A Program Logic for Verifying Secure Routing Protocols (Technical Report)

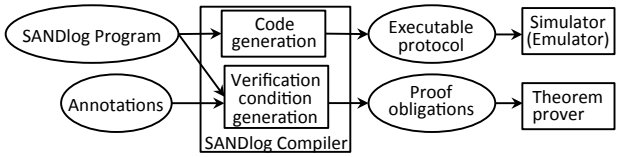
Chen Chen[1], Limin Jia[2], Hao Xu[1], Cheng Luo[1], Wenchao Zhou[3], and Boon Thau Loo[1]

[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 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 the *Border Gateway Protocol* (BGP), where routers are grouped into Autonomous Systems (*AS*) 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,



The round objects are code (proofs), which are the input or output of the framework. The rectangular objects are software components of the framework.

Fig. 1. Architecture of a unified framework for implementing and verifying secure routing protocols.

ASes can advertise non-existent routes or claim to own IP prefixes that they do not. These faults may lead to long periods of interruption of the Internet; best epitomized by recent high-profile attacks [9, 22].

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) [17], ps-BGP [26], so-BGP [27]), to "clean-slate" Internet architectural redesigns such as SCION [28] and ICING [20]. 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, 11, 13] are rarely used in analyzing routing protocols because they 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 explore a novel proof methodology to verify these protocols. We augment prior work on declarative networking (NDLog) [19] with cryptographic libraries to provide compact encoding of secure routing protocols. We call this extension SANDLog (stands for *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 [19]. 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 a Coq file, where proof can be filled. VCGen is integrated into the SANDLog compiler, an cryptography-augmented extension to the declarative networking engine RapidNet [24]. The compiler is able to translate our 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).

2 SANDLog

We introduce the syntax and operational semantics of SANDLog, which extends the *Network Datalog* (NDLog) [19] 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 [19] 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.

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

To support distributed execution, SANDLog assumes that each node has a unique identifier denoted ι. 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 executing such a rule, 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.

We list all available cryptographic functions in Table 1. Note that we currently do not implement all cryptographic operations, such as symmetric encryption and MD5, because they are not used in our encoding yet. However, it is not hard to include them when needed.

Function name	Description
f_sign_asym(info, key)	Create a signature of *info* using *key*
f_verify_asym(cipher, key)	Decrypt a *cipher* with *key*
f_mac(info, key)	Create a message authentication code of *info* using *key*
f_verifymac(info, MAC, key)	Verify *info* against *MAC* with *key*

Table 1. Cryptographic functions in SANDLog

Example Program. The following program computes the best path between each pair of nodes in a network. s is the location parameter of the program. It stores three 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 $\mathsf{path}(@s, d, c, p) :\!\!-\ \mathsf{link}(@s, d, c), p := [s, d].$
sp2 $\mathsf{path}(@z, d, c, p) :\!\!-\ \mathsf{link}(@s, z, c1), \mathsf{path}(@s, d, c2, p1), c := c1 + c2, p := z::p1.$
sp3 $\mathsf{bestPath}(@s, d, \mathsf{min}\langle c\rangle, p) :\!\!-\ \mathsf{path}(@s, d, c, p).$

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 [18], 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*	$\mathcal{C} ::= \mathcal{Q} \triangleright \mathcal{S}_1, \cdots, \mathcal{S}_n$

We write P to denote tuples. The database for storing all derived tuples on a node is denoted Ψ. 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 ι, the database Ψ, a list of unprocessed updates $\mathcal{U}$, and the program *prog* that ι 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 Ψ_A, Ψ_B and Ψ_C. They have constructed paths to their neighbors (i.e., the corresponding path and bestPath

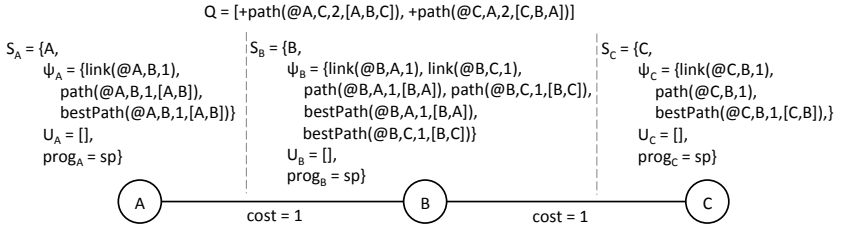


Fig. 2. An Example Scenario.

$\boxed{\mathcal{S} \hookrightarrow \mathcal{S}', \mathcal{U}}$

$$\frac{\begin{array}{c}\mathcal{U}_{in} = [+p_1(@\iota, \vec{t}), ..., +p_m(@\iota, \vec{t})] \qquad \forall j \in [1, m], p_j(@\iota, \vec{t}) \in \mathit{BaseOf}(\mathit{prog}) \\ \mathcal{U}_{ext} = [+q_1(@\iota_1, \vec{t}), ..., +q_k(@\iota_k, \vec{t})] \quad \forall j \in [1, k], q_j(@\iota_j, \vec{t}) \in \mathit{BaseOf}(\mathit{prog}), \iota_j \neq \iota\end{array}}{(\iota, \varnothing, [], \mathit{prog}) \hookrightarrow (\iota, \varnothing, \mathcal{U}_{in}, \mathit{prog}), \mathcal{U}_{ext}} \textsc{Init}$$

$$\frac{(\mathcal{U}_{in}, \mathcal{U}_{ext}) = \mathit{fireRules}(\iota, \Psi, u, \Delta \mathit{prog})}{(\iota, \Psi, u :: \mathcal{U}, \mathit{prog}) \hookrightarrow (\iota, \Psi \uplus u, \mathcal{U} \circ \mathcal{U}_{in}, \mathit{prog}), \mathcal{U}_{ext}} \textsc{RuleFire}$$

$\boxed{\mathcal{C} \xrightarrow{t} \mathcal{C}'}$

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

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

Fig. 3. Operational Semantics

$\boxed{\mathit{fireRules}(\iota, \Psi, u, \Delta \mathit{prog}) = (\mathcal{U}_{in}, \mathcal{U}_{ext})}$

$$\frac{}{\mathit{fireRules}(\iota, \Psi, u, []) = ([], [])} \textsc{Empty}$$

$$\frac{\begin{array}{c}\mathit{fireSingleR}(\iota, \Psi, u, \Delta r) = (\Psi', \mathcal{U}_{in1}, \mathcal{U}_{ext1}) \\ \mathit{fireRules}(\iota, \Psi', u, \Delta \mathit{prog}) = (\mathcal{U}_{in2}, \mathcal{U}_{ext2})\end{array}}{\mathit{fireRules}(\iota, \Psi, u, (\Delta r, \Delta \mathit{prog})) = (\mathcal{U}_{in1} \circ \mathcal{U}_{in2}, \mathcal{U}_{ext1} \circ \mathcal{U}_{ext2})} \textsc{Seq}$$

Fig. 4. Definition of *fileRules*.

tuples are stored). In addition, node B has applied *sp2* and generated updates +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 τ is the time of the transition step. A trace $\mathcal{T}$ is a sequence of transitions: $\xrightarrow{\tau_0} \mathcal{C}_1 \xrightarrow{\tau_1} \mathcal{C}_2 \cdots \xrightarrow{\tau_n} \mathcal{C}_{n+1}$. We assume that the effects of a transition take place at time τ_i (reflected in $\mathcal{C}_{i+1}$). Figure 3 defines the rules for these two transition relations.

– **Global state transition ($\mathcal{C} \xrightarrow{t} \mathcal{C}'$).** 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 $\mathcal{S}$.
– **Local state transition ($\mathcal{S} \hookrightarrow \mathcal{S}', \mathcal{U}$).** Rule Init applies when the program starts to run. Here, only base rules–rules that do not have a rule body–can fire. The auxiliary function *BaseOf*(*prog*) returns all the base rules in *prog*. In the resulting state, the internal update list ($\mathcal{U}_{in}$) contains all the insertion updates located at ι, and the external update list ($\mathcal{U}_{ext}$) contains only updates meant to be stored at a node different from ι. Rule RuleFire (Figure 4) computes new updates based on the program and the first update in the update list. It uses a relation *fireRules*, which processes an update u, and returns a pair of update lists, one for node ι itself, the other for other nodes. Rules for *fireRules* can be found in Appendix **??**.After u is processed, the database of ι is updated with the update u ($\Psi \uplus u$). The $\uplus$ operation increases (decreases) the reference count of P in Ψ by 1, when u is an insertion (deletion) update $+P$ ($-P$). Finally, the update list in the resulting state is augmented with the new updates generated from processing u.

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

Incremental Maintenance. Now we explain in more detail how the database is maintained incrementally by processing updates one at a time. Following the strategy proposed in [18], the local database is maintained incrementally by processing updates one at a time. The rules are rewritten into Δ *rules*, which can efficiently generate all the updates triggered by one update. For any given rule r, containing k body tuples, k Δ rules of the following form are generated, one for each $i \in [1, k]$.

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

Δp_i in the body denotes the update currently being considered. Δp in the head denotes new updates that are generated as the result of firing this rule. Here p^{ν} 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 Δ rules for *sp2* are:

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

Rules *sp2a* and *sp2b* are Δ 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, path^{ν} includes the tuple path(@A,C,2,[A,B,C]), while path does not. On the other hand, link^{ν} 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. The difference between p^{ν} and p is only manifested when a rule body contains more than one p.

Rule Firing. We present in Figure 5 a selected set of rules for firing a single Δ rule given an insertion update. *fireSingleR* is used in the base case of *fireRules*. We write Ψ^{ν} to denote the table resulted from updating Ψ with the current update: $\Psi^{\nu} = \Psi \uplus u$.

When the tuple to be inserted already exists, we do not need to further propagate the update (Rule InsExists). Instead, the reference count is increased. In this case, both update lists are empty. Rule InsNew handles the case where new updates are generated by firing rule r. We use an auxiliary function $\rho(\Psi^{\nu},\Psi,r,i,\vec{t})$ to extract the complete list of substitutions that allows the rule r to fire. Here i and $\vec{t}$ indicate that $q_i(\vec{t})$ is the current update, where q_i is the i^{th} body tuple of rule r. Every substitution σ in that set is a general unifier of the body tuples and constraints. Formally:

(1) $\sigma(\vec{t}) = \sigma(agB_i)$, (2) $\forall j \in [1,i-1], \exists \vec{s}, \vec{s} = \sigma(agB_j)$ and $q_j(\vec{s}) \in \Psi^{\nu}$
(3) $\forall j \in [i+1,n], \exists \vec{s}, \vec{s} = \sigma(agB_j)$ and $q_j(\vec{s}) \in \Psi$ (4) $\forall k \in [1,m], \sigma[a_k]$ is true

We write $[a]$ to denote the constraint that a represents. When a is an assignment (i.e., $x := f(\vec{t})$), $[a]$ is the equality constraint $x = f(\vec{t})$; otherwise, $[a]$ is a.

We use a selection function *sel* to decide which substitution to propagate. It is needed when multiple tuples with the same key are derived using this rule. In NDLog run time, similar to a relational database, a key value of a stored tuple $p(\vec{t})$ uniquely identifies that tuple. When a different tuple $p(\vec{t'})$ with the same key is derived, the old value $p(\vec{t})$ and any tuple derived using it need to be deleted. For instance, the first two arguments of path are its key. $\mathsf{path}(A,B,1)$ and $\mathsf{path}(A,B,2)$ cannot both exist in the database. When multiple tuples with the same key are derived at the same time, we need to make a decision as to which one to keep. We use a *genUpd* function to generate appropriate updates based on the selected substitutions. It may generate deletion updates in addition to an insertion update of the new value. Use the previous example, assume that $\mathsf{path}(A,B,3)$ is in Ψ^{ν}. If we were to choose $\mathsf{path}(A,B,1)$ because it appears earlier in the update list, then *genUpd* returns $\{+\mathsf{path}(A,B,1), -\mathsf{path}(A,B,3)\}$. We omit the details of the definitions of *sel* and *genUpd* here, as there are many possible strategies for implementing these two functions. The only relevant part to the logic we introduce later is that the substitutions used for an insertion update come from the ρ function, and that the substitutions satisfy the property we defined above.

The rest of the rules deal with generating an aggregate tuple. To efficiently implement aggregates, for each tuple p that has an aggregate function in its

arguments, there is an internal tuple p_{agg} that records all candidate values of p. When there is a change to the candidate set, the aggregate is re-computed. In our example (Figure 2), $\mathsf{bestpath}_{agg}$ maintains all candidate path tuples. We omit these auxiliary tuples from the figure for brevity.

We require that the location specifier of a rule head containing an aggregate function be the same as that of the rule body. With this restriction, the state of an aggregate is maintained in one single node. If the result of the aggregate is needed by a remote node, we can write an additionally rule to send the result after the aggregate is computed. The aggregation rules for *fireSingleR* return a new table Ψ' because of updates to these candidate sets p_{agg}.

Rule InsAggNew applies when the aggregate is generated for the first time. We only need to insert the new aggregate value to the table. Additional rules are required to handle aggregates where the new aggregate is the same as the old one or replaces the old one. We omit these rules due to space constraints.

We revisit the example in Figure 2. Upon receiving +path(@A,C,2,[A,B,C]), Δ rule *sp2b* will be triggered 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 (rule InsNew). The Δ rule for *sp3* will also be triggered, and generate a new update +bestPath(@A,C,2,[A,B,C]), which will be included in $\mathcal{U}_{in}$ (rule InsAggNew). After evaluating the Δ 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]).

Rule InsAggSame applies when the new aggregates is the same as the old one. In this case, only the candidate set is updated, and no new update is propagated. Rule InsAggUpd applies when there is a new aggregate value. In this case, we need to generate a deletion update of the old tuple before inserting the new one.

Figure 6 summaries the deletion rules. When the tuple to be deleted has multiple copies, we only reduce its reference count. The rest of the rules are the dual of the corresponding insertion rules.

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 [21] 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 [21]. SANDLog additionally allows aggregates, which are not included in [21]. 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 oper-

$$\boxed{\mathit{fireSingleR}(\iota, \Psi, u, \Delta r) = (\Psi', \mathcal{U}_{in}, \mathcal{U}_{ext})}$$

$$\frac{q_i(\vec{t}) \in \Psi}{\mathit{fireSingleR}(\iota, \Psi, +q_i(\vec{t}), \Delta r) = (\Psi, [], [])}\ \textsc{InsExists}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota_1, ags) :\!\!- \cdots, \Delta q_i(agB_i) \cdots \\ q_i(\vec{t}) \notin \Psi \qquad ags \text{ does not contain any aggregate} \\ \Sigma = \rho(\Psi^\nu, \Psi, r, i, \vec{t}) \qquad \Sigma' = sel(\Sigma, \Psi^\nu) \qquad \mathcal{U} = genUpd(\Sigma, \Sigma', p, \Psi^\nu) \\ \text{if } \iota_1 = \iota \text{ then } \mathcal{U}_i = \mathcal{U}, \mathcal{U}_e = [] \text{ otherwise } \mathcal{U}_i = [], \mathcal{U}_e = \mathcal{U}\end{array}}{\mathit{fireSingleR}(\iota, \Psi, +q_i(\vec{t}), \Delta r) = (\Psi, \mathcal{U}_i, \mathcal{U}_e)}\ \textsc{InsNew}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota, ags) :\!\!- \cdots, \Delta q_i(agB_i) \cdots \qquad q_i(\vec{t}) \notin \Psi \\ ags \text{ contains an aggregate } F_{agg} \qquad \{\sigma_1, \cdots, \sigma_k\} = \rho(\Psi^\nu, \Psi, r, i, \vec{t}) \\ \Psi' = \Psi \uplus \{p_{agg}(@\iota, \sigma_1(ags)), \cdots, p_{agg}(@\iota, \sigma_k(ags))\} \\ Agg(p, F_{agg}, \Psi') = p(@\iota, \vec{s}) \qquad p(@\iota, \vec{s}) \in \Psi\end{array}}{\mathit{fireSingleR}(\iota, \Psi, +q_i(\vec{t}), \Delta r) = (\Psi', [], [])}\ \textsc{InsAggSame}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota, ags) :\!\!- \cdots, \Delta q_i(agB_i) \cdots \qquad q_i(\vec{t}) \notin \Psi \\ ags \text{ contains an aggregate } F_{agg} \qquad \{\sigma_1, \cdots, \sigma_k\} = \rho(\Psi^\nu, \Psi, r, i, \vec{t}) \\ \Psi' = \Psi \uplus \{p_{agg}(@\iota, \sigma_1(ags)), \cdots, p_{agg}(@\iota, \sigma_k(ags))\} \\ Agg(p, F_{agg}, \Psi') = p(@\iota, \vec{s}) \qquad p(@\iota, \vec{s}_1) \in \Psi \\ \vec{s} \text{ and } \vec{s}_1 \text{ share the same key but different aggregate value}\end{array}}{\mathit{fireSingleR}(\iota, \Psi, +q_i(\vec{t}), \Delta r) = (\Psi', [-p(@\iota, \vec{s}_1), +p(@\iota, \vec{s})], [])}\ \textsc{InsAggUpd}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota, ags) :\!\!- \cdots, \Delta q_i(agB_i) \cdots \qquad q_i(\vec{t}) \notin \Psi \\ ags \text{ contains an aggregate } F_{agg} \qquad \{\sigma_1, \cdots, \sigma_k\} = \rho(\Psi^\nu, \Psi, r, \vec{t}) \\ \Psi' = \Psi \uplus \{p_{agg}(@\iota, \sigma_1(ags)), \cdots, p_{agg}(@\iota, \sigma_k(ags))\} \\ Agg(p, F_{agg}, \Psi') = p(@\iota, \vec{s}) \qquad \nexists p(@\iota, \vec{s}') \in \Psi \\ \text{such that } \vec{s} \text{ and } \vec{s}' \text{ share the same key but different aggregate value}\end{array}}{\mathit{fireSingleR}(\iota, \Psi, +q_i(\vec{t}), \Delta r) = (\Psi', [+p(@\iota, \vec{s})], [])}\ \textsc{InsAggNew}$$

Fig. 5. Insertion rules for evaluating a single Δ rule

ational 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 logic for reasoning about cryptographic protocols [11, 14], 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 interacting 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

$$\frac{(n, q_i(\vec{t})) \in \Psi \qquad n > 1}{\mathit{fireSingleR}(\iota, \Psi, -q_i(\vec{t}), \Delta r) = (\Psi, [], [])}\ \textsc{DelExists}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota_1, ags) \coloneq \cdots, \Delta q_i(agB_i) \cdots \\ (1, q_i(\vec{t})) \in \Psi \qquad ags \text{ does not contain any aggregate} \\ \{\sigma_1, \cdots, \sigma_k\} = sel(\rho(\Psi^\nu, \Psi, r, i, \vec{t}), \Psi^\nu) \\ \mathcal{U} = [-p(@\iota_1, \sigma_1(ags)), \cdots, -p(@\iota_1, \sigma_k(ags))] \\ \text{if } \iota_1 = \iota \text{ then } \mathcal{U}_i = \mathcal{U}, \mathcal{U}_e = [] \text{ otherwise } \mathcal{U}_i = [], \mathcal{U}_e = \mathcal{U}\end{array}}{\mathit{fireSingleR}(\iota, \Psi, -q_i(\vec{t}), \Delta r) = (\Psi, \mathcal{U}_i, \mathcal{U}_e)}\ \textsc{DelNew}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota, ags) \coloneq \cdots, \Delta q_i(agB_i) \cdots \qquad (1, q_i(\vec{t})) \in \Psi \\ ags \text{ contains an aggregate } F_{agg} \qquad \{\sigma_1, \cdots, \sigma_k\} = \rho(\Psi^\nu, \Psi, r, i, \vec{t}) \\ \Psi' = \Psi \backslash \{p_{agg}(@\iota, \sigma_1(ags)), \cdots, p_{agg}(@\iota, \sigma_k(ags))\} \\ Agg(p, F_{agg}, \Psi') = p(@\iota, \vec{s}) \qquad p(@\iota, \vec{s}) \in \Psi\end{array}}{\mathit{fireSingleR}(\iota, \Psi, -q_i(\vec{t}), \Delta r) = (\Psi', [], [])}\ \textsc{DelAggSame}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota, ags) \coloneq \cdots, \Delta q_i(agB_i) \cdots \qquad (1, q_i(\vec{t})) \in \Psi \\ ags \text{ contains an aggregate } F_{agg} \qquad \{\sigma_1, \cdots, \sigma_k\} = \rho(\Psi^\nu, \Psi, r, i, \vec{t}) \\ \Psi' = \Psi \backslash \{p_{agg}(@\iota, \sigma_1(ags)), \cdots, p_{agg}(@\iota, \sigma_k(ags))\} \\ Agg(p, F_{agg}, \Psi') = p(@\iota, \vec{s}) \qquad p(@\iota, \vec{s}_1) \in \Psi \\ \vec{s} \text{ and } \vec{s}_1 \text{ share the same key but different aggregate value}\end{array}}{\mathit{fireSingleR}(\iota, \Psi, -q_i(\vec{t}), \Delta r) = (\Psi', [-p(@\iota, \vec{s}_1), +p(@\iota, \vec{s})], [])}\ \textsc{DelAggUpd}$$

$$\frac{\begin{array}{c}\Delta r = \Delta p(@\iota, ags) \coloneq \cdots, \Delta q_i(agB_i) \cdots \qquad (1, q_i(\vec{t})) \in \Psi \\ ags \text{ contains an aggregate } F_{agg} \qquad \{\sigma_1, \cdots, \sigma_k\} = \rho(\Psi^\nu, \Psi, r, i, \vec{t}) \\ \Psi' = \Psi \backslash \{p_{agg}(@\iota, \sigma_1(ags)), \cdots, p_{agg}(@\iota, \sigma_k(ags))\} \\ Agg(p, F_{agg}, \Psi') = \mathsf{NULL}\end{array}}{\mathit{fireSingleR}(\iota, \Psi, -q_i(\vec{t}), \Delta r) = (\Psi', [-p(@\iota, \vec{s'})], [])}\ \textsc{DelAggNone}$$

Fig. 6. Deletion rules for evaluating a single Δ rule

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 our 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 run arbitrary SANDLog programs; (2) assumptions about cryptographic functions are admitted as axioms.

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

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

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

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

Fig. 7. Selected Rules in Program Logic

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

Predicate $P(\vec{t})@(\iota,\tau)$ means that tuple $P(\vec{t})$ is derivable at time τ by node ι. The first element in $\vec{t}$ is a location identifier ι', which may be different from ι. When a tuple $P(\iota', ...)$ is derived at node ι, it is sent to ι'. 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 ι has received a tuple $P(\vec{t})$ at time τ. $\mathsf{honest}(\iota, prog(\iota), \tau)$ means that node ι starts to run program $prog(\iota)$ at time τ. Using these atoms and first-order logic connectives, we can specify security properties such as authenticity.

Logical Judgments. The logical judgments use two contexts: context Σ contains all the free variables and Γ contains logical assumptions.

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

Judgment (1) states that φ is provable given the assumptions in Γ. Judgment (2) is an assertion about SANDLog programs. We write $\varphi(\vec{x})$ when $\vec{x}$ are free in φ. $\varphi(\vec{t})$ 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 ι, starting at time τ_b, satisfies $\varphi(\iota, \tau_b, \tau_e)$, given any time point τ_e later than τ_b. Intuitively, the trace contains several threads running concurrently, only one of them runs the program, and the others can be malicious. Since τ_e is any time after τ_b (the time *prog* starts), φ 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 exist 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 7) 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

φ_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, let p be path, and $\varphi_p(i,t,\vec{x})$ be that every link in argument *path* must exist 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 φ_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 ι has received a tuple P before time t is closed under trace extension; the property that node ι 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 — φ holds; and (2) when p is derived using other tuples, φ holds. The second (last) 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 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. Recall that we write $[x := f(\vec{t})]$ to rewrite 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 case (1) mentioned above.

The last (second) premise of Inv can be automatically generated given a SANDLog program and all the corresponding φ_ps. 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 ι 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.

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

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

$\mathcal{T} \vDash \mathsf{recv}(\iota,\mathsf{tp}(P,\vec{t}))@\tau$ 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 $P(\vec{t}) \in \mathcal{U}$

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

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

Fig. 8. Trace-based semantics

Soundness. We prove the soundness of our logic with regard to the trace semantics. First, we define the semantics for our logical judgments in Figure 8. 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 ι at time τ, if $P(\vec{t})$ is either an internal update or an external update generated at a time point τ' no later than τ. A node ι sends out a tuple $P(\iota',\vec{t})$ if that tuple was derived by node ι. Because ι' is different from ι, it is sent over the network. A *received tuple* is one that comes from the network (obtained using DeQueue). Finally, an honest node ι runs *prog* at time τ, if at time τ the local state of ι is the initial state with an empty table and update queue.

The semantics of invariant assertion states that if a trace $\mathcal{T}$ contains the execution of *prog* by node ι (formally defined as the node running *prog* is one of the nodes in the configuration $\mathcal{C}$), then given any time point τ_e after τ_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 σ for Σ, given any trace $\mathcal{T}$, $\mathcal{T} \vDash \Gamma\sigma$ implies $\mathcal{T} \vDash \varphi\sigma$;*

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

Proof of soundness can be found in Appendix B.

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 [24], 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.

Lemmas. For each rule in a SANDLog program, VCGen generates a lemma in the form of the last premise in inference rule INV (Figure 7). 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 φ_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. All SANDLog specifications and Coq proofs can be found online at `http://netdb.cis.upenn.edu/forte2014/`.

5.1 S-BGP

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. The SANDLog encoding of S-BGP ($prog_{sbgp}$) is presented in Appendix C. All honest nodes in a network – nodes that follow S-BGP protocol – run this code. 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 9. Here n is the parameter representing the identifier of the node that runs $prog_{sbgp}$.

$\mathsf{link}(@n, n')$	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. 9. Tuples for $prog_{sbgp}$

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 2.

Empirical Evaluation. We use RapidNet [24] 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 φ_{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 φ_{auth1} asserts a property $\mathsf{goodPath}(t, p, d)$ on any advertisement tuple generated by an honest node n. We define $\mathsf{Honest}(n)$ to mean that n's private key has not been compromised and that n runs S-BGP. Formally: $\mathsf{Honest}(n) \triangleq \mathsf{honest}(n, prog_{sbgp}(n), -\infty)$. Here, the starting time is set to be the earliest possible time point. SANDLog's semantics allows a node to begin execution at any

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 2. Summary of $prog_{sbgp}$ encoding

time after the specified starting time, so using $-\infty$ gives us the most flexibility. $\mathsf{goodPath}(t, p, d)$ 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.

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

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

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

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 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 goodPath. By changing the recursive definition of goodPath, we can verify different properties like whether a link was authorized to use even if it existed. Furthermore, finer granularity of S-BGP encoding allows us to assume more axioms about the network, thus verifying more properties based on φ_{auth1}. By varying the definition of goodPath, we can specify different properties such as one that requires each subpath be authorized by the sender. φ_{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 Γ. 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 node n verified 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 φ_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 φ_p in φ_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 $\varphi = \forall n\ t, \mathsf{Honest}(n) \supset \varphi_I(n, t, -\infty)$. φ is injected into the assumptions (Γ) by

VCGen, and is safe to be used in any future proof. Finally, $\varphi \supset \varphi_{auth1}$ is also proven in Coq by applying standard first-order logic rules.

We provide more details of how to prove (a). Instead of proving (a) directly, we first prove a stronger invariant φ_{I1} of $prog_{sbgp}$,

(a_1) $\cdot;\cdot \vdash prog_{sbgp}(i) : \{i, y_b, y_e\}.\varphi_{I1}$

(a_1) only differs from (a) in invariants for head tuples. And we have:

$\varphi_{signature}(i, t, n, m, s) = \exists n', d, m = d :: n' :: p \wedge \varphi_{link2}(p, n, d, n', t)$

$\varphi_{advertisement}(i, t, n', n, d, p, sl) = \varphi_{link2}(p, n, d, t)$

All head tuples p other than *signature* and *advertisement* have invariants:

$\varphi'_p(i, y_b, y_e) = \varphi_{link1}(p, n, d, t)$

Formulas $\varphi_{link1}(p, n, d, t)$ and $\varphi_{link2}(p, n, d, t)$ are defined as follows:

$$\varphi_{link1}(p, n, d, t) = \exists p',\ p = n :: p' \wedge (p' = \mathsf{nil} \supset \mathsf{prefix}(n, d)@(n, t)) \wedge \forall p'', m', p' = m' :: p'' \supset \mathsf{link}(n, m')@(n, t)$$

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

The first invariant $\varphi_{link1}(p, n, d, t)$ states that n is the first node on the path p, and the link from n to the next node on p exists. The second sub-invariant $\varphi_{link2}(p, n, d, n', t)$ extends the first one by including the receiving node n' as an argument, asserting that the link between n and n' also exists. These two invariants match the non-recursive conditions in the definition of goodPath.

We also prove φ_{I1} using Inv rule. Using Honest rule on (a_1) and keeping the only clause related to signature, we derive the following:

$$(a_2)\ \cdot;\cdot \vdash \forall n, \forall t, \mathsf{Honest}(n) \wedge \mathsf{signature}(n, m, s)@(n, t) \supset \exists n', d, m = d :: n' :: p \wedge \varphi_{link2}(p, n, d, n', t)$$

Now (a_2) has connected an honest node's signature to the existence of links related to it. Combining (a_2) and the axiom A_{sig} in Section 4, we can use φ_{I1} whenever a signature *sig* of a node n is properly verified in $prog_{sbgp}$. Since the verification of signature list is performed recursively, this fills in the recursive part in the definition of goodPath.

The complete proof can be found in the Coq file. One technical challenge is that in proving that invariants specified in φ_I are maintained by rules r3 and r4, the direction in which we check the signature list is different from the direction in which the route is created: we verify the signature for the longest path first. The definition of the invariant for tuple verifyPath is non-trivial. It needs to convey that part of a path p has been verified, and part of it (namely q in the formal definition below) still needs to be verified. To do so, we use an implication, stating that if the path to be verified satisfies the invariant goodPath, then the entire path satisfies the invariant goodPath, formally defined as follows:

$\varphi_{verifyPath}(t, p, q, d) = \exists l, p = l{+}{+}q \wedge \mathsf{goodPath}(t, q, d) \supset \mathsf{goodPath}(t, p, d)$

The construction of the invariant $\varphi_{verifyPath}$ and the proofs that rules r3 and r4 maintain this invariant demonstrates that rule Inv is general enough to handle complicated recursions in the program.

Given a route advertisement of a path p made by an honest node, property φ_{auth1} only associates each honest node n in p to the existence of links to its neighbors. S-BGP satisfies a stronger property that additionally associates the generated route by each honest node in p to the sub-path it heads in p.

We define a stronger property $\mathsf{goodPath2}(t, p, d)$ below. The meaning of the variables remains the same as before.

$$\frac{\mathsf{Honest}(n) \supset \exists t', t' \leqslant t \wedge \mathsf{prefix}(n, d)@(n, t')}{\mathsf{goodPath2}(t, n :: nil, d)}$$

$$\frac{\mathsf{Honest}(n) \supset \exists t', c, s, t' \leqslant t \wedge \mathsf{link}(n, n')@(n, t') \wedge \mathsf{route}(n, d, c, n :: nil, sl)@(n, t') \quad \mathsf{goodPath2}(t, n :: nil, d)}{\mathsf{goodPath2}(t, n' :: n :: nil, d)}$$

$$\frac{\begin{array}{l}\mathsf{Honest}(n) \supset \exists t', t' \leqslant t \wedge \mathsf{link}(n, n')@(n, t') \wedge \\ \qquad \exists t'', c, s, t'' \leqslant t \wedge \mathsf{link}(n, n'')@(n, t'') \wedge \\ \qquad \mathsf{route}(n, d, c, n :: n'' :: p'', sl)@(n, t') \\ \mathsf{goodPath2}(t, n :: n'' :: p'', d)\end{array}}{\mathsf{goodPath2}(t, n' :: n :: n'' :: p'', d)}$$

Compare the above definition with the one for $\mathsf{goodPath}$, the last two rules additionally assert the existence of a route tuple. The predicate $\mathsf{route}(n, d, c, n :: p', sl)@(n, t')$ states that node n generates a route tuple for path $n :: p'$ at time t', and that sl is the signature list that authenticates the path $n :: p'$. This property ensures that an attacker cannot use n's route advertisement for another path p', which happens to share the two links that n connects to, to fake a route advertisement. For instance, $p = n1 :: n :: n2 :: p1$ and $p' = n1 :: n :: n2 :: p2$ and $p1 \neq p2$. If however, a protocol only requires a node n to sign the links to its neighbors, this would have been a valid attack.

5.2 SCION

SCION [28] is a clean-slate design of Internet architecture that offers more flexible route selection and failure isolation, in addition to route authenticity. In SCION, autonomous domains (AD) – networks under the same administration responsibility, such as a country or company – are grouped into a trust domain (TD). TD core, typically a top-tier ISP in each TD, provides routing service inside and across the border of TD. A TD core periodically generates path construction beacons to its customer ADs to initiate the process of path construction. Each endpoint AD, upon receiving a beacon, (1) verifies the information inside the beacon, (2) if it is one of the path along which this AD is willing to forward packets, attaches itself to the end of the received path, constructing a new beacon b, and (3) forwards b to its customer ADs. After receiving a set of beacons, an endpoint AD, if not part of TD core, selects k paths and uploads them to the TD core, thus finishing path construction. When later an AD n intends to send a packet to n', it first queries the TD core for the paths that n' has uploaded, and then constructs a forwarding path based on the query result. In this way, each end AD n' has a say in the choice of forwarding path.

Path construction beacon plays an important role in SCION's routing. A beacon is a sequence of objects. Each object comprises four fields: an interface field, a time field, an opaque field and a signature. The interface field contains a list of ADs, representing the path. It also includes each AD's interfaces to neighbors, called *ingress* and *egress*. Ingress is the incoming interface and egress is the outgoing interface. They are used to eliminate forwarding table look-up during data forwarding phase. The time field registers the time when beacon is received. The opaque field, while sharing "ingress" and "egress" with interface field, adds a message authentication code (MAC) of them using AD's private key. The opaque field is only computed and forwarded, but not used, in path construction phase. It will be later included in all data packet that tries to traverse the AD. We will return to opaque field in Section 5.3. The signature is that of all the above threes fields along with the previous signature from the preceding AD. In addition, signature includes a certificate authenticating the identity of current AD.

We list the definitions of several relevant tuples in SCION encoding in Figure 10. coreTD determines whether an AD is TD core. provider and consumer state the provider/consumer relation between each pair of neighbor ADs. The beaconIni and beaconFwd are path construction beacons for beacon initialization and beacon dissemination respectively. verifiedbeacon stores a valid (verified) beacon. upPath contains information about a legitimate path to the TD core, including an interface field, an opaque field list and a time list. pathUpload contains information about a specific path to be uploaded to the TD core.

coreTD$(@n, c, td, ctf)$	c is the core of TD td with certificate ctf attesting to that fact
provider$(@n, m, ig)$	m is n's provider, with traffic into n through interface ig
customer$(@n, m, eg)$	m is n's customer, with traffic out of n through interface eg
beaconIni$(@m, n, td,$	itf, containing a path, is initialized by n and sent to m.
$itf, tl, ol, sl, sg)$	tl is a list of time stamps,
	ol is a list of opaque fields, whose meaning is not relevant here.
	sl is list of signatures for route attestation.
	sg is a signature for certain global information,
	which is not relevant here.
verifiedBeacon$(@n, td,$	itf is the stored interface fields from n to the TD core in td.
$itf, tl, ol, sl, sg)$	Rest of the fields have the same meaning as those in beaconIni
beaconFwd$(@m, n, td,$	itf is forwarded to m with corresponding signature list sl
$itf, tl, ol, sl, sg)$.	Rest of the fields have the same meaning as those
	in beaconIni
upPath$(@n, td, itf,$	$opqU$ is a list of opaque fields indicating a path.
$opqU, tl)$	Rest of the fields have the same meaning as those in beaconIni.
pathUpload$(@m, n,$	src is the node (AD) who initiated the path upload process.
$src, c, itf, opqD,$	c is TD core of an implicit TD.
$opqU, pt)$	$opqD$ is the opaque fields uploaded.
	pt indicates the next opaque field in $opqU$ to be checked.
	itf and $opqU$ have the same meaning as those in upPath.

Fig. 10. Tuples for SCION

$$\frac{\mathsf{coreTD}(ad, c, td, ctf)@(ad, t) \quad \mathsf{Honest}(c) \supset \exists t', t' \leqslant t \wedge \mathsf{customer}(c, n, ceg)@(c, t')}{\mathsf{goodInfo}(t, td, ad, nil, c :: ceg :: n :: nil)}$$

$$\frac{\begin{array}{l}\mathsf{coreTD}(ad, c, td, ctf)@(ad, t) \\ \mathsf{Honest}(n) \supset \exists t', t' \leqslant t \wedge \mathsf{provider}(n, c, nig)@(n, t') \wedge \mathsf{customer}(n, m, neg)@(n, t') \\ \qquad \wedge \exists td', tl, ol, sg, s, \mathsf{verifiedBeacon}(n, td', c :: ceg :: n :: nig :: nil, tl, \\ \qquad\qquad ol, s :: nil)@(n, t') \\ \mathsf{goodInfo}(t, td, ad, nil, (c :: ceg :: n :: nil))\end{array}}{\mathsf{goodInfo}(t, td, ad, s :: nil, c :: ceg :: n :: nig :: neg :: m :: nil)}$$

$$\frac{\begin{array}{l}\mathsf{Honest}(n) \supset \exists t', t' \leqslant t \wedge \mathsf{provider}(n, h, nig)@(n, t') \wedge \mathsf{customer}(n, m, meg)@(n, t') \\ \qquad \wedge \exists td', tl, ol, sg, s, sl, \mathsf{verifiedBeacon}(n, td', p' {+}{+} h :: hig :: heg :: n :: nig, \\ \qquad\qquad tl, ol, s :: sl)@(n, t'). \\ \mathsf{goodInfo}(t, td, ad, sl, p' + +h :: hig :: heg :: n :: nil)\end{array}}{\mathsf{goodInfo}(t, td, n, s :: sl, \quad p' {+}{+} h :: hig :: heg :: n :: nig :: neg :: m :: nil)}$$

Fig. 11. Definitions of goodInfo

SCION also satisfies similar route authenticity properties as S-BGP. Each path in SCION is composed of two parts: up path and down path. We only prove the properties for the up paths. The proof for the down paths can be obtained by switching the role of provider and customer. The definition of route authenticity on up path, denoted φ_{authS}, is very similar to φ_{auth2} of S-BGP. Tuples provider and customer in SCION are counterparts of the link tuple in S-BGP, and beaconIni and beaconFwd correspond to advertisement.

$$\begin{array}{l}\varphi_{authS} = \forall n, m, t, td, itf, tl, ol, sl, sg, \\ \qquad \mathsf{honest}(n) \wedge \\ \qquad (\mathsf{beaconIni}(@m, n, td, itf, tl, ol, sl, sg)@(n, t) \\ \qquad \vee \mathsf{beaconFwd}(@m, n, td, itf, tl, ol, sl, sg)@(n, t)) \\ \qquad \supset \mathsf{goodInfo}(t, td, n, sl, itf)\end{array}$$

Formula φ_{authS} asserts a property $\mathsf{goodInfo}(t, td, n, sl, itf)$ on any beacon tuple generated by node n, which is either a TD core or an ordinary AD. Predicate $\mathsf{goodInfo}(t, td, n, sl, itf)$ takes five arguments: t represents the time, td is the identity of the TD that the path lies in, n is the node that verifies the beacon containing the interface field itf, and sl is the signature list associated with the path. $\mathsf{goodInfo}(t, td, n, sl, itf)$ makes sure that each AD in a path, which is represented by the interface field itf, does have a link to its provider and customer, as stated in the path. Also, for each AD, there always exists a verified beacon for a path up to, but excluding it (TD core does not have any providers). The definition of goodInfo is shown in Figure 11.

The definition of goodInfo considers three cases. The base case is when a TD core c initializes an interface field $c :: ceg :: n :: nig :: nil$ and sends it

to AD n. We require that c be a TD core and n be its customer. The next two cases are similar, they both require the current AD n have a link to its preceding neighbor, represented by provider, as well as one to its down stream neighbor, represented by customer. In addition, a verifiedBeacon tuple should exist, representing an authenticated route stored in database, with all signatures inside properly verified. The difference of these two cases is caused by two possible types of the preceding AD: TD core and non-TD core. The proof strategy is exactly the same as that used in proof of goodPath about S-BGP. To prove φ_{authS}, we first prove $prog_{scion}$ has an invariant property φ_I:

(b) $\cdot;\cdot \vdash prog_{scion}(n) : \{i, y_b, y_e\}.\varphi_I(i, y_b, y_e)$
where $\varphi'_I(i, y_b, y_e) = \bigwedge_{p \in hdOf(prog_{scion})} \forall t, \forall \vec{x}, y_b \leqslant t < y_e \wedge p(\vec{x})@(i, t) \supset \varphi_p(i, t, \vec{x})$.

For instance, the invariant for beaconIni and beaconFwd are as follows.
$\varphi_{beaconIni}(i, t, m, n, td, itf, tl, ol, sl, sg) = \mathsf{goodInfo}(t, td, n, sl, itf)$
$\varphi_{beaconFwd}(i, t, m, n, td, itf, tl, ol, sl, sg) = \mathsf{goodInfo}(t, td, n, sl, itf)$
(b) can be proved using Inv rule, whose premises are also verified in Coq. Applying Honest rule to (b), we can deduce $\varphi' = \forall n, t', \mathsf{Honest}(n) \supset \varphi'_I(n, t', -\infty)$, which directly implies φ_{authS}.

5.3 Comparison: S-BGP & SCION

In terms of practical route authenticity, there is little difference between what S-BGP and SCION can offer. This is not surprising, as the kind of information that S-BGP and SCION sign at path construction phase is very similar. Though both use layered-signature to protect the routing information, ASes in S-BGP only sign the path information and a layered signature in S-BGP is a list of signatures. On the other hand, ADs in SCION sign the previous signature so a layered signature in SCION a nested signature. Consider an AS n in S-BGP that signed the path p twice, generating two signatures: s and s'. An attacker, upon receiving a sequence of signatures containing s, can replace s with s' without being detected. This attack is not possible in SCION, as attackers cannot extract signatures from a nested signature.

SCION provide stronger security guarantee than S-BGP in the data forwarding phase. SCION, enables an AD to verify its willingness to carry traffic through specific interfaces. SCION attaches each data packet with a list of opaque fields. These opaque fields are extracted from beacons received during path construction phase, which are MACs of an AD's ingress and egress. An AD, upon receiving a data packet, will re-compute the MAC of intended ingress and egress, along with opaque field of previous neighbor. This MAC is compared with the one contained in the opaque field embedded in the packet. If they are the same, the AD knows that it has agreed to receiving/sending packets from/to its neighbors. Otherwise, it drops the data packet. The formal definition of data path authenticity in SCION can be expressed as φ_{authD}.

$\varphi_{authD} = \forall m, n, t, src, core, itf, opqD, opqU, pt,$

$$\frac{pt = 0}{\mathsf{goodFwdPath}(t, n, opqU, pt)}$$

$$\frac{\begin{array}{l}\mathsf{Honest}(n) \supset \exists t', m, td, ctf, t' \leqslant t \\ \qquad\qquad \wedge\ \mathsf{coreTD}(n, n, td, ctf)@(n, t') \\ \qquad\qquad \wedge\ \mathsf{customer}(n, m, ceg)@(n, t')\end{array}}{\begin{array}{l}\mathsf{goodFwdPath}(t, n, opq'{+}{+}[ceg :: mac :: nil] :: nil, \\ \qquad\qquad\qquad\qquad length(opq'{+}{+}[ceg :: mac :: nil] :: nil))\end{array}}$$

$$\frac{\begin{array}{l}0 < pt \wedge pt < length(opq'{+}{+}[nig :: neg :: mac' :: nil] :: [ceg :: mac :: nil] :: nil \wedge \\ \mathsf{Honest}(n) \supset \exists t', h, m, t' \leqslant t \\ \qquad\qquad \wedge\ \mathsf{provider}(n, h, nig)@(n, t') \\ \qquad\qquad \wedge\ \mathsf{customer}(n, m, meg)@(n, t') \\ \qquad\qquad \wedge\ \exists td', tl, sg, sl, \mathsf{verifiedBeacon}(n, td', h :: ceg :: n :: nig, \\ \qquad\qquad\qquad\qquad\qquad\qquad tl, [ceg :: mac :: nil] :: nil, sl, sg)@(n, t')\end{array}}{\mathsf{goodFwdPath}(t, n, opq'{+}{+}[nig :: neg :: mac' :: nil] :: [ceg :: mac :: nil] :: nil, pt)}$$

$$\frac{\begin{array}{l}0 < pt \wedge \\ pt < length(opq'{+}{+}[nig :: neg :: mac' :: nil] :: [hig :: heg :: mac :: nil]{+}{+}opq'') \wedge \\ \mathsf{Honest}(n) \supset \exists t', h, m, t' \leqslant t \\ \qquad\qquad \wedge\ \mathsf{provider}(n, h, nig)@(n, t') \\ \qquad\qquad \wedge\ \mathsf{customer}(n, m, meg)@(n, t') \\ \qquad\qquad \wedge\ \exists td', tl, sg, sl, p', p'', \mathsf{verifiedBeacon}(n, td', p'{+}{+}h :: hig :: heg :: n :: nig, \\ \qquad\qquad\qquad\qquad\qquad\qquad tl, p''{+}{+}[hig :: heg :: mac :: nil] :: nil, sl, sg)@(n, t')\end{array}}{\mathsf{goodFwdPath}(t, n, opq'{+}{+}[nig :: neg :: mac' :: nil] :: [hig :: heg :: mac :: nil]{+}{+}opq'', pt)}$$

Fig. 12. Definitions of goodFwdPath

$$\begin{array}{l}\mathsf{honest}(n) \wedge \\ \mathsf{pathUpload}(@m, n, src, core, itf, opqD, opqU, pt)@(n, t) \supset \\ \mathsf{goodFwdPath}(t, n, opqU, pt)\end{array}$$

Formula φ_{authD} asserts property $\mathsf{goodFwdPath}(t, n, opqU, pt)$ on any tuple pathUpload sent by a customer AD to its provider. There are four arguments in $\mathsf{goodFwdPath}(t, n, opqU, pt)$: t is the time. n is the node who sent out pathUpload tuple. $opqU$ is a list of opaque fields for forwarding. pt is a pointer to $opqU$. Except time t, all arguments in $\mathsf{goodFwdPath}(t, n, opqU, pt)$ are the same as those in pathUpload, whose arguments are described in Figure 10. $\mathsf{goodFwdPath}(t, n, opqU, pt)$ states that whenever an AD receives a packet, it should connect to its provider and customer, as indicated by the opaque field in the packet. In addition, it must have verified a beacon with a path containing this neighboring relationship. The property is also an invariant for other tuples such as pathLookup and upMessage.

The definition of $\mathsf{goodFwdPath}(t, n, opqU, pt)$ is given in Figure 12. There are four cases. The base case is when pt is 0. Since pt points to the next opaque field to be checked, nothing is currently verified. In this case goodFwdPath holds trivially. If pt is equal to the length of opaque field list, meaning all opaque fields

have been verified already, then based on SCION specification, the last opaque field should be that of the TD core. Being a TD core requires a certificate (coreTD), and a neighbor customer along the path (customer). When pt does not point to the head or the tail of the opaque field list, node n should have a neighbor provider(provider), and a neighbor customer(customer). It must also have received and processed a verifiedBeacon during path construction. The last two cases both cover this scenario, with the only difference being that the node n's provider might be a TD core.

SCION uses MAC for integrity check during data forwarding, so we use the following axiom about about MAC. It states that if message msg's MAC, computed by n with n''s private key k, is m and node n' is honest, then n' must have generated such a mac tuple at an earlier time t'.

$$\begin{aligned} A_{mac} = \forall & msg, m, k, n, n', t, \\ & \mathsf{verifyMac}(msg, m, k)@(n, t) \wedge \mathsf{privateKeys}(n, n', k)@(n, t) \wedge \\ & \mathsf{Honest}(n') \supset \exists t', t' < t \wedge \mathsf{mac}(n', msg, m)@(n', t') \end{aligned}$$

In SCION, each node should not share its own private key with other nodes. This means, for each specific MAC, only the node who generated it can verify its validity. This fact simplifies the axiom:

$$\begin{aligned} A'_{mac} = \forall & msg, m, k, n, t, \\ & \mathsf{verifyMac}(msg, m, k)@(n, t) \wedge \mathsf{privateKeys}(n, k)@(n, t) \wedge \\ & \mathsf{Honest}(n) \supset \exists t', t' < t \wedge \mathsf{mac}(n, msg, m)@(n, t') \end{aligned}$$

The rest of the proof follows the same strategy as that of goodPath and goodInfo. We prove invariant property of $prog_{scion}$ which implies φ_{authD}.

Though S-BGP does not explicitly state the process of data forwarding, we can still compare its IP-based forwarding to SCION's forwarding mechanism. Like BGP, an AS running S-BGP maintains a routing table on all BGP speaker routers that connect to peers in other domains. The routing table is an ordered collection of forwarding entries, each represented as a pair of ⟨IP prefix, next hop⟩. Upon receiving a packet, the speaker searches its routing table for IP prefix that matches the destination IP address in the IP header of the packet, and forwards the packet on the port corresponding to the next hop based on table look-up. Whenever a packet arrives, an AS that runs S-BGP will forward it to the next hop, based on forwarding table look-up. This next hop must have been authenticiated, because only after an S-BGP update message has been properly verified, will the AS insert the next hop into the forwarding table.

However, SCION provides stronger security guarantees over S-BGP in terms of last hop of the packet. An AS running S-BGP has no way of detecting whether a received packet is from legitimate neighbor ASes – those neighbor ASes that have received an update message advertising the path. Imagine that an AS n has two neighbor ASes, m and m'. n knows a route to an IP prefix p and is only willing to advertise the route to m. Ideally, any packet from m' through n to p should be rejected by n. However, this may not happen in practice. As long as its IP destination is p, a packet will be forwarded by n, regardless of whether

it is from m or m'. On the other hand, SCION routers are able to discard such packets by verifying the MAC, since the MAC of the ingress and egress cannot be forged.

6 Related Work

Cryptographic Protocol Analysis. The analysis of cryptographic protocols [11, 25, 16, 23, 13, 6, 4, 14] has been an active area of research. Compared with cryptographic protocols, secure routing protocols have to deal with arbitrary network topologies and the programs of 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 [11, 14]. 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.

Networking Protocol Verification. Recently, several papers have investigated the verification of route authenticity properties on specific wireless routing protocols for mobile networks [2, 3, 10]. 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, 15, 12]. 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 a 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 [20]. In addition, as an alternative to Coq, we are also exploring the use of automated first-order logic theorem provers to automate our proofs.

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. CNET: How pakistan knocked youtube offline, `http://news.cnet.com/8301-10784_3-9878655-7.html`
10. Cortier, V., Degrieck, J., Delaune, S.: Analysing routing protocols: four nodes topologies are sufficient. In: Proceedings of POST (2012)
11. Datta, A., Derek, A., Mitchell, J.C., Roy, A.: Protocol Composition Logic (PCL). Electronic Notes in Theoretical Computer Science 172, 311–358 (2007)
12. Engler, D., Musuvathi, M.: Model-checking large network protocol implementations. In: Proceedings of NSDI (2004)
13. Escobar, S., Meadows, C., Meseguer, J.: A rewriting-based inference system for the NRL protocol analyzer: grammar generation. In: Proceedings of FMSE (2005)
14. Garg, D., Franklin, J., Kaynar, D., Datta, A.: Compositional system security with interface-confined adversaries. ENTCS 265, 49–71 (September 2010)
15. Goodloe, A., Gunter, C.A., Stehr, M.O.: Formal prototyping in early stages of protocol design. In: Proceedings of ACM WITS (2005)
16. 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)
17. 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)
18. 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)
19. 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)
20. Naous, J., Walfish, M., Nicolosi, A., Mazieres, D., Miller, M., Seehra, A.: Verifying and enforcing network paths with ICING. In: Proceedings of CoNEXT (2011)

21. Nigam, V., Jia, L., Loo, B.T., Scedrov, A.: Maintaining distributed logic programs incrementally. In: Proceedings of PPDP (2011)
22. 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
23. Paulson, L.C.: Mechanized proofs for a recursive authentication protocol. In: Proceedings of CSFW (1997)
24. RapidNet: A Declarative Toolkit for Rapid Network Simulation and Experimentation: http://netdb.cis.upenn.edu/rapidnet/
25. Roy, A., Datta, A., Derek, A., Mitchell, J.C., Jean-Pierre, S.: Secrecy analysis in protocol composition logic. In: Proceedings of ESORICS (2007)
26. Wan, T., Kranakis, E., Oorschot, P.C.: Pretty secure BGP (psBGP). In: Proceedings of 12th NDSS (2005)
27. White, R.: Securing bgp through secure origin BGP (soBGP). The Internet Protocol Journal 6(3), 15–22 (2003)
28. 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 Oakland S&P (2011)

A First-order logic rules

The syntax of the logic formulas is shown below.

$$
\begin{array}{lll}
\textit{Atoms} & A ::= P(\vec{t})@(\iota,\tau) \mid \mathsf{send}(\iota, \mathsf{tp}(P,\iota',\vec{t}))@\tau \mid \mathsf{recv}(\iota, \mathsf{tp}(P,\vec{t}))@\tau \\
& \quad \mid \mathsf{honest}(\iota, \mathit{prog}, \tau) \mid t_1 \ \mathit{bop}\ t_2 \\
\textit{Formulas} & \varphi ::= \top \mid \bot \mid A \mid \varphi_1 \wedge \varphi_2 \mid \varphi_1 \vee \varphi_2 \mid \varphi_1 \supset \varphi_2 \mid \neg\varphi \mid \forall x.\varphi \mid \exists x.\varphi \\
\textit{Variable Ctx} & \Sigma ::= \cdot \mid \Sigma, x \qquad \textit{Logical Ctx} \quad \Gamma ::= \cdot \mid \Gamma, \varphi
\end{array}
$$

$$\frac{\Sigma;\Gamma \vdash \varphi \qquad \Sigma;\Gamma,\varphi \vdash \varphi'}{\Sigma;\Gamma \vdash \varphi'}\ \textsc{Cut} \qquad \frac{\varphi \in \Gamma}{\Sigma;\Gamma \vdash \varphi}\ \textsc{Init} \qquad \frac{\Sigma;\Gamma,\varphi \vdash \cdot}{\Sigma;\Gamma \vdash \neg\varphi}\ \neg\mathrm{I}$$

$$\frac{\Sigma;\Gamma \vdash \neg\varphi}{\Sigma;\Gamma,\varphi \vdash \cdot}\ \neg\mathrm{E} \qquad \frac{\Sigma;\Gamma \vdash \varphi_1 \qquad \Sigma;\Gamma \vdash \varphi_2}{\Sigma;\Gamma \vdash \varphi_1 \wedge \varphi_2}\ \wedge\mathrm{I}$$

$$\frac{i \in [1,2], \Sigma;\Gamma \vdash \varphi_1 \wedge \varphi_2}{\Sigma;\Gamma \vdash \varphi_i}\ \wedge\mathrm{E} \qquad \frac{i \in [1,2], \Sigma;\Gamma \vdash \varphi_i}{\Sigma;\Gamma \vdash \varphi_1 \vee \varphi_2}\ \vee\mathrm{I}$$

$$\frac{\Sigma;\Gamma \vdash \varphi_1 \vee \varphi_2 \qquad \Sigma;\Gamma,\varphi_1 \vdash \varphi \qquad \Sigma;\Gamma,\varphi_2 \vdash \varphi}{\Sigma;\Gamma \vdash \varphi}\ \vee\mathrm{E}$$

$$\frac{\Sigma,x;\Gamma \vdash \varphi}{\Sigma;\Gamma \vdash \forall x.\varphi}\ \forall\mathrm{I} \qquad \frac{\Sigma;\Gamma \vdash \forall x.\varphi}{\Sigma;\Gamma \vdash \varphi[t/x]}\ \forall\mathrm{E} \qquad \frac{\Sigma;\Gamma \vdash \varphi[t/x]}{\Sigma;\Gamma \vdash \exists x.\varphi}\ \exists\mathrm{I}$$

$$\frac{\Sigma;\Gamma \vdash \exists x{:}\tau.\varphi \qquad \Sigma,a;\Gamma,\varphi[a/x] \vdash \varphi' \qquad a \notin \mathsf{fv}(\varphi')}{\Sigma;\Gamma \vdash \varphi'}\ \exists\mathrm{E}$$

B Proof of Theorem 1

By mutual induction on the derivation $\mathcal{E}$. The rules for standard first-order logic formulas are straightforward. We show the case when $\mathcal{E}$ ends in the HONEST rule.

Case: The last step of $\mathcal{E}$ is HONEST.

$$\mathcal{E} = \frac{\begin{array}{c}\mathcal{E}_1 :: \Sigma;\Gamma \vdash \mathit{prog}(i) : \{i, y_b, y_e\}.\varphi(i, y_b, y_e) \\ \mathcal{E}_2 :: \Sigma;\Gamma \vdash \mathsf{honest}(\iota, \mathit{prog}(\iota), t)\end{array}}{\Sigma;\Gamma \vdash \forall t', t' > t, \varphi(\iota, t, t')}\ \textsc{Honest}$$

Given σ, $\mathcal{T}$ s.t. $\mathcal{T} \vDash \Gamma\sigma$, by I.H. on $\mathcal{E}_1$ and $\mathcal{E}_2$

(1) $\Gamma\sigma \vDash (\mathit{prog})\sigma(i) : \{i, y_b, y_e\}.(\varphi(i, y_b, y_e))\sigma$

(2) $\mathcal{T} \vDash (\mathsf{honest}(\iota, \mathit{prog}(\iota), t))\sigma$

By (2),

(3) at time $t\sigma$, $\iota\sigma$ starts to run program $((\mathit{prog})\sigma)$

By (1) and (3), given any T s.t. $T > t\sigma$

(4) $\mathcal{T} \vDash \varphi\sigma(\iota\sigma, t\sigma, T)$

Therefore,

(5) $\mathcal{T} \vDash (\forall t', t' > t, \varphi(\iota, t, t'))\sigma$

Case: $\mathcal{E}$ ends in INV rule.

Given $\mathcal{T}$, σ such that $\mathcal{T} \vDash \Gamma\sigma$, and at time τ_b, node ι's local state is (ι, [], [], $prog(\iota)$), given any time point τ_e such that $\tau_e \geqslant \tau_b$,
let $\varphi = (\bigwedge_{p \in hdOf(prog)} \forall t, \forall \vec{x}, \tau_b \leqslant t < \tau_e \wedge p(\vec{x})@(\iota, t) \supset \varphi_p(\iota, t, \vec{x}))\sigma$
we need to show $\mathcal{T} \vDash \varphi$
By induction on the length of $\mathcal{T}$

subcase: $|\mathcal{T}| = 0$, $\mathcal{T}$ has one state and is of the form $\xrightarrow{\tau} \mathcal{C}$

By assumption (ι, [], [], $[prog]_\iota$) $\in \mathcal{C}$
Because the update list is empty, $\nexists \sigma_1$, s.t. $\mathcal{T} \vDash (p(\vec{x})@(\iota, t))\sigma\sigma_1$
Therefore, $\mathcal{T} \vDash \varphi$ trivially.

subcase: $\mathcal{T} = \mathcal{T}' \xrightarrow{\tau} \mathcal{C}$

We examine all possible steps allowed by the operational semantics.
To show the conjunction holds, we show all clauses in the conjuction are true by construct a generic proof for one clause.

case: DEQUEUE is the last step.

Given a substitution σ_1 for t and $\vec{x}$ s.t. $\mathcal{T} \vDash (\tau_b \leqslant t < \tau_e \wedge p(\vec{x})@(\iota, t))\sigma\sigma_1$
By the definitions of semantics, and DEQUEUE merely moves messages around
(1) $(p(\vec{x}))\sigma\sigma_1$ is on trace $\mathcal{T}'$
(2) $\mathcal{T}' \vDash (\tau_b \leqslant t < \tau_e \wedge p(\vec{x})@(\iota, t))\sigma\sigma_1$
By I.H. on $\mathcal{T}'$,
(3) $\mathcal{T}' \vDash (\forall t, \forall \vec{x}, \tau_b \leqslant t < \tau_e \wedge p(\vec{x})@(\iota, t) \supset \varphi_p(\iota, t, \vec{x}))\sigma$
By (2) and (3)
(4) $\mathcal{T}' \vDash \varphi_p(\iota, t, \vec{x})\sigma\sigma_1$
By φ_p is closed under trace extension and (4),
$\mathcal{T} \vDash \varphi_p(\iota, t, \vec{x})\sigma\sigma_1$
Therefore, $\mathcal{T} \vDash \varphi$ by taking the conjunction of all the results for such p's.

case: NODESTEP is the last step. Similar to the previous case, we examine every tuple p generated by *prog* to show $\mathcal{T} \vDash \varphi$. When p was generated on $\mathcal{T}'$, the proof proceeds in the same way as the previous case. We focus on the cases where p is generated in the last step.
We need to show that $\mathcal{T} \vDash (\forall t, \forall \vec{x}, \tau_b \leqslant t < \tau_e \wedge p(\vec{x})@(\iota, t) \supset \varphi_p(\iota, t, \vec{x}))\sigma$
Assume the newly generated tuple is $(p(\vec{x})@(\iota, \tau_p))\sigma\sigma_1$, where $\tau_p \geqslant \tau$
We need to show that $\mathcal{T} \vDash (\varphi_p(\iota, \tau_p, \vec{x}))\sigma\sigma_1$

subcase: INIT is used

In this case, only rules with an empty body are fired ($r = h(\vec{v}) :- .$).
By expanding the the last premise of the INV rule, and $\vec{v}$ are all ground terms,
(1) $\mathcal{E}_1 :: \Sigma; \Gamma \vdash \forall i, \forall t, \varphi_h(i, t, \vec{v})$
By I.H. on $\mathcal{E}_1$
(2) $\mathcal{T} \vDash (\forall i, \forall t, \varphi_h(i, t, \vec{v}))\sigma$
By (2)
$\mathcal{T} \vDash (\varphi_h(\iota, \tau_p, \vec{y}))\sigma\sigma_1$

subcase: RULEFIRE is used.

We show one case where p is not an aggregate and one where p is.

subsubcase: INSNEW is fired

By examine the Δr rule,
(1) exists $\sigma_0 \in \rho(\Psi^\nu, \Psi, r, k, \vec{s})$ such that $(p(\vec{x})@(\iota, t))\sigma\sigma_1 = (p(\vec{v})@(\iota, t))\sigma_0$
(2) for tuples (p_j) that are derived by node ι, $(p_j(\vec{s_j}))\sigma_0 \in \Psi^\nu$ or $(p_j(\vec{s_j}))\sigma_0 \in \Psi$
By operational semantics, p_j must have been generated on $\mathcal{T}'$
(3)$\mathcal{T}' \vDash (p_j(\vec{s}_j)@(\iota, \tau_p))\sigma_0$
By I.H. on $\mathcal{T}'$ and (3), the invariant for p_j holds on $\mathcal{T}'$

(4) $\mathcal{T}' \vDash (\varphi_{p_j}(\iota, \tau_p, \vec{s}_j))\sigma_0$

By φ_p is closed under trace extension

(5) $\mathcal{T} \vDash (p_j(\vec{x}_j)@(\iota, \tau_p) \wedge \varphi_{p_j}(\iota, \tau_p, \vec{s}_j))\sigma_0$

For tuples (q_j) that are received by node ι, using similar reasoning as above

(6) $\mathcal{T} \vDash (\mathsf{recv}(i, \mathsf{tp}(q_j, \vec{s}_j))@\tau_p)\sigma_0$

(7) For constraints (a_j), $\mathcal{T} \vDash a_j\sigma_0$

By I.H. on the last premise in Inv and (5) (6) (7)

(8) $\mathcal{T} \vDash (\varphi_p(i, \tau_p, \vec{v}))\sigma_0$

By (1) and (8), $\mathcal{T} \vDash (\varphi_p(i, \tau_p, \vec{y}))\sigma\sigma_1$

subsubcase: InsAggNew is fired.

When p is an aggregated predicaete, we additionally prove that every aggregate candidate predicate p_{agg} has the same invariant as p.

That is (1) $\mathcal{T} \vDash (\forall t, \forall \vec{x}, \tau_b \leqslant t < \tau_e \wedge p_{agg}(\vec{x})@(\iota, t) \supset \varphi_p(\iota, t, \vec{x}))\sigma$

The reasoning is the same as the previous case.

We additionally show that (1) is true on the newly generated $p_{agg}(\vec{t})$.

C S-BGP encoding

```
r1 route(@N,Prefix,Cost,Path,SigList) :-
      prefixs(@N,Prefix), List := f_empty(), Cost := 0,
      Path := f_prepend(N,List), SigList := f_empty().

r2 bestRoute(@N,Prefix,a_MIN<Cost>,Path,SigList) :-
      route(@N,Prefix,Cost,Path,SigList).

r3 verifyPath(@N,Neighbor,Prefix,PathToVerify,SigList,OrigPath,OrigSigList) :-
      advertisements(@N,Neighbor,Prefix, ReceivedPath,SigList),
      link(@N,Neighbor),
      PathToVerify := f_prepend(N,ReceivedPath),
      OrigPath := PathToVerify, OrigSigList := SigList,
      f_member(ReceivedPath,N) == 0,
      Neighbor == f_first(ReceivedPath).

r4 verifyPath(@N,Neighbor,Prefix,PathTemp,SigList1,OrigPath,OrigSigList) :-
      verifyPath(@N,Neighbor,Prefix,PathToVerify,SigList,OrigPath,OrigSigList),
      f_size(SigList) > 0, f_size(PathToVerify) > 1,
      PathTemp := f_removeFirst(PathToVerify),
      Node2 := f_first(PathTemp), publicKeys(@N,Node2,PubKey),
      SigInfo := f_first(SigList),
      InfoToVerify := f_prepend(Prefix,PathToVerify),
      f_verify(InfoToVerify,SigInfo,PubKey) == 1,
      SigList1 := f_removeFirst(SigList).

r5 route(@N,Prefix,Cost,OrigPath,OrigSigList) :-
      verifyPath(@N,Node,Prefix,PathToVerify,SigList,OrigPath,OrigSigList),
      f_size(SigList) == 0, f_size(PathToVerify) == 1,
      Cost:= f_size(OrigPath) - 1.

r6 signature(@N,InfoToSign,Sig) :-
      bestRoute(@N,Prefix,Cost,BestPath,SigList),
      link(@N,Neighbor), privateKeys(@N,PrivateKey),
      PathToSign := f_prepend(Neighbor,BestPath),
      InfoToSign := f_prepend(Prefix,PathToSign),
      Sig := f_sign(InfoToSign,PrivateKey).
```

```
r7 advertisements(@Neighbor,N,Prefix,BestPath, NewSigList) :-
      bestRoute(@N,Prefix,Cost,BestPath,SigList),
      link(@N,Neighbor),
      PathToSign := f_prepend(Neighbor,BestPath),
      InfoToSign == f_prepend(Prefix,PathToSign),
      signature(@N,InfoToSign,Sig),
      NewSigList := f_prepend(Sig,SigList).
```