# A PROGRAM LOGIC FOR VERIFYING SECURE ROUTING PROTOCOLS

CHEN CHEN, LIMIN JIA, HAO XU, CHENG LUO, WENCHAO ZHOU, AND BOON THAU LOO

University of Pennsylvania
*e-mail address*: chenche@cis.upenn.edu

Carnegie Mellon University
*e-mail address*: liminjia@cmu.edu

University of Pennsylvania
*e-mail address*: haoxu@cis.upenn.edu

University of Pennsylvania

Georgetown University
*e-mail address*: wzhou@cs.georgetown.edu

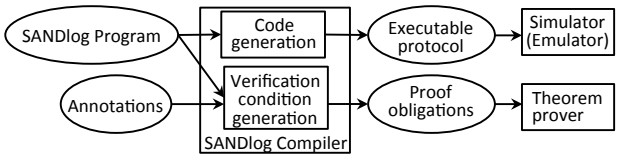
University of Pennsylvania
*e-mail address*: boonloo@cis.upenn.edu

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 encoded several proposed secure routing mechanisms in SANDLog, verified variants of path authenticity properties by manually discharging the generated verification conditions in Coq, and generated executable code based on SANDLog specification and ran the code in simulation.

*2012 ACM CCS:* [**Theory of computation**]: Logic—Logic and verification; [**Security and privacy**]: Network security—Security protocols.

*Key words and phrases:* Declarative networking; Program logic; Routing protocols.

LOGICAL METHODS IN COMPUTER SCIENCE DOI:10.2168/LMCS-???




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

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

## 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, 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, 25].

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) [18], ps-BGP [31], so-BGP [32]), to "clean-slate" Internet architectural redesigns such as SCION [33] and ICING [22]. However, *none* of the proposals formally analyzed their security properties. These protocols are implemented from scratch, evaluated primarily experimentally, and their security properties shown via informal reasoning.

Existing protocol analysis tools [6, 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) [20] with cryptographic libraries to provide compact encoding of secure routing protocols. We call our language SANDLog (stands for *Secure and Authenticated Network DataLog*). We develop a program logic for reasoning about SANDLog programs that execute in an adversarial environment. The properties proved on a SANDLog program hold even when the program interact with potentially malicious programs in the network.

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 [27]. The compiler is able to translate our SANDLog specification into executable code, which is amenable to implementation and evaluation.

We choose to use a declarative language as our specification language for two reasons. First, it has been shown that declarative languages such as NDLog can specify a variety of network protocols concisely [20]. Second, SANDLog is the specification language for both verification and generating low-level implementations. As a result, verification and empirical evaluation of secure routing protocols can be carried out in a 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).

Compared to our conference paper [8] in FORTE 2014, we have added the new case study of SCION, a clean-slate Internet architecture for inter-domain routing. We encode SCION in SANDLog, simulate the code in RapidNet, and verify variants of route authenticity properties. We also provide a comparison between S-BGP and SCION. It shows that SCION's security guarantee in routing is similar to S-BGP, as they both use layered signatures to protect advertised path from being tampered with by an attacker. SCION, however, enforces stronger security properties during data forwarding, enabling an AS to authenticate an upstream neighbor. On the other hand, S-BGP does not provide any guarantee regarding data forwarding, which means an AS could forward packets coming from any neighbor. SANDLog specification and formal verification of both solutions can be found online (`http://netdb.cis.upenn.edu/secure_routing/`.)

## 2. SANDLog

We specify secure routing protocols in a distributed declarative programming language called SANDLog. SANDLog is an extension to Network Datalog (NDLog) [20], which is proved to be a compact and clean way of specifying network routing protocols [21]. SANDLog inherits the expressiveness of NDLog, and is augmented with security primitives (e.g. asymmetric encryption) necessary for specifying secure routing protocols.

### 2.1. Syntax.

SANDLog's syntax is summarized in Figure 2. A typical SANDLog program is composed of a set of rules, each of which consists of a rule head and a rule body. The rule head is a predicate, or tuple (we use predicate and tuple interchangeably). A rule body consists of a list of body elements which are either tuples or atoms (i.e. assignments and inequality constraints). The head tuple supports aggregation functions as its arguments, whose semantics will be introduced in Section 2.2. SANDLog also defines (and implements) a number of cryptographic functions, which represent common encryption operations such as signature generation and verification. Intuitively, a SANDLog rule specifies that the head tuple is derivable if all the body tuples are derivable and all the constraints represented by the body atoms are satisfied. SANDLog distinguishes between base tuples and derived tuples. Base tuples are populated upon system initialization. Rules for populating base tuples are denoted as $b$.

| | | | |
|---|---|---|---|
| *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* | $t$ | ::= | $x \mid c \mid \iota \mid f(\vec{t}) \mid f_c(\vec{t})$ |
| *Predicate* | $pred$ | ::= | $p(agH) \mid p(agB)$ |
| *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* | $agH$ | ::= | $agB \mid @\iota, ags, F_{agr}\langle x \rangle, ags$ |
| *Base tp rules* | $b$ | ::= | $p(agH).$ |
| *Program* | $prog(\iota)$ | ::= | $b_1, \cdots, b_n, r_1, \cdots, r_k$ |

Figure 2: Syntax of SANDLog

To support distributed execution, a SANDLog program *prog* is parametrized over the node it runs on. Each tuple in the program is supposed to have a location specifier, written $@\iota$, which specifies where a tuple resides and serves as the first argument of a tuple. A rule head can specify a location different from its body tuples. When such a rule is executed, the derived tuple is sent to the remote node represented by the location specifier of the head tuple. We discuss the operational semantics of SANDLog in detail in Section 2.2.

To specify security operations in secure routing protocols, our syntax definition also includes cryptographic functions. Figure 3 gives detailed explanation of these functions. Users can add additional cryptographic primitives to SANDLog based on their needs.

**An example program.** In Figure 4, we show an example program for computing the shortest path between each pair of nodes in a network. $s$ is the location parameter of the program, representing the ID of the node where the program is executing. Each node stores three kinds of tuples: $\mathsf{link}(@s, d, c)$ means that there is a direct link from $s$ to $d$ with cost $c$; $\mathsf{path}(@s, d, c, p)$ means that $p$ is a path from $s$ to $d$ with cost $c$; and $\mathsf{bestPath}(@s, d, c, p)$ states that $p$ is the lowest-cost path between $s$ and $d$. Here, link is a base tuple, whose values are determined by the concrete network topology. path and bestPath are derived tuples. Figure 4 only shows the rules common to all network nodes. Rules for initializing the base tuple link depend on the topology and are omitted from the figure.

In the program, 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

| Function | Description |
|---|---|
| f_sign_asym(*info*, *key*) | Create a signature of *info* using *key* |
| f_verify_asym(*info*, *sig*, *key*) | Verify that *sig* is the signature of *info* using *key* |
| f_mac(*info*, *key*) | Create a message authentication code of *info* using *key* |
| f_verifymac(*info*, *MAC*, *key*) | Verify *info* against *MAC* using *key* |

Figure 3: Cryptographic functions in SANDLog

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

Figure 4: A SANDLog program for computing all-pair shortest paths

($s$ and $d$) to compute the shortest path. The arguments that appear before the aggregation denotes the group-by keys.

We can construct a more secure variant of the shortest path protocol by deploying signature authentication in the rules involving cross-node communications (e.g. $sp2$). In the following rule $sp2'$, a signature $sig$ for the path becomes an additional argument to the path tuple. When node $s$ receives such a tuple, it verifies the signature of the path $\mathsf{f\_verify}(p1, sig, pk)$. When $s$ sends out a path to its neighbor, it generates a signature by assigning $sig := \mathsf{f\_sign}(p, sk)$. Here f_sign and f_verify are user-defined asymmetric cryptographic functions (e.g. RSA).

$sp2'$ $\mathsf{path}(@z, d, c, p, sig) :-$
$\mathsf{link}(@s, z, c1), \mathsf{path}(@s, d, c2, p1, sig1), c := c1 + c2, p := z::p1,$
$\mathsf{pubK}(@s, d, pk), \mathsf{f\_verify}(p1, sig1, pk) = 1, \mathsf{privK}(@s, sk), sig := \mathsf{f\_sign}(p, sk).$

To execute the program, a user provides rules for initializing base tuples. For example, if we would like to run the shortest-path program over the topology given in Figure 5, the following rules will be included in the program. Rules $rb1$ lives at node $A$, rules $rb2$ and $rb3$ live at node $B$, and rule $rb4$ lives at node $C$.

$rb1$ $\mathsf{link}(@A, B, 1).$ $\quad$ $rb3$ $\mathsf{link}(@B, C, 1).$
$rb2$ $\mathsf{link}(@B, A, 1).$ $\quad$ $rb4$ $\mathsf{link}(@C, B, 1).$

2.2. **Operational Semantics.** The operational semantics of SANDLog adopts a distributed state transition model. Each node runs a designated SANDLog program, and maintains a database of derived tuples as its local state. Nodes can communicate with each other by sending tuples over the network, which is represented as a global network queue. The evaluation of the SANDLog programs follows the PSN algorithm [19], and updates 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.

At a high-level, each node computes its local fixed-point by firing the rules on newly-derived tuples. The fixed-point computation can also be triggered when a node receives tuples from the network. When a tuple is derived, it is sent to the node specified by its location specifier. Instead of blindly computing the fixed-point, we make sure that only rules whose body tuples are updated are fired. The operational semantics also support deletion of tuples. A deletion is propagated through the rules similar to an insertion.

More formally, the constructs needed for defining the operational semantics of SANDLog are presented below.

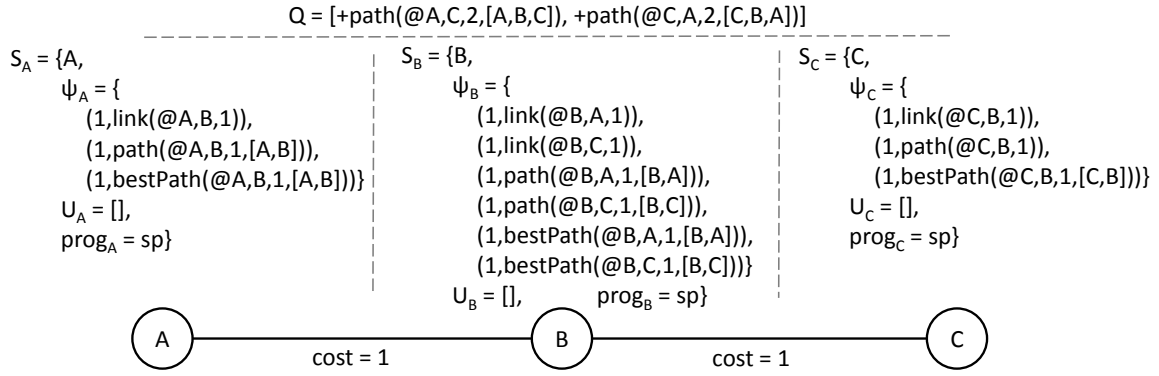



Figure 5: An Example Scenario.

$$
\begin{array}{llllllll}
\textit{Table} & \Psi & ::= & \cdot \mid \Psi, (n, P) & \textit{Network Queue} & \mathcal{Q} & ::= & \mathcal{U} \\
\textit{Update} & u & ::= & -P \mid +P & \textit{Local State} & \mathcal{S} & ::= & (\iota, \Psi, \mathcal{U}, \textit{prog}(\iota)) \\
\textit{Update List} & \mathcal{U} & ::= & [u_1, \cdots, u_n] & \textit{Configuration} & \mathcal{C} & ::= & \mathcal{Q} \rhd \mathcal{S}_1, \cdots, \mathcal{S}_n \\
\textit{Trace} & \mathcal{T} & ::= & \xrightarrow{\tau_0} \mathcal{C}_1 \xrightarrow{\tau_1} \mathcal{C}_2 \cdots \xrightarrow{\tau_n} \mathcal{C}_{n+1} & & & &
\end{array}
$$

We write $P$ to denote tuples. The database for storing all derived tuples on a node is denoted $\Psi$. Because there could be multiple derivations of the same tuple, we associate each tuple with a reference count $n$, recording the number of valid derivations for that tuple. An update is either an insertion of a tuple, denoted $+P$, or a deletion of a tuple, denoted $-P$. We write $\mathcal{U}$ to denote a list of updates. A node's local state, denoted $\mathcal{S}$, consists of the node's identifier $\iota$, the database $\Psi$, a list of unprocessed updates $\mathcal{U}$, and the program *prog* that $\iota$ runs. A configuration of the network, written $\mathcal{C}$, is composed of a network update queue $\mathcal{Q}$, and the set of the local states of all the nodes in the network. The queue $\mathcal{Q}$ models the update messages sent across the network. Finally, a trace $\mathcal{T}$ is a sequence of time-stamped (i.e. $\tau_i$) configuration transitions.

Figure 5 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. Each node's local state is displayed right above the node. For example, the local state of the node A is given by $S_A$ above it. The network queue $Q$ is presented at the top of Figure 5. In the current state, all three nodes are aware of their direct neighbors, i.e., link tuples are in their databases $\Psi_A$, $\Psi_B$ and $\Psi_C$. They have constructed paths to their neighbors (i.e., the corresponding path and bestPath tuples are stored). The current network queue $Q$ stores two tuples: +path(@A,C,2,[A,B,C]) and +path(@C,A,2,[C,B,A]), waiting to be delivered to their destinations (node $A$ and $C$ respectively). These two tuples are the result of running *sp2* at node $B$. We will explain further how configurations are updated based on the updates in the network queue when introducing the transition rules.

**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 $\mathcal{C} \rightarrow \mathcal{C}'$, where $\mathcal{C}$ and $\mathcal{C}'$ respectively represent the states of all nodes along with the network queue before and after the transition. Figure 6 defines the rules for system state transition.

- **Global state transition ($\mathcal{C} \rightarrow \mathcal{C}'$).**

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

$$\frac{\mathcal{U}_{in} = [+p_1(@\iota, \vec{t_1}), ..., +p_m(@\iota, \vec{t_m})] \qquad [p_1(@\iota, \vec{t_1}), \cdots, p_m(@\iota, \vec{t_m})] = \mathit{BaseOf}(\mathit{prog})}{(\iota, \varnothing, [], \mathit{prog}) \hookrightarrow (\iota, \varnothing, \mathcal{U}_{in}, \mathit{prog}), []} \textsc{Init}$$

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

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

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

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

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

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

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

Figure 6: Operational Semantics

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 $\iota$, and the external update list ($\mathcal{U}_{ext}$) contains only updates meant to be stored at a node different from $\iota$. In this case, it is empty. Rule RuleFire (Figure 6) 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 $\iota$ itself, the other for other nodes. The last argument for *fireRules*, $\Delta$*prog*, transforms every rule $r$ in the program *prog* into a *delta* rule, $\Delta r$, for $r$, which we explain when we discuss incremental maintenance. After $u$ is processed, the database of $\iota$ is updated with the update $u$ ($\Psi \uplus u$). The $\uplus$ operation increases (decreases) the reference count of $P$ in $\Psi$ by 1, when $u$ is an insertion (deletion) update $+P$ ($-P$). The update list in the resulting state is augmented with the new updates generated from processing $u$.

- **Fire rules** ($\mathit{fireRules}(\iota, \Psi, u, \Delta\mathit{prog}) = (\mathcal{U}_{in}, \mathcal{U}_{ext})$). Given one update, we fire rules in the program *prog* that are affected by this update. Rule EMPTY is the base case where all rules have been fired, so we directly return two empty sets. Given a program with at least one rule ($\Delta r, \Delta\mathit{prog}$), rule SEQ first fires the rule $\Delta r$, then recursively calls itself to process the rest of the rules in $\Delta\mathit{prog}$. The resulting updates are the union of the updates from firing $\Delta r$ and $\Delta\mathit{prog}$.

Given the example scenario in Figure 5, now node $A$ dequeues the update +path(@A,C,2,[A,B,C]) from the network queue $Q$ at the top of Figure 5, and puts it into the unprocessed update list $\mathcal{U}_A$ (rule DEQUEUE). Node $A$ then locally processes the update by firing all rules that are triggered by the update, and generates new updates $\mathcal{U}_{in}$ and $\mathcal{U}_{ext}$. In the resulting state, the local state of node $A$ ($\Psi_A$) is updated with path(@A,C,2,[A,B,C]), and $\mathcal{U}_A$ now includes $\mathcal{U}_{in}$. The network queue is also updated to include $\mathcal{U}_{ext}$ (rule NODESTEP).

Our operational semantics does not specify the time gaps between two consecutive reductions and, therefore, does not determine time points as associated with a concrete trace—such as $\mathcal{C} \xrightarrow{\tau} \mathcal{C}'$, where $\tau$ represents the time at which a concrete transition takes place. Instead, a trace (without time points) generated by the operational semantics—e.g., $\mathcal{C} \rightarrow \mathcal{C}'$—is an abstraction of all its corresponding annotations with time points that satisfy monotonicity. In our assertions and proofs, we use time points only to specify a relative order between events on a specific trace, so their concrete values are irrelevant.

**Incremental maintenance.** Now we explain in more detail how the database of a node is maintained incrementally by processing updates in its internal update list $\mathcal{U}_{in}$ one at a time. Following the strategy proposed in declarative networking [19], the rules in a SANDLog program are rewritten into $\Delta$ *rules*, which can efficiently generate all the updates triggered by one update. For any given rule $r$ that contains $k$ body tuples, $k$ $\Delta$ rules of the following form are generated, one for each $i \in [1, k]$.

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

$\Delta p_i$ in the body denotes the update currently being considered. $\Delta p$ in the head denotes new updates that are generated as the result of firing this rule. Here $p_i^{\nu}$ denotes a tuple of name $p_i$ in the database $\Psi$ or the internal update list $\mathcal{U}_{in}$. In comparison, $p_i$ (without $\nu$) denotes a tuple of name $p_i$ only in $\Psi$. For example, the $\Delta$ 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 $\Delta$ rules triggered by updates of the link and path relation respectively. For instance, when node $A$ processes +path(@A,C,2,[A,B,C]), only rule *sp2b* is fired. In this step, $\mathsf{path}^{\nu}$ includes the tuple path(@A,C,2,[A,B,C]), while path does not. On the other hand, $\mathsf{link}^{\nu}$ and link denote the same set of tuples, because $\mathcal{U}_{in}$ does not contain any tuple of name link. The rule evaluation then generates +path(@B,C,3,[B,A,B,C]), which will be communicated to node $B$ and further triggers rule *sp2b* at node $B$. Such update propagates until no further new tuples are generated.

**Rule Firing.** We present in Figure 7 the set of rules for firing a single $\Delta$ rule given an insertion update. We write $\Psi^{\nu}$ to denote the table resulted from updating $\Psi$ with the current update: $\Psi^{\nu} = \Psi \uplus u$.

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

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

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

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

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

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

Figure 7: Insertion rules for evaluating a single $\Delta$ rule

Rule InsExists specifies the case where the tuple to be inserted (i.e. $q_i(\vec{t})$) already exists. We do not need to further propagate the update. Rule InsNew handles the case where new updates are generated by firing rule $r$. In order to fire a rule $r$, we need to map its bodies to concrete tuples in the database or the update list. We use an auxiliary function $\rho(\Psi^\nu, \Psi, r, i, \vec{t})$ to extract the complete list of substitutions for variables in the rule. 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 $\sigma$ in that set is a general unifier of the body tuples and constraints. Formally: (1) $\vec{t} = \sigma(agB_i)$,

(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 $a$.

When multiple tuples with the same key are derived using a rule, a selection function *sel* is introduced to decide which substitution to propagate. In SANDLog 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, we can demand that each pair of nodes in the network have a unique path between them. This is equivalent to designating the first two arguments of path as its key. As a result, path(A,B,1,[A,B]) and path(A,B,2,[A,D,B]) cannot both exist in the database.

We also 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. For example, assume that path(A,B,3,[A,C,D,B]) is in $\Psi^\nu$. If we were to choose path(A,B,1,[A,B]) because it appears earlier in the update list, then *genUpd* returns $\{+$path(A,B,1,[A,B]), $-$path(A,B,3,[A,C,D,B])$\}$. We leave the definitions of *sel* and *genUpd* abstract here, as there are many possible strategies for implementing these two functions. Aside from the strategy of picking the first update in the queue (illustrated above), another possible strategy is to pick the last, as it is the freshest. Once the strategy of *sel* is fixed, *genUpd* is also fixed. However, the only relevant part to the logic we introduce later is that the substitutions used for an insertion update come from the $\rho$ function, and that the substitutions satisfy the property we defined above. In other words, our program logic can be applied to a number of different implementation of *sel* and *genUpd*.

The rest of the rules in Figure 7 deal with generating an aggregate tuple. 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 (i.e. InsAggSame and InsAggUpd) are required to handle aggregates where the new aggregate is the same as the old one or replaces the old one.

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. For example, bestpath$_{agg}$ maintains all candidate path tuples.

We also 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 additional rule to send the result after the aggregate is computed.

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 8 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.

We revisit the example in Figure 5 to illustrate how incremental maintenance is performed on the shortest-path program. Upon receiving +path(@A,C,2,[A,B,C]), $\Delta$ rule *sp2b* will be

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

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

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

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

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

Figure 8: Deletion rules for evaluating a single $\Delta$ rule

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 $\Delta$ 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 $\Delta$ rules triggered by the update +path(@A,C,2,[A,B,C]), we have $\mathcal{U}_{in}$ = {+bestPath(@A,C,2,[A,B,C])} and $\mathcal{U}_{ext}$ = {+path(@B,C,3,[B,A,B,C])}. In addition, $\mathsf{bestpath}_{agg}$, the auxiliary relation that maintains all candidate tuples for bestpath, is also updated to reflect that a new candidate tuple has been generated. It now includes bestpath(@A,C,2,[A,B,C]).

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

The operational semantics is correct if the results are the same as one where all rules reside in one node and a global fixed point is computed at each round. The proof of correctness is out of the scope of this paper. We are working on correctness definitions and proofs for variants of PSN algorithms. Our initial results for a simpler language can be found in [23]. SANDLog additionally allows aggregates, which are not included in [23]. The soundness of our logic only depends on the specific evaluation strategy implemented by the compiler, and is orthogonal to the correctness of the operational semantics. Updates to the operational semantics is likely to come in some form of additional bookkeeping in the representation of tuples, which we believe will not affect the overall structure of the program logic; as these metadata are irrelevant to the logic.

## 3. A Program Logic for SANDLog

To verify correctness of secure routing protocols encoded in SANDLog, we introduce a program logic for SANDLog. The program logic enables us to prove program invariants—that is, properties holding throughout the execution of SANDLog programs—even if the nodes running the program interact with potential attackers, whose behaviors are unpredictable. The properties of secure routing protocols that we are interested in are all safety properties and can be verified by analyzing programs' invariant properties.

**Attacker model.** We assume *connectivity-bound* network attackers, a variant of the Dolev-Yao network attacker model. The attacker can perform cryptographic operations with correct keys, such as encryption, decryption, and signature generation, but is not allowed to eavesdrop or intercept packets. This attacker model manifests itself 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 are malicious and run arbitrary SANDLog programs; (2) safety of cryptography is admitted as axioms in our proofs.

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

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

Predicate $P(\vec{t})@(\iota,\tau)$ means that tuple $P(\vec{t})$ is derived at time $\tau$ by node $\iota$. The first element in $\vec{t}$ is a location identifier $\iota'$, which may be different from $\iota$. When a tuple $P(\iota',...)$ is derived at node $\iota$, it is sent to $\iota'$. This *send* action is captured by predicate $\mathsf{send}(\iota,\mathsf{tp}(P,\iota',\vec{t}))@\tau$. Correspondingly, predicate $\mathsf{recv}(\iota,\mathsf{tp}(P,\vec{t}))@\tau$ denotes that node $\iota$ has received a tuple $P(\vec{t})$ at time $\tau$. A user could determine send and recv tuples by inspecting rules whose head tuple locates differently from body tuples. For example, the head tuple $\mathsf{path}(@z,d,c,p)$ in the rule *sp2* of the shortest-path program (Figure 4) corresponds to a tuple $\mathsf{send}(s,\mathsf{tp}(\mathit{path},z,(z,d,c,p)))@t$ in our logic. $\mathsf{honest}(\iota,\mathit{prog}(\iota),\tau)$ means that node $\iota$ starts to run program $\mathit{prog}(\iota)$ at time $\tau$. Since predicates take time points as an argument, we are effectively encoding linear temporal logic (LTL) in first-order logic [17]. The domain of the time points is the set of natural numbers. Each time point represents the number of clock ticks from the initialization of the system.

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

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

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

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

$$\frac{\Sigma;\Gamma\vdash\exists x.\varphi \qquad \Sigma,a;\Gamma,\varphi[a/x]\vdash\varphi' \qquad a \text{ is fresh}}{\Sigma;\Gamma\vdash\varphi'}\ \exists\text{E}$$

Figure 9: Rules in first-order logic.

**Logical judgments.** The logical judgments in our program logic use two contexts: context $\Sigma$, which contains all the free variables; and context $\Gamma$, which contains logical assumptions.

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

Judgment (1) states that $\varphi$ is provable given the assumptions in $\Gamma$. Judgment (2) is an assertion about SANDLog programs, i.e., a program invariant. We write $\varphi(\vec{x})$ when $\vec{x}$ are free in $\varphi$. $\varphi(\vec{t})$ denotes the resulting formula of substituting $\vec{t}$ for $\vec{x}$ in $\varphi(\vec{x})$. Recall that *prog* is parametrized over the identifier of the node it runs on. The program invariant 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 a program *prog* by a node $\iota$, starting at time $\tau_b$, satisfies $\varphi(\iota,\tau_b,\tau_e)$, for any time point $\tau_e$ later than $\tau_b$. Note that the trace could also contain threads that run malicious programs. Since $\tau_e$ is any time after $\tau_b$ (the time *prog* starts), $\varphi$ is an invariant property of *prog*.

**Inference rules.** The inference rules of our program logic include all standard first-order logic ones (e.g. Modus ponens), shown in Figure 3. Reasoning about the ordering between time points are carried out in first-order logic using theory on natural numbers (in Coq, we use Omega). We choose first-order logic because it is better supported by proof assistants (e.g. Coq).

In addition, we introduce two key rules (Figure 10) into our proof system. Rule Inv proves an invariant property of a program *prog*. The program invariant takes on a specific form as the conjunction of all the invariants of the tuples derived by *prog*, and means that if any head tuple is derived by *prog*, then its associated property should hold; 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 name of the head tuple, and $\varphi_p(i,t,\vec{x})$ is an invariant property associated with $p(\vec{x})$. For example, $p$ can be $\mathsf{path}$, and $\varphi_p(i,t,\vec{x})$ be that every link in argument *path* must have existed in the past. In the INV rule, the function *rlOf*(*prog*) returns rules generating derivation tuples for a given program, and the function $\mathsf{fv}(r)$ returns all free variables in a given rule.

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

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

$$\boxed{\Sigma;\Gamma \vdash \varphi}$$

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

Figure 10: Rules in program logic

Intuitively, the premises of Inv need to establish that *each* derivation rule's body tuples and its associative invariants together imply the invariant of the rule's head tuple. For each derivation rule $r$ in *prog*, we assume that the body of $r$ is arranged so that the first $m$ tuples (i.e. $p_1(\vec{s}_1), ..., p_m(\vec{s}_m)$) are derived by *prog*, the next $n$ tuples (i.e. $q_1(\vec{u}_1), ..., q_n(\vec{u}_n)$) are received from the network, and constraints (i.e. $a_1, ..., a_k$) constitute the rest of the body. For tuples derived by *prog* (i.e. $p_j$'s), we can safely assume that their invariants $\varphi_{p_j}$ hold at time $t$. On the other hand, properties of received tuples (i.e. $q_j$) are excluded from the premises, as in an adversarial environment, messages from the network are not trusted by default.

Each premise of the INV rule provides the strongest assumption that allows us to prove the conclusion in that premise. In most cases, arithmetic constraints are enough for proving the invariant. But in some special cases—for example, the invariant explicitly specifies the existence of a received tuple—the predicate representing the action of a tuple receipt is needed in the assumption. In other words, $\varphi_{p_j}$ is the inductive hypothesis in this inductive proof. In our case study, we frequently need to invoke the inductive hypothesis to complete the proof.

We make sure that each tuple in an SANDLog program is either derived locally or received from the network, but not both. For a program that violates this property, the user can rewrite the program by creating a copy tuple of a different name for the tuple that can be both derived locally or received from the network. For example, the path tuple in the shortest-path program in Figure 4 could be both derived locally (rule *sp1*) and received from a remote node (rule *sp2*). The user could rewrite the head tuple path in *sp2* to recvPath to differentiate it from path. In this way, the invariant property associated with the path tuple can be trusted and used in the proof of the program invariant.

$$\frac{\begin{array}{l}\Sigma;\Gamma \vdash \forall s, \forall d, \forall c, \forall t, \\ (\mathsf{link}(s,d,c)@(s,t) \wedge p = [s,d]) \supset \\ (\exists z, c', \mathsf{link}(s,z,c')@(s,t) \vee \mathsf{link}(z,s,c')@(z,t)) \\ \\ \Sigma;\Gamma \vdash \forall s, \forall d, \forall c1, \forall c2, \forall p1, \forall z, \forall t, \\ (\mathsf{link}(s,z,c1)@(s,t) \wedge \mathsf{recv}(s, \mathsf{tp}(\mathsf{path}, s, d, c2, p1))@t \wedge \\ c = c1 + c2 \wedge p = z::p1) \supset \\ (\exists z'', c'', \mathsf{link}(z,z'',c'')@(z,t) \vee \mathsf{link}(z'',z,c')@(z'',t)) \\ \\ \Sigma;\Gamma \vdash \mathsf{true}\end{array}}{\Sigma;\Gamma \vdash \varphi_{sp}}\ \textsc{Inv}$$

Figure 11: Proof of $\varphi_{sp}$

We also require that an invariant $\varphi_p$ be closed under trace extension. Formally: if $\mathcal{T} \vDash \varphi(\iota, t, \vec{s})$ and $\mathcal{T}$ is a prefix of $\mathcal{T}'$, then $\mathcal{T}' \vDash \varphi(\iota, t, \vec{s})$. For instance, the property that node $\iota$ has received a tuple $p$ before time $t$ is closed under trace extension, while the property that node $\iota$ never sends $p$ to the network is not closed under trace extension.

We do not allow invariants to be specified over base tuples. The INV rule cannot be used to derive properties of base rules (e.g., link), because the function *rlOf*() only returns rules for derivation tuples.

As an example, we use INV to prove a simple program invariant of the shortest-path program in Figure 4. The property is specified as

$\varphi_{sp} = prog(x) : \{x, y_b, y_e\}.$
$(\forall t, \forall y, \forall c, \forall pt, y_b \leqslant t < y_e \wedge$
$\mathsf{path}(x,y,c,pt)@(x,t) \supset$
$(\exists z, c', \mathsf{link}(x,z,c')@(x,t) \vee$
$\mathsf{link}(z,x,c'@(z,t))) \wedge$
$(\forall t, \forall y, \forall c, \forall pt, y_b \leqslant t < y_e \wedge$
$\mathsf{bestPath}(x,y,c,pt)@(x,t) \supset \mathsf{true}$

Intuitively, $\varphi_{sp}$ specifies an invariant property for the path tuple, which says a path tuple must imply a path tuple to/from the direct neighbor. $\varphi_{sp}$ also assigns true as the invariant property for bestPath tuples. The proof is established using INV (Figure 11). The whole proof has three premises, each corresponding to a rule in the shortest-path program in Figure 4. For example, in the second premise corresponding to *sp2*, we include the local link tuple and the received path as well as constraints in the assumption, while leaving out the invariant property of the path tuple, because a received path tuple should not be trusted in an adversarial environment.

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

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

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

$\mathcal{T} \vDash \mathsf{recv}(\iota, \mathsf{tp}(P, \vec{t}))@\tau$ 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 $\tau$, node $\iota$'s local state is $(\iota, [], [], prog\ (\iota))$

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

Figure 12: Trace-based semantics

the prescribed protocols. The invariant property of an honest node holds even when it interacts with other malicious nodes in the network, which is required by the soundness of the inference rules. We explain in more detail next.

**Soundness.** We prove the soundness of our logic with regard to the trace semantics. First, we define the trace-based semantics for our logic and judgments in Figure 12. Different from semantics of first-order logic, in our semantics, formulas are interpreted on a trace $\mathcal{T}$. We elide the rules for first-order logic connectives. A tuple $P(\vec{t})$ is derivable by node $\iota$ at time $\tau$, if $P(\vec{t})$ is either an internal update or an external update generated at a time point $\tau'$ no later than $\tau$. A node $\iota$ sends out a tuple $P(\iota', \vec{t})$ if that tuple was derived by node $\iota$. Because $\iota'$ is different from $\iota$, it is sent over the network. A *received tuple* is one that comes from the network (obtained using DeQueue). Finally, an honest node $\iota$ runs *prog* at time $\tau$, if at time $\tau$ and the local state of $\iota$ at time $\tau$ 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 $\iota$ (formally defined as the node running *prog* is one of the nodes in the configuration $\mathcal{C}$), then given any time point $\tau_e$ after $\tau_b$, the trace $\mathcal{T}$ satisfies $\varphi(\iota, \tau_b, \tau_e)$. Here, the semantic definition requires that the invariant of an honest node holds in the presence of attackers, because we examine all traces that include the honest node in their configurations. This means that those traces can contain arbitrary other nodes, some of which are malicious.

Our program logic is proven to be sound with regard to the trace semantics:

**Theorem 3.1** (Soundness)**.** (1) *If* $\Sigma; \Gamma \vdash \varphi$*, then for all grounding substitution* $\sigma$ *for* $\Sigma$*, 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* $\sigma$ *for* $\Sigma$*,* $\Gamma\sigma \vDash (prog)\sigma(i) : \{i, y_b, y_e\}.(\varphi(i, y_b, y_e))\sigma$*.*

The detailed proof of Theorem 3.1 can be found in Section A. The intuition behind the soundness proof is that the invariant properties $\varphi_p$ specified for the predicate $p$ are local properties that will not be affected by the attacker. For instance, we can specify basic arithmetic constraints of arguments derived by the honest node and the existence of base tuples. These invariants can be checked by examining the program of the honest node and are not affected by how the honest node interacts with the rest of the network. We never use any invariant of received tuples, because they could be sent from an attacker, and the attacker does not need to generate those tuples following protocols. However, we can use

the fact that those received tuples must have arrived at the honest node; otherwise, the rule will not fire. In other words, we trust the runtime of an honest node.

**Discussion.** Our program logic enables us to prove invariant properties that hold even in adversarial environment. The network trace $\mathcal{T}$ in Theorem 3.1 could involve attacker threads who run arbitrary malicious programs. For example, a trace may contain attacker threads who keep propagating invalid route advertisement for a non-existent destination. Properties proved with our logic, however, still hold in such traces. The key observation here is that in the rule INV, the correctness of the program property does not rely on received tuples, which could have been manipulated by malicious attackers. This guarantee is further validated by our logic semantics and soundness, where we demand that a proved conclusion should hold in *any* trace.

Our program logic could possibly prove false program invariants for SANDLog programs only generating empty network traces. A such example program is as follows:

$$r1\ \mathsf{p}(@a) :- \mathsf{q}(@a).$$
$$r2\ \mathsf{q}(@a) :- \mathsf{p}(@a).$$

A user could assign $\mathsf{false}$ to both $\mathsf{p}$ and $\mathsf{q}$, and prove the program invariant with the rule INV. However, this program, when executing in bottom-up evaluation, produces an empty set of tuples. The INV rule is still sound in this case as there is no trace that generates tuples $\mathsf{p}$ and $\mathsf{q}$. Instead, a SANDLog program should have rules of the form "$\mathsf{p}$ :-" to generate base tuples. If a false program invariant is given for such a program, the user is obliged to prove $\vdash \mathsf{false}$ in the logic, which is impossible.

## 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 [27], 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 10). Rule *sp1* of example program in Section 2.1, for instance, corresponds to the following lemma:

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

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

**Axioms.** For each invariant $\varphi_p$ of a rule head $p$, VCGen produces an axiom of the form: $\forall i, t, \vec{x}, \mathsf{Honest}(i) \supset p(\vec{x})@(i,t) \supset \varphi_p(i, \vec{x})$. These axioms are conclusions of the HONEST rule after invariants are verified. Soundness of these axioms is backed by Theorem 3.1. Since we always assume that the program starts at time $-\infty$, the condition that $t > -\infty$ is always true, thus omitted.

## 5. Case Studies

In this section, we investigate two proposed secure routing solutions: S-BGP (Section 5.1) and SCION (Section 5.2). We encode both solutions in SANDLog and prove that they preserve route authenticity, a key property stating that route announcements are trustworthy. Our case studies not only demonstrate the effectiveness of our program logic, but provide a formal proof supporting the informal guarantees given by the solution designers. Interested readers can find SANDLog specification and formal verification of both solutions online (`http://netdb.cis.upenn.edu/secure_routing/`.)

### 5.1. S-BGP.

Secure Border Gateway Protocol (S-BGP) [30] is a comprehensive solution that aims to eliminate security vulnerabilities of BGP, while maintaining compatibility with original BGP specifications. S-BGP requires that each node sign the route information (route attestation) using asymmetric encryption (e.g. RSA [28]) before advertising the message to its neighbor. The route information is supposed to include the destination address (represented by an IP prefix), the known path to the destination, and the identifier of the neighbor to whom the route information will be sent. The sender also attaches a signature list to the route information, containing all signatures received from the previous neighbors. A node receiving the route attestation would not trust the routing information unless all signatures inside are properly checked.

**Encoding.** Figure 13 presents our encoding of S-BGP in SANDLog. The meaning of tuples in the program can be found in Figure 14. In rule r1 of Figure 13, when a node N receives an advertise tuple from its neighbor Nb, it generates a verifyPath tuple, which serves as an entry point for recursive signature verification. In rule 2, N recursively verifies all signatures in Osl, which stands for "original signature list". Sl in verifyPath is a sub-list of Osl, representing the signatures that have not been checked. When all signatures have been verified — this is ensured by "$f\_size(Sl) == 0$" in rule 3 — N accepts the route and stores the path as a route tuple in the local database. Rule 4 also allows $N$ to generate a route tuple storing the path to its self-owned IP prefixes (i.e. prefix(@N, Pfx)). Given a specific destination Pfx, in rule 5, $N$ aggregates all route tuples storing paths to Pfx, and computes a bestPath tuple for the shortest path. The bestPath is intended to be propagated to downstream ASes. Before propagation, however, S-BGP requires $N$ to sign the path information. This is captured in rule 6, where $N$ uses its private key (i.e. privateKeys(@N, PriK)) to generate a signature based on the selected bestPath tuple. Finally, in rule 7, $N$ embeds the routing information (i.e. bestPath) along with its signature (i.e. signature) into a new route advertisement (i.e. advertise), and propagates the message to its neighbors.

```
r1 verifyPath(@N,Nb,Pfx,Pvf,
               Sl,OrigP,Osl) :-
    advertise(@N,Nb,Pfx,RcvP,Sl),
    link(@N,Nb),
    Pvf := f_prepend(N,RcvP),
    OrigP := Pvf,
    Osl := Sl,
    f_member(RcvP,N) == 0,
    Nb == f_first(RcvP).
r2 verifyPath(@N,Nb,Pfx,PTemp,
                Sl1,OrigP,Osl) :-
    verifyPath(@N,Nb,Pfx,Pvf,
                Sl,OrigP,Osl),
    publicKeys(@N,Nd,PubK),
    f_size(Sl) > 0,
    f_size(Pvf) > 1,
    PTemp := f_removeFirst(Pvf),
    Nd := f_first(PTemp),
    SigM := f_first(Sl),
    MsgV := f_prepend(Pfx,Pvf),
    f_verify(MsgV,SigM,PubK) == 1,
    Sl1 := f_removeFirst(Sl).
r3 route(@N,Pfx,C,OrigP,Osl) :-
    verifyP(@N,Nd,Pfx,Pvf,
             Sl,OrigP,Osl),
    f_size(Sl) == 0,
    f_size(Pvf) == 1,
    C:= f_size(OrigP) - 1.
r4 route(@N,Pfx,C,P,Sl) :-
    prefixs(@N,Pfx),
    List := f_empty(),
    C := 0,
    P := f_prepend(N,List),
    Sl := f_empty().
r5 bestRoute(@N,Pfx,a_MIN<C>,P,Sl) :-
    route(@N,Pfx,C,P,Sl).
r6 signature(@N,Msg,Sig) :-
    bestRoute(@N,Pfx,C,BestP,Sl),
    privateKeys(@N,PriK),
    link(@N,Nb),
    Pts := f_prepend(Nb,BestP),
    Msg := f_prepend(Pfx,Pts),
    Sig := f_sign(Msg,PriK).
r7 advertise(@Nb,N,Pfx,BestP,NewSl) :-
    bestRoute(@N,Pfx,C,BestP,Sl),
    link(@N,Nb),
    Pts := f_prepend(Nb,BestP),
    Msg == f_prepend(Pfx,Pts),
    signature(@N,Msg,Sig),
    NewSl := f_prepend(Sig,Sl).
```

Figure 13: S-BGP encoding

**Property specification.** Route authenticity of S-BGP ensures that no route announcement can be tampered with by an attacker. In other words, it requires that any route announcement *accepted* by a node is authentic. We encode it as $\varphi_{auth1}$ below.

| | |
|---|---|
| $\mathsf{link}(@n, n')$ | 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 $p$. |
| $\mathsf{verifyPath}(@n, n', d, p, sl, pOrig, sOrig)$ | a path $p$ to a destination $d$ is verified against signature list $sl$. $p$ 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. |
| $\mathsf{advertise}(@n', n, d, p, sl)$ | $n$ advertises path $p$ to neighbor $n'$ with signature list $sl$. |

Figure 14: Tuples for $prog_{sbgp}$

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

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

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

Figure 15: Definitions of goodPath

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

$\varphi_{auth1}$ is a general topology-independent security property. It asserts that whenever an honest node $n$, denoted as $\mathsf{Honest}(n)$, sends out an advertise tuple to its neighbor $m$, the property $\mathsf{goodPath}(t, d, p)$ holds. $\mathsf{Honest}(n)$ means that $n$ runs S-BGP and $n$'s private key is not compromised. 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 time after the specified starting time, so using $-\infty$ gives us the most flexibility. $\mathsf{goodPath}(t, d, p)$ is recursively defined in Figure 15, which asserts that all links in the path $p$ towards the destination $d$ exist no later than $t$. Each link $(m, n)$ is represented by two tuples: $\textcircled{=}$ link(@n, m) and $\textcircled{=}$ link(@m, n). These two tuples reside on two endpoints respectively.

To be more specific, the definition of $\mathsf{goodPath}(t, d, p)$ involves three cases (Figure 15). The base case is when $p$ contains only one node. 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, but impose no constraint on $n'$'s database, because $n'$ has not received the advertisement. The last case is when the length of $p$ is larger than two; we check 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.

goodPath can serve as a template for a number of useful properties. For example, by substituting link ($n$,$n''$) with announce_link ($n$, $n''$), we are able to express whether a node is willing to let its neighbor know of that link. We can also require each subpath be authorized by the sender.

**Axiom of signature.** To use the authenticity property of signatures in the proof of $\varphi_{auth1}$, we include the following axiom $A_{sig}$ in the logical context $\Gamma$. This axiom states that if a signature $s$ is verified by the public key of a node $n'$, and $n'$ is honest, then $n'$ must have generated a signature tuple. Predicate $\mathsf{verify}(m, s, k)@(n, t)$ means that node $n$ verifies, using key $k$ at time $t$, that $s$ is a valid signature of message $m$.

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

**Verification.** Our goal is to prove that $\varphi_{auth1}$ is an invariant property that holds on all possible execution traces. However, directly proving $\varphi_{auth1}$ is hard, as it involves verification over all the traces. Instead, we take the indirect approach of using our program logic to prove a program invariant, which is stronger than $\varphi_{auth1}$, and, more importantly, whose validity implies the validity of $\varphi_{auth1}$. To be concrete, we show that $prog_{sbgp}$ has the following invariant property $\varphi_I$:

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

where $\varphi_I$ is defined as:

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

Every $\varphi_p$ in $\varphi_I$ denotes the invariant property associated with each head tuple in $prog_{sbgp}$, and needs to be specified by the user. Table 1 gives the invariants associated with all head tuples in the program. Especially, the invariant associated with the advertise tuple (goodPath) is the same as the conclusion of $\varphi_{auth1}$.

We prove (a) using the INV rule in Section 3, by showing that all the premises hold. The INV rule has two types of premises: (1) Premises that ensure each rule of the program maintains the invariant of its rule head; and (2) Premises that ensure all invariants for head tuples are closed under trace extension. Premises of the second type are guaranteed through manual inspection of all the invariants, thus omitted in the formal proof. In terms of premises of the first type, since $prog_{sbgp}$ has seven rules, this corresponds to seven premises to be proved. For example, the premise corresponding to rule 2 is represented by ($a_0$), shown below.

| Rule | Head Tuple | Invariant |
|---|---|---|
| r1,r2 | verifyPath (N,Nb,Pfx,Pvf, Sl,OrigP,Osl)@(N,t) | $\exists l, \mathsf{Osl} = l{+}{+}\mathsf{Pvf} \wedge$ (goodPath(t,Pfx,Pvf) $\supset$ goodPath(t,Pfx,Osl)) |
| r3,r4 | route (N,Pfx,C,OrigP,Osl)@(N,t) | goodPath (t,Pfx,OrigP) |
| r5 | bestRoute (N,Pfx,C,P,Sl)@(N,t) | goodPath (t,Pfx,OrigP) |
| r6 | signature (N,Msg,Sig)@(N,t) | $\exists \mathsf{p}, \mathsf{m}, \mathsf{pfx}, \mathsf{Msg} = \mathsf{pfx} :: \mathsf{nei} :: \mathsf{p}$ |
| r7 | advertise (Nb,N,Pfx,BestP,NewSl) | goodPath (t,Pfx,Nb::BestP) |

Table 1: Tuple invariants in $\varphi_I$ for S-BGP route authenticity

$(a_0)$ $\cdot;\cdot \vdash \forall \mathsf{N}, \forall \mathsf{Nb}, \forall \mathsf{Pfx}, \forall \mathsf{Pvf}, \forall \mathsf{Sl}, \forall \mathsf{Sl1}, \forall \mathsf{OrigP}, \forall \mathsf{Osl}, \forall \mathsf{t}, \forall \mathsf{Nd},$
$\forall \mathsf{PubK}, \forall \mathsf{m}, \forall \mathsf{p}, \forall \mathsf{SigM}, \forall \mathsf{MsgV}, \forall \mathsf{PTemp}, \forall \mathsf{Osl},$
$\mathsf{verifyPath}(\mathsf{N},\mathsf{Nb},\mathsf{Pfx},\mathsf{Pvf},\mathsf{SL},\mathsf{OrigP},\mathsf{Osl})@(\mathsf{N},\mathsf{t}) \wedge$
$\exists l, \mathsf{Osl} = l{+}{+}\mathsf{Sl} \wedge$
$(\mathsf{goodPath}(\mathsf{t},\mathsf{Pfx},\mathsf{Pvf}) \supset \mathsf{goodPath}(\mathsf{t},\mathsf{Pfx},\mathsf{Osl})) \wedge$
$\mathsf{publicKeys}(\mathsf{N},\mathsf{Nd},\mathsf{PubK})@(\mathsf{N},\mathsf{t}) \wedge$
$\mathsf{length}(\mathsf{Sl}) > 0 \wedge$
$\mathsf{length}(\mathsf{Pvf}) > 0 \wedge$
$\mathsf{Pvf} = \mathsf{m} :: \mathsf{Nd} :: \mathsf{p} \wedge$
$\mathsf{PTemp} = \mathsf{Nd} :: \mathsf{p} \wedge$
$\mathsf{Sl} = \mathsf{SigM} :: \mathsf{Sl1} \wedge$
$\mathsf{MsgV} = \mathsf{Pfx} :: \mathsf{Pvf} \wedge$
$\mathsf{verify}(\mathsf{MsgV},\mathsf{SigM},\mathsf{PubK})@(\mathsf{N},\mathsf{t}) \supset$
$(\exists l, \mathsf{Osl} = l{+}{+}\mathsf{Sl1} \wedge$
$(\mathsf{goodPath}(\mathsf{t},\mathsf{Pfx},\mathsf{PTemp}) \supset \mathsf{goodPath}(\mathsf{t},\mathsf{Pfx},\mathsf{Osl})))$

Here, $(\exists l, \mathsf{Osl} = l{+}{+}\mathsf{Sl1} \wedge (\mathsf{goodPath}(\mathsf{t,Pfx,PTemp}) \supset \mathsf{goodPath}(\mathsf{t,Pfx,Osl})))$ is the invariant of rule 2's head tuple verifyPath (Figure 1). Other rule-related premises are constructed in a similar way. We prove all the premises in Coq, thus proving $(a)$.

After (a) is proved, by applying the HONEST rule to (a), we can deduce $\varphi = \forall n\ t, \mathsf{Honest}(n) \supset \varphi_I(n, t, -\infty)$. $\varphi_I$ can then be injected into the assumptions ($\Gamma$) by VCGen (as do $\varphi_{I1}$) and is safe to be used as theorem in proving other properties. Finally, $\varphi_{auth1}$ is proved by discharging $\varphi \supset \varphi_{auth1}$ in Coq with standard elimination rules.

**Proof details.** Among the others, the premise corresponding to rule 2 in the program turns out to be the most challenging one, as it involves recursion and signature verification. Recursion in rule 2 makes it hard to find the proper invariant specification for the head tuple verifyPath, as the invariant needs to maintain correctness for both the head tuple and the body tuple, which have different arguments. In our specification, we specify the invariant in a way that reversely verify the signature list by checking the signature for the longest path first. More concretely, we use an implication, stating that if the path to be verified satisfies the invariant goodPath, then the entire path satisfies the invariant goodPath (Table 1).

Another challenge in proving the invariant for verifyPath is to reason about the existence of *link* tuples at the previous nodes. We solve the problem in two steps: (1) we prove a stronger auxiliary program invariant $(a_1)$, which asserts the existence of the local *link* tuple when a node signs the path information. (2) we then use the axiom $A_{sig}$ to allow a node who verifies a signature to assure the existence of the *link* tuple at the remote node who signs the signature.

More concretely, $(a_1)$ is defined as:

$$(\mathrm{a}_1)\ \cdot;\cdot \vdash prog_{sbgp}(i) : \{i, y_b, y_e\}.\varphi_{I1}$$

In $(\mathrm{a}_1)$, all head tuples $p$ other than *signature* and *advertise* takes on the same invariant $\varphi_{link1}(p, n, d, t)$:

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

It states that node $n$ is the first element in path $p$, and the *link* tuple from $n$ to its neighbor in $p$ exists in $n$'s database.

For *signature* and *advertise*, we introduce another property:

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

$\varphi_{link2}(p, n, d, n', t)$ extends $\varphi_{link1}(p, n, d, t)$ by including the receiving node $n'$ as an argument, asserting that the link between $n$ and $n'$ also exists. And the invariants of *signature* and *advertise* are:

$$\begin{array}{l}\varphi_{signature}(i, t, n, m, s) = \exists n', d, m = d :: n' :: p \wedge \varphi_{link2}(p, n, d, n', t)\\ \varphi_{advertise}(i, t, n', n, d, p, sl) = \varphi_{link2}(p, n, d, n', t)\end{array}$$

We prove $(a_1)$ using the Inv rule. Then, by applying Honest rule to $(\mathrm{a}_1)$ and only keeping the clause in $\varphi_{I2}$ related to signature, we derive the following:

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

$(a_2)$ connects an honest node's signature to the existence of related link tuples at a previous node in the path $p$.

Next, we use $(a_2)$ along with $A_{sig}$ to prove $(\mathrm{a}_0)$. Applying $A_{sig}$ to tuples publicKeys and verify in $(a_0)$, we can get:

$$\begin{array}{l}(\mathrm{a}_3)\ \cdot;\cdot \vdash \forall \mathsf{Nd}, \forall \mathsf{MsgV}, \forall \mathsf{SigM}, \forall \mathsf{t},\\ \mathsf{Honest}(\mathsf{Nd}) \supset \exists \mathsf{t'}, \mathsf{t'} < \mathsf{t} \wedge \mathsf{signature}(\mathsf{Nd}, \mathsf{MsgV}, \mathsf{SigM})@(\mathsf{Nd}, \mathsf{t'})\end{array}$$

We further apply $(a_2)$ to $(a_3)$ to obtain:

$$\begin{array}{l}(\mathrm{a}_4)\ \cdot;\cdot \vdash \forall \mathsf{Nd}, \forall \mathsf{t}, \forall \mathsf{MsgV},\\ \exists n', d, p, \mathsf{Nd} = d :: n' :: p \wedge \varphi_{link2}(p, \mathsf{Nd}, d, n', \mathsf{t})\end{array}$$

Combining $(\mathrm{a}_4)$ and the assumptions in $(\mathrm{a}_0)$, we are able to prove the conclusion of $(\mathrm{a}_0)$. Other premises can be proved similarly. For non-recursive rules, the premises for them are straightforward. The detailed proof can be found online.

**Discussion.** $\varphi_{auth1}$ is a general template for proving different kinds of route authenticity. For example, S-BGP satisfies a stronger property that guarantees authentication of each subpath in a given path $p$. The property, called $\mathsf{goodPath2}(t, d, p)$, is defined in Figure 16. The meaning of the variables remains the same as before.

Compared with goodPath, the last two rules of goodPath2 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

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

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

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

Figure 16: Definitions of goodPath2

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 direct links of $n$. More specifically, given $p = n1 :: n :: n2 :: p1$ and $p' = n1 :: n :: n2 :: p2$, with $p1 \neq p2$, an attacker could not replace $p$ with $p'$ without being detected. However, a protocol that only requires a node $n$ to sign the links to its direct neighbors would be vulnerable to such attack.

5.2. **SCION.** SCION [33] is a clean-slate design of Internet routing architecture that offers more flexible route selection and failure isolation along with route authenticity. Our case study focuses on the routing mechanism proposed by SCION. We only provide high-level explanation of SCION. Detailed encoding can be found under the following link (`http://netdb.cis.upenn.edu/secure_routing/`).

In SCION, Autonomous Domains (AD) — a concept similar to Autonomous Systems (AS) in BGP — are grouped into different Trust Domains (TD). Inside each Trust Domain, top-tier ISP's are selected as the TD core, which provide routing service inside and across the border of TD. Figure 17 presents an example deployment of SCION with two TD's. Each AD can communicate with its neighbors. The direction of direct edges represents provider-customer relationship in routing; the arrow goes from a provider to its customer.

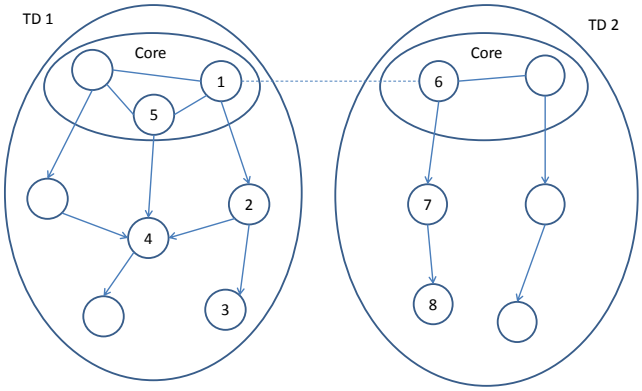



Figure 17: An example deployment of SCION

To initiate the routing process, a TD core periodically generates a path construction announcement, called a beacon, to all its customer ADs. Each non-core AD, upon receiving a beacon, (1) verifies the information inside the beacon, (2) attaches itself to the path inside the received beacon to construct a new beacon, and (3) forwards the new beacon to its customer ADs. Each beacon represents a path towards the TD core (e.g. path "1-2-3" in Figure 17). After receiving $k$ beacons , an downstream AD selects $m$ paths out of $k$ and uploads them to the TD core, thus finishing path construction ($k$ and $m$ can be set by the administrator). When later an AD $n$ intends to send a packet to another AD $n'$, it first queries the TD core for the paths that $n'$ has uploaded, and then constructs a forwarding path combining its own path to the TD core with the query result. For example, in Figure 17, when node 4 wants to communicate with node 3, it would query from the TD core for path "1-2-3", and combine it with its own path to the TD core (i.e. "4-5"), to get the desired path "4-5-1-2-3".

In Table 2, we summarize the SANDLog encoding of the path construction phase in SCION. Definitions of important tuples can be found in Figure 18. The path construction beacon plays an important role in SCION routing mechanism. A beacon is composed of four fields: an interface field, a time field, an opaque field and a signature. The interface field in SCION is identical to the announced path in S-BGP. An interface field contains a list of AD identifiers representing the routing path. As its name suggests, the interface field also includes each AD's interfaces to direct neighbors in the path — SCION calls the interface to an AD's provider as *ingress* and the one to a customer as *egress*. Each AD attaches his own identifier along with its *ingress* and *egress* to the end of the received interface field, generating the new interface field. For example, in Figure 17, assume the ingress interface of AD 2 against AD1 is "a", and the egress interface of AD 2 against AD 4 is "b". Given an interface field {c::1} from AD 1 — c represents the egress interface of AD 1 against AD 2 — the newly generated interface field at AD 2 targeting AD 4 will be {c::1::a::b::2}.

The time field is a list of time stamps which record the arrival time of the beacon at each AD. The opaque field adds a message authentication code (MAC) on each AD's *ingress* and *egress* fields using the AD's private key, for the purpose of path authentication during data forwarding. The final part is called the signature list. Each AD constructs a signature by signing the above threes fields (i.e. the interface field, the opaque field and the time field) along with the signature received from preceding ADs. The newly generated signature is appended to the end of the signature list.

SCION also satisfies similar route authenticity properties as S-BGP. Each path in SCION is composed of two parts: a path from the sender to the TD core (called "up path") and a path from the TD core to the receiver (called "down path"). We only prove the properties for the up paths. The proof for the down paths can be obtained similarly by switching the role of provider and customer. Tuples provider and customer in SCION can be seen as counterparts of the link tuple in S-BGP, and tuple beaconIni and tuple beaconFwd correspond to tuple advertise. The definition of route authenticity in SCION, denoted $\varphi_{authS}$, is defined as:

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

| Rule | Summary | Head Tuple |
|---|---|---|
| **b1:** | TD core generates a signature. | signature(@$core$, $info$, $sig$, $time$) |
| **b2:** | TD core signs beacon global information. | signature(@$core$, $info$, $sig$, $time$) |
| **b3:** | TD core initiates an opaque field. | mac(@$core$, $info$, $hash$) |
| **b4:** | TD core initiates global info of beacon. | beaconPrep(@$core$, $glb$, $sigG$, $time$) |
| **b5:** | TD core sends a new beacon to neighbor. | beaconIni(@$nei$, $core$, $td$, $itf$, $tl$, $ol$, $sl$, $sigG$) |
| **b6:** | AD receives a beacon from TD core. | beaconRev(@$ad$, $td$, $td$, $itf$, $tl$, $ol$, $ing$, $sigG$) |
| **b7:** | AD receives a beacon from non-core AD. | beaconRev(@$ad$, $td$, $ing$, $itf$, $tl$, $ol$, $sl$, $sigG$) |
| **b8:** | AD verifies global information. | beaconToVeri(@$ad$, $td$, $itf$, $l$, $ol$, $sl$, $sigG$, $itfv$, $pos$) |
| **b9:** | AD recursively verifies signatures. | beaconToVeri(@$ad$, $td$, $itf$, $tl$, $ol$, $sl$, $sigG$, $itfv$, $pos$) |
| **b10:** | AD validates a beacon. | verifiedBeacon(@$ad$, $td$, $ing$, $itf$, $tl$, $ol$, $sl$, $sigG$) |
| **b11:** | AD creates signature for new beacon. | signature(@$ad$, $info$, $sig$, $time$) |
| **b12:** | AD initiates opaque field for new beacon. | mac(@$ad$, $info$, $hash$) |
| **b13:** | AD sends the new beacon to neighbor. | beaconFwd(@$nei$, $ad$, $td$, $itf$, $tl$, $ol$, $sl$, $sigG$) |
| **pc1:** | AD extracts path information. | upPath(@$ad$, $td$, $itf$, $ol$, $tl$) |
| **pc2:** | AD initiates path upload. | pathUpload(@$nei$, $ad$, $src$, $core$, $itf$, $ol$, $op$, $pos$) |
| **pc3:** | AD sends path to upstream neighbor. | pathUpload(@$nei$, $ad$, $src$, $core$, $itf$, $ol$, $op$, $pos$) |
| **pc4:** | TD core stores received path. | downPath(@$core$, $src$, $itf$, $op$) |

Table 2: SANDLog encoding of path construction in SCION

Formula $\varphi_{authS}$ asserts a property goodInfo($t, td, n, sl, itf$) on any beacon tuple generated by node $n$, which is either a TD core or an ordinary AD. The definition of goodInfo is shown in Figure 19. Predicate 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. goodInfo($t, td, n, sl, itf$) makes sure that each AD present in the interface field $itf$ does have the specified links to its provider and customer respectively. Also, for each non-core AD, there always exists a verified beacon corresponding to the path from the TD core to it.

More concretely, 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 downstream neighbor, represented by customer. In addition, a verifiedBeacon tuple should exist, representing an authenticated route stored in the database, with all inside signatures properly verified. The difference between these two cases is caused by two possible types of an AD's provider: TD core and non-TD core.

| | |
|---|---|
| $\mathsf{coreTD}(@n, c, td, ctf)$ | $c$ is the core of TD $td$ with certificate $ctf$ attesting to that fact |
| $\mathsf{provider}(@n, m, ig)$ | $m$ is $n$'s provider, with traffic into $n$ through interface $ig$ |
| $\mathsf{customer}(@n, m, eg)$ | $m$ is $n$'s customer, with traffic out of $n$ through interface $eg$ |
| $\mathsf{beaconIni}(@m, n, td,$ | $itf$, 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. |
| $\mathsf{verifiedBeacon}(@n, td,$ | $itf$ 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 $\mathsf{beaconIni}$ |
| $\mathsf{beaconFwd}(@m, n, td,$ | $itf$ 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 $\mathsf{beaconIni}$ |
| $\mathsf{upPath}(@n, td, itf,$ | $opqU$ is a list of opaque fields indicating a path. |
| $opqU, tl)$ | Rest of the fields have the same meaning as those in $\mathsf{beaconIni}$. |
| $\mathsf{pathUpload}(@m, n,$ | $src$ 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 $\mathsf{upPath}$. |

Figure 18: Tuples for SCION

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

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

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

Figure 19: Definitions of $\mathsf{goodInfo}$

The proof strategy is exactly the same as that used in proof of $\mathsf{goodPath}$ about S-BGP. To prove $\varphi_{authS}$, we first prove $prog_{scion}$ has a stronger program invariant $\varphi_I$:

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

where $\varphi_I(i, y_b, y_e)$ is defined as:

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

Especially, $\varphi_p$ for beaconIni and beaconFwd are as follows:

$\varphi_{beaconIni}(i, t, m, n, td, itf, tl, ol, sl, sg) = \mathsf{goodInfo}(t, td, n, sl, itf)$
$\varphi_{beaconFwd}(i, t, m, n, td, itf, tl, ol, sl, sg) = \mathsf{goodInfo}(t, td, n, sl, itf)$

As in S-BGP, (b) can be proved using Inv rule, whose premises are verified in Coq. After (b) is proved, we can deduce $\varphi' = \forall n, t', \mathsf{Honest}(n) \supset \varphi_I(n, t', -\infty)$ by applying Honest rule to (b). Finally, $\varphi_{authS}$ is proved by showing that $\varphi' \supset \varphi_{authS}$, which is straightforward.

At the end of the path construction phase, an AD needs to upload its selected paths to the TD core for future queries (i.e. rules $pc1 - pc4$ in Table 2). The uploading process uses the forwarding mechanism in SCION, which provides hop-by-hop authentication. An AD who wants to send traffic to another AD attaches each data packet with the opaque field extracted from a beacon received during the path construction phase. The opaque field contains MACs of the ingress and egress of all ADs on the intended path. When the data packet is sent along the path, each AD en-route re-computes the MAC of intended ingress and egress using its own private key. This MAC is compared with the one contained in the opaque field. If they are the same, the AD knows that it has agreed to receiving/sending packets from/to its neighbors during path construction phase and forwards the packet further along the path. Otherwise, it drops the data packet.

The formal definition of data path authenticity in SCION is defined as:

$\varphi_{authD} = \forall m, n, t, src, core, itf, opqD, opqU, pt,$
$\mathsf{honest}(n) \wedge$
$\mathsf{pathUpload}(@m, n, src, core, itf, opqD, opqU, pt)@(n, t) \supset$
$\mathsf{goodFwdPath}(t, n, opqU, pt)$

Formula $\varphi_{authD}$ 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 $opqU$, indicating the next opaque field to be checked. 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 18. $\mathsf{goodFwdPath}(t, n, opqU, pt)$ states that whenever an AD receives a packet, it has direct links 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 definition of $\mathsf{goodFwdPath}(t, n, opqU, pt)$ is given in Figure 20. There are four cases. The base case is when $pt$ is 0, which means nothing has been 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 second and third cases both cover this scenario. The second case applies when a node $n$'s provider is TD core, while in the third case, $n$'s provider and customer are both ordinary TDs.

SCION uses MAC for integrity check during data forwarding, so we use the following axiom about MAC. It states that if a node $n$ verifies a MAC, using $n'$'s key $k$, there must have been a node $n''$ who created the MAC at an earlier time $t'$.

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

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

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

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

Figure 20: Definitions of goodFwdPath

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

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:

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

The rest of the proof follows the same strategy as that of goodPath and goodInfo. Interested readers can refer to our proof online for details.

5.3. **Comparison between S-BGP and SCION.** In this section, we compare the difference between the security guarantees provided by S-BGP and 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, signatures in S-BGP are not technically layered—ASes in S-BGP only sign the path information, not including previous signatures. On the other hand, ADs in SCION sign

the previous signature so signatures in SCION are nested. 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 also provides stronger security guarantees than S-BGP in data forwarding. 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 $\langle$IP prefix, next hop$\rangle$. 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. This next hop must have been authenticated, 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 guarantee over S-BGP in terms of the last hop of the packet. An AS $n$ running S-BGP has no way of detecting whether a received packet is from legitimate neighbor ASes who are authorized to forward packets to $n$. Imagine that $n$ 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 for AS's who run S-BGP for routing. 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 in the opaque field, since $m$ cannot forge the MAC information embedded in the beacon.

5.4. **Empirical evaluation.** We use RapidNet [27] to generate low-level implementation of S-BGP and SCION from SANDLog encoding. We validate the low-level implementation in the ns-3 simulator [24]. 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.

## 6. Related Work

**Cryptographic Protocol Analysis.** The analysis of cryptographic protocols [11, 29, 16, 26, 13, 5, 3, 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 [1, 2, 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 [7], we have verified route authenticity properties on variants of S-BGP using a combination of manual proofs and an automated tool, Proverif [6]. 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 [4, 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 the declarative language SANDLog. We have integrated verification into a unified framework for formal analysis and empirical evaluation of secure routing protocols. As future work, we plan to expand our use cases, for example, to investigate mechanisms for securing the data (packet forwarding) plane [22]. In addition, as an alternative to Coq, we are also exploring the use of automated first-order logic theorem provers to automate our proofs.

## References

[1] Mathilde Arnaud, Véronique Cortier, and Stéphanie Delaune. Modeling and verifying ad hoc routing protocols. In *Proceedings of CSF*, 2010.
[2] Mathilde Arnaud, Véronique Cortier, and Stéphanie Delaune. Deciding security for protocols with recursive tests. In *Proceedings of CADE*, 2011.
[3] J. Bau and J.C. Mitchell. A security evaluation of DNSSEC with NSEC3. In *Proceedings of NDSS*, 2010.
[4] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Formal verification of standards for distance vector routing protocols. *J. ACM*, 49(4), 2002.
[5] Bruno Blanchet. Automatic verification of correspondences for security protocols. *J. Comput. Secur.*, 17(4), December 2009.
[6] Bruno Blanchet and Ben Smyth. Proverif 1.86: Automatic cryptographic protocol verifier, user manual and tutorial. `http://www.proverif.ens.fr/manual.pdf`.
[7] Chen Chen, Limin Jia, Boon Thau Loo, and Wenchao Zhou. Reduction-based security analysis of internet routing protocols. In *WRiPE*, 2012.
[8] Chen Chen, Limin Jia, Hao Xu, Cheng Luo, Wenchao Zhou, and Boon Thau Loo. A program logic for verifying secure routing protocols. June 2014.
[9] CNET. How pakistan knocked youtube offline.
[10] Véronique Cortier, Jan Degrieck, and Stéphanie Delaune. Analysing routing protocols: four nodes topologies are sufficient. In *Proceedings of POST*, 2012.

[11] Anupam Datta, Ante Derek, John C. Mitchell, and Arnab Roy. Protocol Composition Logic (PCL). *Electronic Notes in Theoretical Computer Science*, 172:311–358, 2007.
[12] Dawson Engler and Madanlal Musuvathi. Model-checking large network protocol implementations. In *Proceedings of NSDI*, 2004.
[13] Santiago Escobar, Catherine Meadows, and José Meseguer. A rewriting-based inference system for the NRL protocol analyzer: grammar generation. In *Proceedings of FMSE*, 2005.
[14] Deepak Garg, Jason Franklin, Dilsun Kaynar, and Anupam Datta. Compositional system security with interface-confined adversaries. *ENTCS*, 265:49–71, September 2010.
[15] Alwyn Goodloe, Carl A. Gunter, and Mark-Oliver Stehr. Formal prototyping in early stages of protocol design. In *Proceedings of ACM WITS*, 2005.
[16] Changhua He, Mukund Sundararajan, Anupam Datta, Ante Derek, and John C. Mitchell. A modular correctness proof of IEEE 802.11i and TLS. In *Proceedings of CCS*, 2005.
[17] Hans W. Kamp. *Tense Logic and the Theory of Linear Order*. Phd thesis, Computer Science Department, University of California at Los Angeles, USA, 1968.
[18] Stephen Kent, Charles Lynn, Joanne Mikkelson, and Karen Seo. Secure border gateway protocol (S-BGP). *IEEE Journal on Selected Areas in Communications*, 18:103–116, 2000.
[19] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative Networking: Language, Execution and Optimization. In *SIGMOD*, 2006.
[20] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. In *Communications of the ACM*, 2009.
[21] Boon Thau Loo, Joseph M. Hellerstein, Ion Stoica, and Raghu Ramakrishnan. Declarative Routing: Extensible Routing with Declarative Queries. In *SIGCOMM*, 2005.
[22] Jad Naous, Michael Walfish, Antonio Nicolosi, David Mazieres, Michael Miller, and Arun Seehra. Verifying and enforcing network paths with ICING. In *Proceedings of CoNEXT*, 2011.
[23] Vivek Nigam, Limin Jia, Boon Thau Loo, and Andre Scedrov. Maintaining distributed logic programs incrementally. In *Proceedings of PPDP*, 2011.
[24] ns 3 project. Network Simulator 3. `http://www.nsnam.org/`.
[25] One Hundred Eleventh Congress. 2010 report to congress of the u.s.-china economic and security review commission, 2010.
[26] L. C. Paulson. Mechanized proofs for a recursive authentication protocol. In *Proceedings of CSFW*, 1997.
[27] RapidNet: A Declarative Toolkit for Rapid Network Simulation and Experimentation. `http://netdb.cis.upenn.edu/rapidnet/`.
[28] Ronald L. Rivest, Adi Shamir, and Leonard M. Adleman. A method for obtaining digital signatures and public-key cryptosystems (reprint). *Commun. ACM*, 26(1):96–99, 1983.
[29] Arnab Roy, Anupam Datta, Ante Derek, John C. Mitchell, and Seifert Jean-Pierre. Secrecy analysis in protocol composition logic. In *Proceedings of ESORICS*, 2007.
[30] Secure BGP. http://www.ir.bbn.com/sbgp/.
[31] Tao Wan, Evangelos Kranakis, and P. C. Oorschot. Pretty secure BGP (psBGP). In *Proceedings of 12th NDSS*, 2005.
[32] Russ White. Securing bgp through secure origin BGP (soBGP). *The Internet Protocol Journal*, 6(3):15–22, 2003.
[33] Xin Zhang, Hsu-Chun Hsiao, Geoffrey Hasker, Haowen Chan, Adrian Perrig, and David G. Andersen. Scion: Scalability, control, and isolation on next-generation networks. In *Proceedings of Oakland S&P*, 2011.

## Appendix A. Proof of Theorem 3.1

By mutual induction on the derivation $\mathcal{E}$. The rules for standard first-order logic inference rules $\Sigma;\Gamma \vdash \varphi$ are straightforward. We show the case when $\mathcal{E}$ ends in the Honest rule.

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

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

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

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

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

By (2),

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

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 $\mathcal{T}$, $\sigma$ such that $\mathcal{T} \vDash \Gamma\sigma$, and at time $\tau_b$, node $\iota$'s local state is ($\iota$, [], [], $prog(\iota)$), given any time point $\tau_e$ such that $\tau_e \geqslant \tau_b$,

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 ($\iota$, [], [], $[prog]_\iota$) $\in \mathcal{C}$

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 conjunction are true by construct a generic proof for one clause.

**case::** DeQueue is the last step.

Given a substitution $\sigma_1$ 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 $\varphi_p$ 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 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 $\mathcal{E}_1$ (here $\mathcal{E}_1$ is a smaller derivation than $\mathcal{E}$, so we can directly invoke 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 $\Delta r$ 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 $\iota$, $(p_j(\vec{s_j}))\sigma_0 \in \Psi^\nu$ 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 $\varphi_p$ 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 $\iota$, 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 predicate, 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 $p_{agg}(\vec{t})$.