HAL open science

# A Program Logic for Verifying Secure Routing Protocols

Chen Chen, Limin Jia, Hao Xu, Cheng Luo, Wenchao Zhou, Boon Thau Loo

**To cite this version:**

Chen Chen, Limin Jia, Hao Xu, Cheng Luo, Wenchao Zhou, et al.. A Program Logic for Verifying Secure Routing Protocols. 34th Formal Techniques for Networked and Distributed Systems (FORTE), Jun 2014, Berlin, Germany. pp.117-132, 10.1007/978-3-662-43613-4_8 . hal-01398011

**HAL Id: hal-01398011**

**https://inria.hal.science/hal-01398011v1**

Submitted on 16 Nov 2016

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

# A Program Logic for Verifying Secure Routing Protocols

Chen Chen[1], Limin Jia[2], Hao Xu[1], Cheng Luo[1], Wenchao Zhou[3], and Boon Thau Loo[1]

[1] University of Pennsylvania {chenche,haoxu,boonloo}@cis.upenn.edu
[2] Carnegie Mellon University liminjia@cmu.edu
[3] Georgetown University wzhou@cs.georgetown.edu

**Abstract.** The Internet, as it stands today, is highly vulnerable to attacks. However, little has been done to understand and verify the formal security guarantees of proposed secure inter-domain routing protocols, such as Secure BGP (S-BGP). In this paper, we develop a sound program logic for SANDLog—a declarative specification language for secure routing protocols—for verifying properties of these protocols. We prove invariant properties of SANDLog programs that run in an adversarial environment. As a step towards automated verification, we implement a verification condition generator (VCGen) to automatically extract proof obligations. VCGen is integrated into a compiler for SANDLog that can generate executable protocol implementations; and thus, both verification and empirical evaluation of secure routing protocols can be carried out in this unified framework. To validate our framework, we (1) encoded several proposed secure routing mechanisms in SANDLog, (2) verified variants of path authenticity properties by manually discharging the generated verification conditions in Coq, and (3) generated executable code based on SANDLog specification and ran the code in simulation.

## 1 Introduction

In recent years, we have witnessed an explosion of services provided over the Internet. These services are increasingly transferring customers' private information over the network and being used in mission-critical tasks. Central to ensuring the reliability and security of these services is a secure and efficient Internet routing infrastructure. Unfortunately, the Internet infrastructure, as it stands today, is highly vulnerable to attacks. The Internet runs *Border Gateway Protocol* (BGP), where routers are grouped into Autonomous Systems (*ASes*) administrated by Internet Service Providers (*ISP*s). Individual ASes exchange route advertisements with neighboring ASes using the *path-vector* protocol. Each originating AS first sends a route advertisement (containing a single AS number) for the IP prefixes it owns. Whenever an AS receives a route advertisement, it adds itself to the AS *path*, and advertises the best route to its neighbors based on its routing policies. Since these route advertisements are not authenticated, ASes can advertise non-existent routes or claim to own IP prefixes that they do

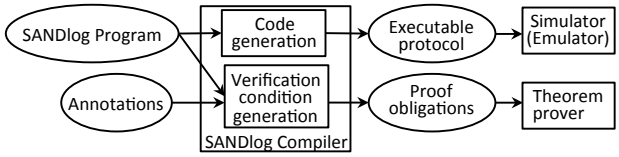


**Fig. 1.** Architecture of a unified framework for implementing and verifying secure routing protocols. The round objects represent the inputs and outputs of the framework, which are either code or proofs. The rectangular objects are software components of the framework.

not. These faults may lead to long periods of interruption of the Internet; best epitomized by recent high-profile attacks [10, 24].

In response to these vulnerabilities, several new Internet routing architectures and protocols for a more secure Internet have been proposed. These range from security extensions of BGP (Secure-BGP (S-BGP) [19], ps-BGP [28], so-BGP [30]), to "clean-slate" Internet architectural redesigns such as SCION [31] and ICING [22]. However, *none* of the proposals formally analyzed their security properties. These protocols are implemented from scratch, evaluated primarily experimentally, and their security properties shown via informal reasoning.

Existing protocol analysis tools [7, 12, 14] are rarely used in analyzing routing protocols because routing protocols are considerably more complicated than cryptographic protocols: they often compute local states, are recursive, and their security properties need to be shown to hold on arbitrary network topologies. As the number of models is infinite, model-checking-based tools, in general, cannot be used to prove the protocol secure.

To overcome the above limitations, we develop a novel proof methodology to verify these protocols. We augment prior work on declarative networking (ND-Log) [21] with cryptographic libraries to provide compact encoding of secure routing protocols. We call this extension SANDLog (*Secure and Authenticated Network DataLog*). It has been shown that such a Datalog-like language can be used for implementing a variety of network protocols [21]. We develop a program logic for reasoning about SANDLog programs that run in an adversarial environment. Based on the program logic, we implement a verification condition generator (VCGen), which takes as inputs the SANDLog program and user-provided annotations, and outputs intermediary proof obligations as lemma statements in Coq's syntax. Proofs for these lemmas are later completed manually. VCGen is integrated into the SANDLog compiler, which augments the declarative networking engine RapidNet [26] to handle cryptographic functions. The compiler is able to translate SANDLog specification into executable code, which is amenable to implementation and evaluation. Both verification and empirical evaluation of secure routing protocols can be carried out in this unified framework (Figure 1).

We summarize our technical contributions:

1. We define a program logic for verifying SANDLog programs in the presence of adversaries (Section 3). We prove that our logic is sound.
2. We implement VCGen for automatically generating proof obligations and integrate VCGen into a compiler for SANDLog (Section 4).

3. We encode S-BGP and SCION in SANDLog, verify path authenticity properties of these protocols, and run them in simulation (Section 5).

Due to space constraints, we omit many details, which can be found in our companion technical report [9].

## 2 SANDLog

We introduce the syntax and operational semantics of SANDLog, which extends the *Network Datalog* (NDLog) [21] with a library for cryptographic functions. The complete definitions can be found in our TR.

### 2.1 Syntax

SANDLog's syntax is summarized below. A SANDLog program is composed of a set of rules, each of which consists of a rule head and a rule body. A rule head is a tuple. A rule body consists of a list of body elements which are either tuples or atoms. Atoms include assignments and inequality constraints. The binary operator *bop* denotes inequality relations. Each SANDLog rule specifies that if all the tuples in the body are derivable and all the constraints specified by the atoms in the body are satisfied, then the head tuple is derivable. These features are shared between NDLog [21] and SANDLog. Unique to SANDLog, are the cryptographic functions denoted $f_c$, implemented as a library. This library includes commonly used functions such as signature generation and verification.

$$
\begin{array}{llll}
\textit{Crypt func} & f_c & ::= & \mathsf{f_sign_asym} \mid \mathsf{f_verify_asym} \cdots \\
\textit{Atom} & a & ::= & x := t \mid t_1 \ \textit{bop}\ t_2 \\
\textit{Terms} & t & ::= & x \mid c \mid \iota \mid f(\vec{t}) \mid f_c(\vec{t}) \\
\textit{Body Elem} & B & ::= & p(agB) \mid a \\
\textit{Arg List} & ags & ::= & \cdot \mid ags, x \mid ags, c \\
\textit{Rule Body} & body & ::= & \cdot \mid body, B \\
\textit{Body Args} & agB & ::= & @\iota, ags \\
\textit{Rule} & r & ::= & p(agH) :- \ body \\
\textit{Head Args} & agH & ::= & agB \mid @\iota, ags, F_{agg}\langle x \rangle, ags \\
\textit{Program} & prog(\iota) & ::= & r_1, \cdots, r_k
\end{array}
$$

To support distributed execution, SANDLog assumes that each node has a unique identifier denoted $\iota$. A SANDLog program *prog* is parametrized over the identifier of the node it runs on. A location specifier, written $@\iota$, specifies where a tuple resides and is the first argument of a tuple. We require all body tuples to reside on the same node as the program. A rule head can specify a location different from its body tuples. When such a rule is executed, the derived head tuple is sent to the specified remote node. Finally, SANDLog supports aggregation functions (denoted $F_{agg}\langle x \rangle$), such as max and min, in the rule head.

**An example program.** The following program can be used to compute the best path between each pair of nodes in a network. $s$ is the location parameter of the program, representing the ID of the node where the program is executing.

Each node stores three kinds of tuples: $\mathsf{link}(@s, d, c)$ means that there is a direct link from $s$ to $d$ with cost $c$; $\mathsf{path}(@s, d, c, p)$ means that $p$ is a path from $s$ to $d$ with cost $c$; and $\mathsf{bestPath}(@s, d, c, p)$ states that $p$ is the lowest-cost path between $s$ and $d$.

*sp1* $\mathsf{path}(@s, d, c, p) \text{:-} \mathsf{link}(@s, d, c), p := [s, d].$
*sp2* $\mathsf{path}(@z, d, c, p) \text{:-} \mathsf{link}(@s, z, c1), \mathsf{path}(@s, d, c2, p1), c := c1 + c2, p := z{::}p1.$
*sp3* $\mathsf{bestPath}(@s, d, \mathsf{min}\langle c\rangle, p) \text{:-} \mathsf{path}(@s, d, c, p).$

Rule *sp1* computes all one-hop paths based on direct links. Rule *sp2* expresses that if there is a link from $s$ to $z$ of cost *c1* and a path from $s$ to $d$ of cost *c2*, then there is a path from $z$ to $d$ with cost *c1+c2* (for simplicity, we assume links are symmetric, i.e. if there is a link from $s$ to $d$ with cost $c$, then a link from $d$ to $s$ with the same cost $c$ also exists). Finally, rule *sp3* aggregates all paths with the same pair of source and destination ($s$ and $d$) to compute the best path. The arguments that appear before the aggregation denotes the group-by keys.

### 2.2 Operational Semantics

The operational semantics of SANDLog adopts a distributed execution model. Each node runs a designated program, and maintains a database of derived tuples in its local state. Nodes can communicate with each other by sending tuples over the network. The evaluation of the SANDLog programs follows the PSN algorithm [20], and maintains the database incrementally. The semantics introduced here is similar to that of NDLog except that we make explicit, which tuples are derived, which are received, and which are sent over the network. This addition is crucial to specifying and proving protocol properties. The constructs needed for defining the operational semantics of SANDLog are presented below.

| | | | |
|---|---|---|---|
| *Table* | $\Psi ::= \cdot \mid \Psi, (n, P)$ | *Network Queue* | $\mathcal{Q} ::= \mathcal{U}$ |
| *Update* | $u ::= -P \mid +P$ | *Local State* | $\mathcal{S} ::= (\iota, \Psi, \mathcal{U}, prog(\iota))$ |
| *Update List* | $\mathcal{U} ::= [u_1, \cdots, u_n]$ | *Configuration* | $\mathcal{C} ::= \mathcal{Q} \rhd \mathcal{S}_1, \cdots, \mathcal{S}_n$ |

We write $P$ to denote tuples. The database for storing all derived tuples on a node is denoted $\Psi$. Because there could be multiple derivations of the same tuple, we associate each tuple with a reference count $n$, recording the number of valid derivations for that tuple. An update is either an insertion of a tuple, denoted $+P$, or a deletion of a tuple, denoted $-P$. We write $\mathcal{U}$ to denote a list of updates. A node's local state, denoted $\mathcal{S}$, consists of the node's identifier $\iota$, the database $\Psi$, a list of unprocessed updates $\mathcal{U}$, and the program *prog* that $\iota$ runs. A configuration of the network, written $\mathcal{C}$, is composed of a network update queue $\mathcal{Q}$, and the set of the local states of all the nodes in the network. The queue $\mathcal{Q}$ models the update messages sent across the network.

Figure 2 presents an example scenario of executing the shortest-path program shown in Section 2.1. The network consists of three nodes, $A$, $B$ and $C$, connected by two links with cost 1. In the current state, all three nodes are aware of their direct neighbors, i.e., link tuples are in their databases $\Psi_A$, $\Psi_B$ and $\Psi_C$. They have constructed paths to their neighbors (i.e., the corresponding path and bestPath tuples are stored). In addition, node $B$ has applied *sp2* and generated updates

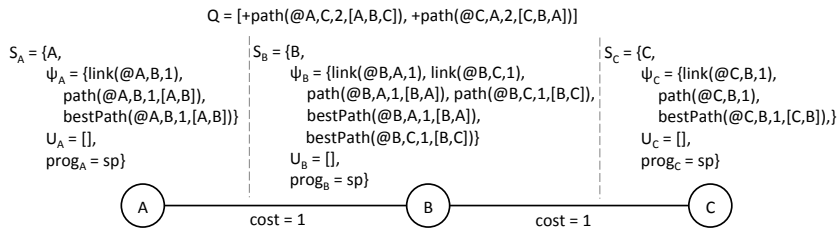


**Fig. 2.** An Example Scenario.

+path(@A,C,2,[A,B,C]) and +path(@C,A,2,[C,B,A]), which are currently queued and waiting to be delivered to their destinations (node $A$ and $C$ respectively).

**Top-level transitions.** The small-step operational semantics of a node is denoted $\mathcal{S} \hookrightarrow \mathcal{S}', \mathcal{U}$. From state $\mathcal{S}$, a node takes a step to a new state $\mathcal{S}'$ and generates a set of updates $\mathcal{U}$ for other nodes in the network. The small-step operational semantics of the entire system is denoted $\mathcal{C} \xrightarrow{\tau} \mathcal{C}'$, where $\tau$ is the time of the transition step. A trace $\mathcal{T}$ is a sequence of transitions: $\xrightarrow{\tau_0} \mathcal{C}_1 \xrightarrow{\tau_1} \mathcal{C}_2 \cdots \xrightarrow{\tau_n} \mathcal{C}_{n+1}$, where the time points on the trace are monotonically increasing ($\tau_0 < \tau_1 < \cdots < \tau_n$). We assume that the effects of a transition take place at time $\tau_i$ (reflected in $\mathcal{C}_{i+1}$). Figure 3 defines the rules for system state transition.

Rule NodeStep states that the system takes a step when one node takes a step. As a result, the updates generated by node $i$ are appended to the end of the network queue. We use $\circ$ to denote the list append operation. Rule DeQueue applies when a node receives updates from the network. We write $\mathcal{Q}_1 \oplus \mathcal{Q}_2$ to denote a merge of two lists. Any node can dequeue updates sent to it and append those updates to the update list in its local state. Here, we overload the $\circ$ operator, and write $\mathcal{S} \circ \mathcal{Q}$ to denote a new state, which is the same as $\mathcal{S}$, except that the update list is the result of appending $\mathcal{Q}$ to the update list in $\mathcal{S}$.

We omit the detailed rules for state transitions within a node. Instead, we explain it through examples. At a high-level, those rules either fire base rules—rules that do not have a rule body—at initialization; or computes new updates based on the program and the first update in the update list. Continue the example scenario, node $A$ dequeues +path(@A,C,2,[A,B,C]), and puts it into the unprocessed update list $\mathcal{U}_A$ (rule DeQueue). Node $A$ then fires all rules that are triggered by the update, and generates new updates $\mathcal{U}_{in}$ and $\mathcal{U}_{ext}$ ($\mathcal{U}_{in}$ and $\mathcal{U}_{ext}$ denote updates to local (internal) states and remote (external) states respectively.) In the resulting state, the local state of node $A$ is updated: path(@A,C,2,[A,B,C]) is

$\boxed{\mathcal{C} \rightarrow \mathcal{C}'}$

$$\frac{\mathcal{S}_i \hookrightarrow \mathcal{S}'_i, \mathcal{U} \qquad \forall j \in [1,n] \wedge j \neq i,\ \mathcal{S}'_j = \mathcal{S}_j}{\mathcal{Q} \rhd \mathcal{S}_1, \cdots \mathcal{S}_n \rightarrow \mathcal{Q} \circ \mathcal{U} \rhd \mathcal{S}'_1, \cdots \mathcal{S}'_n} \ \textsc{NodeStep}$$

$$\frac{\mathcal{Q} = \mathcal{Q}' \oplus \mathcal{Q}_1 \cdots \oplus \mathcal{Q}_n \qquad \forall j \in [1,n] \qquad \mathcal{S}'_j = \mathcal{S}_j \circ \mathcal{Q}_j}{\mathcal{Q} \rhd \mathcal{S}_1, \cdots \mathcal{S}_n \rightarrow \mathcal{Q}' \rhd \mathcal{S}'_1, \cdots \mathcal{S}'_n} \ \textsc{DeQueue}$$

**Fig. 3.** Operational Semantics

inserted into $\Psi_A$, and $\mathcal{U}_A$ now includes $\mathcal{U}_{in}$. The network queue is updated to include $\mathcal{U}_{ext}$ (rule NodeStep).

**Incremental maintenance.** Following the strategy proposed in [20], the local database is maintained incrementally by processing updates one at a time. The rules are rewritten into $\Delta$ *rules*, which can efficiently generate all the updates triggered by one update. For any given rule $r$ that contains $k$ body tuples, $k$ $\Delta$ rules of the following form are generated, one for each $i \in [1, k]$.

$$\Delta p(agH) :- p_1^{\nu}(agB_1), ..., p_{i-1}^{\nu}(agB_{i-1}), \Delta p_i(agB_i), \\ p_{i+1}(agB_{i+1}), ..., p_k(agB_k), a_1, ..., a_m$$

$\Delta p_i$ in the body denotes the update currently being considered. $\Delta p$ in the head denotes new updates that are generated as the result of firing this rule. Here $p^{\nu}$ denotes the set of tuples whose name is $p$ and includes the current update being considered. $p$ is drawn only from the set of tuples that does not include the current update. For example, the $\Delta$ rules for *sp2* are:

$$sp2a \quad \Delta\mathsf{path}(@z, d, c, p) :- \Delta\mathsf{link}(@s, z, c1), \mathsf{path}(@s, d, c2, p1), c := c1 + c2, p := z::p1.$$
$$sp2b \quad \Delta\mathsf{path}(@z, d, c, p) :- \mathsf{link}^{\nu}(@s, z, c1), \Delta\mathsf{path}(@s, d, c2, p1), c := c1 + c2, p := z::p1.$$

Rules *sp2a* and *sp2b* are $\Delta$ rules triggered by updates of the link and path relation respectively. For instance, when node $A$ processes +path(@A,C,2,[A,B,C]), only rule *sp2b* is fired. In this step, $\mathsf{path}^{\nu}$ includes the tuple path(@A,C,2,[A,B,C]), while path does not. On the other hand, $\mathsf{link}^{\nu}$ and link denote the same set of tuples, because the update is a path tuple, and thus does not affect tuples with a different name.

**Rule firing.** Here we explain through examples how they work. Informally, a $\Delta$ rule is fired if instantiations of its body tuples are present in the derived tuples and available updates. The resulting rule head will be put into the update lists, depending on whether it needs to be sent to another node, or consumed locally.

We revisit the example in Figure 2. Upon receiving +path(@A,C,2,[A,B,C]), $A$ will trigger $\Delta$ rule *sp2b* and generate a new update +path(@B,C,3,[B,A,B,C]), which will be included in $\mathcal{U}_{ext}$ as it is destined to a remote node $B$. The $\Delta$ rule for *sp3* will also be triggered and will generate a new update +bestPath(@A,C,2,[A,B,C]), which will be included in $\mathcal{U}_{in}$. After evaluating the $\Delta$ rules triggered by the update +path(@A,C,2,[A,B,C]), we have $\mathcal{U}_{in} = \{$+bestPath(@A,C,2,[A,B,C])$\}$ and $\mathcal{U}_{ext} = \{$+path(@B,C,3,[B,A,B,C])$\}$. In addition, $\mathsf{bestpath}_{agg}$, the auxiliary relation that maintains all candidate tuples for bestpath, is also updated to reflect that a new candidate tuple has been generated. It now includes $\mathsf{bestpath}_{agg}$(@A,C,2,[A,B,C]).

**Discussion.** The semantics introduced here will not terminate for programs with a cyclic derivation of the same tuple, even though set-based semantics will. Most routing protocols do not have such issue (e.g., cycle detection is well-adopted in routing protocols). Our prior work [23] has proposed improvements to solve this issue. It is a straightforward extension to the current semantics and is not crucial for demonstrating the soundness of the program logic we develop.

The operational semantics is correct if the results are the same as one where all rules reside in one node and a global fixed point is computed at each round. The proof of correctness is out of the scope of this paper. We are working on correctness definitions and proofs for variants of PSN algorithms. Our initial

results for a simpler language can be found in [23]. SANDLog additionally allows aggregates, which are not included in [23]. The soundness of our logic only depends on the specific evaluation strategy implemented by the compiler, and is orthogonal to the correctness of the operational semantics. Updates to the operational semantics is likely to come in some form of additional bookkeeping in the representation of tuples, which we believe will not affect the overall structure of the program logic; as these metadata are irrelevant to the logic.

## 3 A Program Logic for SANDLog

Inspired by program logics for reasoning about cryptographic protocols [12, 15], we define a program logic for SANDLog. The properties we are interested in are safety properties, which should hold throughout the execution of SANDLog programs that interact with attackers.

**Attacker model.** We assume *connectivity-bound* network attackers, a variant of the Dolev-Yao network attacker model. An attacker can send and receive messages to and from its neighbors. We assume a symbolic model of the cryptographic functions: an attacker can operate cryptographic functions to which it has the correct keys, such as encryption, decryption, and signature generation. This model does not allow an attacker to eavesdrop or intercept packets. This makes sense in the application domain that we consider, as attackers are malicious nodes in the network that participate in the routing protocol exchange. All the links we consider represent dedicated physical cables that connect neighboring nodes, which are hard to eavesdrop without physical intrusion.

This attacker model manifests in the formal system in two places: (1) the network is modeled as connected nodes, some of which run the SANDLog program that encodes the prescribed protocol and others are malicious and run arbitrary SANDLog programs; (2) assumptions about cryptographic functions are admitted as axioms in proofs.

**Syntax.** We use first-order logic formulas, denoted $\varphi$, as property specifications. The atoms, denoted $A$, include predicates and term inequalities.

$$\begin{array}{lll} \textit{Atoms} & A ::= & P(\vec{t})@(\iota,\tau) \mid \mathsf{send}(\iota, \mathsf{tp}(P, \iota', \vec{t}))@\tau \mid \mathsf{recv}(\iota, \mathsf{tp}(P, \vec{t}))@\tau \\ & \mid & \mathsf{honest}(\iota, \mathit{prog}(\iota), \tau) \mid t_1 \ \mathit{bop}\ t_2 \end{array}$$

Predicate $P(\vec{t})@(\iota,\tau)$ means that tuple $P(\vec{t})$ is derivable at time $\tau$ by node $\iota$. The first element in $\vec{t}$ is a location identifier $\iota'$, which may be different from $\iota$. When a tuple $P(\iota', ...)$ is derived at node $\iota$, it is sent to $\iota'$. This *send* action is captured by predicate $\mathsf{send}(\iota, \mathsf{tp}(P, \iota', \vec{t}))@\tau$. Predicate $\mathsf{recv}(\iota, \mathsf{tp}(P, \vec{t}))@\tau$ denotes that node $\iota$ has received a tuple $P(\vec{t})$ at time $\tau$. $\mathsf{honest}(\iota, \mathit{prog}(\iota), \tau)$ means that node $\iota$ starts to run program $\mathit{prog}(\iota)$ at time $\tau$. Since predicates take time points as an argument, we are effectively encoding linear temporal logic (LTL) in first-order logic [18]. Using these atoms and first-order logic connectives, we can specify security properties such as route authenticity (see Section 5 for details).

**Logical judgments.** The logical judgments use two contexts: context $\Sigma$ contains all the free variables and $\Gamma$ contains logical assumptions.

$$\boxed{\Sigma;\Gamma \vdash \mathit{prog}(i) : \{i, t_b, t_e\}.\varphi(i, t_b, t_e)}$$

$$\frac{\begin{array}{l}\forall p \in \mathit{hdOf}(\mathit{prog}),\ \varphi_p \text{ is closed under trace extension} \\ \forall r \in \mathit{rlOf}(\mathit{prog}),\quad r = h(\vec{v}) :\!- p_1(\vec{s}_1), ..., p_m(\vec{s}_m), q_1(\vec{u}_1), ..., q_n(\vec{u}_n), a_1, ..., a_k \\ \quad \Sigma;\Gamma \vdash \forall i, \forall t, \forall \vec{y} \\ \qquad \bigwedge_{j\in[1,k]} [a_j] \wedge \bigwedge_{j\in[1,m]} (p_j(\vec{s}_j)@(i,t) \wedge \varphi_{p_j}(i,t,\vec{s}_j)) \wedge \bigwedge_{j\in[1,n]} \mathsf{recv}(i, \mathsf{tp}(q_j, \vec{u}_j))@t \\ \qquad \supset \varphi_h(i,t,\vec{v}) \qquad \text{where } \vec{y} = \mathit{fv}(r)\end{array}}{\Sigma;\Gamma \vdash \mathit{prog}(i) : \{i, y_b, y_e\}. \bigwedge_{p\in \mathit{hdOf}(\mathit{prog})} \forall t, \forall \vec{x}, y_b \leqslant t < y_e \wedge p(\vec{x})@(i,t) \supset \varphi_p(i,t,\vec{x})} \ \textsc{Inv}$$

$$\boxed{\Sigma;\Gamma \vdash \varphi} \quad \frac{\Sigma;\Gamma \vdash \mathit{prog}(i) : \{i, y_b, y_e\}.\varphi(i, y_b, y_e) \quad \Sigma;\Gamma \vdash \mathsf{honest}(\iota, \mathit{prog}(\iota), t)}{\Sigma;\Gamma \vdash \forall t', t' > t, \varphi(\iota, t, t')} \ \textsc{Honest}$$

**Fig. 4.** Selected Rules in Program Logic

(1) $\Sigma;\Gamma \vdash \varphi$ (2) $\Sigma;\Gamma \vdash \mathit{prog}(i) : \{i, y_b, y_e\}.\varphi(i, y_b, y_e)$

Judgment (1) states that $\varphi$ is provable given the assumptions in $\Gamma$. Judgment (2) is an assertion about SANDLog programs. We write $\varphi(\vec{x})$ when $\vec{x}$ are free in $\varphi$. $\varphi(\vec{t})$ denotes the resulting formula of substituting $\vec{t}$ for $\vec{x}$ in $\varphi(\vec{x})$. Recall that *prog* is parametrized over the identifier of the node it runs on. The assertion of an invariant property for such a program is parametrized over not only the node ID $i$, but also the starting point of executing the program ($y_b$) and a later time point $y_e$. Judgment (2) states that any trace $\mathcal{T}$ *containing* the execution of program *prog* by a node $\iota$, starting at time $\tau_b$, satisfies $\varphi(\iota, \tau_b, \tau_e)$ for any time point $\tau_e$ later than $\tau_b$. Intuitively, the trace contains several threads running concurrently, only one of them runs the program and the other threads can be malicious. Since $\tau_e$ is any time after $\tau_b$ (the time *prog* starts), $\varphi$ is an invariant property of *prog*. For example, $\varphi(i, y_b, y_e)$ could specify that whenever $i$ derives a path tuple, every link in the path must have existed in the past.

**Inference rules.** The inference rules of our program logic include all standard first-order logic ones (e.g. Modus ponens), omitted for brevity. We explain two key rules (Figure 4) in our proof system.

Rule Inv derives an invariant property of a program *prog*. The invariant property states that if a tuple $p$ is derived by this program, then some property $\varphi_p$ must be true; formally: $\forall t, \forall \vec{x}, y_b \leqslant t < y_e \wedge p(\vec{x})@(i,t) \supset \varphi_p(i,t,\vec{x})$, where $p$ is the tuple name of a rule head of *prog*, and $\varphi_p(i,t,\vec{x})$ is an invariant property associated with $p(\vec{x})$. For example, $p$ can be path, and $\varphi_p(i,t,\vec{x})$ be that every link in argument *path* must have existed in the past. Rule Inv states that the invariant of the program is the conjunction of all the invariants of the tuples it derives.

We require that the invariants $\varphi_p$ be closed under trace extension (the first premise of Inv). Formally: $\mathcal{T} \vDash \varphi(\iota, t, \vec{s})$ and $\mathcal{T}$ is a prefix of $\mathcal{T}'$ then $\mathcal{T}' \vDash \varphi(\iota, t, \vec{s})$. For instance, the property that node $\iota$ has received a tuple $P$ before time $t$ is closed under trace extension; the property that node $\iota$ never sends $P$ to the network is not closed under trace extension. This restriction has not affected

our case studies: the invariants used in verification only assert what happened in the past, or facts independent of time (e.g., arithmetic constraints).

Intuitively, the premises of Inv need to establish that (1) when $p$ is a base tuple—its derivation is independent of any other tuples—$\varphi$ holds; and (2) when $p$ is derived using other tuples, $\varphi$ holds. The last (second) premise of Inv does precisely that. It checks every rule in *prog* and proves that the body tuples and the invariants associated with the body tuples together imply the invariant of the head tuple. For example, for *sp1*, to show that the invariant associated with path is true, we can use the fact that there is a link tuple, that the invariant associated with that link tuple is true, and that the constraint $p = [s, d]$ is true. This is sound because we are inducting over the derivation tree of the head tuple.

This premise looks complicated because the body tuples need to be treated differently depending on whether they are derived locally, received from the network, or constraints. For each rule $r$ in *prog*, we assume that the body of $r$ is arranged so that the first $m$ tuples are derived by *prog*, the next $n$ tuples are received from the network, and constraints constitute the rest of the body. The right-hand-side of the implication of the last (second) premise is the invariant associated with tuple $h$. A rule head is only derivable when all of its body tuples are derivable and constraints satisfied. For tuples that are derived earlier by *prog* (denoted $p_j$), we can safely assume that their invariants hold at time $t$. All received tuples ($q_j$) should have been received prior to rule firing. Finally, the atoms (constraints, denoted $a_j$) should be true. Here, $[x := f(\vec{t})]$ rewrites the assignment statement into an equality check $x = f(\vec{t})$. The left-hand-side of that implication is a conjunction of formulas denoting the above conditions. When $r$ only has a rule head, this premise is reduced to the right-hand-side of that implication, which is what case (1) mentioned above.

The last (second) premise of Inv can be automatically generated given a SANDLog program and all the corresponding $\varphi_p$s. In Section 4, we detail the implementation of the verification condition generator for Coq.

The Honest rule proves properties of the entire system based on invariants of a SANDLog program. If $\varphi(i, y_b, y_e)$ is the invariant of *prog*, and a node $\iota$ runs the program *prog* at time $t_b$, then any trace containing the execution of this program satisfies $\varphi(\iota, t_b, t_e)$, where $t_e$ is a time point after $t_b$. SANDLog programs never terminate: after the last instruction, the program enters a stuck state.

**Soundness.** We prove the soundness of our logic with regard to the trace semantics. First, we define the semantics for our logic and judgments in Figure 5. Formulas are interpreted on a trace $\mathcal{T}$. We elide the rules for first-order logic connectives. A tuple $P(\vec{t})$ is derivable by node $\iota$ at time $\tau$, if $P(\vec{t})$ is either an internal update or an external update generated at a time point $\tau'$ no later than $\tau$. A node $\iota$ sends out a tuple $P(\iota', \vec{t})$ if that tuple was derived by node $\iota$. Because $\iota'$ is different from $\iota$, it is sent over the network. A *received tuple* is one that comes from the network (obtained using DeQueue). Finally, an honest node $\iota$ runs *prog* at time $\tau$ and the local state of $\iota$ at time $\tau$ is the initial state with an empty table and update queue.

$\mathcal{T} \models P(\vec{t})@(\iota,\tau)$ iff $\exists \tau' \leqslant \tau$, $\mathcal{C}$ is the configuration on $\mathcal{T}$ prior to time $\tau'$,
$(\iota,\Psi,\mathcal{U},prog(\iota)) \in \mathcal{C}$, at time $\tau'$, $(\iota,\Psi,\mathcal{U},prog(\iota)) \hookrightarrow (\iota,\Psi',\mathcal{U}' \circ \mathcal{U}_{in},prog(\iota)),\mathcal{U}_e$,
and either $P(\vec{t}) \in \mathcal{U}_{in}$ or $P(\vec{t}) \in \mathcal{U}_e$

$\mathcal{T} \models \mathsf{send}(\iota,\mathsf{tp}(P,\iota',\vec{t}))@\tau$ iff $\mathcal{C}$ is the configuration on $\mathcal{T}$ prior to time $\tau$,
$(\iota,\Psi,\mathcal{U},prog(\iota)) \in \mathcal{C}$, at time $\tau$, $(\iota,\Psi,\mathcal{U},prog(\iota)) \hookrightarrow \mathcal{S}',\mathcal{U}_e$ and $P(@\iota',\vec{t}) \in \mathcal{U}_e$

$\mathcal{T} \models \mathsf{recv}(\iota,\mathsf{tp}(P,\vec{t}))@\tau$ iff $\exists \tau' \leqslant \tau$, $\mathcal{C} \xrightarrow{\tau'} \mathcal{C}' \in \mathcal{T}$,
$\mathcal{Q}$ is the network queue in $\mathcal{C}$, $P(\vec{t}) \in \mathcal{Q}$, $(\iota,\Psi,\mathcal{U},prog(\iota)) \in \mathcal{C}'$ and $P(\vec{t}) \in \mathcal{U}$

$\mathcal{T} \models \mathsf{honest}(\iota,prog(\iota),\tau)$ iff at time $\tau$, node $\iota$'s local state is $(\iota,\ [],\ [],\ prog\ (\iota))$

$\Gamma \models prog(i) : \{i,y_b,y_e\}.\varphi(i,y_b,y_e)$ iff Given any trace $\mathcal{T}$ such that $\mathcal{T} \models \Gamma$,
and at time $\tau_b$, node $\iota$'s local state is $(\iota,\ [],\ [],\ prog(\iota))$
given any time point $\tau_e$ such that $\tau_e \geqslant \tau_b$, it is the case that $\mathcal{T} \models \varphi(\iota,\tau_b,\tau_e)$

**Fig. 5.** Trace-based semantics

The semantics of invariant assertion states that if a trace $\mathcal{T}$ contains the execution of *prog* by node $\iota$ (formally defined as the node running *prog* is one of the nodes in the configuration $\mathcal{C}$), then given any time point $\tau_e$ after $\tau_b$, the trace $\mathcal{T}$ satisfies $\varphi(\iota,\tau_b,\tau_e)$. This definition allows *prog* to run concurrently with other programs, some of which may be controlled by the adversary.

The program logic is proven to be sound with regard to the trace semantics.

**Theorem 1 (Soundness)** *1. If $\Sigma;\Gamma \vdash \varphi$, then for all grounding substitution $\sigma$ for $\Sigma$, given any trace $\mathcal{T}$, $\mathcal{T} \models \Gamma\sigma$ implies $\mathcal{T} \models \varphi\sigma$;*

*2. If $\Sigma;\Gamma \vdash prog(i) : \{i,y_b,y_e\}.\varphi(i,y_b,y_e)$, then for all grounding substitution $\sigma$ for $\Sigma$, $\Gamma\sigma \models (prog)\sigma(i) : \{i,y_b,y_e\}.(\varphi(i,y_b,y_e))\sigma$.*

## 4 Verification Condition Generator

As a step towards automated verification, we implement a verification condition generator (VCGen) to automatically extract proof obligations from a SANDLog program. VCGen is implemented in C++ and fully integrated to RapidNet [26], a declarative networking engine for compiling SANDLog programs. We target Coq, but other interactive theorem provers such as Isabelle HOL are possible.

More concretely, VCGen generates lemmas corresponding to the last premise of rule Inv. It takes as inputs: the abstract syntax tree of a SANDLog program *sp*, and type annotations *tp*. The generated Coq file contains the following: (1) definitions for types, predicates, and functions; (2) lemmas for rules in the SANDLog program; and (3) axioms based on Honest rule.

**Definition.** Predicates and functions are declared before they are used. Each predicate (tuple) $p$ in the SANDLog program corresponds to a predicate of the same name in the Coq file, with two additional arguments: a location specifier and a time point. For example, the generated declaration of the *link* tuple $\mathsf{link}(@node, node)$ is the following

```
Variable link: node → node → node → time → Prop.
```

For each user-defined function, a data constructor of the same name is generated, unless it corresponds to a Coq's built-in operator (e.g. list operations). The function takes a time point as an additional argument.

| | |
|---|---|
| $\mathsf{link}(@n, n')$ | there is a link between $n$ and $n'$. |
| $\mathsf{route}(@n, d, c, p, sl)$ | $p$ is a path to $d$ with cost $c$. |
| | $sl$ is the signature list associated with $p$. |
| $\mathsf{prefix}(@n, d)$ | $n$ owns prefix (IP addresses) $d$. |
| $\mathsf{bestRoute}(@n, d, c, p, sl)$ | $p$ is the best path to $d$ with cost $c$. |
| | $sl$ is the signature list associated with $p$. |
| $\mathsf{verifyPath}(@n, n', d, p, sl,$ | a path $p$ to $d$ needs verifying against signature list $sl$. |
| $pOrig, sOrig)$ | $p$ is a sub-path of $pOrig$, and $s$ is a sub-list of $sOrig$. |
| $\mathsf{signature}(@n, m, s)$ | $n$ creates a signature $s$ of message $m$ with private key. |
| $\mathsf{advertisement}(@n', n, d, p, sl)$ | $n$ advertises path $p$ to neighbor $n'$ with signature list $sl$. |

**Fig. 6.** Tuples for $prog_{sbgp}$

**Lemmas.** For each rule in a SANDLog program, VCGen generates a lemma in the form of the last premise in inference rule Inv (Figure 4). Rule *sp1* of example program in Section 2.1, for instance, corresponds to the following lemma:

```
Lemma r1: forall(s:node)(d:node)(c:nat)(p:list node)(t:time),
  link s d c s t → p = cons (s (cons d nil)) → p-path s t s d c p t.
```

Here, *cons* is Coq's built-in list appending operation. and *p-path* is the invariant associated with predicate *path*.

**Axioms.** For each invariant $\varphi_p$ of a rule head $p$, VCGen produces an axiom of the form: $\forall i, t, \vec{x}, \mathsf{Honest}(i) \supset p(\vec{x})@(i, t) \supset \varphi_p(i, \vec{x})$. These axioms are conclusions of the Honest rule after invariants are verified. Soundness of these axioms is backed by Theorem 1. Since we always assume that the program starts at time $-\infty$, the condition that $t > -\infty$ is always true, thus omitted.

## 5 Case Studies

We investigate two secure routing protocols: S-BGP and SCION. Due to space constraints, we present in detail the verification of one property of S-BGP. All SANDLog specifications and Coq proofs can be found online at `http://netdb.cis.upenn.edu/forte2014/`.

**Encoding.** Secure Border Gateway Protocol (S-BGP) provides security guarantees such as origin authenticity and route authenticity over BGP through PKI and signature-based attestations. Our SANDLog encoding includes all necessary details of S-BGP's route attestation mechanisms. S-BGP requires that each node sign the route information it advertises to its neighbor, which includes the path, the destination prefix (IP address), and the identifier of the intended neighbor. Along with the advertisement, a node sends its own signature as well as all signatures signed by previous nodes on the subpaths. Upon receiving an advertisement, a node verifies all signatures.

Key tuples generated at each node executing $prog_{sbgp}$ are listed in Figure 6. Here $n$ is the parameter representing the identifier of the node that runs $prog_{sbgp}$. All tuples except advertisement are stored at node $n$. An advertisement tuple encodes a route advertisement that, once generated, is sent over the network to one of $n$'s neighbors. We summarize $prog_{sbgp}$ encoding in Table 1.

**Empirical evaluation.** We use RapidNet [26] to generate low-level implementation from SANDLog encoding of S-BGP and SCION. We validate the low-level implementation in the ns-3 simulator [1]. Our experiments are performed on a synthetically generated topology consisting of 40 nodes, where each node runs the generated implementation of the SANDLog program. The observed execution traces and communication patterns match the expected protocol behavior. We also confirm that the implementation defends against known attacks such as adversely advertising non-existent routes.

**Property specification.** We focus on route authenticity, encoded as $\varphi_{auth1}$ below. It holds on any execution trace of a network where some nodes run S-BGP, and those who do not are considered malicious.

$$\varphi_{auth1} = \forall n, m, t, d, p, sl, \\ \mathsf{Honest}(n) \wedge \mathsf{advertisement}(m, n, d, p, sl)@(n, t) \supset \mathsf{goodPath}(t, p, d)$$

Formula $\varphi_{auth1}$ asserts a property $\mathsf{goodPath}(t, p, d)$ on any $\mathsf{advertisement}$ tuple generated by an honest node $n$. $\mathsf{goodPath}(t, p, d)$ defined below asserts that all links in path $p$ reaching the destination IP prefix $d$ must have existed at a time point no later than $t$. This means that every pair of adjacent nodes $n$ and $m$ in path $p$ had in their databases: tuple `link(@n, m)` and `link(@m, n)` respectively.

$$\frac{\mathsf{Honest}(n) \supset \exists t', t' \leqslant t \wedge \mathsf{prefix}(n, d)@(n, t')}{\mathsf{goodPath}(t, n :: nil, d)}$$

$$\frac{\mathsf{Honest}(n) \supset \exists t', t' \leqslant t \wedge \mathsf{link}(n, n')@(n, t') \qquad \mathsf{goodPath}(t, n :: nil, d)}{\mathsf{goodPath}(t, n' :: n :: nil, d)}$$

$$\frac{\begin{array}{c}\mathsf{Honest}(n) \supset \exists t', t' \leqslant t \wedge \mathsf{link}(n, n')@(n, t') \wedge \exists t'', t'' \leqslant t \wedge \mathsf{link}(n, n'')@(n, t'') \\ \mathsf{goodPath}(t, n :: n'' :: p'', d)\end{array}}{\mathsf{goodPath}(t, n' :: n :: n'' :: p'', d)}$$

The base case is when $p$ has only one node, and we require that $d$ be one of the prefixes owned by $n$ (i.e., the $\mathsf{prefix}$ tuple is derivable). When $p$ has two nodes $n'$ and $n$, we require that the link from $n$ to $n'$ exist from $n$'s perspective, assuming that $n$ is honest. The last case checks that both links (from $n$ to $n'$ and from $n$ to $n''$) exist from $n$'s perspective, assuming $n$ is honest. In the last two rules, we also recursively check that the subpath also satisfies $\mathsf{goodPath}$. By varying the definition of $\mathsf{goodPath}$, we can specify different properties such as

| Rule | Summary | Head Tuple |
|---|---|---|
| **r1:** | Generate a route for prefix of own. | $\mathsf{route}(@n, d, c, p, sl)$ |
| **r2:** | Generate a best route for destination. | $\mathsf{bestRoute}(@n, d, c, p, sl)$ |
| **r3:** | Receive advertisement from neighbor. | $\mathsf{verifyPath}(@n, n', d, p, sl, pOrig, sOrig)$ |
| **r4:** | Recursively verify signature list. | $\mathsf{verifyPath}(@n, n', d, p, sl, pOrig, sOrig)$ |
| **r5:** | Generate a route for verified path. | $\mathsf{route}(@n, d, c, p, sl)$ |
| **r6:** | Generate a signature for new route. | $\mathsf{signature}(@n, m, s)$ |
| **r7:** | Send route advertisement to neighbors. | $\mathsf{advertisement}(@n', n, d, p, sl)$ |

**Table 1.** Summary of $prog_{sbgp}$ encoding

one that requires each subpath be authorized by the sender. $\varphi_{auth1}$ is a general topology-independent security property.

**Verification.** To use the authenticity property of the signatures, we include the following axiom $A_{sig}$ in the logical context $\Gamma$. This axiom states that if $s$ is verified by the public key of $n'$, and the node $n'$ is honest, then $n'$ must have generated a signature tuple. Predicate $\mathsf{verify}(m, s, k)@(n, t)$, generated by VCGen when function *f_verify* in SANDLog returns true, means that node $n$ verifies at time $t$ that $s$ is a valid signature of message $m$ according to key $k$.

$$A_{sig} = \forall m, s, k, n, n', t, \mathsf{verify}(m, s, k)@(n, t) \wedge \mathsf{publicKeys}(n, n', k)@(n, t) \wedge \mathsf{Honest}(n') \supset \exists t', t' < t \wedge \mathsf{signature}(n', m, s)@(n', t')$$

We first prove that $prog_{sbgp}$ has an invariant property $\varphi_I$:

(a) $\cdot;\cdot \vdash prog_{sbgp}(n) : \{i, y_b, y_e\}.\varphi_I(i, y_b, y_e)$

with $\varphi_I(i, y_b, y_e) = \bigwedge_{p \in hdOf(prog_{sbgp})} \forall t\ \vec{x}, y_b \leqslant t < y_e \wedge p(\vec{x})@(i, t) \supset \varphi_p(i, t, \vec{x})$.

Here, every $\varphi_p$ in $\varphi_I$ denotes the invariant property associated with each head tuple in $prog_{sbgp}$, and needs to be specified by the user. For instance, the invariant associated with the advertisement tuple is denoted $\varphi_{advertisement}$:

$\varphi_{advertisement}(i, t, n', n, d, p, sl) = \mathsf{goodPath}(t, p, d)$.

The proof of (a) is carried out in Coq; we manually discharged all lemmas generated by VCGen. Next, applying the Honest rule to (a), we can deduce $\varphi = \forall n\ t, \mathsf{Honest}(n) \supset \varphi_I(n, t, -\infty)$. $\varphi$ is injected into the assumptions ($\Gamma$) by VCGen, and is safe to be used in subsequent proof steps. Finally, $\varphi \supset \varphi_{auth1}$ is also proven in Coq by applying standard first-order logic rules.

We explain interesting steps of proving (a). Similar to verifying (a), using Inv, Honest and keeping the only clause related to signature, we derive the following:

$$(\mathrm{a}_2)\ \cdot;\cdot \vdash \forall n, \forall t, \mathsf{Honest}(n) \wedge \mathsf{signature}(n, m, s)@(n, t) \supset \exists n', d, m = d :: n' :: p \wedge \varphi_{link2}(p, n, d, n', t)$$

Formula $\varphi_{link2}(p, n, d, n', t)$ states that $n$ is the first node on the path $p$, the link from $n$ to the next node on $p$ exists, and the link between $n$ and the receiving node $n'$ also exists. This matches the non-recursive conditions in the definition of goodPath.

$$\begin{aligned}&\varphi_{link2}(p, n, d, n', t) = \mathsf{link}(n, n')@(n, t) \wedge \\ &\quad \exists p',\ p = n :: p' \wedge (p' = \mathsf{nil} \supset \mathsf{prefix}(n, d)@(n, t)) \\ &\quad \wedge \forall p'', m', p' = m' :: p'' \supset \mathsf{link}(n, m')@(n, t)\end{aligned}$$

Now ($\mathrm{a}_2$) has connected an honest node's signature to the existence of links related to it. Combining ($\mathrm{a}_2$) and $A_{sig}$, each time a signature $sig$ of a node $n$ is properly verified in $prog_{sbgp}$, the invariant link2 (link tuples existed) holds under the assumption that $n$ is honest.

Defining the invariant for tuple verifyPath is technically challenging. The direction in which we check the signature list is different from the direction in which the route is created. The invariant needs to convey that part of a path has been verified, and part of it still needs to be verified. The solution is to use

implication to state that if the path to be verified satisfies goodPath, then the entire path satisfies goodPath.

## 6 Related Work

**Cryptographic protocol analysis.** Analyzing cryptographic protocols [12, 27, 17, 25, 14, 6, 4, 15] has been an active area of research. Compared to cryptographic protocols, secure routing protocols have to deal with arbitrary network topologies and the protocols are more complicated: they may access local storage and commonly include recursive computations. Most model-checking techniques are ineffective in the presence of those complications.

**Verification of trace properties.** A closely related body of work is logic for verifying trace properties of programs (protocols) that run concurrently with adversaries [12, 15]. We are inspired by their program logic that requires the asserted properties of a program to hold even when that program runs concurrently with adversarial programs. One of our contributions is a general program logic for a declarative language SANDLog, which differs significantly from an ordinary imperative language. The program logic and semantics developed here apply to other declarative languages that use bottom-up evaluation strategy.

Wang et al. [29] have developed a proof system for proving correctness properties of networking protocols specified in NDLog. Built on proof-theoretic semantics of Datalog, they automatically translate NDLog programs into equivalent first-order logic axioms. Those axioms state that all the body tuples are derivable if and only if the head tuple is derivable. One main difference is that unlike theirs, we made explicit in our semantics, the trace associated with the distributed execution of a SANDLog program. Another important difference is that we verify invariants associated with each derived tuple in the presence of attackers, which are not present in their system. Therefore, their system cannot be directly used to verify the security properties of secure routing protocols.

**Networking protocol verification.** Recently, several papers have investigated the verification of route authenticity properties on specific wireless routing protocols for mobile networks [2, 3, 11]. They have showed that identifying attacks on route authenticity can be reduced to constraint solving, and that the security analysis of a specific route authenticity property that depends on the topologies of network instances can be reduced to checking these properties on several four-node topologies. In our own prior work [8], we have verified route authenticity properties on variants of S-BGP using a combination of manual proofs and an automated tool, Proverif [7]. The modeling and analysis in these works are specific to the protocols and the route authenticity properties. Some of the properties that we verify in our case study are similar. However, we propose a general framework for leveraging a declarative programming language for verification and empirical evaluation of routing protocols. The program logic proposed here can be used to verify generic safety properties of SANDLog programs.

There has been a large body of work on verifying the correctness of various network protocol design and implementations using proof-based and model-checking techniques [5, 16, 13, 29]. The program logic presented here is customized

to proving safety properties of SANDLog programs, and may not be expressive enough to verify complex correctness properties. However, the operational semantics for SANDLog can be used as the semantic model for verifying protocols encoded in SANDLog using other techniques.

## 7 Conclusion and Future Work

We have designed a program logic for verifying secure routing protocols specified in the declarative language SANDLog. We have integrated verification into a unified framework for formal analysis and empirical evaluation of secure routing protocols. As future work, we plan to expand our use cases, for example, to investigate mechanisms for securing the data (packet forwarding) plane [22]. In addition, as an alternative to Coq, we are also exploring the use of automated first-order logic theorem provers to automate our proofs.

## 8 Acknowledgments

The authors wish to thank the anonymous reviewers for their useful suggestions. Our work is supported by NSF CNS-1218066, NSF CNS-1117052, NSF CNS-1018061, NSF CNS-0845552, NSFITR-1138996, NSF CNS-1115706, AFOSR Young Investigator award FA9550-12-1-0327 and NSF ITR-1138996.

## References

1. ns 3 project: Network Simulator 3, `http://www.nsnam.org/`
2. Arnaud, M., Cortier, V., Delaune, S.: Modeling and verifying ad hoc routing protocols. In: Proceedings of CSF (2010)
3. Arnaud, M., Cortier, V., Delaune, S.: Deciding security for protocols with recursive tests. In: Proceedings of CADE (2011)
4. Bau, J., Mitchell, J.: A security evaluation of DNSSEC with NSEC3. In: Proceedings of NDSS (2010)
5. Bhargavan, K., Obradovic, D., Gunter, C.A.: Formal verification of standards for distance vector routing protocols. J. ACM 49(4) (2002)
6. Blanchet, B.: Automatic verification of correspondences for security protocols. J. Comput. Secur. 17(4) (Dec 2009)
7. Blanchet, B., Smyth, B.: Proverif 1.86: Automatic cryptographic protocol verifier, user manual and tutorial, `http://www.proverif.ens.fr/manual.pdf`
8. Chen, C., Jia, L., Loo, B.T., Zhou, W.: Reduction-based security analysis of internet routing protocols. In: WRiPE (2012)
9. Chen, C., Jia, L., Xu, H., Luo, C., Zhou, W., Loo, B.T.: A program logic for verifying secure routing protocols. Tech. rep., CIS Dept. University of Pennsylvania (February 2014), `http://netdb.cis.upenn.edu/forte2014`
10. CNET: How pakistan knocked youtube offline, `http://news.cnet.com/8301-10784_3-9878655-7.html`
11. Cortier, V., Degrieck, J., Delaune, S.: Analysing routing protocols: four nodes topologies are sufficient. In: Proceedings of POST (2012)
12. Datta, A., Derek, A., Mitchell, J.C., Roy, A.: Protocol Composition Logic (PCL). Electronic Notes in Theoretical Computer Science 172, 311–358 (2007)

13. Engler, D., Musuvathi, M.: Model-checking large network protocol implementations. In: Proceedings of NSDI (2004)
14. Escobar, S., Meadows, C., Meseguer, J.: A rewriting-based inference system for the NRL protocol analyzer: grammar generation. In: Proceedings of FMSE (2005)
15. Garg, D., Franklin, J., Kaynar, D., Datta, A.: Compositional system security with interface-confined adversaries. ENTCS 265, 49–71 (September 2010)
16. Goodloe, A., Gunter, C.A., Stehr, M.O.: Formal prototyping in early stages of protocol design. In: Proceedings of ACM WITS (2005)
17. He, C., Sundararajan, M., Datta, A., Derek, A., Mitchell, J.C.: A modular correctness proof of IEEE 802.11i and TLS. In: Proceedings of CCS (2005)
18. Kamp, H.W.: Tense Logic and the Theory of Linear Order. Phd thesis, Computer Science Department, University of California at Los Angeles, USA (1968)
19. Kent, S., Lynn, C., Mikkelson, J., Seo, K.: Secure border gateway protocol (S-BGP). IEEE Journal on Selected Areas in Communications 18, 103–116 (2000)
20. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative Networking: Language, Execution and Optimization. In: SIGMOD (2006)
21. Loo, B.T., Condie, T., Garofalakis, M., Gay, D.E., Hellerstein, J.M., Maniatis, P., Ramakrishnan, R., Roscoe, T., Stoica, I.: Declarative networking. In: Communications of the ACM (2009)
22. Naous, J., Walfish, M., Nicolosi, A., Mazieres, D., Miller, M., Seehra, A.: Verifying and enforcing network paths with ICING. In: Proceedings of CoNEXT (2011)
23. Nigam, V., Jia, L., Loo, B.T., Scedrov, A.: Maintaining distributed logic programs incrementally. In: Proceedings of PPDP (2011)
24. One Hundred Eleventh Congress: 2010 report to congress of the u.s.-china economic and security review commission (2010), `http://www.uscc.gov/annual_report/2010/annual_report_full_10.pdf`
25. Paulson, L.C.: Mechanized proofs for a recursive authentication protocol. In: Proceedings of CSFW (1997)
26. RapidNet: A Declarative Toolkit for Rapid Network Simulation and Experimentation: `http://netdb.cis.upenn.edu/rapidnet/`
27. Roy, A., Datta, A., Derek, A., Mitchell, J.C., Jean-Pierre, S.: Secrecy analysis in protocol composition logic. In: Proceedings of ESORICS (2007)
28. Wan, T., Kranakis, E., Oorschot, P.C.: Pretty secure BGP (psBGP). In: Proceedings of NDSS (2005)
29. Wang, A., Basu, P., Loo, B.T., Sokolsky, O.: Declarative network verification. In: Proceedings of PADL (2009)
30. White, R.: Securing bgp through secure origin BGP (soBGP). The Internet Protocol Journal 6(3), 15–22 (2003)
31. Zhang, X., Hsiao, H.C., Hasker, G., Chan, H., Perrig, A., Andersen, D.G.: Scion: Scalability, control, and isolation on next-generation networks. In: Proceedings of IEEE S&P (2011)