preference for one route over another[1]. A node can refer to an AS or internal router, depending on whether we are considering iBGP or eBGP policy configurations respectively. We use an example policy of shortest hop-count routing to show the relationship between the abstract algebra and a concrete policy. We will use the terms *route* and *path* interchangeably in the paper.

An abstract routing algebra is a tuple $\langle \Sigma, \preceq, \mathcal{L}, \oplus \rangle$, with the following components:

Path signatures (Σ): Path signatures describe the attributes of the paths, so that routes can be ranked. A special element $\phi \in \Sigma$, represents the signature for prohibited paths. In the hop-count example, Σ is the set of natural numbers (corresponding to the path length) and ∞ is the signature for prohibited path (i.e., all paths with cost ∞ are excluded from consideration).

Path preference relation ($\preceq$): Intuitively, if a $\preceq$ b, then a is preferred to b. Since the ordering is total over the elements of Σ, one can use this ordering for selecting the best path. To ensure prohibited paths are never selected, $\alpha \prec \phi$ for any signature $\alpha \neq \phi$. To select the shortest path, the "less than or equal" ($\leq$) relation on natural numbers is used as $\preceq$.

Link labels ($\mathcal{L}$): Link labels describe the attributes of links between immediate neighbors. In the shortest hop-count routing example, neighboring nodes are one hop apart, so each link's attribute is set to 1 (i.e., $\mathcal{L} = \{1\}$).

Path concatenation ($\oplus$): The concatenation function captures how an AS computes a new route based on a route received from a neighbor. The function takes a label and a signature, and returns a new signature. To compute path lengths, $\oplus$ is the addition (+) function on natural numbers for summing up the cost of a link and an existing path.

[1]The algebra presented in [36] differs slightly from the algebra in the later publication on metarouting [13], which is the algebra we use here.

Complex policies can be represented as compositions of simpler policies [13]. For example, ASes often rank routes based on multiple attributes (e.g., the next-hop AS, the path length, and so on) in a series of "tie-breaking" steps. This is naturally captured by the *lexical product* operator, where $A \otimes B$ denotes the lexical product of algebras $A = \langle \Sigma_A, \preceq_A, \mathcal{L}_A, \oplus_A \rangle$ and $B = \langle \Sigma_B, \preceq_B, \mathcal{L}_B, \oplus_B \rangle$. Each link label for a link uv in the resulting algebra is a pair, consisting of the labels for uv in A and B. Similarly, each signature for a path P is a pair composed of signatures from A and B. The concatenation function is the pairwise concatenation of the labels and signatures. The preference relation is also pairwise in lexical order: the first components are compared using $\preceq_A$, if equal then the second components are compared using $\preceq_B$. For instance, the widest shortest hop-count policy is the lexical product of a policy that prefers higher bandwidths with a policy that prefers shorter paths.

B. Example: Gao-Rexford Guideline

To illustrate the use of algebra to encode policy guidelines, we consider an example based on business relationships between neighboring ASes. We focus our discussion on a policy guideline where an AS prefers routes through customers over routes through peers or providers (called "guideline A" in [9]). Similar algebraic encodings have been presented in prior work [36], but our illustration here serves to highlight a shortcoming of existing algebraic representations, in addition to being used as one driving example in the paper.

Links and paths are distinguished based on their attributes, mapping naturally to label set $\mathcal{L}$, and signature set Σ, respectively. Consequently, the representation of the policy guideline in algebra is straightforward:

Link labels and route signatures. Routes are classified based on the business relationship between neighboring ASes. Routes received from a customer, provider, or peer are classified with path signatures *C*, *P*, and *R*, respectively. In addition, the signature ϕ explicitly denotes all prohibited routes. Therefore, $\Sigma = \{C, P, R, \phi\}$. Likewise, labels $c/p/r$ denote three classes of links to customers, providers, and peers, and $\mathcal{L} = \{c, p, r\}$.

Preference relations. Each AS prefers routes via customers over those via providers or peers, which is straightforwardly encoded as $C \prec P$ and $C \prec R$. To have a total ordering on the signatures, we must define a preference relation between provider (P) and peer (R) routes. Using $P = R$ implies that an AS can decide which routes are preferred based on other tie-breaking methods. So, our encoding uses the following three constraints: $C \prec P$, $C \prec R$, and $P = R$.

Concatenation. The signature of a new route depends only on the node's relationship with its neighbor, as captured by the link label; for example $p \oplus C = P$, $p \oplus R = P$, and $p \oplus P = P$. However, an AS does not export routes learned from one peer or provider to other peers and providers, as illustrated in Figure 2. The figure shows a node u deciding whether to export (to its neighbor v) a route to destination d. Figure 2(a) shows that u is a provider of v, making uv a provider link and vu a customer link. Node u can export customer routes (C) to v, but any peer (R) and provider (P) routes are filtered. In algebra, route filtering is expressed by generating a prohibited path (ϕ). For import policies, if v decides not to import a path of signature s from u, we can encode this import policy as $l \oplus s = \phi$ where l is the label for link vu. Our example policy involves *export* filtering. An export filter at node u can be modeled as an import filter at the receiving node v. The export filters in Figure 2(a) can be represented by $c \oplus P = \phi$ and $c \oplus R = \phi$, where the customer v filters any routes that u learned from its own peers or providers. The complete definition of the concatenation operator is:

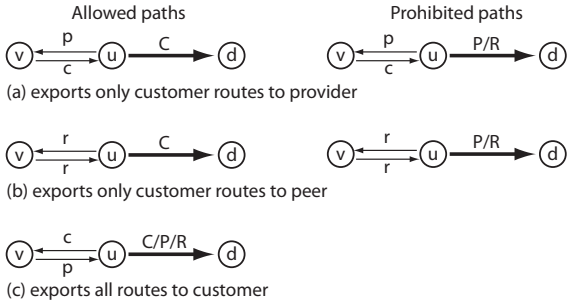

Fig. 2. Export policy for Gao-Rexford guideline. The bold line indicates a route to destination d, with an associated route signature. Each unidirectional link between nodes u and v has a link label.

$\oplus$	C	R	P
c	C	ϕ	ϕ
r	R	ϕ	ϕ
p	P	P	P

The row names refer to link labels (c, r, and p), while column names refer to signatures (C, R, and P). Note that the above table is used to enumerate all possible $\oplus$ operator outputs given input labels and signatures. A closed form policy such as shortest-path can be written by expressing $\oplus$ as a function directly, as we described in Section II-A. Converting export policies at one node to import policies at another is fine for analyzing safety—the two representations are equivalent. However, the distinction matters when generating distributed implementations, as we do in *FSR*. Our extended algebra clearly identifies which node performs the filtering, so *FSR* can generate faithful distributed implementations, as discussed in Section II.

III. Unified Policy Specification

FSR analyzes safety and generates a distributed implementation, given an input policy configuration. We support a wide range of policy configurations, ranging from high-level *guidelines* to specific *instances*, as summarized in Table I. For example, the shortest hop-count routing policy discussed in Section II does not specify the network topology but *completely specifies* the path preferences; in contrast, the Gao-Rexford guideline merely *constrains* the preferences and filters based on business relationships. In other settings, a researcher may analyze specific BGP "gadgets" that violate a proposed safety guideline; similarly, network operators may verify the safety of their network configuration. In these settings, the topology, preferences, and permitted paths are much more

Policy	Topology	Preferences	Filters
Hop-count	General	Specific	None
Gao-Rexford	General	Constrained	Constrained
IGP-cost	Specific	Specific	Constrained
SPP instance	Specific	Specific	Specific

TABLE I
SPECTRUM OF POLICY CONFIGURATIONS

concrete, and can be expressed naturally as instances of the Stable Paths Problem (SPP) [12].

While the existing algebra can express a wide range of policies, the concatenation operator does not indicate which node performs route filtering—the importing node, the exporting node, or a combination of the two. In this section, we introduce separate operators for import and export filtering, to enable automated translation from a policy configuration to a distributed protocol implementation. In the second half of this section, we propose an automatic way to translate SPP instances to an algebraic representation. Together, these two extensions enable *FSR* to automatically generate distributed implementations and analyze safety for a wide range of policy configurations.

A. Separating Import and Export Filters

The routing algebra in Section II does not distinguish whether routes are filtered during export or import—an important distinction when generating distributed protocol implementations. To specify the two filters separately, we replace the original $\oplus$ operator with three concatenation functions for *export* ($\oplus_E$) and *import* ($\oplus_I$) filtering, and a *simple concatenation* function for route generation $\oplus_P$. The result of $l \oplus_E s$ is either E (export), or F (filtered). For example, if node u does not export routes with signature s to node v, we can encode the export filter as $l \oplus_E s = F$, where the label of link uv is l; otherwise $l \oplus_E s = E$. The result of $l \oplus_I s$ is either I (import) or F (filtered). For example, if u does not import a path with signature s from w, we can encode this import filter as $l \oplus_I s = F$ where l is the label for link wu; otherwise $l \oplus_I s = I$. Whenever an incoming route advertisement is received, the import filter ($\oplus_I$) is first applied. If accepted, a new route is generated ($\oplus_P$), and exported after filtering ($\oplus_E$).

A similar extension that distinguishes between import and export labels was proposed in [39]. Their approach is equivalent to ours, for the purpose of safety analysis. We chose ours because it provides a straightforward translation to declarative networking implementations.

Revisiting the Gao-Rexford example from Section II-B, the three concatenation operators $\oplus_I$, $\oplus_P$, and $\oplus_E$ are defined as follows:

$\oplus_I$	C	P	R
c	I	I	I
r	I	I	I
p	I	I	I

$\oplus_P$	C	P	R
c	C	C	C
r	R	R	R
p	P	P	P

$\oplus_E$	C	P	R
c	E	F	F
r	E	F	F
p	E	E	E

Each row of the leftmost (rightmost) table corresponds to one import (export) policy in Figure 2, from top to bottom; for example, in the rightmost table, the first row exports only customers routes to a provider. Since there are no import restrictions, the leftmost table has I for all its entries. The center table shows the $\oplus_P$ operator, where new routes have their signatures (C, R, and P) set according to the labels (c, r, and p) respectively.

For safety analysis, we need to combine the import and export filters into a single concatenation operator ($\oplus$). At a high level, this is as simple as assigning the signature ϕ (for prohibited paths) to any path filtered by *either* the import or the export policy. However, for a path $vu \circ P_{ud}$, the import filter at v depends on the label l of the link vu, but the export filter at u depends on the label $\bar{l}$ of the reverse link uv. We can generate $\oplus$ as follow: for each label l and signature s, if $\bar{l} \oplus_E s = F$ or $l \oplus_I s = F$, then $l \oplus s = \phi$; otherwise $l \oplus s = l \oplus_P s$. In the Gao-Rexford example, neighboring ASes have a bilateral business relationship, leading to link labels of $\bar{p} = c$, $\bar{c} = p$, and $\bar{r} = r$ and the combined $\oplus$ table shown earlier in Section II-B.

In addition to the above Gao-Rexford guideline example, our algebra extensions can be used to specify a variety of import and export filters. For example, if the signature includes the entire AS path, we can easily specify an import (export) policy that disallows routes that traverse a particular AS, by expressing $\oplus_E$ ($\oplus_I$) to output F values whenever a route passes through a particular AS. The lexical product [13] can then be used to compose multiple policies, for instance, combining the Gao-Rexford guideline with a policy that excludes particular paths by AS.

B. Converting SPP Instances to Algebra

Researchers and network operators often want to analyze concrete policy configurations to explore small "gadgets" that violate a policy guideline or verify a real network configuration is safe. The Stable Paths Problem (SPP) [12] is a common way to represent concrete policy configurations. An SPP instance consists of a topology, where each node has a ranked list of "permitted paths" that it could learn from its neighbors. For real router configurations, one can use existing approaches [28] to extract per-node rankings. In the absence of these router configurations, SPP instances can still be extracted by observing actual protocol executions over a period of time.

As an illustrative example, Figure 3 as presented in [6] shows an example internal BGP (iBGP) configuration, where the squares (a, b, and c) are route reflectors and the circles (d, e, and f) are egress nodes that each have an externally-learned route (r_1, r_2, and r_3) to the destination. The solid lines denote iBGP sessions (labeled with its IGP cost) and dotted lines denote additional (IGP) links. Each node has an ordered list of permitted paths, ranked from most to least preferred; for instance, node a prefers the route $aber_2$ over adr_1.

To convert an SPP instance to an algebraic representation, we need an automatic way to construct the equivalent link labels, path signatures, preference relations, and concatenation operator. [19] has shown a formal translation of SPP to routing algebra . While the goal of our translation is the same as theirs, the algebra we use is of a slightly different form. At a high level, we assign a unique label to each link, and a unique signature to each path. Then, we convert the ranking of

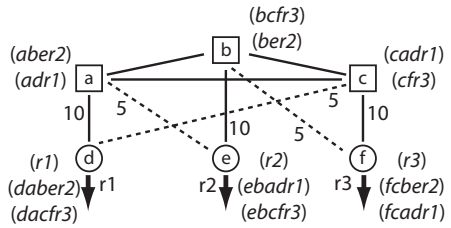

Fig. 3. iBGP configuration instance

permitted paths into a series of preference relations, and define the concatenation function to connect the permitted paths and exclude any filtered routes. Using the SPP instance in Figure 3 as an example:

Labels and signatures. Since a concrete configuration does not have any meaningful classification of links and routes, we assign each link uv a unique label constant l_{uv}, and each permitted path $p = u_n \cdots u_0$ a unique signature r_p. In our iBGP example, the label set $\mathcal{L}$ is $\{l_{ab}, l_{ac}, l_{ad}, l_{ba}, l_{bc}, l_{be}, l_{ca}, l_{cb}, l_{cf}, l_{da}, l_{eb}, l_{fc}\}$, and the signature set Σ is $\{r_1, r_2, r_3, r_{aber2}, r_{adr1}, r_{bcfr3}, r_{ber2}, r_{cadr1}, r_{cfr3}, r_{daber2}, r_{dacfr3}, r_{ebadr1}, r_{ebcfr3}, r_{fcber2}, r_{fcadr1}\}$.

Preference relations. Each node has a ranked list of permitted paths, of the form $r_1, r_2, ...r_n$. We translate this list into the equivalent pairwise preferences: $r_1 \prec r_2, r_2 \prec r_3, ...r_{n-1} \prec r_n$. For instance, at node a, $r_{aber2} \prec r_{adr1}$. The preference relation is defined as the collection of preference relations at each node.

Concatenation. The $\oplus$ operator constrains the relationship between label and signature constants. In particular, for any permitted path r_{uvp} at node u, $r_{uvp} = l_{uv} \oplus r_{vp}$; for instance, $r_{aber2} = l_{ab} \oplus r_{ber2}$. Any other paths are disallowed by assigning the signature ϕ; for instance, $l_{cb} \oplus r_{ber2} = \phi$ because path $cber_2$ is not listed as a permitted path.

Using the above process, we have also encoded various eBGP gadgets [12] in algebra as SPP instances.

C. Expressiveness of Algebra

Routing algebra is a useful abstraction for specifying a wide range of routing policies, which we demonstrate through our case studies (Section VI). There are a few known limitations with routing algebra as defined in [36], [13]. For instance, the inability to express the incompatibility of MEDs from two different neighbors. However any concrete routing configuration that uses MED can be easily encoded in algebra as SPP instances described in Section III-B, which is what is needed by *FSR* to analyze concrete iBGP configurations.

IV. Automated Safety Analysis

Given any algebra, *FSR* fully automates the process of safety analysis, relieving users from the manual and error-prone process of proving safety for each new algebra. The key insight is that the safety analysis can be translated automatically into integer constraints checkable by a standard SMT solver. Applying our technique of encoding SPP instances using algebra (Section III-B), *FSR* can check safety for both high-level policy guidelines and concrete configurations. After a brief review of safety analysis based on routing algebra, we explain how to generate the integer constraints and present three examples that illustrate the conversion process and resulting safety analysis.

A. Strict Monotonicity Implies Safety

FSR uses the safety requirement of *strict monotonicity*, in order to automatically check that a given policy configuration converges. This is an important property of a routing algebra, which ensures that a path does not become more preferred as it grows longer. Formally:

Monotonicity: $s \preceq l \oplus s,\ \forall l \in \mathcal{L}, \forall s \in \Sigma$

Monotonicity ensures that a path P_v from v to the destination is not less preferred than a longer path $uv \circ P_v$, where uv is a link from u to v. In the definitions above, s represents the signature of P_v and l is the label for uv.

FSR uses the stricter form of monotonicity, where l and $l \oplus s$ cannot be equally preferred, defined as follows:

Strict Monotonicity: $s \prec l \oplus s,\ \forall l \in \mathcal{L}, \forall s \in \Sigma$

Sobrinho has proved the following theorem [36].

Theorem 4.1: If the routing algebra is strictly monotonic, then the path-vector protocol converges.

Theorem 4.1 reduces the convergence analysis of protocols to modeling the routing policies in a routing algebra, and proving that the algebra is strictly monotonic. Note that strict monotonicity is a *sufficient*, not *necessary* condition. Hence, there are safe systems that cannot be specified as a strictly monotonic algebra. Consequently, *FSR* will report these systems as unsafe (i.e. false positives). However, this sufficient condition is still very useful in practice to analyze the safety of BGP systems.

The strict monotonicity is actually the most general condition known that guarantees safety regardless of the network topology. This enables researchers and network operators using *FSR* to benefit from this theoretical result by providing automated tool support.

B. Converting Policies to Yices Constraints

Policy configurations expressed in routing algebra have a natural representation as integer constraints. Path signatures can be mapped to integers, and path preferences can be expressed as comparisons ($\leq$) between these integers. By definition [36], the preference relation $\preceq$ needs to be a total relations, and $\leq$ is indeed a total order. This mapping is also complete because we can always map the signatures onto the integer domain, when the $\preceq$ is a total order. Strict monotonicity imposes additional constraints on the preference relation, also naturally captured by comparing integers. This observation allows *FSR* to leverage SMT solvers, which determine whether a set of constraints (i.e., first-order logic formulas) are satisfiable based on a set of theories (e.g., integer theory). Translating from algebraic input to SMT constraints is straightforward, making this approach preferable to other alternatives (e.g., SAT solvers) that would require greater effort to generate encoding.

In addition, an SMT solver produces valuable output, beyond the basic "yes/no" answer. If the constraints can be

satisfied, the solver returns a concrete instance (example) that satisfies all of the constraints. For instance, we consider the simple constraint $x < 2$ when x is an integer. An SMT solver can prove that there exists a value instantiating x that makes $x < 2$ true, and returns $x = 1$ as an example. If the constraints cannot be satisfied, the solver returns a smallest subset of constraints that are not satisfiable—an invaluable aid in identifying the problematic parts of the policy configuration. In our *FSR* implementation, we utilize the Yices [42] SMT solver, although the technique we present here can be applied to SMT solvers in general. Our technique generalizes to other SMT solvers well because our encoding only requires the basic integer theory which is readily included in most SMT solvers.

Input to SMT solver: Given a policy configuration written in routing algebra, *FSR* generates integer constraints for safety analysis recognizable by the solver. *FSR* generates two kinds of constraints based on the sufficient conditions required for safety in Section IV-A: (1) route preference constraints based on $\preceq$ relation (2) strictly monotonic constraints based on $\oplus$ function. *FSR* automatically generates these integer comparison constraints, allowing us to leverage Yices built-in integer support for enforcing total ordering. More concretely, we generate constraints from the algebraic specification via the following three steps:

- **Step 1:** For each signature, we define a variable of the type positive integer.
- **Step 2:** For each $s_1 \preceq s_2$ in the specification, we generate a constraint $s_1 \leq s_2$. Since signatures are integers, the $\leq$ relation imposes a total ordering.
- **Step 3:** For any signature s and s', and label l, for each definition of $s' = l \oplus s$ in the specification, a constraint $s < s'$ is generated. This constraint enforces strict monotonicity. To check for (non-strict) monotonicity, we could generate $s \leq s'$ instead.

SMT solver output: The conjunctions of all constraints are checked by Yices for satisfiability. If Yices returns `sat`, an assignment of integers to variables (signatures) exists that satisfies all of the constraints. This means that the algebra is strictly monotonic, and by Theorem 4.1, any path-vector protocol that implements the policy configurations converges. On the other hand, if Yices returns `unsat`, specific input constraints that form an *unsatisfiable core* are provided. Unsatisfiable core (or unsat core) is a minimal set of inputs constraints that cannot be conjunctively satisfied. It is often significantly smaller than the set of input constraints.

Given the natural mapping of the original input specifications in algebra and Yices constraints, one can easily identify the preference relation for each violating constraint. The user can use these violating preferences as hints to identify (and fix) specific problematic parts of the policy configuration. Note that, there can be multiple unsatisfiable cores (i.e. many configuration conflicts), and Yices only outputs one of them at each invocation. To fix all the configuration problems, the user can attempt removing all unsatisfiable cores one by one in an iterative fashion.

Policy compositions: The lexical product (Section II-A) of a monotonic algebra and a strict monotonic algebra is strictly monotonic [13]. For policy configurations in the form of a lexical product over algebras, safety analysis can be performed by analyzing each algebra separately. Consider the lexical product $A \otimes B$ of two algebras A, B. Analysis starts from algebra A, and if it is strictly monotonic, the composed policy is safe. If A is monotonic, then B is checked. If B is strictly monotonic, then the composed algebra is safe, otherwise it is deemed unsafe. If A is not even monotonic, then the composed policy is deemed unsafe.

C. Yices Examples

We present several examples to demonstrate the three-step process of generating Yices constraints from algebraic input and the analysis process.

Shortest Hop-Count: We start with the simplest example using shortest hop-count. The algebraic specification of this policy is presented in Section II-A. We show the Yices encoding of the constraints below:

```
(define-type Sig (subtype (n::nat) (> n 0)))
(assert (forall (s::Sig) (< s s+1)))
```

The first line declares a type (`Sig`) for signatures, which is the subset of positive integers. Yices provides the built-in type `nat` for integers. Yices uses prefix syntax, so $n > 0$ is encoded as $(> n\ 0)$. Step 1 and 2 are omitted since the signatures are already integers, and the preference relation $\preceq$ is already specified using $\leq$.

The second line corresponds to step 3, and encodes the strict monotone constraint. `assert` is the keyword to tell Yices to insert this constraint into the logical context to be checked for satisfiability. Since the domain of the signatures is infinite, we cannot enumerate all strict-monotonicity constraints; instead, we universally quantify using `forall` over all signatures.

As expected, Yices returns `sat` for this policy.

Gao-Rexford Guideline A: Our second example analyzes the safety of Gao-Rexford guideline A, with the routing algebra presented earlier in Section II-B. The constraints are expressed in Yices as:

```
(define-type Sig (subtype (n::nat) (> n 0)))
(define C::Sig)    (define P::Sig)    (define R::Sig)
;; preference relations
(assert (< C R))   (assert (< C P))   (assert (= R P))
;; strict monotonicity
(assert (< C C))   (assert (< C R))   (assert (< C P))
(assert (< R P))   (assert (< P P))
```

The first four statements define the three classes of signatures—customer (C), provider (P), and peer (R)—as positive integers (step 1). The next three constraints correspond to step 2, encoding the route preference constraints of $C < R$, $C < P$ and $R = P$. The next five constraints correspond to entries in the combined concatenation operator in Section II-B after omitting constraints in the form $S < \phi$, which are already ensured to be true because any signature is strictly preferred over the signature for prohibited path ϕ by definition. This corresponds to step 3, which encodes strict monotonicity of the algebra.

Interestingly, Yices returns `unsat` for the above input, indicating that the algebra is not strictly monotonic. One of the violating constraints is resulted from $c \oplus C = C$, which states that a customer route that is sent from a customer link is still a

customer route. This is a known property of the Gao-Rexford guideline, which requires an additional constraint on acyclicity in the customer-provider relationship for safety.

Another approach to guaranteeing safety in Gao-Rexford is to use another algebra that is strictly monotonic as the tie breaker, in the event of a tie between two equally preferred route classes (e.g., provider and peer, or routes from same classes). As an example of policy composition, we first use Yices to prove that the algebra encoding guideline A is *monotonic* ($s \preceq l \oplus s \; \forall l \in \mathcal{L}, \forall s \in \Sigma$), and then compose guideline A with a strictly monotonic algebra such as shortest hop-count; the resulting protocol converges.

To perform the above analysis, we modify the strict monotonicity constraints in the above Yices encoding to check for monotonicity constraints. This requires changing each $<$ to $\leq$, e.g. `(assert (<= C C))`, etc. When we check the constraints, Yices returns `sat`, and provides a possible instantiation `C=1`, `P=2`, `R=2`.

In addition to guideline A, we have applied *FSR* to analyze a number of guidelines including Gao-Rexford guideline B [9] and also guidelines that ensure safe backup routing [8].

Internal BGP Configuration Instance: As our final example, we use *FSR* to analyze the six-node iBGP configuration in Figure 3, using our technique for encoding SPP instances in algebra (Section III-B). In Section VI, we present our experiences analyzing a larger network based on the Rocketfuel [37] dataset, and also the analysis of well-known eBGP gadgets.

We use the same three-step process to generate solver constraints. First, each permitted path in the SPP instance is mapped to an integer variable. Second, a constraint is generated for each route preference, derived from the per-node rankings in SPP. Finally, for each entry of the concatenation function, we generate a strict monotonic constraint as described in step 3. All in all, eighteen constraints are generated.

The number of constraints depends on the number of permitted paths which, in turn, depends on the network topology. In contrast, the previous two examples are independent of the network topology. These differences reflect the broad applicability of *FSR*, in analyzing policy configurations that range from partially to fully specified (see Table I).

For these input constraints, Yices returns `unsat`, meaning that the algebra violates strict monotonicity. In fact, this iBGP system is known to be unsafe [6].

However, given eighteen constraints, pinpointing the problem manually is quite difficult. For larger networks with even more constraints, manual analysis becomes even harder. Fortunately, Yices can generate the minimal set of constraints (unsat core) that cannot be satisfied. More details on pinpointing configuration problems with unsat core are in Section VI-B.

Here, the unsat core includes the rankings of nodes a, b and c's and strict monotonicity constraints involving available routes of those nodes, but does *not* include constraints for the route preferences of node d, e, and f. This leads to the conclusion that there are potential problems with the configurations of the route reflectors a, b, and c. In fact, each reflector prefers other reflector's client over its own, which causes an oscillation [6].

Once the problem is identified, the network operator can change the network settings such as topology, so that the route preferences of nodes a, b, and c are changed, and use *FSR* to analyze the safety of the new configuration. As validation, we rerun Yices with a modified configuration that does not include the preference cycle among the reflector nodes, and the solver returns `sat`.

Soundness of SPP Safety Analysis. For an SPP instance, each AS only knows the preference relation among the routes that are in its own routing table, and the policy configurations do not enforce any order among routes that originate from different nodes. However, the safety analysis of the routing algebra requires a total ordering of all routes. The output of `sat` means that there exists one strictly monotonic algebra that extends the route preference relation specified in policy configurations to be a total order. Applying Theorem 4.1 directly, we know that a protocol that implements this extended algebra is safe. We argue that a protocol that implements the extended algebra has exactly the same behavior as a protocol that implements the original policy configurations where the preference relation is only a partial order. The reason is that additional preferences in the extended relation describes preferences between routes that have different sources, which are not relevant to the route selection process in practice anyway; and therefore will not affect the protocol behavior.

V. Distributed Implementations

In addition to convergence analysis, *FSR* generates a protocol implementation from the policy configurations. *FSR* uses a declarative networking language called *Network Datalog* (*NDlog*) as an intermediary language to bridge the gap between the abstract routing algebra and efficient distributed implementations.

Our choice of *NDlog* is motivated by the following. First, the declarative features of *NDlog* allows for straightforward translation from the algebra to *NDlog* programs. Second, *NDlog* enables a variety of routing protocols and overlay networks to be specified in a natural and concise manner. In fact, *NDlog* specifications are orders of magnitude less code than imperative implementations. For example, traditional routing protocols such as the path vector and distance-vector protocols can be expressed in a few lines of code [24], and a more complex protocol such as the Chord distributed hash table can be expressed in 47 lines. This makes possible a clean and concise proof (via logical inductions) of the correctness of the generated *NDlog* programs with regard to the algebra. The compact specifications also makes it easy to incorporate alternative routing mechanisms to the basic path-vector protocol, as we will later demonstrate in our evaluation section. Finally, when compiled and executed, these declarative protocols perform efficiently relative to imperative implementations [22].

Declarative techniques have been widely used for distributed protocol implementations, including, overlay networks [23], fault tolerance protocols [35], cloud computing [1], sensor networks [2], overlay network compositions [25], and wireless

routing [21]. All these are evidences of *NDlog*'s widespread applicability.

In *FSR* prototype, we use the open-source RapidNet [31] declarative networking engine as a basis for executing *NDlog* programs. *NDlog* programs are compiled by RapidNet into distributed execution plans that are based on the Click [20] execution model. Our generated *NDlog* implementation is composed of two components: one implements the routing mechanisms, the other implements the routing policies. *FSR* provides a built-in module implementing the path-vector mechanism, which we discuss in detail in Section V-A. The component implementing policies is directly translated from the algebra. In Section V-B, we show how *FSR* translates the algebra into *NDlog* programs.

A. Generalized Path Vector Mechanism

FSR takes as input, a *generalized path-vector* protocol, as its default routing mechanism. The *NDlog* implementation is shown below, and for the rest of this paper we refer to it as the GPV program. GPV implements a path-vector protocol that computes the most preferred path based on a routing algebra.

NDlog is a distributed variant of Datalog. An *NDlog* program is composed of several *rules*. Each rule has the form `p :- q1, q2, ..., qn.`, which can be read informally as "`q1` and `q2` and ... and `qn` implies `p`". Here, `p` is the *head* of the rule, and `q1`, `q2`,...,`qn` is a list of *predicates* that constitutes the *body* of the rule. A rule is triggered (evaluated) once all the body predicate values (tuples) are generated. Once triggered, the head tuple is generated. Rule execution is done in a continuous, long-running fashion using a distributed query processor, where rule head tuples are continuously updated (inserted or deleted) in an incremental fashion [26] as the body tuples are updated.

```
//GPV program
gpvRecv sig(@U,SNew,PNew) :- msg(@U,V,D,S,P),
         PNew=f_concatPath(U,P), V=f_head(P),
         SNew=f_concatSig(L,S), label(@U,V,L),
         f_import(L,S)=true.

gpvStore route(@U,D,S,P) :- sig(@U,S,P), D=f_last(P).

gpvSelect localOpt(@U,D,a_pref<S>,P) :- route(@U,D,S,P).

gpvSend msg(@N,U,D,S,P) :- localOpt(@U,D,S,P),
          label(@U,N,L), f_export(L,S)=true.
```

In *NDlog*, the names of predicates, function symbols, and constants begin with a lower-case letter, while variable names begin with an upper-case letter. Similar to most implementations of Datalog, *NDlog* includes a limited set of function calls beginning with "`f_`", and user-defined arithmetic functions beginning with "`a_`". These functions include boolean predicates, arithmetic computations, and simple list operations.

The above program manipulates the following tuples. `label(@U,V,L)`[2] tuples, where each tuple represents an edge from the node itself (`U`) to one of its neighbors (`V`) of attribute `L`. A set of computed routes, stored as `sig(@U,S,P)` tuples at each source node `U`, where `S` and `P` are the signature and path of the route respectively. Route advertisement messages exchanged among nodes are represented by `msg(@U,V,D,S,P)` tuples. Each tuple denotes a message that is sent by node `V` to `U`, and the advertised route is for destination `D` with path `P` and signature `S`. We provide a high-level description of the above program, broken down by rules:

- **Receiving routes.** Rule `gpvRecv` is triggered upon receiving a route advertisement (`msg` tuple) from a neighboring node. Based on the route advertisement, the rule generates a new route with a new path `PNew` and a new signature `SNew`. The `f_concatSig` implements the simple concatenation function $\oplus_P$, while the function `f_import(L,S)`, implements the import filter $\oplus_I$ in algebra. It evaluates to true if and only if `L`$\oplus_I$`S`$= I$.
- **Storing routes.** Rule `gpvStore` builds a `route` table at each node, which stores all the candidate routes to the destination, by using the information in its locally maintained `sig` table.
- **Selecting routes.** Rule `gpvSelect` computes the optimal route (represented as `localOpt` tuples) based on the `route` table. The user-defined aggregate function `a_pref` computes the optimal route by using the route preference function `f_pref` (as its comparison function) , which implements the $\preceq$ relation in algebra.
- **Sending routes.** Rule `gpvSend` propagates new routes to neighbors. Whenever a node's local optimal routes `localOpt(@U,D,S,P)` to destination `D` is updated, the updated route is re-advertised to all neighbors `N`. Similar to import policies, we use the `f_export` function to filter out routes: rule `gpvSend` only generates a message if the route is not filtered by the export policy. `f_export(L,S)` implements $\oplus_E$, and it returns true if and only if `L`$\oplus_E$`S`$= E$.

GPV provides a template for users to plug in customize policy configurations. One of the advantages of using *NDlog* is its ease of incorporating routing policies in algebraic form with routing mechanisms (e.g. GPV). Signature generation is achieved by performing a predicate unification of labels and signatures recursively in *NDlog* rules, and applying the appropriate function (`f_concatSig`) for generating new signatures. The recursive signature generation (from other signatures) is encoded in only 4 rules in *NDlog*. Import and export filters are simply boolean functions (`f_export`, `f_import`) in rule bodies which are triggered when true. While it is certainly possible to use an imperative language instead, *NDlog* provides the right balance of features in terms of compact specifications, ease of proofs and translation from algebra.

Algebra	*NDlog* Predicates / functions
$\preceq$	`f_pref`
$\oplus_P$	`f_concatSig`
$\oplus_I$	`f_import`
$\oplus_E$	`f_export`

TABLE II
ALGEBRA AND *NDlog* MAPPING.

B. Converting Policies to NDlog Functions

Table II summaries the correspondence between definitions in algebra, and the function names in the generated *NDlog*

[2] *NDlog* supports a *location specifier*, expressed with "`@`" symbol followed by an attribute. This attribute is used to denote the source location of the corresponding tuple. For example, `label` tuples are stored based on the value of the `U` attribute.

programs. We use the extended algebra introduced in Section III-A, which distinguishes between simple concatenation function $\oplus_P$, import filter $\oplus_I$, and export filter $\oplus_E$. Functions `f_pref`, `f_concatSig`, `f_import`, and `f_export` are directly generated from input routing algebra as follows:

- **Step 1.** For each $s_1 \preceq s_2$ in the specification, add a statement to `f_pref(S1,S2)` that returns true if `S1`$= s_1$ and `S2`$= s_2$.
- **Step 2.** For any signature s and s', and label l, for each definition of $s' = l \oplus_P s$ in the specification, generate a statement in `f_concatSig(L,S)` that returns s' if `L`$= l$ and `S`$= s$.
- **Step 3.** For any signature s and label l, if $l \oplus_E s = F$, generate a statement in `f_export(L,S)` that returns `false` if `L`$= l$ and `S`$= s$. Similarly define `f_import(L,S)` for import filter $\oplus_I$.

To deploy the *NDlog* implementation on a concrete topology, each router takes additional configuration information automatically generated from the topology:

- **Step 4.** For each link in the input topology, generate a corresponding `label` tuple (assigned a value from the set $\mathcal{L}$). A `sig` tuple is also generated for each one-hop path to the destination. Signatures associated with these one-hop paths are typically known as the *origination set* [13], a subset of Σ defined as part of the input algebra.

Note that the above steps can be generated on a *per-node* basis, based on each node's input algebra. If the algebra directly uses functions and relations for which *NDlog* has built-in support (e.g. integer arithmetic), then steps 1 to 3 can simply use *NDlog*'s built-in functions.

Policy Composition: If the network designers choose to use the compositional feature of the routing algebra, the compositional operators can be straightforwardly mapped to *NDlog* templates as well. In particular, the lexical product of two policy algebras can also be concisely represented in *NDlog* by encoding the labels and signatures as a pair, and customizing the `f_pref` comparator function to check the first attribute, and then the second attribute in the case of a tie-breaker. We omit a detailed discussion due to space constraints.

C. NDlog Examples

We present examples to demonstrate the process of generating *NDlog* programs from input algebra.

Shortest Hop-Count: For shortest hop-count, the label for each link is 1, so for a node `u`, for each of `u`'s neighbor `v`, *FSR* generates a tuple `label(@u,v,1)`. If `u` has a direct link to the destination `d`, then a tuple `sig(@u,1,[ud])` is defined. This completes Step 4.

Next the concatenation and preference function are generated (Step 1 and 2). The concatenation function is defined as integer addition, and the preference relation is integer $\leq$ relation.

```
#def_func f_concatSig(L,S) { return L+S }
#def_func f_pref(S1,S2) { return S1 <= S2 }
```

Finally, the shortest hop-count policy does not have any import or export filtering, so they are the constant `true` function (Step 3).

```
#def_func f_export(L,S) { return true }
#def_func f_import(L,S) { return true }
```

Gao-Rexford Guideline A: Based on the network topology, for a node `u`, *FSR* generates a `label(@u,v,ch)` tuple for each of its neighbor `v`, and `ch` is 'c' if `v` is `u`'s customer; 'p', if `v` is `u`'s provider; and 'r', if `v` is `u`'s peer. Similarly, for each initial route of length 1, `sig(@u,ch,[ud])` is defined and `ch` is 'C' if the link `ud` is a customer link; 'P', if `ud` is a provider link; and 'R', if `ud` is a peer link. This corresponds to Step 4.

Next, in Step 1 and 2, definitions for functions implementing $\oplus_P$ and $\preceq$ are generated as follows.

```
#def_func f_concatSig(L,S) {
  if (L=='c') && (S=='C') return 'C'
  if (L=='c') && (S=='P') return 'C'
  if (L=='c') && (S=='R') return 'C'
  if (L=='p') && (S=='C') return 'P'
  ....  }

#def_func f_pref(S1,S2) {
  return (S1=='C' && S2=='R') ||  // C < R
         (S1=='C' && S2=='P')     // C < P  }
```

`f_concatSig` returns the signature `S` based on the link `L`, as defined by the earlier input algebra $c \oplus_P * = C$, $p \oplus_P * = P$, $r \oplus_P * = R$, where $*$ stands in for any signature C, P, or R. For each entry in $\oplus_P$, *FSR* generates an `if` clause, and we omit the rest of the definitions. `f_pref` returns `true` if `S1` is a customer route (C). This forces a customer route to be preferred over a peer/provider routes (R and P respectively). This is a direct translation from the earlier input algebra for Gao-Rexford, namely $C \prec P$ and $C \prec R$.

Finally, import and export functions are generated based on the filters $\oplus_I$ and $\oplus_E$. Since guideline A does not specify import filters, `f_import` is the constant function that always returns `true`. The export function returns `true` if the route is not filtered ($l \oplus_E s = E$); `false` if the route is filtered ($l \oplus_E s = F$).

```
#def_func f_import(L,S) { return true }
#def_func f_export(L,S) {
  if (L=='c' && S=='P') return false
  if (L=='c' && S=='R') return false
  if (L=='r' && S=='P') return false
  if (L=='r' && S=='R') return false
  return true       }
```

SPP Instances: Since SPP imposes explicit rankings, the `f_pref` function would compare signatures for a given source/destination pair. Based on per-node rankings of paths, `f_pref(S1,S2)` will return true if `S1` is preferred over `S2`, and false otherwise. To speed up the comparison process, one possible optimization (enhancement to step 2) is to store the per-node rankings in an ordered table for fast retrieval. Similarly, for export filters, one can maintain a table of permitted paths to be exported, and the `f_export` simply checks that a particular path is in the permitted export list, before it is exported. Import filters can be implemented similarly.

D. Correctness of NDlog implementation

In order to apply Theorem 4.1 and show that the *NDlog* implementation of a strictly monotonic algebra converges, we need to show the correctness of the *NDlog* implementation. The correctness depends on two conditions: first, the *NDlog* program correctly implements the path-vector protocol, and second, the *NDlog* program correctly implements the input

algebra. Prior work has experimentally validated [22] and formally proven [40] the correctness of an *NDlog* implementation of the path-vector protocol. In addition, [30] has formally proven correct *NDlog*'s operational semantics. We hence focus on the second condition.

We introduce several notations to set up our proofs. We define ι to be a function that maps the set of links in the network topology to the set of labels in $\mathcal{L}$. Given a concrete network topology, ι is the correct assignment of labels to links, i.e. $\iota(uv) = l$ if the label of link uv is l. The function σ_0 maps initial routes (route of length 1) to their signatures. σ_0 is the correct signature assignments to initial routes. Given a destination d, $\sigma_0([ud]) = s$ if the signature of route $[ud]$ is s.

Given ι, σ_0 and an algebra $\mathcal{A}$, function $\sigma_{\iota,\sigma_0,\mathcal{A}}$ maps each route to its signature. When it is clear from the context, we omit the subscripts, and write σ.

$$\sigma(p) = \begin{cases} \sigma_0(p) & p = [\text{ud}] \\ \iota(uv) \oplus \sigma(p') & p = uv \circ p' \end{cases}$$

Finally we define a function $nd(t)$ that returns the *NDlog* term that represents t.

A key aspect is to prove that *NDlog* computes the signatures for routes correctly, formally:

Theorem 5.1 (Correctness of NDlog *translation):* Given any path p, if $\texttt{sig}(nd(u), nd(s), nd(p))$ is generated by *prog*, and $s \neq \phi$, then $s = \sigma(p)$.

Detailed proofs can be found in Appendix A

VI. Evaluation

We present several case studies that span analysis and implementation to demonstrate several ways of using *FSR*: (1) automatically generating a proof of safety or pinpointing configuration problems of both policy guidelines and specific instances, (2) empirically evaluating protocol dynamics and temporal properties that cannot be easily checked in formal analysis, and (3) deploying and evaluating alternative routing mechanisms.

In all cases, the inputs required to our tool for analysis and experimentation are the routing mechanism, input policies (specified in the form of algebra), and a network topology (synthetically generated or obtained from either CAIDA [3] or Rocketfuel [37]).

Evaluation environment. *FSR* provides an interface for users to specify policy configurations using algebraic specifications, which are compiled into *NDlog* programs. *FSR* uses the RapidNet [31], [27] declarative networking engine to compile the *NDlog* programs into applications (with an execution model similar to Click [20]) executable in ns-3 [29], an emerging discrete event-driven simulator similar to the popular ns-2. Like its predecessor, ns-3 emulates all layers of the network stack, supporting configurable loss, packet queuing, and network topology models. It also allows for a *simulation mode*, enabling a comprehensive examination under various network topologies and conditions, as well as an *deployment mode* where different hosts in a testbed environment execute the deployed system over a real network. The ability to run the same application in these two modes enables us to execute each *NDlog* program at scale in simulation and in an actual implementation running on a testbed, providing two avenues for augmenting the formal analysis.

A. Convergence Time vs. Network Size

Our first case study presents a scenario where a researcher empirically evaluates policy guidelines using the distributed *NDlog* implementation automatically generated from the algebraic specifications. To ensure strict monotonicity, we compose the basic Gao-Rexford guideline A policy with the shortest hop-count as the tie-breaker (using algebra's composition operator), as described in Section IV-C. The researcher has already analyzed the composed policy for its safety properties using Yices, but would like to measure the convergence time with respect to the depth of the AS hierarchy. A prior study [32] proved that, the worst case upper-bound of the convergence time for Gao-Rexford guideline is $2 \times (d + 1)$ *phases* (rounds of route advertisements), where d is the length of the longest customer-provider chain. The researcher can use the implementation that *FSR* generates from GPV (Section V), and policy configurations (Section III-A). We present our results using RapidNet's simulation and deployment modes.

Simulation mode: Our first experiment is carried out in RapidNet's simulation mode. As our input topologies, we utilize the AS-level network graph (with annotated customer-provider relationships) provided in the CAIDA dataset [3]. The simulation is performed in a quad-core machine with Intel Xeon 2.33GHz CPUs and 4GB memory running Linux 2.6. In the simulation setup, all links have 100 Mbps in bandwidth and 10 ms latency.

To fit the simulation into memory and use a similar network size for our subsequent testbed evaluation, we extract subgraphs from CAIDA's global network topology as follows: we remove all stub ASes[3], randomly select an AS *R* as the *root*, and then extract the AS hierarchy (transitively) provided by the AS. We include an AS to be part of the subgraph rooted at *R* if there exists a route consisting only of peer/customer links from *R* to the AS. We choose 14 such subgraphs with the length of the longest customer-provider chains ranging from 3-16. For each subgraph, we executed the GPV protocol with guideline A, and measured the convergence time (from start of protocol until all nodes have computed routes to all destinations).

Figure 4 (CAIDA-Sim) shows the protocol convergence time as the length of the longest provider-customer chain increases. As a basis of comparison, we plot the theoretical worst-case convergence time [32]. Our protocol mechanism is configured to batch and propagate routes every second, a feature easily achieved using *NDlog*'s time-based predicates [23]. For instance, given the longest customer-provider chain of 10, the execution should converge within at most $2 \times (10 + 1)$ phases, namely 22 seconds. We make the following two observations from our simulation results. First, the convergence time increases linearly with the length of the longest customer-provider chain, validating the trend shown in the prior theoretical results. Second, we observe that, in practice, the protocol converges faster than the theoretical worst case. Upon further investigation based on execution logs, we realize the faster convergence is because customers at the "leaves" of the customer-provider tree typically have multiple

[3]The pruned topology contains 5220 ASes and 23101 links.

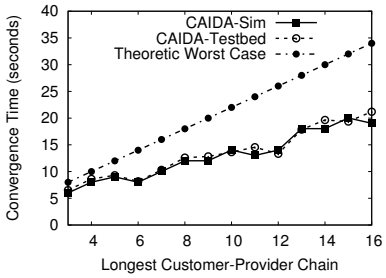

Fig. 4. Convergence time (seconds) for BGP against longest customer-provider chain.

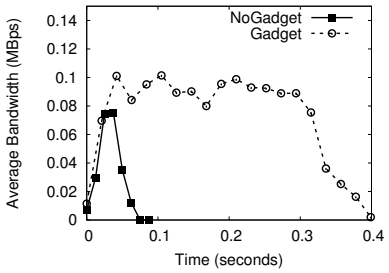

Fig. 5. Average per-node bandwidth utilization (MBps) for iBGP with gadget.

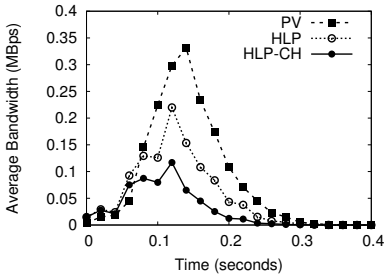

Fig. 6. Average per-node bandwidth utilization (MBps) for HLP.

paths to the root providers and can leverage peer-to-peer links, and hence rarely require the full depth to propagate routes.

Deployment mode: Our second experiment validates our simulation results using RapidNet's deployment mode. Here, we utilize 32 quad-core machines with a similar hardware/-software configuration as our simulation experiment. The machines are connected using high-speed Gigabit Ethernet. We run up to 5 RapidNet instances per machine, and configure the neighbor links among RapidNet instances to be consistent with the earlier CAIDA setup in simulation. As before, we set the propagation period to 1 second.

Figure 4 (CAIDA-Testbed) shows that the convergence time obtained in the testbed closely mirrors that of our earlier simulation results. Our tool can switch between simulation and deployment based evaluation easily. Simulation and deployment modes of RapidNet uses the same compiled code base, with a configuration flag indicating running the network stack in simulation or using actual sockets. In the rest of this section, we primarily present results obtained in the simulation mode.

All in all, our first set of experiments based on the Gao-Rexford guideline is encouraging. Not only are we able to use Yices to check the guideline for safety, we are able to (with minimal effort) generate distributed implementations that provide additional performance insights using actual Internet topologies.

B. Pinpoint iBGP Configuration Errors

We emulate a scenario where a network operator uses our *FSR* toolkit to study the safety properties of an existing iBGP network configuration. As the input topology, we utilize the intradomain topology (with inferred link weights) of AS 1755 from the Rocketfuel [37] dataset, which contains 87 routers and 322 links. Pairwise IGP costs are computed a priori based on the shortest paths. The iBGP reflector-client topology is synthetically configured as a 6-level hierarchy with 53 reflectors.

Given the above input topology, we execute a GPV protocol on all 87 routers, and have each router within the AS compute the best route to a remote destination outside the AS, under the condition that several egress routers are aware of external routes to this particular destination. At each router, the route preference is based on the IGP cost from the router to the egress routers, i.e., the route with the lowest IGP cost is selected. This policy is similarly configured using routing algebra, and compiled into *NDlog* implementations.

To experiment with *FSR*'s ability to detect configuration errors, we embed a gadget similar to Figure 3 into the iBGP topology. This embedding is achieved by selecting three neighboring routers from the graph and setting their IGP cost to the egress routers the same as those in Figure 3. One goal of our experiment is to see whether our tool can detect this unsafe gadget embedded in a larger network instance, and observe its performance implications in actual executions.

Safety analysis: The algebraic representation of the SPP instance of the network is extracted and analyzed for safety. In the absence of real router configurations, we extract the per-node rankings from *NDlog* implementation runs as follows. We execute the GPV protocol in *NDlog* on all 87 routers, and populate the permitted paths of each router based on its incoming route advertisements. These permitted paths are then sorted based on IGP costs described above, to generate per-node rankings. *FSR* directly translates these per-node rankings expressed in algebra into constraints used by our SMT solver to perform safety analysis.

In total, the extracted SPP instance contains 259 constraints generated for strict monotonicity, and 292 constraints for per-node rankings. On a quad-core machine in our testbed, the SMT solver returns `unsat` within 100 ms, and reports a minimal unsatisfiable core consisting of six constraints. Interestingly, these six constraints not only form a dispute wheel, but are also directly attributed to the routers in the embedded gadget that we deliberately introduced earlier. This provides a "hint" for network operators to fix the configuration error starting from the errant constraints.

Experimentation: Upon fixing the configuration errors, we experimentally evaluated both iBGP configurations implemented using *NDlog*. Similar to the earlier CAIDA experiments, all the links are set to 100 Mbps bandwidth, 10 ms latency, and up to 3ms jitter. Figure 5 shows our comparison of average per-node bandwidth utilization over time for the iBGP protocol with and without the embedded gadget (shown as *Gadget* and *NoGadget* after the fix). Compared with *Gadget*, we observe a 91% decrease in communication overhead, and 82% decrease in convergence time in *NoGadget*. Note that the embedded IBGP gadget [6] causes transient oscillation, and hence result in higher bandwidth utilization for *Gadget*, as compared to *NoGadget*.

C. eBGP Gadget Analysis

FSR's applicability extends beyond high-level guidelines and iBGP configurations. We briefly summarize our experiences of using *FSR* to analyze well-known eBGP gadgets: GOODGADGET, BADGADGET and DISAGREE [12]. These experiments highlight the use of *NDlog* implementations generated from *NDlog* implementations of SPP instances.

Analysis: The input algebra for these three gadgets are SPP instances described in Section III-B, where the algebra is used to encode per-node permitted paths and rankings. Our analysis results are as expected: GOODGADGET is safe, while BADGADGET and DISAGREE are unsafe. These results match the manual proofs in prior work [12], but are obtained automatically by our solver.

Experimentation: We further experimentally evaluate the gadgets using the automatically-generated *NDlog* implementation. In all cases, we provide an input topology, which contains one or more gadgets on a subset of the nodes. For GOODGADGET, as the number of gadgets increases, both the convergence time and communication cost increase. The increase is due to route recomputation, which occurs when a previously computed best path is overwritten by a longer path with a higher local preference. Nevertheless, all GOODGADGET scenarios converge as expected. On the other hand, the BADGADGET execution never converges—the protocol continued to transmit a high rate of update messages indefinitely. For DISAGREE, a gadget that can temporarily oscillate between two stable states before eventually converging, the protocols takes a longer time to converge as the percentage of *conflicting links* increases[4].

D. Alternative Routing Mechanism

While we have adopted GPV as the default mechanism, given that *FSR* is an extensible framework, other routing mechanisms can also be used, as long as they are implemented in *NDlog*. In our final case study, we demonstrate how researchers can supply *FSR* with a *different* routing mechanisms to study their impact on convergence behavior. We consider the *Hybrid Link-State and Path-Vector* (HLP) [38] protocol that has been proposed as an enhancement to the path-vector protocol. HLP capitalizes on the assumption that the ASes running BGP can be partitioned into domains that form a customer-provider hierarchy. HLP uses the regular link-state protocol within each customer-provider hierarchy, and a path-vector protocol (called *Fragment Path-Vector*, where paths that are internal to the hierarchy are hidden) across different hierarchies. We implement HLP in *NDlog* by using just 10 rules (11 rules if we also specify that internal paths are hidden).

We configure the network topology as a 10-domain network. Each domain is a 20-node acyclic hierarchical structure rooted by a top provider, where each node (with the exception of the top provider) has 1 or 2 providers. We configure the topology and policies within a domain based on the Gao-Rexford guideline A. Link latencies within one domain are set to 10 ms. In addition, there are a total of 84 cross-domain links throughout the network; these links are configured to have 50 ms latency. In all cases, links are set to have a bandwidth of 100 Mbps. For cost hiding, we set 5 as the threshold.

Figure 6 shows the bandwidth utilization of HLP over time, with and without cost hiding (shown as *HLP-CH* and *HLP*, respectively). As a basis of comparison, we execute the path-vector protocol (shown as *PV*). We note that as expected, *HLP* converges faster than *PV*, requiring 0.35 seconds compared to 0.4 seconds for *PV*. Moreover, the per-node communication cost for *HLP* and *PV* is 1.09 MB and 1.75 MB, respectively. *HLP-CH* further reduces the communication cost to 0.59 MB per node.

Beside HLP, other possibilities include multi-path routing protocols, and neighbor-specific BGP mechanisms, which typically require further customization to user-defined functions, for instance, propagating the top-k paths instead of the current best. Such comparisons are tremendously useful for researchers to study the full design space in both policies and mechanisms.

VII. RELATED WORK

We discuss two bodies of work most related to *FSR*:

Formal models for safe BGP systems. *FSR* does not propose new formal model or sufficient conditions for safe inter-domain routing. Instead, *FSR* leverages routing algebra [36], [13], and adds practical extensions in order to generate distributed implementations. By casting convergence analysis as a constraint satisfiability problem, *FSR* automates safety analysis of routing polices using SMT-solvers. *FSR* can be viewed as a practical toolkit that can be applied to recent advances in formal models [39], [6], [16], [19] for inter-domain routing.

Tools for BGP analysis. Existing tools for analyzing BGP today comes in the form of configuration checkers or runtime debugging of deployed systems. For instance, *rcc* is a tool for statically checking BGP configurations for possible faults. Other runtime debugging platforms include [5], [18]. Prior to deployment, one may also perform custom simulations, using platforms such as simBGP [34]. These systems are often debugged in a *post-mortem* fashion. Routing protocols are developed first, and then debugged over time as errors are uncovered. As a result, subtle bugs may require a long time to be encountered, and in some cases, once identified, errors are difficult to isolate in a distributed setting. While simulations provide an arguably more controlled environment, they are removed from actual implementations and also require the programmer to correctly implement the protocols in the simulator.

Compared with all of the above tools, the methodology of *FSR* is fundamentally different. The input to *FSR* are routing algebras that encode policy configurations. Instead of analyzing the implementation, *FSR* performs safety analysis on the algebra representation. *FSR* generate provably correct *NDlog* implementations from the algebra, and the safety results obtained in analysis carry over to the implementation.

[4] A conflicting link is a link where the two adjacent nodes always prefer to route through each other.

VIII. Conclusion

In this paper, we present *Formally Safe Routing* (*FSR*), a unified toolkit for analyzing and evaluating BGP policy configurations, ranging from high-level guidelines to specific network instances. *FSR* leverages recent advances in routing algebra and declarative networking. One key contribution of *FSR* is its ability to perform safety analysis and generate an implementation from the same algebraic representation of routing policy. We show that routing algebra has a very natural translation to both integer constraints and declarative networking programs. This allows research on inter-domain routing to leverage mature technologies, such as SMT solvers and RapidNet, to automate complex and error-prone tasks for researchers and practitioners alike.

Our experiences with *FSR* have been promising. Using our prototype, we have analyzed a wide range of policy configurations. We have combined safety analysis and experiments with the protocol implementations to pinpoint configuration errors and gain insights into the performance of existing protocols. To encourage widespread adoption, we plan to release *FSR* as an open-source toolkit for the research community. To illustrate *FSR* in action, please see [7] for a video demonstration of our current *FSR* prototype, based on the scenario used in Section VI-A.

Another interesting avenue of exploration is enhancements to routing algebra itself. As we discussed in Section III-C and IV-A, *FSR* has some limitations with regards to expressing configurations with MEDs, and the safety requirement (strict monotonicity) is only a sufficient condition. These shortcomings are tied to the underlying routing algebra that *FSR* uses. In the future, we are interested in adapting *FSR* to use recent advances in routing algebra [17], as the underlying formalism to capture a wider range of safe configurations.

Our longer-term goal is to fundamentally change the way inter-domain routing protocols are designed, implemented, and configured. We believe that *FSR*'s rigorous and formal approach is a significant step in the right direction. Starting from the foundations laid in this paper, we plan to extend *FSR* in two main directions. First, we want to incorporate the most recent advances in routing algebra [16] to analyze, for example, new policy guidelines such as neighbor-specific BGP [41] and the interactions between iBGP and eBGP. Second, we plan to exploit the close connection between *NDlog* programs and state-transition systems. This would allow *FSR* to use a model-checker to generate traces of protocol oscillations for unsafe policy configurations.

Acknowledgment

The authors would like to thank Shivkumar Muthukumar and Harjot Gill for their valuable help in the RapidNet code base. Alexander Gurney provided useful insights into routing algebra, in particular handling of MEDs and the strict monotonicity condition. This research is funded in part by NSF grants (CCF-0820208, CNS-0830949, CNS-0845552, CNS-1040672, CPS-0932397, IIS-0812270, and TC-0905607), AFOSR grants (FA9550-08-1-0352, FA9550-09-1-0643), ONR grants (N00014-09-1-0770, N00014-11-1-0555), a gift from Cisco System, and the Alexander von Humboldt Foundation.

References

[1] Alvaro, P., Condie, T., Conway, N., Elmeleegy, K., Hellerstein, J. M., and Sears, R. Boom analytics: exploring data-centric, declarative programming for the cloud. In *Proceedings of the 5th European conference on Computer systems* (2010), EuroSys '10, ACM.

[2] Chu, D., Popa, L., Tavakoli, A., Hellerstein, J. M., Levis, P., Shenker, S., and Stoica, I. The design and implementation of a declarative sensor network system. In *Sensys* (2007).

[3] Dimitropoulos, X., Krioukov, D., Fomenkov, M., Huffaker, B., Hyun, Y., claffy, k., and Riley, G. AS relationships: inference and validation. *ACM SIGCOMM Computer Communication Review* (2007).

[4] Feamster, N., Johari, R., and Balakrishnan, H. Implications of autonomy for the expressiveness of policy routing. In *ACM SIGCOMM* (2005).

[5] Feldmann, A., Maennel, O., Mao, Z. M., Berger, A., and Maggs, B. Locating Internet routing instabilities. In *ACM SIGCOMM* (2004).

[6] Flavel, A., and Roughan, M. Stable and flexible iBGP. In *ACM SIGCOMM* (2009).

[7] FSR Demonstration Video. http://netdb.cis.upenn.edu/rapidnet/sigcomm11demo.html.

[8] Gao, L., Griffin, T. G., and Rexford, J. Inherently safe backup routing with BGP. In *IEEE INFOCOM* (2001).

[9] Gao, L., and Rexford, J. Stable Internet routing without global coordination. In *ACM SIGMETRICS* (2000).

[10] Griffin, T. G. The stratified shortest-paths problem. In *COMSNETS* (2010).

[11] Griffin, T. G., Jaggard, A., and Ramachandran, V. Design principles of policy languages for path vector protocols. In *ACM SIGCOMM* (2003).

[12] Griffin, T. G., Shepherd, F. B., and Wilfong, G. The stable paths problem and interdomain routing. *IEEE Trans. on Networking 10* (2002), 232–243.

[13] Griffin, T. G., and Sobrinho, J. L. Metarouting. In *ACM SIGCOMM* (2005).

[14] Griffin, T. G., and Wilfong, G. An analysis of BGP convergence properties. In *SIGCOMM* (1999).

[15] Griffin, T. G., and Wilfong, G. A Safe Path Vector Protocol. In *INFOCOM* (2000).

[16] Gurney, A., and Griffin, T. G. Neighbor-specific BGP: An algebraic exploration. In *ICNP* (2010).

[17] Gurney, A. J. Construction and Verification of Routing Algebras. Ph.D. Thesis. University of Cambridge. 2009.

[18] Haeberlen, A., Avramopoulos, I., Rexford, J., and Druschel, P. NetReview: Detecting when interdomain routing goes wrong. In *NSDI* (2009).

[19] Jaggard, A. D., and Ramachandran, V. Relating two formal models of path-vector routing. In *INFOCOM* (2005).

[20] Kohler, E., Morris, R., Chen, B., Jannotti, J., and Kaashoek, M. F. The Click Modular Router. *ACM TOCS 18(3)* (2000), 263–297.

[21] Liu, C., Correa, R., Li, X., Basu, P., Loo, B. T., and Mao, Y. Declarative Policy-based Adaptive MANET Routing. In *ICNP* (2009).

[22] Loo, B. T., Condie, T., Garofalakis, M., Gay, D. E., Hellerstein, J. M., Maniatis, P., Ramakrishnan, R., Roscoe, T., and Stoica, I. Declarative networking. In *Communications of the ACM* (2009).

[23] Loo, B. T., Condie, T., Hellerstein, J. M., Maniatis, P., Roscoe, T., and Stoica, I. Implementing Declarative Overlays. In *SOSP* (2005).

[24] Loo, B. T., Hellerstein, J. M., Stoica, I., and Ramakrishnan, R. Declarative Routing: Extensible Routing with Declarative Queries. In *SIGCOMM* (2005).

[25] Mao, Y., Loo, B. T., Ives, Z., and Smith, J. M. MOSAIC: Unified Platform for Dynamic Overlay Selection and Composition. In *CoNEXT* (2008).

[26] Mengmeng Liu, Nicholas Taylor, Wenchao Zhou, Zachary Ives, and Boon Thau Loo. Recursive Computation of Regions and Connectivity in Networks. In *ICDE* (2009).

[27] Muthukumar, S. C., Li, X., Liu, C., Kopena, J. B., Oprea, M., and Loo, B. T. Declarative toolkit for rapid network protocol simulation and experimentation. In *SIGCOMM (demo)* (2009).

[28] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI* (2005).

[29] Network Simulator 3. http://www.nsnam.org/.

[30] NIGAM, V., JIA, L., LOO, B. T., AND SCEDROV, A. Maintaining Distributed Logic Programs Incrementally. In *ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP)* (2011).
[31] RAPIDNET: A DECLARATIVE TOOLKIT FOR RAPID NETWORK SIMULATION AND EXPERIMENTATION. http://netdb.cis.upenn.edu/rapidnet/.
[32] SAMI, R., SCHAPIRA, M., AND ZOHAR, A. Searching for stability in interdomain routing. In *IEEE INFOCOM* (2009).
[33] SCHAPIRA, M., ZHU, Y., AND REXFORD, J. Putting BGP on the right path: A case for next-hop routing. In *ACM SIGCOMM HotNets* (Oct. 2010).
[34] SIMBGP. http://www.bgpvista.com/simbgp.php.
[35] SINGH, A., DAS, T., MANIATIS, P., DRUSCHEL, P., AND ROSCOE, T. Bft protocols under fire. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation* (2008), NSDI'08, USENIX Association.
[36] SOBRINHO, J. An algebraic theory of dynamic network routing. *IEEE/ACM Trans. Netw. 13* (October 2005).
[37] SPRING, N., MAHAJAN, R., AND WETHERALL, D. Measuring ISP topologies with Rocketfuel. In *ACM SIGCOMM* (2002).
[38] SUBRAMANIAN, L., CAESAR, M., EE, C. T., HANDLEY, M., MAO, M., SHENKER, S., AND STOICA, I. HLP: A next-generation interdomain routing protocol. In *SIGCOMM* (2005).
[39] TAYLOR, P., AND GRIFFIN, T. A model of configuration languages for routing protocols. In *PRESTO* (2009).
[40] WANG, A., BASU, P., LOO, B. T., AND SOKOLSKY, O. Declarative network verification. In *PADL* (2009).
[41] WANG, Y., SCHAPIRA, M., AND REXFORD, J. Neighbor-specific BGP: More flexible routing policies while improving global stability. In *ACM SIGMETRICS* (2009).
[42] YICES. http://yices.csl.sri.com/.

APPENDIX A
PROOF OF CORRECTNESS

To prove Theorem 5.1, we make a few assumptions. First, ι and the complement of the label operation $\bar{l}$ has the property that the label assigned to a link vu is the complement of the link assigned to uv.

(**Property A**): $\overline{\iota(uv)} = \iota(vu)$.

For example, in the Gao-Rexford guideline, the reverse direction of a customer link is a provider link.

We also assume that generated functions and predicates faithfully implement the algebraic specifications, which are formally stated below.

(**Property B**):

- `f_export`($nd(l), nd(s)$) = `true` iff $l \oplus_E s = E$.
- `f_import`($nd(l), nd(s)$) = `true` iff $l \oplus_I s = I$.
- `f_concatSig`($nd(l), nd(s)$) = $nd(s')$ iff $l \oplus_P s = s'$.
- `f_pref`($nd(s1), nd(s2)$) = `true` iff $s1 \preceq s2$.
- `label`(@$nd(u), nd(v), nd(l)$) `:-` . is in *prog* iff $\iota(uv) = l$.
- `sig`(@$nd(u), nd(s), nd(p)$) `:-` . is in *prog* iff $\sigma_0(p) = s$.

Given an algebra $\mathcal{A}$, and a network topology represented by ι and σ_0, let *prog* be the *NDlog* program that is translated from $\mathcal{A}$, and ι and σ_0.

We first prove the following lemma (containing two parts i and ii), which state that the generated signature tuples are correct; and that if a route update message is generated, then the route's signature is correctly computed, and the export policies have been applied:

Lemma A.1:

(i) Given any path p, if `sig`($nd(u), nd(s), nd(p)$) is generated by *prog*, and $s \neq \phi$, then $s = \sigma(p)$.

(ii) Given any path p, if `msg`($nd(u), nd(v), nd(d), nd(s), nd(p)$) is generated by *prog*, and `s` $\neq \phi$, then $s = \sigma(p)$, and $\iota(vu) \oplus_E s = E$.

Proof (sketch): By induction of the length of p.

We abbreviate $nd(t)$ to `t` when it is clear from the context that *NDlog* representation of t is required.

In the base case, the length of p is 1. We know that $\sigma(p) = \sigma_0(p)$ and by our assumptions (Property B), we know that if `sig(u,s,p)` is generated by *prog*, then $s = \sigma_0(p)$. So part (i) holds.

By examining the GPV program, `msg(n,u,d,s,p)` tuple is only generated when `gpvExport` is applied. So we know that $s = \sigma(p)$ (s and p comes from the `sig` tuple), and that `f_export(l,s) = true` and `label(u,n,l)` is true. Use Property B again, we know that $l \oplus_E s = E$, and $\iota(un) = l$, so part (ii) holds.

In the inductive case, to prove part (i), we examine the `gpvSig` rule. The new path and signature is generated from the tuple `msg(u,v,d,s,p)`. Using induction hypotheses, we know that $s = \sigma(p)$, and $\iota(vu) \oplus_E s = E$. If a new `sig(u,snew,pnew)` is generated, then it must be the case that `f_import(l,s) = I` where `l` = $\iota(uv)$. Using property A, we know that $\overline{\iota(uv)} = \iota(vu)$. By examining the way we generate $\oplus$ from $\oplus_p$, $\oplus_E$ and $\oplus_I$, we know that `snew` = $\iota(uv) \oplus \sigma(p)$, which is equal to $\sigma(pnew)$ (Property B). We can prove part (ii) in similar ways as we prove part (i) in the base case. ■

Lemma A.1 implies Theorem 5.1.

Anduo Wang is a Ph.D. student in the Computer and Information Science department at the University of Pennsylvania. She received her B.S. degree from Tianjin University in 2004 and M.S. Degree from University of Pennsylvania in 2009,

Limin Jia is a Systems Scientist in the Cylab at Carnegie Mellon University. She received her Ph.D. in Computer Science from Princeton University in 2008.

Wenchao Zhou is a Ph.D. student in the Computer and Information Science department at the University of Pennsylvania. He received his B.S. degree from Tsinghua University in 2006 and M.S. Degree from University of Pennsylvania in 2009,

Yiqing Ren is a masters student in the Computer and Information Science department at the University of Pennsylvania. She received her B.S. degree from Shanghai Jiaotong University in 2006.

Boon Thau Loo is an Assistant Professor in the Computer and Information Science department at the University of Pennsylvania. He received his Ph.D. degree in Computer Science from the University of California, Berkeley in 2006.

Jennifer Rexford is a Professor in the Computer Science department at Princeton University. Jennifer received her BSE degree in electrical engineering from Princeton University in 1991, and her MSE and Ph.D. degrees in computer science and electrical engineering from the University of Michigan in 1993 and 1996, respectively.

Vivek Nigam is an Alexander von Humboldt Fellow in the Computer Science department at the Ludwig-Maximilians University of Munich. He was previously a post-doctoral researcher in the Mathematics department at the University of Pennsylvania. He received his Ph.D. from LIX - École Polytechnique in 2009.

Andre Scedrov is a Professor in the Mathematics department, and holds a joint appointment in the Computer and Information Science department at the University of Pennsylvania. He received his Ph.D. at the State University of New York, Buffalo in 1981.

Carolyn Talcott is a Program Director in the Computer Science Laboratory of SRI International, where she leads the Symbolic Systems Technology group and the Pathway Logic Project. She holds a Ph.D. in computer science from Stanford University in 1985 and a Ph.D. in chemistry from the University of California, Berkeley in 1966.