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

Wenchao Zhou
supervised by Prof. Boon Thau Loo
CIS Department, University of Pennsylvania
Philadelphia, PA, 19104
wenchaoz@cis.upenn.edu

1. 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 [13, 10, 12, 11, 25] allow researchers to explore how local policies affect BGP stability and identify policy guidelines that, if universally adopted by ISPs, ensure global safety [7, 6, 9, 2, 8, 23]. 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* [11, 25] with recent advances in *declarative networking* [17] to produce provably-correct implementations of safe interdomain routing.

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:

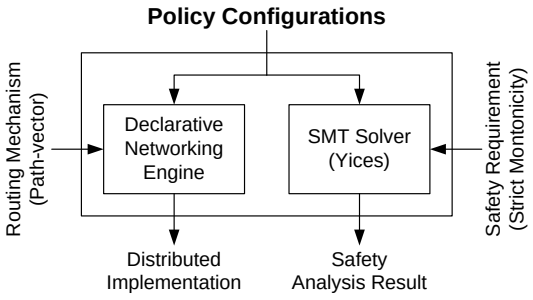

Figure 1: *FSR* Architecture.

Policy configuration as algebra: Our extensions to routing algebra [11, 25] 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 [32]. 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 [25]). 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 [22, 20]. *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 [27]. 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.

2. BACKGROUND AND RELATED WORK

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 [25] to analyze safety. In this section, we present a brief overview of routing algebra, and discuss previous work related of *FSR*.

2.1 Abstract Routing Algebra

Routing algebra is an abstract structure that describes how network nodes calculate routes, and the preference for one route over another. 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.

Complex policies can be represented as compositions of simpler policies [11]. 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 $\preceq_B$.

2.2 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 [25, 11], 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 [28, 4, 14, 16] 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 [3, 15]. Prior to deployment, one may also perform custom simulations, using platforms such as simBGP [24]. 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 require the programmer to correctly implement the protocols in the simulator.

Compared with 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.

3. APPROACH AND RESULTS

Next, we describe in details the design of the *FSR* toolkit. Section 3.1 defines safety analysis as a constraint satisfiability problem, and describes how *FSR* fully automates the analysis using an SMT solver. We describe how FSR automatically generates a distributed *NDlog* implementation in Section 3.2, and case studies in Section 3.3 to demonstrate the usages of the *FSR* toolkit.

3.1 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. After a brief review of safety analysis based on routing algebra, we explain how to generate the integer constraints and present an example that illustrates the conversion process and resulting safety analysis.

3.1.1 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:

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

Strict monotonicity ensures that a path P_v from v to the destination is 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. Sobrinho has proved the following theorem [25].

THEOREM 1. *If the routing algebra is strictly monotonic, then the path-vector protocol converges.*

Theorem 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.

3.1.2 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 [25], 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 our *FSR* implementation, we utilize the Yices [32] SMT solver, although the technique can be applied to SMT solvers in general.

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 3.1.1: (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 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) problematic parts of the policy configuration.

3.1.3 An Example Yices Encoding

We present an example that encodes shortest hop-count to demonstrate the three-step process of generating Yices constraints from algebraic input and the analysis process. We have encoded other policy in Yices as well; we omit them here due to the space limit.

The algebraic specification of the shortest hop-count policy is presented in Section 2.1. 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, indicting the convergence of the routing protocol.

3.2 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 [18]. This makes possible a clean and concise proof (via logical inductions) of the correctness of the generated *NDlog* programs with regard to the algebra.

3.2.1 NDlog Implementation

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 [19] 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_`".
The above program manipulates the following tuples. `label(@U,V,L)`[1] 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 concatenation function $\oplus$, while the function `f_import(L,S)`, implements the import filter.

- **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.

One of the advantages of using *NDlog* is its ease of incorporating routing policies in algebraic form with routing mechanisms. 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*. 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.

3.2.2 Correctness of NDlog implementation

In order to apply Theorem 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 [17] and formally proven [29] the correctness of an *NDlog* implementation of the path-vector protocol. In addition, [21] 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 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 assignment of signatures 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 (detailed proofs can be found in [30]):

THEOREM 2 (CORRECTNESS OF *NDlog* TRANSLATION). *Given any path p, if $\mathtt{sig}(nd(u), nd(s), nd(p))$ is generated by prog, and $s \neq \phi$, then $s = \sigma(p)$.*

3.3 Case Studies

We present 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

[1] *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.

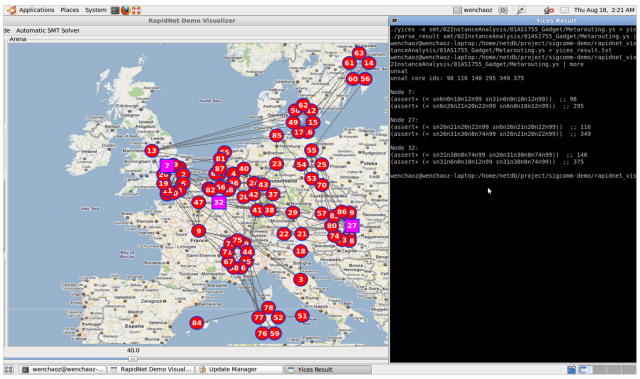

Figure 2: A snapshot of the FSR toolkit, where the SMT's results "pinpoint" the problematic routers.

instances, and (2) empirically evaluating protocol dynamics and temporal properties that cannot be easily checked in formal analysis. 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 [1] or Rocketfuel [26]). Due to space limit, we present here one case study [31]that explores the safety properties of the iBGP protocol. Section 4 summarizes the rest case studies (and the details are available in [30]).

As the input topology, we utilize the intradomain topology (with inferred link weights) of AS 1755 from the Rocketfuel [26] 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 the iBGP 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 into the iBGP topology. 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 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.

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*. All the links are set to 100 Mbps bandwidth, 10 ms latency, and up to 3ms jitter. We compare the average per-node bandwidth utilization over time for the iBGP protocol with and without the embedded gadget (*Gadget* and *NoGadget* respectively). 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 [4] causes transient oscillation, and hence result in higher bandwidth utilization for *Gadget*, as compared to *NoGadget*.

4. CONCLUSION

In this work, 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. Our main research contributions are:

- **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.

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 illustrate *FSR* in action, please see [5] for a video demonstration of our current *FSR* prototype.

Acknowledgment

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.

5. REFERENCES

[1] 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).

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[20] 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 *Proc. ACM SIGCOMM - demo* (2009).

[21] Nigam, V., Jia, L., Loo, B. T., and Scedrov, A. Maintaining Distributed Logic Programs Incrementally. In *Proc. ACM PPDP* (2011).

[22] RapidNet: A Declarative Toolkit for Rapid Network Simulation and Experimentation. `http://netdb.cis.upenn.edu/rapidnet/`.

[23] Schapira, M., Zhu, Y., and Rexford, J. Putting BGP on the right path: A case for next-hop routing. In *Proc. ACM SIGCOMM HotNets* (Oct. 2010).

[24] simBGP. `http://www.bgpvista.com/simbgp.php`.

[25] Sobrinho, J. An algebraic theory of dynamic network routing. *IEEE/ACM Trans. Netw. 13* (October 2005).

[26] Spring, N., Mahajan, R., and Wetherall, D. Measuring ISP topologies with Rocketfuel. In *Proc. ACM SIGCOMM* (2002).

[27] Subramanian, L., Caesar, M., Ee, C. T., Handley, M., Mao, M., Shenker, S., and Stoica, I. HLP: A next-generation interdomain routing protocol. In *Proc. ACM SIGCOMM* (2005).

[28] Taylor, P., and Griffin, T. A model of configuration languages for routing protocols. In *Proc. PRESTO Workshop* (2009).

[29] Wang, A., Basu, P., Loo, B. T., and Sokolsky, O. Declarative network verification. In *Proc. PADL* (2009).

[30] Wang, A., Jia, L., Wenchao Zhou, Ren, Y., Loo, B. T., Rexford, J., Nigam, V., Scedrov, A., and Talcott, C. FSR: Formal Analysis and Implementation Toolkit for Safe Inter-domain Routing. *IEEE/ACM Trans. Networking* (2012).

[31] Wenchao Zhou, Ren, Y., Wang, A., Jia, L., Gurney, A. J., Loo, B. T., and Rexford, J. FSR: Formal Analysis and Implementation Toolkit for Safe Inter-domain Routing. In *Proc. ACM SIGCOMM - Demonstration* (2011).

[32] Yices. `http://yices.csl.sri.com/`.