Reasoning About Secrecy for Active Networks

Pankaj Kakkar
University of Pennsylvania
pankaj@gradient.cis.upenn.edu

Carl A. Gunter
University of Pennsylvania
gunter@cis.upenn.edu

Martín Abadi
abadi@soe.ucsc.edu

Abstract

In this paper we develop a language of mobile agents called *uPLAN* for describing the capabilities of active (programmable) networks. We use a formal semantics for uPLAN to demonstrate how capabilities provided for programming the network can affect the potential flows of information between users. In particular, we formalize a concept of security against attacks on secrecy by an 'outsider' and show how basic protections are preserved in the presence of programmable network functions.

1 Introduction

The goal of research on *programmable* or *active* networks [21, 20] is to make computer networks more flexible. This flexibility threatens the security of the network, so good techniques for modeling and analyzing the consequences of providing new network services are desirable. This paper describes an approach based on a simple language of mobile programs. The primary contributions are a definition of secrecy for a simple model of active networks and routing, an analysis of how this definition relates to one based on bisimulation, and a collection of case studies.

Properties that Aid Scalability. The Internet has proved the value of high connectivity in networking. Active networking research has attempted to achieve scalability too, aiming to accommodate internetworks composed of thousands of physical networks, multiple administrative domains, and a diversity of users. However, some users may harbor ill-will towards others, or even towards the network as a whole. Even when no bad intentions are involved, a programmable network may be vulnerable to mistakes or unexpected interactions between its users. The mechanisms for programming the network may also be helpful in damaging it. Such problems are not limited to new kinds of programmable networks; the current Internet has problems of its own. Complex interactions between features may lead to vulnerabilities in systems thought to be well-understood. For example, Bellovin [7] showed how to combine remote log-in, the domain name system (DNS) directory, router table updates, and other features to compromise Unix hosts on the Internet. Clearly the growth of mobile agents has created new risks for hosts; programmable networks that aim to move this form of agent into routers bring additional risks and challenges.

Research on programmable networks has sought to address these concerns by a variety of mechanisms. One is to imitate features of the Internet that have caused it to be scalable and survivable. A simple example is the way the Internet protocol (IP) uses a Time-To-Live (TTL) field to limit the effects of routing loops. The TTL field also relates the ability of a host to affect the network to the size of its attachment: a user with a modem is not able to send very many packets in the first place, and the packets sent cannot circulate infinitely or create many new packets to waste bandwidth. Programming systems for active networks, generally known as Execution Environments (EE's), often provide some analog of the TTL field to control network resource utilization.

The aim of this paper is to look at another form of protection that aids the usability and survivability of the Internet. In this paper we call it *Security Against Outsiders* or *AO-security.* The Internet has been able to survive the threat of *password sniffing,* that is, the ability to listen to network traffic and find passwords that can be used to gain illegitimate access to hosts or routers. Various protocols are available to avoid sending passwords in cleartext over the network, but the use of cleartext passwords is still very common. A primary reason is the fact that when a password is sent between one host and another, it does not visit every network in the Internet on the way. A party seeking to learn the password may not have an obvious mechanism for listening on a network if he does not have access through a host attached to it. While the password is vulnerable to sniffing by 'insiders' who are on the networks over which it will pass, it may be less vulnerable to 'outsiders' who must sniff from afar with little toe-hold from which to grab and redirect the sensitive information. Protocols that enable internetwork routing systems to adapt to topological changes such as router failures also cause packets to appear in unexpected places in the network, so topological guarantees are generally viewed as a weak form of protection. However, topological guarantees are a recognized practical part of current protections against vulnerabilities like password sniffing and also provide protections against more sophisticated attacks such as traffic analysis. For example, topological guarantees can make it harder to collect large amounts of ciphertext or determine whether an encrypted email was sent between two parties at a given time. AO-security is sometimes dumb luck; but it is often crucial, for example, in corporate networks behind firewalls and with moderate security requirements. The aim of this paper is to study AO-security in the context of active networks, where luck may not be enough. Our study provides insight into the protection afforded by programming interfaces and physical topology.

The Challenge of AO-Security for Active Networks. When we allow a network to become programmable, some of the apparent AO-security enjoyed by the Internet protocols may be lost if programmability of routers is available to outsiders and the right kinds of operations are made available to them. For example, researchers in the SwitchWare Project at the University of Pennsylvania have experimented with downloadable OCaml modules capable of dynamically altering switching and queuing strategies on routers [13, 6]. The privilege to use such a function will probably be limited by some form of cryptographic authorization system [5], but the underlying capabilities easily accommodate a man-in-the-middle attack on all of the packets passing through the affected router. Password sniffing would clearly be a possibility from any site that was able to obtain these capabilities. Execution environments like PLAN [14], ANTS [23], and others strive to control the tradeoff between security and flexibility by limiting the programming interface available to active network agents. This strategy can be applied quite flexibly. For instance, Hicks and Keromytis [12] describe an active firewall where outsiders passing through are allowed to use active network capabilities but only on a limited interface available to 'visitor' packets. In particular, the firewall places wrappers on visitor packets that limit the symbol table of the packet when it evaluates on a router.

To realize the goal of understanding the limits of AO-security in active networks we need to develop a theory for explaining the effect on security of offering new services on routers. Such an explanation is not only important to 'purely active' networks, but also to deployments of active network EE's within the Internet, for instance in the ABONE (`www.csl.sri.com/activate`) active network testbed. Indeed, some of the examples we give below are meaningful to some degree even for the current Internet, especially as moves are being made to extend the capabilities of routers to include things like multicast, resource reservation, and even web caching. The ability to reason about interactions between routing and computation at endpoints may also be important as mobile agent technology progresses and connections between hosts and routers become more complex.

Requirements for a Suitable Theory. There are at least two problems that must be overcome in developing a suitable theory for relating router interfaces to AO-security in the network. First, it is necessary to develop a rigorous but simple concept of secrecy. Second, it is essential to represent the techniques in a sufficiently general way that the methods are applicable to more than one protocol or active network architecture.

Our approach to modeling secrecy is inspired by work on the spi calculus [2, 4, 3], which uses ideas from concurrency theory to model secrecy, describing who has access to pieces of data via scoping rules. Our approach is more basic because we aim for less generality for channel mobility. Our strategy is to model the capabilities of the network via a collection of agents that run over a fixed graph representing the underlying physical networks, routers, and hosts. (The spi calculus lacks a concept of location, but our model defines one.) Information flow is modeled via a concept of ‘signature enrichment’ in which agents that see capabilities pass across networks to which they are attached obtain these capabilities themselves and can use them in agents they create. This allows us to model secrecy in terms of where information does *not* flow, and we are able to show for certain cases that this is equivalent to conditions for secrecy based on noninterference.

Our approach to obtaining sufficient generality for protocol modeling is to select a small but expressive calculus of network agents and use this as the focus for translating capabilities into invariants. The calculus provides for agents able to make calls of the form ‘evaluate expression e at location l’. We name it *uPLAN* because it is inspired by the Packet Language for Active Networks (PLAN) [14] but it is not specific to the PLAN approach. One architectural distinction in active networks is between programs that are carried in packets that are evaluated when the packet arrives versus programs that are installed on a router and process the packets passing though it. PLAN is a scripting language for packets to invoke router-resident programs. Other approaches focus on router-resident services without using a packet language, or on a packet language for a fixed set of router-resident services. This distinction is important from a practical perspective, but from a theoretical perspective the distinction is small. If f is a router-resident function and x is the data contained in a packet one can think of f as being applied to a range of values of x, but one can also think of x as operating on a range of values of f. The properties of information flow will be the same from either perspective.

Outline. After this introduction we provide some background on networks, routing, and AO-security. We then introduce our language for describing network programmability and its semantics. We show how the language can be used to express and prove properties about secrecy and integrity of data. The next section generalizes the concept of secrecy, and provides an alternative way of expressing secrecy in networks using bisimulation. We then consider AO-security for an active networking variant of labeled routing. A final section concludes.

2 Background

The Internet achieves connectivity by connecting *networks* through *routers.* Each network supports the connectivity of a collection of *hosts* via network interfaces. Such an interface can be viewed abstractly as an *address* pair consisting of a network and a *location,* where a location is a host or router. Figure 1 shows an example of this model in which ‘clouds’ represent networks, squares represent routers, and circles represent hosts. To simplify matters we make no distinction between specific users on hosts and the hosts themselves, so the model in the figure includes computational agents like Alice and Bob, who have addresses on networks $n1$ and $n3$ respectively. For Alice to communicate with Bob she sends one or more packets to router $r1$ using network $n1$. Routers

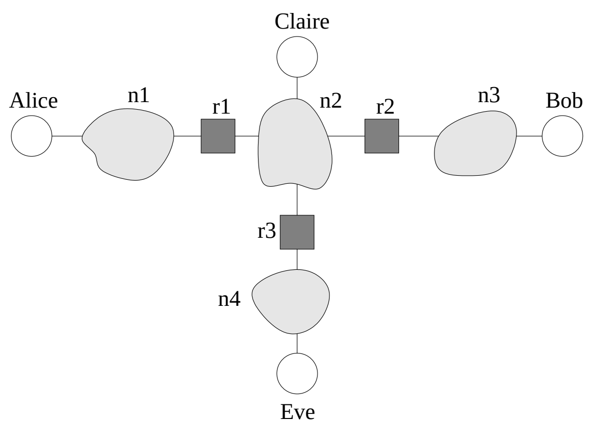

Figure 1: Internetwork Example

specialize in knowing how to get packets to where they belong based on a destination address provided by the sender; in particular, $r1$ will probably decide to forward the packets from Alice to Bob to router $r3$ across network $n2$. This router, in turn, will notice that it can get the packets directly to Bob and will communicate them to him across network $n3$. This process causes the communication medium consisting of the networks and routers to perform some of the functions of a single network. We refer to the collection consisting of the networks, routers, and hosts as an *internetwork* in this paper.

In general, a location l maintains a table that associate a 'next hop' with each destination address. The next hop is an address on a network to which l is attached. The routing table of a host is often simple; for instance, Alice is probably configured to send packets to $(n1, r1)$ whenever they are for an address not on $n1$. Routers generally maintain more interesting routing tables which they develop in a distributed computation involving other routers; the aim of this computation is finding paths between each pair of locations on a portion of the internetwork. Several protocols exist to achieve this goal. Two of the most widely-used protocols are RIP [18] and OSPF [19]. In both protocols, each router sends out regular updates of information to aid other routers in determining how to forward packets. Routers examine the updates that they get from each router adjacent to them and modify their routing tables accordingly. In RIP, the updates contain the length and first hop of the shortest known path to a given destination, while in OSPF the updates contain link information about routers around the internetwork.

Many network technologies are subject to 'sniffing' of information. For instance, an Ethernet LAN broadcasts each message to all of its attached hosts, any one of which may chose to 'listen' to messages not addressed to it. For instance, when $r1$ communicates packets from Alice to Bob across $n2$ for $r2$, hosts like Claire and routers like $r3$ that are attached to $n2$ can collect these packets and read them. This problem can be addressed by encryption, but many organizations have not yet implemented encrypted versions of the protocols. For instance, in the RIP and OSPF protocols, the router updates are protected only by passwords, which are exchanged in cleartext and hence knowable to anyone who is on a network that the updates go through. This lack of secrecy threatens integrity since anyone who knows the password can issue updates either by mistake or with bad intent. Here is how the RIP RFC on applicability of the protocol describes the security intention of the passwords [17, page 4]:

> The need for authentication in a routing protocol is obvious. It is not usually important to conceal the information in the routing messages, but it is essential to prevent the insertion of bogus routing information into the routers. So, while the authentication mechanism specified in RIP-2 is less than ideal, it does prevent anyone who cannot directly access the network (i.e., someone who cannot sniff the routing packets to determine the password) from inserting bogus routing information.

This is a description of AO-security for the RIP tables. The OSPF protocol allows passwords on a per-interface basis, and suggests the possibility of using a password per network. The protection provided by such passwords is characterized in the OSPF standard as a safeguard against mistakes [19, pages 205 and 206]:

> The authentication type is configurable on a per-area basis. Additional authentication data is configurable on a per-interface basis. For example, if an area uses a simple password scheme for authentication, a separate password may be configured for each network contained in the area. ...
>
> This guards against routers inadvertently joining the area. They must first be configured with their attached networks' passwords before they can participate in the routing domain.

Returning to Figure 1, these properties imply that although Claire may sniff passwords used between $r1$ and $r2$ and therefore may corrupt the routing updates with bogus updates, this option will not be available to Eve, who will never see these passwords on the network to which she is attached. Consider, on the other hand, the internetwork in Figure 2. There is now another router adjacent to Eve, making it possible for Eve to sniff the router update password between the two routers on her network and thus acquire the capability to change routing tables on those routers. Eve may use the information acquired in this way to corrupt the routers attached to her network. These routers, in turn, may mislead routers to which they send updates, and so on throughout a portion of the internetwork. This may allow Eve to cause some or all packets from Alice to Bob to go through her network, at least for a while.

Our aim, beginning in the next section, is to develop a language for expressing capabilities so we can talk about AO-security for this internetwork and internetworks in general. Note first that even for a fixed internetwork the issue is non-trivial. We must characterize what can happen under all of the message sequences that could be sent by Eve, taking into account all of the ways these

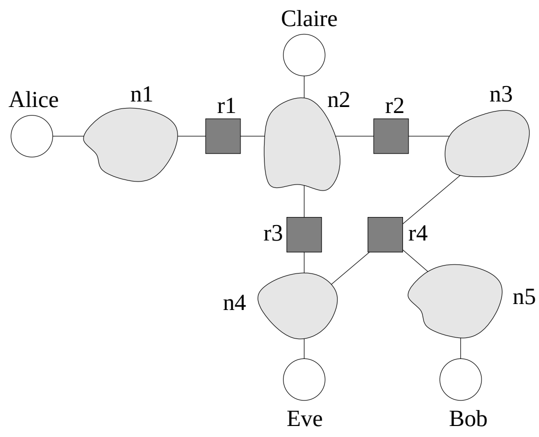

Figure 2: Internetwork Case Study

messages might interact with the updates passing between the routers or the messages between the hosts. Our strategy is to view the internetwork as providing a primitive programming capability. This primitive capability is bolstered by services offered by the routers and hosts, such as the routing protocol updates and any other services that may aid communication on the internetwork.

3 A Language-Oriented View of Networking

In this section, we define a language uPLAN inspired by PLAN [14]; very roughly, it is a small subset of PLAN with the key features needed for our study.

There are other systems for the study of security in distributed and mobile systems. Specifically, the pi calculus and related formalisms emphasize the concept of channel mobility (the capability of creating and communicating channels dynamically). In contrast, uPLAN deals with a fixed set of communication links and makes the internetwork topology and the location of computations explicit. In comparison with some other calculi for distributed computation (for example, the distributed join calculus), our model of distribution is rather concrete, in part because we are trying to model actual active networks, in part because of the limitations on mobility. All these calculi have in common, however, the view of some pieces of data as unforgeable and unguessable. This view plays an important role in the spi calculus, for example, and is crucial in our treatment of security in this paper.

3.1 Grammar

The syntax of uPLAN is given in Table 1. In the programming examples in the paper, we will use some syntactic sugar, such as allowing functions with multiple

Table 1: Grammar of uPLAN

Hosts	h	$\in$	Host
Routers	r	$\in$	Route
Locations (Loc)	l	$::=$	$h \mid r$
Networks	n	$\in$	Net
Addresses (Addr)	a	$::=$	(n, l)
Integers	i	$\in$	Int
Booleans	b	$::=$	$\mathsf{true} \mid \mathsf{false}$
Constants	c	$::=$	$i \mid b \mid a \mid ()$
Base Unary Ops	α	$\in$	$\neg \mid \mathsf{hd} \mid \mathsf{tl} \mid -$
Base Binary Ops	β	$\in$	$+ \mid * \mid - \mid / \mid$
			$\wedge \mid \vee \mid \mathsf{cons} \mid =$
Services	σ	$\in$	$\mathsf{Services}$
Data	δ	$\in$	Data
Variables	x	$\in$	Var
Variable Lists	$\vec{x}$	$::=$	$x \mid x, \vec{x}$
Definitions	d	$::=$	$\mathsf{val}\ x = e \mid$
			$\mathsf{fun}\ x(x) = e$
Definition Lists	$\vec{d}$	$::=$	$d \mid d\ \vec{d}$
Expressions	e	$::=$	$x \mid c \mid \delta \mid$
Chunks			$\|x\|(e) \mid \|\sigma\|(e)$
Remote Evaluation			$e\,@\,e \mid$
Unary Applications			$x(e) \mid \alpha(e) \mid \sigma(e) \mid$
Binary Applications			$e\,\beta\,e \mid$
Lists and tuples			$[\,] \mid [\vec{e}] \mid (e, \vec{e}) \mid \#i\ e \mid$
Conditionals			$e\,?\,e, e \mid$
Sequence			$e; e \mid$
Local Bindings			$\vec{d}\ \mathsf{in}\ e$
Expression Lists	$\vec{e}$	$::=$	$e \mid e, \vec{e}$

arguments. A simple example based on the internetwork of Figure 1 is a good place to begin understanding the concepts in uPLAN. The following program defines a function for *source routing* and uses it to route a datum through Alice to Bob. The program could be evaluated at any location.

```
fun sourceRoute(l,d) =
  ¬(l=[]) ?
    (|sourceRoute|(tl(l),d))  @  (hd(l)),
    deliver(d)
in
sourceRoute([(n1,Alice),(n3,Bob)], δ)
```

A program consists of a declaration part and an expression, which is to be evaluated locally. Let us suppose that this program is evaluated by Eve. The function `sourceRoute` is invoked on a list of two addresses, which happen to be the interfaces of Alice and Bob, in that order. The function first checks if the list is empty, and, finding that it is not, makes the following call:

```
    (|sourceRoute|(tl(l),d))  @  (hd(l))
```

This means the function `sourceRoute` should be invoked on the list `[(n3,Bob)]`, and this should be done at Alice, the location associated with the address at the head of the list. The bars surrounding the `sourceRoute` function are intended to indicate that this function may not be invoked locally. An expression of the form $|f|(x)$ is called a *chunk* (short for 'code hunk') and represents data at the local node. If we write:

```
val x = |deliver|(3)
in
x  @  (n1,Alice)
```

The value of `x` is bound to a pair consisting of a function name deliver and an argument 3. The function associated with `deliver` depends on the location where it is evaluated.

3.2 Semantics

The semantics of uPLAN is given as a set of transition rules representing computational steps applied to a multiset of terms representing network state. This form of description can be understood intuitively using a metaphor called a 'chemical abstract machine' [9] where the state is viewed as a collection of 'particles' and computation as a set of 'reactions' between these particles. Each reaction consumes one or more particles and produces one or more. In our model a particle may also be introduced into the mix by an agent (like Alice, Eve, or a router) or removed from it (to represent termination of a computation for instance). Multiset rewriting is well understood in programming language theory and is easily automated for model checking. For instance, Maude

Table 2: Semantic Objects in uPLAN

Chunk values	ch	$::=$	$\mathsf{chunk}(x, v, E)\ \mid$
			$\mathsf{chunk}(\sigma, v, E)$
Values (Val)	v	$::=$	$c \mid \delta \mid \bot \mid$
			$ch \mid$
			$[\,] \mid [\vec{v}] \mid$
			$(v, \vec{v}) \mid$
			$\mathsf{close}(x, x, e, E)$
Value lists	$\vec{v}$	$::=$	$v \mid v, \vec{v}$
Environment	E	:	$\mathsf{Var} \rightarrow \mathsf{Val}$
Stack	S	$::=$	$\mathbf{nil} \mid v :: S$
Opcodes	o	$\in$	$? \mid \mathbf{ap} \mid \mathbf{serv} \mid$
			$\mathbf{unary} \mid \mathbf{binary} \mid$
			$@ \mid \mathbf{bind} \mid$
			$\alpha \mid \beta \mid \sigma \mid$
			$\mathbf{ch} \mid \mathbf{list}_i \mid \mathbf{tup}_i \mid \mathbf{proj}_i$
Code	C	$::=$	$\mathbf{nil} \mid$
Opcodes			$o :: C \mid$
Expressions			$e :: C \mid$
Definition list			$\vec{d} :: C$
Dumps	D	$::=$	$\mathbf{nil} \mid$
			$\langle S, E, C, D \rangle$
Topology		:	$\mathsf{Net} \rightarrow \mathcal{P}(\mathsf{Loc})$
RouteState	t	:	$\mathsf{Loc} \rightarrow \mathsf{Addr} \rightarrow (\mathsf{Addr} \times \mathsf{Int})$
DataState	s	:	$\mathsf{Loc} \rightarrow \mathcal{P}(\mathsf{Data})$
Dictionary	z	:	$\mathsf{Loc} \rightarrow \mathsf{Data} \rightarrow \mathsf{Val}$
Controlled locations	k	$\in$	$\mathsf{Con} \subseteq \mathsf{Loc}$
Available services	Σ	$\subseteq$	$\mathsf{Services}$
Particles	p	$::=$	$t \mid s \mid z \mid \Sigma \mid$
			$\mathsf{Local}\langle S, E, C, D \rangle_l\ \mid$
			$\mathsf{Transit}\langle a, ch \rangle_l$

Table 3: Abstract Machine

$$
\begin{array}{rcl}
\mathsf{Local}\langle S, E, x :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle E(x) :: S, E, C, D\rangle_l \\
\mathsf{Local}\langle S, E, c :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle c :: S, E, C, D\rangle_l \\
\\
\mathsf{Local}\langle S, E, e \mathbin{?} e', e'' :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E, e :: ? :: e' :: e'' :: C, D\rangle_l \\
\mathsf{Local}\langle \mathsf{true} :: S, E, ? :: e' :: e'' :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E, e' :: C, D\rangle_l \\
\mathsf{Local}\langle \mathsf{false} :: S, E, ? :: e' :: e'' :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E, e'' :: C, D\rangle_l \\
\\
\mathsf{Local}\langle S, E, |x|(e) :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E, e :: \mathbf{ch} :: x :: C, D\rangle_l \\
\mathsf{Local}\langle v :: S, E, \mathbf{ch} :: x :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle \mathsf{chunk}(x, v, E) :: S, E, C, D\rangle_l \\
\\
\mathsf{Local}\langle S, E, |\sigma|(e) :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E, e :: \mathbf{ch} :: \sigma :: C, D\rangle_l \\
\mathsf{Local}\langle v :: S, E, \mathbf{ch} :: \sigma :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle \mathsf{chunk}(\sigma, v, E) :: S, E, C, D\rangle_l \\
\\
\mathsf{Local}\langle S, E, [\,] :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle [\,] :: S, E, C, D\rangle_l \\
\mathsf{Local}\langle S, E, [e_1, \ldots, e_n] :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E, e_1 :: \cdots :: e_n :: \mathbf{list}_n :: C, D\rangle_l \\
\mathsf{Local}\langle v_n :: \cdots :: v_1 :: S, E, \mathbf{list}_n :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle [v_1, \ldots, v_n] :: S, E, C, D\rangle_l \\
\\
\mathsf{Local}\langle S, E, (e_1, \ldots, e_n) :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E, e_1 :: \cdots :: e_n :: \mathbf{tup}_n :: C, D\rangle_l \\
\mathsf{Local}\langle v_n :: \cdots :: v_1 :: S, E, \mathbf{tup}_n :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle (v_1, \ldots, v_n) :: S, E, C, D\rangle_l \\
\\
\mathsf{Local}\langle S, E, \#i\ e :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E, e :: \mathbf{proj}_i :: C, D\rangle_l \\
\mathsf{Local}\langle (v_1, \ldots, v_i, \ldots, v_n) :: S, E, \mathbf{proj}_i :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle v_i :: S, E, C, D\rangle_l \\
 & & \text{where } 1 \leq i \leq n \\
\\
\mathsf{Local}\langle S, E, x(e) :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E, e :: x :: \mathbf{ap} :: C, D\rangle_l \\
\mathsf{Local}\langle \mathsf{close}(x, x', e, E') :: v :: S, E, \mathbf{ap} :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle \mathbf{nil}, E''[v/x], e, \langle S, E, C, D\rangle\rangle_l \\
 & & \text{where } E'' = E'[\mathsf{close}(x, x', e, E')/x] \\
\mathsf{Local}\langle v :: S, E, \mathbf{nil}, \langle S', E', C', D'\rangle\rangle_l & \longrightarrow & \mathsf{Local}\langle v :: S', E', C', D'\rangle_l \\
\\
\mathsf{Local}\langle S, E, \alpha(e) :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E, e :: \alpha :: \mathbf{unary} :: C, D\rangle_l \\
\mathsf{Local}\langle v :: S, E, \alpha :: \mathbf{unary} :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle \mathsf{ConstApply}(\alpha, v) :: S, E, C, D\rangle_l \\
\mathsf{Local}\langle S, E, \sigma(e) :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E, e :: \mathbf{serv} :: \sigma :: C, D\rangle_l \\
\mathsf{Local}\langle v :: S, E, \mathbf{serv} :: \sigma :: C, D\rangle_l, s, t, z & \longrightarrow & [\![\sigma]\!](v, k, s, t, z, (S, E, C, D)) \\
\mathsf{Local}\langle S, E, e\, \beta\, e' :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle e :: e' :: \beta :: \mathbf{binary} :: C, D\rangle_l \\
\mathsf{Local}\langle v' :: v :: S, E, \beta :: \mathbf{binary} :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle \mathsf{ConstApply}(\beta, v, v') :: S, E, C, D\rangle_l \\
\\
\mathsf{Local}\langle S, E, e\, @\, e :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E, e :: e' :: @ :: c, D\rangle_l \\
\mathsf{Local}\langle a :: ch :: S, E, @ :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle () :: S, E, C, D\rangle_l, \mathsf{Transit}\langle a, ch\rangle_l \\
\\
\mathsf{Local}\langle S, E, \vec{d}\ \mathsf{in}\ e :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle \mathbf{nil}, E, \vec{d} :: e, \langle S, E, C, D\rangle\rangle_l \\
\mathsf{Local}\langle S, E, d\vec{d} :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E, d :: \vec{d} :: C, D\rangle_l \\
\\
\mathsf{Local}\langle S, E, \mathsf{val}\ x = e :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E, e :: \mathbf{bind} :: x :: C, D\rangle_l \\
\mathsf{Local}\langle v :: S, E, \mathbf{bind} :: x :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E[v/x], C, D\rangle_l \\
\mathsf{Local}\langle S, E, \mathsf{fun}\ x(x') = e :: C, D\rangle_l & \longrightarrow & \mathsf{Local}\langle S, E[\mathsf{close}(x, x', e, E)/x], C, D\rangle_l \\
\mathsf{Local}\langle S, E, C, D\rangle_l & \longrightarrow & \\
\\
\mathsf{Transit}\langle (n, l), \mathsf{chunk}(x, v, E)\rangle_l & \longrightarrow & \mathsf{Local}\langle \mathbf{nil}, E, x(v), \mathbf{nil}\rangle_l \\
\\
\mathsf{Transit}\langle a, ch\rangle_{l_1}, t, s & \longrightarrow & \mathsf{Transit}\langle a, ch\rangle_{l_2}, t, s' \\
 & & \text{where } ch = \mathsf{chunk}(x, v, E) \text{ or } \mathsf{chunk}(\sigma, v, E) \\
 & & \text{and}((n, l_2), i) = t(l_1)(a) \\
 & & \text{and } s'(l) = \left\{ \begin{array}{l} s(l) \text{ if } l \notin \mathsf{Topology}(n) \\ s(l) \cup \{\delta | \delta \in v \text{ or } \delta \in E\} \text{ otherwise} \end{array} \right. \\
\\
s, \Sigma & \longrightarrow & \mathsf{Local}\langle \mathbf{nil}, \emptyset, e, \mathbf{nil}\rangle_l, s, \Sigma \\
 & & \text{where } e \text{ obeys } s(l) \\
 & & \text{and all services } \sigma \text{ in } e \text{ are in } \Sigma
\end{array}
$$

(`http://maude.csl.sri.com`) provides automated support for multiset rewriting and was used to analyze a protocol [22] based on the kind of labeled routing we discuss below in Section 6.

The chemical abstract machine semantics of uPLAN is given in Tables 2 and 3. A state of this machine is a multiset of terms we call *particles,* and each transition rule is a rewriting rule on this multiset that replaces some particles in the multiset with others. A *computation* Γ is a sequence of states of the abstract machine $M_1 \longrightarrow M_2 \longrightarrow \ldots \longrightarrow M_k$. We write $\longrightarrow^*$ for the transitive closure of $\longrightarrow$. Most of the rules in the table apply to the language constructs of uPLAN, but one rule allows for services σ. The semantics of a service is a function $[\![\sigma]\!]()$ of the machine state. The semantics of uPLAN and the service functions allows packets to stop making progress in the network and does not have any built-in liveness assumptions. In particular, packets will stop making progress if no transition rule applies to the current state. The semantics also allows for the possibility of packets being dropped from a computation, although it does so indirectly by allowing $\mathsf{Local}\langle\cdots\rangle_l$ particles to be dropped, rather than $\mathsf{Transit}\langle\cdots\rangle_l$ particles. In the scenarios that we consider, the possibility that packets might be dropped does not cause any secrets to be leaked to outsiders, but that need not always be the case.

We use the notion of 'controlled' versus 'uncontrolled' locations to model the adversarial behavior of locations. It can be imagined that controlled locations are those that lie within a single administrative domain, like all machines in a lab that a system administrator controls or all computers operating behind a firewall at a business facility. Our results will be based on the assumption that uncontrolled locations may create any particles they wish based on the information they have, but controlled locations will create particles according to restrictions we place on their behavior. The final rule from Table 3 is called the *generation rule* and represents particles created by hosts and routers. In most cases we will state explicitly which particles are produced by the controlled locations by simply listing them, but in other cases we need to stipulate that controlled locations produce particles using this rule according to a specific process they may be running, such as a standard routing protocol. Controlled locations can be viewed as constituting a system which must react not only to actions of its own participants, but also to arbitrary actions of an environment of uncontrolled locations.

The internetwork topology is modeled through a map $\mathsf{Topology}$ from networks to sets of locations—each location in the set has an attachment to the network. In a valid topology, hosts are attached to only one network. (There would be no difficulty in adding multi-homed hosts to the model.) We model secrecy through the notion of a $\mathsf{DataState}$. A $\mathsf{DataState}$, a map from locations to sets of Data, defines exactly what Data is 'known' to a given location. Routing tables are modeled through a $\mathsf{RouteState}$, a map from locations to tables, which are themselves maps from destination addresses to next hop addresses with integer weights. In a valid $\mathsf{RouteState}$, the next hop for any destination and any location must be a neighbor, that is, located on a common network with the location at which the next hop is computed.

To model state on locations, a $\mathsf{Dictionary}$ associates with each location a table mapping Data to uPLAN values. We use a special value, $\perp$, to represent undefined entries in the dictionary. Each entry for which a value hasn't been explicitly defined maps to $\perp$. We also define $\mathsf{Domain}_l(z)$ to be $\{\delta \mid z(l)(\delta) \neq \perp\}$.

Locations are constrained to emit only uPLAN programs containing data that they have 'learned'. For a datum δ, we say $\delta \in e$ if δ occurs syntactically in the particle e, and we say e obeys $s(l)$ if every $\delta \in e$ is also a member of $s(l)$. We also extend the relation $\in$ to environments and particles and the relation obeys to environments. We say l knows δ in $\mathsf{DataState}$ s if $\delta \in s(l)$.

We consider a valid state M to be a multiset in which there is exactly one instance of a $\mathsf{DataState}$, s, one instance of a $\mathsf{RouteState}$, t, one instance of a $\mathsf{Dictionary}$, z, and one instance of a set of services Σ. There may be multiple instances of $\mathsf{Transit}\langle\cdots\rangle_l$ or $\mathsf{Local}\langle\cdots\rangle_l$ particles. A $\mathsf{Transit}\langle a, \mathsf{chunk}(x, v, E)\rangle_l$ particle contains a chunk value and the address where the chunk is to be evaluated. The particle is routed to the specified address and converted into a $\mathsf{Local}\langle\cdots\rangle_l$ particle that evaluates $x(v)$ in the environment E upon arrival at the destination a. A $\mathsf{Local}\langle S, E, C, D\rangle_l$ particle contains a program in uPLAN under evaluation at a location. As the nomenclature of the elements of a Local particle implies, this evaluation happens in an extended version of the SECD abstract machine [15, 16]. This evaluation may give rise to several new $\mathsf{Transit}\langle\cdots\rangle_l$ particles before finishing and disappearing. $\mathsf{Local}\langle\cdots\rangle_l$ particles may also appear spontaneously.

The function $\mathsf{ConstApply}$ applies an operator to one or more values and returns the result of the application. Services are simulated by associating with a service its 'meaning', which is a function that takes as arguments the argument to the service and global internetwork state in the form of current location, $\mathsf{DataState}$, $\mathsf{Dictionary}$, and $\mathsf{RouteState}$, and returns a $\mathsf{DataState}$, $\mathsf{Dictionary}$, and $\mathsf{RouteState}$ along with one or more particles. We could constrain this definition further since services will be expected to work only with state at the location where they are invoked. For instance, a service invoked on a router r will not modify the state on another router r', but it may cause a message to be sent to r' which would cause such a change.

4 Some AO-Security Properties

Let us now show how to prove whether AO-security holds in some basic cases. We carry out two analyses, the first assuming that the routers use static routing tables, and the second assuming that they can be dynamically updated. For the first of these we can describe a set of particles created by controlled locations explicitly, and then analyze the actions the uncontrolled locations may take in terms of these. The dynamic update case is more subtle because controlled locations generate particles based on information they receive from other locations, including possibly uncontrolled locations. The aim of an adversary at an uncontrolled location is to corrupt a controlled router and exploit the spread of this corruption to other controlled locations using standard routing updates produced by the controlled routers. For each of the static and dynamic cases we first consider results for a fixed topology, then express a generalization to arbitrary topologies.

4.1 Static Routing Tables

Let us define a basic service deliver to deliver a datum to an intended recipient. We define the value of

$$[\![\mathsf{deliver}]\!](\delta, l, s, t, z, (S, E, C, D))$$

to be $\mathsf{Local}\langle () :: S, E, C, D\rangle_l, s', t, z$ where

$$s'(l') = \begin{cases} s(l) \cup \{\delta\} & \text{if } l = l' \\ s(l) & \text{otherwise} \end{cases}$$

That is, the deliver service simply causes δ to be added to the data known at the location to which it was delivered. For the results in this section let us define the following particle,

$$p_A = \mathsf{Local}\langle \mathbf{nil}, \emptyset, |\mathsf{deliver}|(\delta) \ @ \ (n3, \mathsf{Bob}), \mathbf{nil}\rangle_{\mathsf{Alice}}.$$

This carries a chunk that causes δ to be added to Bob's known data.

We will first examine properties of the internetwork in Figure 1. We assume that s is a DataState such that $\delta \notin s(l)$ unless $l = \mathsf{Alice}$. That is, only Alice knows δ. We assume that the RouteState t provides shortest routes between all destinations. That is, a packet moving according to the routing tables reaches its destination after traversing as few networks as the topology allows. In particular, this means the routing tables direct a packet from Alice to Bob along the path Alice, $r1$, $r2$, Bob. We assume that z is the empty dictionary, and Σ includes only the deliver service.

The following states that if, after Alice initiates the transmission of δ to Bob, there are no particles generated (using the generation rule) by the controlled locations, then there is no computation in which Eve manages to learn δ. In this and subsequent results, Con denotes the controlled locations, and all other locations are assumed to be uncontrolled. In this first example all locations are included in Con.

Observation 1. *Let*

$$\mathsf{Con} = \{\mathsf{Alice}, \mathsf{Bob}, \mathsf{Claire}, \mathsf{Eve}, r1, r2, r3\}$$

and suppose controlled nodes do not generate any particles. If

$$\{t, s, z, \Sigma, p_A\} \longrightarrow^* \{t', s', z', \Sigma, p_1, \ldots, p_i\},$$

then $\delta \notin s'(\mathsf{Eve})$.

Table 4: Example of a Computation

$$
\begin{array}{l}
\{t, s, z, \Sigma, p_A\}\longrightarrow\\
\{t, s, z, \Sigma,\\
\quad \mathsf{Local}\langle \mathbf{nil}, \emptyset, |\mathsf{deliver}|(\delta) :: (n3, \mathsf{Bob}) :: @, \mathbf{nil}\rangle_{\mathsf{Alice}}\}\longrightarrow\\
\{t, s, z, \Sigma,\\
\quad \mathsf{Local}\langle \mathbf{nil}, \emptyset, \delta :: \mathbf{ch} :: \mathsf{deliver} :: (n3, \mathsf{Bob}) :: @, \mathbf{nil}\rangle_{\mathsf{Alice}}\}\longrightarrow^*\\
\{t, s, z, \Sigma, \mathsf{Transit}\langle (n3, \mathsf{Bob}), \mathsf{chunk}(\mathsf{deliver}, \delta, \emptyset)\rangle_{r1}\}\longrightarrow\\
\{t, s', z, \Sigma, \mathsf{Transit}\langle (n3, \mathsf{Bob}), \mathsf{chunk}(\mathsf{deliver}, \delta, \emptyset)\rangle_{r2}\}\longrightarrow\\
\{t, s'', z, \Sigma, \mathsf{Transit}\langle (n3, \mathsf{Bob}), \mathsf{chunk}(\mathsf{deliver}, \delta, \emptyset)\rangle_{\mathsf{Bob}}\}\longrightarrow\\
\{t, s'', z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, \emptyset, \mathsf{deliver}(\delta), \mathbf{nil}\rangle_{\mathsf{Bob}}\}\longrightarrow\\
\{t, s'', z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, \emptyset, \delta :: \mathbf{serv} :: \mathsf{deliver}, \mathbf{nil}\rangle_{\mathsf{Bob}}\}\longrightarrow\\
\{t, s'', z, \Sigma, \mathsf{Local}\langle \delta, \emptyset, \mathbf{serv} :: \mathsf{deliver}, \mathbf{nil}\rangle_{\mathsf{Bob}}\}\longrightarrow\\
\{t, s'', z, \Sigma, \mathsf{Local}\langle (), \emptyset, \mathbf{nil}, \mathbf{nil}\rangle_{\mathsf{Bob}}\}\longrightarrow\\
\{t, s'', z, \Sigma\}
\end{array}
$$

The proof of the observation is straightforward, noting that the only computations possible with these assumptions are initial segments of the one shown in Table 4, where s' is the same as s except $\delta \in s'(\mathsf{Claire})$, and s'' is the same as s' except $\delta \in s'(\mathsf{Bob})$. A very similar observation can also be stated using an initial assumption that no packets are present, and controlled nodes generate only the packet p_A. Note that $t = t'$ and $z = z'$ since there is no interface for changing the routing tables or dictionary.

The following states the same thing as Observation 1, except this time Eve is an uncontrolled location and thus can initiate transmissions. This extra capability is not enough for Eve to learn the secret being sent from Alice to Bob, even though Eve is able to send packets onto any of the nodes along the path and have them come back to her using source routing.

Observation 2. *Let*

$$\mathsf{Con} = \{\mathsf{Alice}, \mathsf{Bob}, \mathsf{Claire}, r1, r2, r3\}$$

and suppose controlled nodes do not generate any particles. If

$$\{t, s, z, \Sigma, p_A\}\longrightarrow^*\{t', s', z', \Sigma, p_1, \ldots, p_i\},$$

then $\delta \notin s'(\mathsf{Eve})$.

However, an outsider located outside of the topological path of a communication can compromise security with the aid of an insider on the path. The outsider does not even have to participate in the computation.

Observation 3. *Let*

$$\mathsf{Con} = \{\mathsf{Alice}, \mathsf{Bob}, \mathsf{Eve}, r1, r2, r3\}$$

and assume and suppose controlled nodes do not generate any particles. Then there is a state s' such that

$$\{t, s, z, \Sigma, p_A\} \longrightarrow^* \{t, s', z, \Sigma\}$$

and $\delta \in s'(\mathsf{Eve})$.

These observations can be generalized to an arbitrary topology. Before doing this, let us be a little more rigorous about the definition of a route.

Definition: A *route* determined by a $\mathsf{RouteState}$ t is a sequence of addresses $a_1 = (n_1, l_1), \ldots, a_p = (n_p, l_p)$ where $(a_{j+1}, i) = t(l_j)(a_p)$ and $l_j \neq l_p$ whenever $j < p$. p is the length of the route.

It is possible that there is *no* route between a pair of locations, but it is unique if it exits. The number of addresses on a route is its *length.* A topology is connected if it is connected as a graph. In such a topology, routing tables can be configured so that there is a route between any pair of locations. It will be convenient to ignore the difference between a location and an address when the location is a host, since a host determines a unique address.

The theorem simply says that a datum cannot be learned by a location unless the routing tables move it across a network to which the location is attached. Recall that $\mathsf{Topology}(n)$ is the set of locations attached to network n.

Theorem 4. *Let l_1 and l_2 be hosts in an arbitrary internetwork and suppose that l_2 is attached to network n_2. Let*

$$p_L = \mathsf{Local}\langle \mathbf{nil}, \emptyset, |\mathsf{deliver}|(\delta) \ @ \ (n_2, l_2), \mathbf{nil}\rangle_{l_1}$$

and suppose $\delta \notin s(l)$ for all $l \neq l_1$. Suppose there is a route R between l_1 and l_2 and let

$$\mathsf{Con} = \{l \mid (n, l') \in R \text{ and } l \in \mathsf{Topology}(n)\}.$$

Suppose controlled nodes do not generate any particles and

$$\{t, s, z, \Sigma, p_L\} \longrightarrow^* \{t', s', z', \Sigma, p_1, \ldots, p_i\}.$$

Then $\delta \in s'(l)$ only if $l \in \mathsf{Con}$.

The proof of this result and a selection of other results from the paper can be found in an appendix.

A few remarks are in order about the limitations of Theorem 4. We have assumed here that the router tables do not change. In practice, if a network or router fails, the routing tables will be altered to find new routes and packets may consequently pass along new routes. Eve might even be able to make this happen by sending so much traffic that a router misses enough transmissions to conclude that an adjacent router or network is down and modify its routes accordingly. To model this kind of attack would require an extension of our framework. We can, however, model the case in which router tables change because of updates, and Eve may be able to use these updates.

4.2 Dynamic Routing Tables

Routers attempt to be robust with respect to temporary or permanent link changes by periodically contacting their neighbors to exchange information about routes. These updates can be attacked by an adversary seeking to mislead routers about the internetwork topology to defeat AO-security. To keep things concrete we focus on using a specific routing protocol for our analysis, although analogous results probably hold for most protocols in current use. In distance vector routing, the routers keep a next hop address for each destination and a distance estimate. The routers periodically provide their current estimates to their neighbors (locations l_1, l_2 are neighbors if there is a network n such that $l_1, l_2 \in \mathsf{Topology}(n)$) and tables are updated to reflect newly-learned path estimates. Details of these kinds of protocols can be found in [10, 11]; we will suppress some of the details to simplify our exposition here.

A RouteState t is said to have *shortest routes* if there is a route between any pair of locations and, for any such pair l_1, l_2, there is no RouteState t' that provides a shorter path. Distance vector routing finds a shortest path routing table using the following protocol. Each router r periodically creates particles

$$\mathsf{Local}\langle \mathbf{nil}, \emptyset, |\mathsf{routeUpdate}|(r, a_d, i) \ @ \ (n, r'), \mathbf{nil}\rangle_r$$

where a_d is an intended destination, $t(r)(a_d) = (a, i)$, and r' is a neighbor on the common network n. These are called *advertisements* and they indicate distance estimates to the destination a_d. Upon receiving such an advertisement, the router r' invokes a service routeUpdate which may change the routing table entry for a_d at r'. This service takes as its arguments the router r generating the advertisement, the destination a_d that the advertisement concerns, and the distance i where $t(r)(a_d) = (a, i)$. Then r' uses this information to improve its estimate for a route to a_d. The salient feature of this protocol is that it eventually calculates shortest path routes and *stabilizes.* That is, once shortest path routes are computed, the advertisements do not change routes.

As in the previous subsection, let us begin by analyzing cases for the internetwork in Figure 1. Eve is able to alter the routing tables to suit her purposes in defeating AO-security—she simply sends routing update messages to $r1$ and $r3$ that cause all packets meant for Bob to be routed to Eve. (Eve could discard those packets or obscure her eavesdropping by source-routing them to Bob.) The routers exchange routing updates periodically and may therefore change the tables back. However, it is possible for Eve to get in her updates at the right time so the packet containing δ is routed to her instead of Bob, making it possible for her to learn δ. That is, there is (at least) one possible computation where Eve can cause routing tables to be corrupted for a long enough period that she learns the secret.

Observation 5. *Let*

$$\mathsf{Con} = \{\mathsf{Alice}, \mathsf{Bob}, \mathsf{Claire}, r1, r2, r3\}$$

and $\Sigma = \{\mathsf{deliver}, \mathsf{routeUpdate}\}$. *Assume that controlled nodes generate only router advertisements. Then there exist* t', s', z' *such that*

$$\{t, s, z, \Sigma, p_A\} \longrightarrow^* \{t', s', z', \Sigma\}$$

and $\delta \in s'(\mathsf{Eve})$.

The addition of the routeUpdate service enables an attacker to defeat AO-security for δ.

Let us now consider an alternate service, routeUpdateP, which is used to change routing tables, but only after a password check. It takes four arguments, with the first three being the same as in routeUpdate, and the fourth being a password. We assume a mapping from networks to passwords that associates a password p_n to each network n. An invocation of routeUpdateP is *valid* for a router that receives it from network n if, and only if, the supplied password is the password p_n assigned to n. As a simplification, we will assume that this password is also a member of Data, and is established before computations begin between routers attached to a given network.

The analog of Observation 5 will fail when passwords are used to protect routing updates since Eve will be unable to obtain a password. However, there is a threat for the internetwork in Figure 2. In this topology, Eve is on a network that has two routers attached to it, so updates contain passwords that can be sniffed by Eve. Although Eve cannot influence $r1$ directly, she can do so indirectly, by introducing a spurious entry on $r3$'s routing table that claims that Bob is 0 hops away. The next time $r3$ sends an update to $r1$, it will ask $r1$ to change the routing table entry for Bob to $r3$ if the current entry shows a route length greater than 1 and this will be true in this internetwork. We again depend on Eve's ability to make the spurious updates in time.

Observation 6. *Let*

$$\mathsf{Con} = \{\mathsf{Alice}, \mathsf{Bob}, \mathsf{Claire}, r1, r2, r3\}$$

and $\Sigma = \{\mathsf{deliver}, \mathsf{routeUpdateP}\}$. *Assume that controlled nodes generate only valid router advertisements. Then there exist* t', s', z' *such that*

$$\{t, s, z, \Sigma, p_A\} \longrightarrow^* \{t', s', z', \Sigma\}$$

and $\delta \in s'(\mathsf{Eve})$.

Thus even though it is harder for Eve to learn δ in the presence of passwords, she manages to do so. Notice however, that she was only able to do so because she happened to be on a network where advertisements could be seen. If this had not been true, and if routers followed their protocol, then Eve would not have been able to learn δ. The following theorem generalizes this property:

Theorem 7. *Let* l_1 *and* l_2 *be hosts in an arbitrary internetwork, and assume that* l_1 *and* l_2 *are attached to networks* n_1 *and* n_2 *respectively. Let*

$$p_L = \mathsf{Local}\langle \mathbf{nil}, \emptyset, |\mathsf{deliver}|(\delta) \ @ \ (n_2, l_2), \mathbf{nil}\rangle_{l_1}.$$

Suppose $\delta \notin s(l)$ for each $l \neq l_1$, and $\kappa_n \in s(l)$ if and only if $l \in \mathsf{Topology}(n)$. Let $\mathcal{F}$ be the set of all locations that are only connected to networks containing a unique router. Let $\mathsf{Con} = (\mathsf{Loc} - \mathcal{F}) \cup \mathsf{Topology}(n_1) \cup \mathsf{Topology}(n_2)$. Let $\Sigma = \{\mathsf{deliver}, \mathsf{routeUpdateP}\}$. If routers generate only valid advertisements and t provides shortest routes, and

$$\{t, s, z, \Sigma, p_L\} \longrightarrow^* \{t', s', z', \Sigma, p_1, \ldots, p_i\},$$

then, for all $l \in \mathcal{F}$, $\delta \notin s'(l)$.

The theorem relies on the assumption that valid advertisements produced by routers do not change routes because shortest routes have been computed. Uncontrolled locations will be unable to create valid advertisements because they cannot obtain passwords.

It is non-trivial to formulate a converse for the implication in this theorem. In particular, Eve may have access to router control information but still be unable to learn δ if she is 'too far away' from the communication between Alice and Bob. That is, the topology of the network makes a difference.

5 Bisimilarity and Secrecy

The examples in the previous section illustrate that proofs of properties that express the secrecy (or not) of data in the internetwork are possible. We now formalize two notions of secrecy for internetworks and prove their equivalence. These results apply to our system *without* the generation rule. That is, all particles derive from those originally present. Future work will generalize these results so that this restriction is not needed. The formalizations rely on the concept of AO-security. Here, l is the outsider against whom we want to secure δ.

Definition: A machine state $M = \{t, s, z, \Sigma, p_1, \ldots, p_k\}$ provides *δ-secrecy against l* if there is no M' such that $M' = \{t', s', z', \Sigma, p'_1, \ldots, p'_{k'}\}$ and $M \longrightarrow^* M'$ and $\delta \in s'(l)$.

We now define an alternative notion of secrecy that is analogous to the definition of secrecy used in the spi calculus [4]. The idea that this definition expresses is that a datum remains secret to an external observer if that observer's view of the internetwork is independent of the datum. In particular, changing the datum being transmitted does not change the observer's view of the internetwork. The motivation for such a requirement is that any difference in the behavior of the internetwork could form a covert channel. These ideas go back to treatments of secrecy in terms of noninterference and related conditions.

In order to formalize this idea, we first define an observer's view of the internetwork in terms of the packets transmitted on any network the observer is attached to. We do this through a labeled transition system, which is derived from the chemical abstract machine defined earlier by partitioning the transition relation defined in Table 3 into two categories. A transition $M \longrightarrow M'$ *is* *observable at l* if it involved application of the rule

$$\begin{array}{l}\mathsf{Transit}\langle a, \mathsf{chunk}(x, v, E)\rangle_{l'}, t, s \longrightarrow \\ \quad \mathsf{Transit}\langle a, \mathsf{chunk}(x, v, E)\rangle_{l''}, t, s'\end{array}$$

where $t(l')(a) = ((n, l''), i)$ and $l \in \mathsf{Topology}(n)$. All other transitions are not observable at l. In the rest of this section, we use $M \twoheadrightarrow M'$ for non-observable transitions, and $M \overset{p}{\twoheadrightarrow} M'$ for observable transitions of the form above, where $p = \mathsf{Transit}\langle a, \mathsf{chunk}(x, v, E)\rangle_{l'}$.

We now define the notion of l-bisimulation:

Definition: A relation R between machine states is an *l-bisimulation* if M_1 R M_2 implies that whenever $M_1 \twoheadrightarrow^* \overset{p}{\twoheadrightarrow} \twoheadrightarrow^* M_1'$, then there is some M_2' such that $M_2 \twoheadrightarrow^* \overset{p}{\twoheadrightarrow} \twoheadrightarrow^* M_2'$ and M_1' R M_2', and vice versa.

We define l-equivalence $\sim_l$ to be the largest l-bisimulation. Thus, $M \sim_l M'$ if and only if there is an l-bisimulation R such that M R M'. The relation $\sim_l$ is an equivalence. We have the following

Theorem 8. *Let $M = \{t, s, z, \Sigma, p_1, \ldots, p_i\}$ and suppose $\delta \notin s(l)$ and $\delta' \notin M$. If $M \sim_l M[\delta'/\delta]$ then M provides δ-secrecy against l.*

That is, if l's view of the internetwork is the same even when the parties exchanging secret data use different data, then l does not learn the data. The converse is more involved, and requires the following restriction on the kinds of services that a program in uPLAN can invoke:

Definition: A service σ is *secrecy-friendly* if, for every v, l, s, t, S, E, C, D such that $\delta' \notin s, v, S, E, C, D$ and controlled location l,

$$\begin{array}{r}([\![\sigma]\!](v, l, s, t, a, (S, E, C, D)))[\delta'/\delta]) = \\ (s', t', a', p_1, \ldots, p_i)[\delta'/\delta]\end{array}$$

where $[\![\sigma]\!](v, l, s, t, z, (S, E, C, D)) = s', t', z', p_1, \ldots, p_i$.

This definition expresses the requirement that services are transparent with respect to data. With this tool in hand, we obtain a converse of the previous theorem:

Theorem 9. *If $M = \{t, s, z, \Sigma, p_1, \ldots, p_i\}$ provides δ-secrecy against l and $\delta' \notin M$ and all services in Σ are secrecy-friendly, then $M \sim_l M[\delta'/\delta]$*

6 Active Labeled Routing

This section studies AO-security for internetworks in which the routers offer the ability to set labeled routes. The basic idea of labeled routes is to move a packet along a route based on a label it carries, rather than according to the routing table entries for a final destination. Instead of carrying the entire route in the packet itself (as in source routing), the packets carry only a label. Before sending any packets that use that label, a source could set up a route by leaving 'bread crumbs' associating labels to intermediate destinations. This gives the source more control over the route taken by the packets it sends. This control can be exploited to create customized routes that reduce congestion, improve security or reliability, and contribute to other objectives. In scenarios where a number of packets are expected to use the same label, this scheme can be more efficient than source routing and more flexible than default routing using RIP or OSPF. For example, the PLANet [13] testbed was used to study a protocol called *Flow-Based Adaptive Routing (FBAR)* where diagnostic packets would collect information about congestion, use this to determine custom routes around congested links, and configure labels to provide alternate routing. In labeled routing, as in source routing, the next destination supplied by the label need not be an address on an attached network—as it is in a routing table—since default routing (using the routing tables) can be used to get the packet to any destination specified by the label. In the FBAR experiment, routing provided by RIP was used as a default, with customization provided by labels when improvement on the default seemed possible.

The FBAR experiment raised some questions about the security of labels that can be customized by sources. As we have seen, the ability to modify routing table entries raises concerns about AO-security. Will packets using labeled routes be equally insecure? In fact, the labels, if they are as unguessable as passwords, will provide security for labeled routing protocols like FBAR that is generally comparable to that of password-protected routing updates like those considered in Section 4. To illustrate this, let us state some simple observations concerning AO-security for the internetwork in Figure 1 when locations offer labeled routing as a service. In order to express this routing scheme, consider the services in Table 5.

Table 5: Services for Labeled Routing

set: $[\![\mathsf{set}]\!]((\mathtt{lab}, a), l, s, t, z, (S, E, C, D)))$
$= \mathsf{Local}\langle () :: S, E, C, D\rangle_l, s, t, z'$
where $z'(l')(\mathtt{lab}') =$
if $l' = l$ and $\mathtt{lab}' = \mathtt{lab}$ then a else $z(l')(\mathtt{lab}')$

get: $[\![\mathsf{get}]\!](\mathtt{lab}, l, s, t, z, (S, E, C, D))$
$= \mathsf{Local}\langle z(l)(\mathtt{lab}) :: S, E, C, D\rangle_l, s, t, z$

getLabels: $[\![\mathsf{getLabels}]\!]((), l, s, t, z, (S, E, C, D))$
$= \mathsf{Local}\langle \mathsf{Domain}_l(z) :: S, E, C, D\rangle_l, s, t, z$

In order to set up a route, Alice can insert entries into a dictionary at locations on the way (starting with Alice herself) associating the label with the next hop from that location towards the intended final destination, say Bob. This route could be based on any metric that suits Alice. Let us assume here that it is the shortest route. Thus, a dictionary entry at Alice associates the label being used for this route with (n1,r1), an entry at $r1$ associates the label with (n2,r2), and one at $r2$ associates the label with (n3,Bob). Here's a uPLAN program that uses the deliver service defined earlier in conjunction with labeled routing to route a datum from Alice to Bob:

```
fun labelRoute(d,l,c,dest) =
  (c = dest) ?
    (deliver (d)),
    (|labelRoute|(d,l,#2(get(l)),dest))  @  (get(l))
in
labelRoute(δ,lab,Alice,Bob)
```

where δ is the datum to be routed to Bob, and lab is the label that was used to setup the labeled route from Alice to Bob. Call this program e_L. It is similar to source routing, but there is no need for a list of intermediate destinations in the packet. Once the labeled route has been set up, injecting the particle $p = \mathsf{Local}\langle \mathbf{nil}, \emptyset, e_L, \mathbf{nil}\rangle_{\mathsf{Alice}}$ causes δ to be routed to Bob according to the labeled route.

Consider a scenario in which Eve can get a list of all labels in use at $r1$. She can then send packets to $r1$ and try each label in sequence to change the dictionary entries so that the particle containing δ gets routed instead to Eve. This gives Eve a way to learn δ. The following formalizes this property. Here, let s be a DataState such that $\delta \notin s(l)$ unless $l = \mathsf{Alice}$, and $\mathtt{lab} \notin s(\mathsf{Eve})$. That is, only Alice knows δ, and Eve does not know the label used to set up the labeled route. Also, let z be a Dictionary such that $z(\mathsf{Alice})(\mathtt{lab}) = r1$, $z(r1)(\mathtt{lab}) = r2$, $z(r2)(\mathtt{lab}) = \mathsf{Bob}$, and all other dictionary entries are undefined.

Observation 10. *Let*

$$\begin{aligned}&\mathsf{Con} = \{\mathsf{Alice}, \mathsf{Bob}, \mathsf{Claire}, r1, r2, r3\}\\ &\Sigma = \{\mathsf{set}, \mathsf{get}, \mathsf{getLabels}, \mathsf{deliver}\}\\ &p = \mathsf{Local}\langle \mathbf{nil}, \emptyset, e_L, \mathbf{nil}\rangle_{\mathsf{Alice}}\end{aligned}$$

and assume that the controlled nodes do not generate particles. Then $\{s, t, z, \Sigma, p\}$ does not provide δ-secrecy against Eve*.*

However, if such a facility is not available to Eve, then Eve cannot learn δ. Thus by restricting the service interface appropriately, we can prove guarantees about secrecy when labeled routing is in use. The following formalizes this property:

Observation 11. *Let*

$$\begin{aligned}&\mathsf{Con} = \{\mathsf{Alice}, \mathsf{Bob}, \mathsf{Claire}, r1, r2, r3\}\\ &\Sigma = \{\mathsf{set}, \mathsf{get}, \mathsf{deliver}\}\\ &p = \mathsf{Local}\langle \mathbf{nil}, \emptyset, e_L, \mathbf{nil}\rangle_{\mathsf{Alice}}\end{aligned}$$

and assume that the controlled nodes do not generate particles. Then $\{s, t, z, \Sigma, p\}$ *provides* δ*-secrecy against* Eve*.*

We can generalize this property in two directions. The internetwork under consideration can be arbitrary, and the labels can point to addresses that are several hops away, requiring the use of routing tables which might be corrupted. First, we prove a theorem about an arbitrary network topology and a single hop labeled route - i.e., a labeled route in which each label points to a location that is only one hop away.

In the following, $e_L(\delta, \texttt{lab}, l, l')$ is the program:

```
fun labelRoute(d,l,c,dest) =
  (c = dest) ?
    (deliver (d)),
    (|labelRoute|(d,l,#2(get(l)),dest))  @  (get(l))
in
labelRoute(δ,lab,l,l')
```

Theorem 12. *Let* l_0 *be a host attached to the network* n_1 *in an arbitrary internetwork. Let* $a_1 = (n_1, l_1), \ldots, a_k = (n_k, l_k)$ *be the sequence such that* $l_i \in \mathsf{Topology}(n_{i+1})$ *and* $z(l_i)(\texttt{lab}) = (n_{i+1}, l_{i+1})$*, where* $\texttt{lab} \in \mathsf{Data}$*, and* z *is a dictionary in which all other entries are undefined. Let*

$$\Sigma = \{deliver, get, set\}$$
$$p_L = \mathsf{Local}\langle \mathbf{nil}, \emptyset, e_L(\delta, \texttt{lab}, l_0, l_k), \mathbf{nil}\rangle_{l_1}$$
$$\mathsf{Con} = \bigcup_i \mathsf{Topology}(n_i)$$

Suppose $\delta, \texttt{lab} \notin s(l)$ *for all* $l \notin \mathsf{Con}$ *and suppose that controlled nodes do not generate new particles. Then* $\{s, t, z, \Sigma, p_L\}$ *provides* δ*-secrecy against all* $l \notin \mathsf{Con}$*.*

We now include labeled routes that span multiple hops. Between two locations on the labeled route, the packet would then make use of the routing tables to travel to the next hop. However, now the routeUpdateP service gives an attacker the capability to corrupt routing tables provided the attacker has access to a routing password. As in the last section, κ_n is the routing password for network n, and will be known to all locations on the network n.

Theorem 13. *Let* l_0 *be a host attached to the network* n_1 *in an arbitrary internetwork. Let* $a_1 = (n_1, l_1), \ldots, a_k = (n_k, l_k)$ *be the sequence such that* $z(l_i)(\texttt{lab}) = (n_{i+1}, l_{i+1})$*, where* $\texttt{lab} \in \mathsf{Data}$*, and* z *is a dictionary in which all other entries are undefined. Let*

$$\Sigma = \{\mathsf{deliver}, \mathsf{get}, \mathsf{set}, \mathsf{routeUpdateP}\}$$
$$p_L = \mathsf{Local}\langle \mathbf{nil}, \emptyset, e_L(\delta, \texttt{lab}, l_0, l_k), \mathbf{nil}\rangle_{l_1}$$

Let $\mathcal{F}$ *be the set of all locations that are only connected to networks containing a unique router. Let* $\mathsf{Con} = (\mathsf{Loc} - \mathcal{F}) \cup \mathsf{Topology}(n_1) \cup \mathsf{Topology}(n_2)$ *and assume controlled nodes do not generate any more particles. Suppose* $\delta, \texttt{lab} \notin s(l)$ *for each* $l \notin \mathsf{Con}$*, and* $\kappa_n \in s(l)$ *if and only if* $l \in \mathsf{Topology}(n)$*. Let* t *be a shortest-path routing table. Then* $\{s, t, z, \Sigma, p_L\}$ *provides* δ*-secrecy against all* $l \notin \mathsf{Con}$*.*

7 Conclusions

In this paper we have defined uPLAN as a primitive programming interface for active networks and used it to demonstrate how to reason about the secrecy of passwords and labels and the integrity of routers. We believe the approach is simple and direct enough to be used routinely for analyzing interfaces while being powerful enough to work for a variety of interesting protocols. We have also provided a formal treatment of AO-security, which is an important attribute of network protocols. AO-Security plays a significant role in current practice, and it is not always trivial to ensure—as this paper demonstrates in the context of active networks.

Acknowledgment

We appreciated the help we received on this paper from Roch Guerin, Mike Hicks, Trevor Jim, and Jon Moore. This work was supported by DARPA under Contract #N66001-96-C-852 and ONR under Contracts N00014-99-1-0403 and N00014-00-1-0641.

A Proofs of various statements

A.1 Theorem 4

Let us examine the sequence of evaluation of p_L, that is sent from l_1 to l_2. Assuming that no other particles are generated, Table 6 shows the evaluation of p_L.

Table 6: Evaluation of particle p_L

$$\begin{array}{l}
\{t, s, z, \Sigma, p_L\}\longrightarrow \\
\{t, s, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, \emptyset, |\mathsf{deliver}|(\delta) :: (n_2, l_2) :: @, \mathbf{nil}\rangle_{l_1}\}\longrightarrow \\
\{t, s, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, \emptyset, \delta :: \mathbf{ch} :: \mathsf{deliver} :: (n_2, l_2) :: @, \mathbf{nil}\rangle_{l_1}\}\longrightarrow \\
\{t, s, z, \Sigma, \mathsf{Local}\langle \delta, \emptyset, \mathbf{ch} :: \mathsf{deliver} :: (n_2, l_2) :: @, \mathbf{nil}\rangle_{l_1}\}\longrightarrow \\
\{t, s, z, \Sigma, \mathsf{Local}\langle \mathsf{chunk}(\mathsf{deliver}, \delta, \emptyset), \emptyset, (n_2, l_2) :: @, \mathbf{nil}\rangle_{l_1}\}\longrightarrow \\
\{t, s, z, \Sigma, \mathsf{Local}\langle (), \emptyset, \mathbf{nil}, \mathbf{nil}\rangle_{l_1}, \mathsf{Transit}\langle (n_2, l_2), \mathsf{chunk}(\mathsf{deliver}, \delta, \emptyset)\rangle_{l_1}\}\longrightarrow \\
\{t, s, z, \Sigma, \mathsf{Transit}\langle (n_2, l_2), \mathsf{chunk}(\mathsf{deliver}, \delta, \emptyset)\rangle_{l_1}\}\longrightarrow^* \\
\{t, s^\delta, z, \Sigma, \mathsf{Transit}\langle (n_2, l_2), \mathsf{chunk}(\mathsf{deliver}, \delta, \emptyset)\rangle_{l_2}\}\longrightarrow \\
\{t, s^\delta, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, \emptyset, \mathsf{deliver}(\delta), \mathbf{nil}\rangle_{l_2}\}\longrightarrow \\
\{t, s^\delta, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, \emptyset, \delta :: \mathbf{serv} :: \mathsf{deliver}, \mathbf{nil}\rangle_{l_2}\}\longrightarrow \\
\{t, s^\delta, z, \Sigma, \mathsf{Local}\langle \delta, \emptyset, \mathbf{serv} :: \mathsf{deliver}, \mathbf{nil}\rangle_{l_2}\}\longrightarrow \\
\{t, s^\delta, z, \Sigma, \mathsf{Local}\langle (), \emptyset, \mathbf{nil}, \mathbf{nil}\rangle_{l_2}\}\longrightarrow \\
\{t, s^\delta, z, \Sigma\}
\end{array}$$

In Table 6, s^δ is the same as s except all locations on a network on the route

R between l_1 and l_2 have now learned δ. Let

$$\mathcal{P}_L^1 = \{p \mid \{t, s, z, \Sigma, p_L\} \longrightarrow^* \{t, s', z, \Sigma, p\}\}$$

and

$$\mathcal{P}_L^2 = \{p \mid \{t, s, z, \Sigma, p_L\} \longrightarrow^* \{t, s', z, \Sigma, p, p'\}\},$$

and let $\mathcal{P}_L = \mathcal{P}_L^1 \cup \mathcal{P}_L^2$. Notice that $\mathcal{P}_L$ contains all the $\mathsf{Local}\langle \ldots \rangle$ particles from Table 6 along with $\mathsf{Transit}\langle \ldots \rangle$ particles of the following form:

$$\mathsf{Transit}\langle (n_2, l_2), \mathsf{chunk}(\mathsf{deliver}, \delta, \emptyset) \rangle_l$$

where l is a location on the route between l_1 and l_2. This is true because the particle p_L gets turned into a $\mathsf{Transit}\langle\rangle$ particle of the above form with l being l_1, and is intended to reach the address (n_2, l_2).

In the following, let $K(s) = \{l \mid \delta \in s(l)\}$. We now prove Theorem 4:

Proof of Theorem 4. Suppose $\{t, s, z, \Sigma, p_L\} \longrightarrow^* M = \{t', s', z', \Sigma, p_1, \ldots, p_k\}$. Then, all of the following must be true:

- $K(s) \subseteq \mathsf{Con}$.
- $t' = t$.
- For all $p \in M$, either $p \in \mathcal{P}_L$ or $\delta \notin p$.

We prove this by induction on the length of the computation $\{t, s, z, \Sigma, p_L\} \longrightarrow^* M$:

Base Case: If the length of the computation is 0, then $M = \{t, s, z, \Sigma, p_L\}$. The results follow directly from the assumptions in this case.

Induction Step: Let the hypothesis be true for a computation of j steps. Now, let us consider a computation involving $j + 1$ steps, and in particular look at the last transition.

Let $\{t, s, z, \Sigma, p_L\} \longrightarrow^* M' \longrightarrow M''$ where

$$\begin{array}{rcl} M' & = & \{t', s', z', \Sigma, p_1', \ldots, p_m'\} \text{ and} \\ M'' & = & \{t'', s'', z'', \Sigma, p_1'', \ldots, p_n''\}. \end{array}$$

Then we know from the induction hypothesis that $t' = t, K(s) \subseteq \mathsf{Con}$ and for all p', either $p' \in \mathcal{P}_L$ or $\delta \notin p'$.

We now need to verify that the induction hypothesis holds for each possible transition rule used in the last transition. We will not describe every case here, rather we will pick a representative set of rules. We thus examine five cases on the last transition, and claim that all rules except four are similar to the first case while the next four cases are special:

Case

$$\mathsf{Local}\langle v :: S, E, \mathbf{ch} :: \sigma :: C, D\rangle_k \longrightarrow \mathsf{Local}\langle \mathsf{chunk}(\sigma, v, E) :: S, E, C, D\rangle_k$$

Thus, there is some $p' \in M'$ and some $p'' \in M''$ such that $p' = \mathsf{Local}\langle v :: S, E, \mathbf{ch} :: \sigma :: C, D\rangle_k$ and $p'' = \mathsf{Local}\langle \mathsf{chunk}(\sigma, v, E) :: S, E, C, D\rangle_k$. Now there are two subcases:

- $p' \in \mathcal{P}_L$.
  From the definition of P, $\{t, s, z, \Sigma, p_L\} \longrightarrow^* \{t, s', z, \Sigma, p'\}$. Since $\{t, s', z, \Sigma, p'\} \longrightarrow \{t, s', z, \Sigma, p''\}$, we must have $p'' \in \mathcal{P}_L$.
- $p' \notin \mathcal{P}_L$.
  From the induction hypothesis then $\delta \notin p'$, and by an inspection of the rule, $\delta \notin p''$.

In both subcases, this rule does not change either the $\mathsf{DataState}$ or the $\mathsf{RouteState}$, thus $t'' = t' = t$ and $s'' = s'$ therefore $K(s) \subseteq \mathsf{Con}$.

Case

$$\begin{array}{l} \mathsf{Transit}\langle a, ch\rangle_l, t', s' \longrightarrow \mathsf{Transit}\langle a, ch\rangle_{l'}, t', s'' \\ \text{where } ch = \mathsf{chunk}(x, v, E) \text{ or } \mathsf{chunk}(\sigma, v, E) \\ \text{and}((n, l'), i) = t'(l)(a) \\ \text{and } s''(l'') = \left\{ \begin{array}{l} s'(l'') \text{ if } l'' \notin \mathsf{Topology}(n) \\ s'(l'') \cup \{\delta | \delta \in v \text{ or } \delta \in E\} \text{ otherwise} \end{array} \right. \end{array}$$

This rule changes the $\mathsf{DataState}$. Now, there must be some $p' \in M'$ and some $p'' \in M''$ such that $p' = \mathsf{Transit}\langle a, ch\rangle_l$ and $p'' = \mathsf{Transit}\langle a, ch\rangle_{l'}$. Notice that the rule does not change the $\mathsf{RouteState}$, hence $t'' = t' = t$. Again, there are two subcases:

- $p' \in \mathcal{P}_L$.
  Then l is a location on the route from l_1 to l_2. By the definition of a route, l' must also be a location on the route, and from the assumptions in the statement of the Theorem, if $l'' \in \mathit{Topology}(n)$ then $l'' \in \mathsf{Con}$.

  Now, p' must be $\mathsf{Transit}\langle (n_2, l_2), \mathsf{chunk}(\mathsf{deliver}, \delta, \emptyset)\rangle_l$, so $\delta \in p'$. If $l'' \in K(s'')$, then $\delta \in s''(l'')$, so either $\delta \in s'(l'')$ or $l'' \in \mathsf{Topology}(n)$ and in both cases $l'' \in \mathsf{Con}$, by the induction hypothesis for the former and from the preceding paragraph for the latter.

  Also by definition of $\mathcal{P}_L$, $p'' \in \mathcal{P}_L$.
- $p' \notin \mathcal{P}_L$.
  Then $\delta \notin p'$, so $\delta \in s''(l'')$ only if $\delta \in s'(l'')$ only if $l'' \in \mathsf{Con}$. Also, by inspection of the rule, $\delta \notin p''$.

Case

$$\begin{array}{l} s', \Sigma \longrightarrow \mathsf{Local}\langle \mathbf{nil}, \emptyset, e, \mathbf{nil}\rangle_l, s', \Sigma \\ \text{where } e \text{ } \mathsf{obeys} \text{ } s'(l) \\ \text{and all services } \sigma \text{ in } e \text{ are in } \Sigma \end{array}$$

This rule involves the generation of a new particle. The new particle is $\mathsf{Local}\langle \mathbf{nil}, \emptyset, e, \mathbf{nil}\rangle_l$. Since controlled nodes can't generate packets, $l \notin \mathsf{Con}$, so $l \notin K(s')$. But then $\delta \notin s'(l)$, hence $\delta \notin \mathsf{Local}\langle \mathbf{nil}, \emptyset, e, \mathbf{nil}\rangle_l$. Moreover $t'' = t' = t$ and $s'' = s'$ so $\delta \in s''(l)$ only if $l \in \mathsf{Con}$.

Case

$$\mathsf{Local}\langle a :: ch :: S, E, @ :: C, D\rangle_k \longrightarrow \mathsf{Local}\langle () :: S, E, C, D\rangle_k, \mathsf{Transit}\langle a, ch\rangle_k$$

This rule has two particles on the right hand side. There must be a $p' \in M'$ such that $p' = \mathsf{Local}\langle a :: ch :: S, E, @ :: C, D\rangle_k$, and $p'', p''' \in M''$ such that $p'' = \mathsf{Local}\langle () :: S, E, C, D\rangle_k$ and $p''' = \mathsf{Transit}\langle a, ch\rangle_k$. The two subcases are:

- $p' \in \mathcal{P}_L$.
  Then by definition of $\mathcal{P}_L$, $p'', p''' \in \mathcal{P}_L$. The $\mathsf{DataState}$ and $\mathsf{RouteState}$ do not change, so the other invariants are also satisfied.
- $p' \notin \mathcal{P}_L$.
  Then $\delta \notin p'$. By inspection of the rule, $\delta \notin p'', p'''$. Again, the other invariants are also satisfied.

Case

$$\mathsf{Local}\langle v :: S, E, \mathbf{serv} :: \sigma :: C, D\rangle_l, s, t, z \longrightarrow [\![\sigma]\!](v, l, s, t, z, (S, E, C, D))$$

This rule involves application of a service. $\sigma \in \Sigma = \{\mathsf{deliver}\}$, therefore $\sigma = \mathsf{deliver}$. We know that

$$[\![\mathsf{deliver}]\!](v, l, s, t, z, (S, E, C, D)) = \mathsf{Local}\langle () :: S, E, C, D\rangle_l, s', t, z$$

where

$$s'(l') = \begin{cases} s(l) \cup \{v\} & \text{if } l = l' \\ s(l) & \text{otherwise} \end{cases}$$

Now, there must be $p' \in M' = \mathsf{Local}\langle v :: S, E, \mathbf{serv} :: \sigma :: C, D\rangle_l$, and $p'' \in M'' = \mathsf{Local}\langle () :: S, E, C, D\rangle_l$. Let us examine the two subcases:

- $p' \in \mathcal{P}_L$.
  Then l must be l_2 and v must be δ. By definition of $\mathcal{P}_L$, $p'' \in \mathcal{P}_L$. Also, $K(s') = K(s) \cup l_2$. $K(s) \subseteq \mathsf{Con}$ and $l_2 \in \mathsf{Con}$ implies that $K(s') \subseteq \mathsf{Con}$.

- $p' \notin \mathcal{P}_L$.
  Then $\delta \notin p'$. By inspection of the rule, $\delta \notin p''$. Moreover, $v \neq \delta$, so $K(s') = K(s) \subseteq \mathsf{Con}$.

In both subcases, $t'' = t' = t$.

□

A.2 Theorem 7

Before we prove the Theorem, let us look at the sequence of evaluation of a valid routing update originating from router r, containing information about a destination address a_d, and meant for r', a neighbor on the network n. We will assume that the RouteState represents shortest path routing tables. Let

$$p_U(r, a_d, r') = \mathsf{Local}\langle \mathbf{nil}, \emptyset, |\mathsf{routeUpdateP}|(r, a_d, i, \kappa_n) \ @ \ (n, r'), \mathbf{nil}\rangle_r$$

where a_d is an intended destination, $t(r)(a_d) = (a, i)$, and κ_n is the routing password for network n. Assuming that no other particles are generated, Table 7 shows this sequence in a computation.

From Table 7, let

$$\mathcal{P}^1_U(r, a_d, r') = \{p \mid \{t, s, z, \Sigma, p_U(r, a_d, r')\} \longrightarrow^* \{t, s', z, \Sigma, p\}\}$$

and

$$\mathcal{P}^2_U(r, a_d, r') = \{p \mid \{t, s, z, \Sigma, p_U(r, a_d, r')\} \longrightarrow^* \{t, s', z, \Sigma, p, p'\}\},$$

and let $\mathcal{P}_U(r, a_d, r') = \mathcal{P}^1_U(r, a_d, r') \cup \mathcal{P}^2_U(r, a_d, r')$. Also, let $\mathcal{P}_L$ be defined as in the last section.

We first prove that all routers on the shortest path from l_1 to l_2 are controlled nodes as per the definition of controlled nodes in Theorem 7:

Lemma 14. *Let $\mathcal{F}$ be the set of all locations that are only connected to networks containing a unique router. Let* Con *be the set of all locations not in $\mathcal{F}$ except all locations on n_1 and n_2. Let R be any route from l_1 to l_2. If $(n, l) \in R$, then* $\mathsf{Topology}(n) \subseteq \mathsf{Con}$.

Proof. There are three cases on l:

- l is either l_1 or l_2.

  Then n is either n_1 or n_2 (respectively), so that $\mathsf{Topology}(n) \subseteq \mathsf{Con}$.

- l is neither l_1 nor l_2, and the route length is 3.

  Then l must be a router that is connected to both n_1 and n_2, and $n = n_1$, so again, $\mathsf{Topology}(n) \subseteq \mathsf{Con}$.

Table 7: Evaluation of a routing update

$$
\begin{array}{l}
\{t,s,z,\Sigma,p_U(r,a_d,r')\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle \mathbf{nil},\emptyset,|\mathsf{routeUpdateP}|(r,a_d,i,\kappa_n)::(n,r')::\ @\ ,\mathbf{nil}\rangle_r\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle \mathbf{nil},\emptyset,(r,a_d,i,\kappa_n)::\mathbf{ch}::\mathsf{routeUpdateP}::(n,r')::\ @\ ,\mathbf{nil}\rangle_r\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle \mathbf{nil},\emptyset,r::a_d::i::\kappa_n::\mathbf{tup}_4::\mathbf{ch}::\mathsf{routeUpdateP}::(n,r')::\ @\ ,\mathbf{nil}\rangle_r\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle r,\emptyset,a_d::i::\kappa_n::\mathbf{tup}_4::\mathbf{ch}::\mathsf{routeUpdateP}::(n,r')::\ @\ ,\mathbf{nil}\rangle_r\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle a_d::r,\emptyset,i::\kappa_n::\mathbf{tup}_4::\mathbf{ch}::\mathsf{routeUpdateP}::(n,r')::\ @\ ,\mathbf{nil}\rangle_r\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle i::a_d::r,\emptyset,\kappa_n::\mathbf{tup}_4::\mathbf{ch}::\mathsf{routeUpdateP}::(n,r')::\ @\ ,\mathbf{nil}\rangle_r\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle \kappa_n::i::a_d::r,\emptyset,\mathbf{tup}_4::\mathbf{ch}::\mathsf{routeUpdateP}::(n,r')::\ @\ ,\mathbf{nil}\rangle_r\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle (r,a_d,i,\kappa_n),\emptyset,\mathbf{ch}::\mathsf{routeUpdateP}::(n,r')::\ @\ ,\mathbf{nil}\rangle_r\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle \mathsf{chunk}(\mathsf{routeUpdateP},(r,a_d,i,\kappa_n),\emptyset),\emptyset,(n,r')::\ @\ ,\mathbf{nil}\rangle_r\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle (n,r')::\mathsf{chunk}(\mathsf{routeUpdateP},(r,a_d,i,\kappa_n),\emptyset,\ @\ ,\mathbf{nil}\rangle_r\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle (),\emptyset,\mathbf{nil},\mathbf{nil}\rangle_r,\mathsf{Transit}\langle (n,r'),\mathsf{chunk}(\mathsf{routeUpdateP},(r,a_d,i,\kappa_n))\rangle_r\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Transit}\langle (n,r'),\mathsf{chunk}(\mathsf{routeUpdateP},(r,a_d,i,\kappa_n))\rangle_r\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Transit}\langle (n,r'),\mathsf{chunk}(\mathsf{routeUpdateP},(r,a_d,i,\kappa_n))\rangle_{r'}\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle \mathbf{nil},\emptyset,\mathsf{routeUpdateP}(r,a_d,i,\kappa_n),\mathbf{nil}\rangle_{r'}\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle \mathbf{nil},\emptyset,(r,a_d,i,\kappa_n)::\mathsf{routeUpdateP}::\mathbf{serv},\mathbf{nil}\rangle_{r'}\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle \mathbf{nil},\emptyset,r::a_d::i::\kappa_n::\mathbf{tup}_4::\mathsf{routeUpdateP}::\mathbf{serv},\mathbf{nil}\rangle_{r'}\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle r,\emptyset,a_d::i::\kappa_n::\mathbf{tup}_4::\mathsf{routeUpdateP}::\mathbf{serv},\mathbf{nil}\rangle_{r'}\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle a_d::r,\emptyset,i::\kappa_n::\mathbf{tup}_4::\mathsf{routeUpdateP}::\mathbf{serv},\mathbf{nil}\rangle_{r'}\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle i::a_d::r,\emptyset,\kappa_n::\mathbf{tup}_4::\mathsf{routeUpdateP}::\mathbf{serv},\mathbf{nil}\rangle_{r'}\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle \kappa_n::i::a_d::r,\emptyset,\mathbf{tup}_4::\mathsf{routeUpdateP}::\mathbf{serv},\mathbf{nil}\rangle_{r'}\}\longrightarrow\\
\{t,s,z,\Sigma,\mathsf{Local}\langle r,a_d,i,\kappa_n),\emptyset,\mathsf{routeUpdateP}::\mathbf{serv},\mathbf{nil}\rangle_{r'}\}\longrightarrow\\
\{t',s,z,\Sigma,\mathsf{Local}\langle (),\emptyset,\mathbf{nil},\mathbf{nil}\rangle_{r'}\}\longrightarrow
\end{array}
$$

- l is neither l_1 nor l_2, and the route length is greater than 3.

  Then l must be a router. Since $(n, l) \in R, \exists l'$ s.t. $t(l')(n_2, l_2) = (n, l)$. If $l' = l_1$, then l must be connected to n_1, otherwise l' must be a router, and then l and l' are both routers on the same network. In both cases, $\mathsf{Topology}(n) \subseteq \mathsf{Con}$.

$\square$

In the following, let $K_\delta(s) = \{l \mid \delta \in s(l)\}$, and let $K_{\kappa_n}(s) = \{l \mid \kappa_n \in s(l)\}$.

Proof of theorem 7. Suppose $\{t, s, z, \Sigma, p_L\} \longrightarrow^* M = \{t', s', z', \Sigma, p_1, \dots, p_k\}$. Then all of the following must be true:

- $K_\delta(s') \subseteq \mathsf{Con}$.
- $K_{\kappa_n}(s') \subseteq \mathsf{Topology}(n) \cap \mathsf{Con}$ for all n.
- $t' = t$.
- For all $p \in M$, one of the following is true:
  - $p \in \mathcal{P}_L$.
  - $p \in \mathcal{P}_U(r, a_d, r')$ for some r, a_d, r' such that r and r' are neighbors.
  - $\delta \notin p$ and, for all networks n, $\kappa_n \notin p$.

Base Case: If the length of the computation is 0, then $M = \{t, s, z, \Sigma, p_L\}$. The results follow directly from the assumptions in this case.

Induction Step: Let the hypothesis be true for a computation of j steps. Now, let us consider a computation involving $j + 1$ steps, and in particular look at the last transition.

Let $\{t, s, z, \Sigma, p_L\} \longrightarrow^* M' \longrightarrow M''$ where

$$\begin{aligned} M' &= \{t', s', z', \Sigma, p'_1, \dots, p'_m\} \text{ and} \\ M'' &= \{t'', s'', z'', \Sigma, p''_1, \dots, p''_n\}. \end{aligned}$$

We now need to verify that the induction hypothesis holds for each possible transition rule used in the last transition. We will not describe every case here, rather we will pick a representative set of rules. We thus examine five cases on the last transition, and claim that all rules except four are similar to the first case while the next four cases are special:

Case

$$\mathsf{Local}\langle v :: S, E, \mathbf{ch} :: \sigma :: C, D\rangle_k \longrightarrow \mathsf{Local}\langle \mathsf{chunk}(\sigma, v, E) :: S, E, C, D\rangle_k$$

Thus, there is some $p' \in M'$ and some $p'' \in M''$ such that $p' = \mathsf{Local}\langle v :: S, E, \mathbf{ch} :: \sigma :: C, D\rangle_k$ and $p'' = \mathsf{Local}\langle \mathsf{chunk}(\sigma, v, E) :: S, E, C, D\rangle_k$. Now there are three subcases:

- $p' \in \mathcal{P}_L$.

  From the definition of $\mathcal{P}_L$, $\{t, s, z, \Sigma, p_L\} \longrightarrow^* \{t, s', z, \Sigma, p'\}$. Since $\{t, s', z, \Sigma, p'\} \longrightarrow \{t, s', z, \Sigma, p''\}$, we must have $p'' \in \mathcal{P}_L$.

- $p' \notin \mathcal{P}_L$, $p' \in \mathcal{P}_U(r, a_d, r')$ for some r, a_d and r'.

  Again, $p'' \in \mathcal{P}_U(r, a_d, r')$ from the definition of $\mathcal{P}_U(r, a_d, r')$.

- $p' \notin \mathcal{P}_L$, $p' \notin \mathcal{P}_U(r, a_d, r')$ for any r, a_d and r'.

  From the induction hypothesis then $\delta \notin p'$, and for all networks n, $\kappa_n \notin p'$. By an inspection of the rule, $\delta \notin p''$ and for all networks n, $\kappa_n \notin p''$.

In all subcases, this rule does not change either the $\mathsf{DataState}$ or the $\mathsf{RouteState}$, thus $t'' = t' = t$ and $s'' = s'$ therefore $K_\delta(s'') \subseteq \mathsf{Con}$ and $K_{\kappa_n}(s'') \subseteq \mathsf{Topology}(n) \cap \mathsf{Con}$ for all n.

Case

$$\begin{array}{l}
\mathsf{Transit}\langle a, ch\rangle_l, t', s' \longrightarrow \mathsf{Transit}\langle a, ch\rangle_{l'}, t', s'' \\
\text{where } ch = \mathsf{chunk}(x, v, E) \text{ or } \mathsf{chunk}(\sigma, v, E) \\
\text{and}((n, l'), i) = t'(l)(a) \\
\text{and } s''(l'') = \left\{ \begin{array}{l} s'(l'') \text{ if } l'' \notin \mathsf{Topology}(n) \\ s'(l'') \cup \{\delta | \delta \in v \text{ or } \delta \in E\} \text{ otherwise} \end{array} \right.
\end{array}$$

This rule changes the $\mathsf{DataState}$. Now, there must be some $p' \in M'$ and some $p'' \in M''$ such that $p' = \mathsf{Transit}\langle a, ch\rangle_l$ and $p'' = \mathsf{Transit}\langle a, ch\rangle_{l'}$. Notice that the rule does not change the $\mathsf{RouteState}$, hence $t'' = t' = t$. Again, there are three subcases:

- $p' \in \mathcal{P}_L$.

  Then l is a location on the route from l_1 to l_2. By the definition of a route, $(n, l') \in R$, and from Lemma 14, $\mathsf{Topology}(n) \subseteq \mathsf{Con}$.

  Now, p' must be $\mathsf{Transit}\langle(n_2, l_2), \mathsf{chunk}(\mathsf{deliver}, \delta, \emptyset)\rangle_l$, so $\delta \in p'$. If $l'' \in K_\delta(s'')$, then $\delta \in s''(l'')$, so either $\delta \in s'(l'')$ or $l'' \in \mathsf{Topology}(n)$ and in both cases $l'' \in \mathsf{Con}$, by the induction hypothesis for the former and from the preceding paragraph for the latter.

  Now, $\kappa_n \notin p'$, so if $l'' \in K_{\kappa_n}(s'')$ then $\kappa_n \in s''(l'')$, which is only true if $\kappa_n n) \in s'(l'')$, so $l'' \in \mathsf{Topology}(n) \cap \mathsf{Con}$.

  Also by definition of $\mathcal{P}_L$, $p'' \in \mathcal{P}_L$.

- $p' \notin \mathcal{P}_L$, $p' \in \mathcal{P}_U(r, a_d, r')$ for some r, a_d and r'.

  By inspection of Table 7, $l = r$ and $l' = r'$. r, r' are neighbors, i.e., there is a network n such that $r, r' \in \mathsf{Topology}(n)$. Since there are two routers on n, $\mathsf{Topology}(n) \subseteq \mathsf{Con}$.

  Now, $\kappa_n \in p'$. $l'' \in K_{\kappa_n}(s'')$ implies $\kappa_n \in s''(l'')$, then either $n = n'$ and $l'' \in \mathsf{Topology}(n')$ (thus $l'' \in \mathsf{Con}$), or $\kappa_n \in s'(l'')$. The latter implies (by the induction hypothesis) that $l'' \in \mathsf{Topology}(n')$ and $l'' \in \mathsf{Con}$ also, so either way, $K_{\kappa_n}(s'') \subseteq \mathsf{Topology}(n') \cap \mathsf{Con}$.

In this case, $\delta \notin p'$, so $l'' \in K_\delta(s'')$ implies that $\delta \in s''(l'')$ which is only true if $\delta \in s'(l'')$, so $l'' \in \mathsf{Con}$.

Also, by definition of $\mathcal{P}_U(r, a_d, r')$, $p'' \in \mathcal{P}_U(r, a_d, r')$.

- $p' \notin \mathcal{P}_L$, $p' \notin \mathcal{P}_U(r, a_d, r')$ for any r, a_d and r'.

  Then $\delta \notin p'$, so $\delta \in s''(l'')$ only if $\delta \in s'(l'')$ only if $l'' \in \mathsf{Con}$. Also, by inspection of the rule, $\delta \notin p''$. The same argument holds for all κ_n.

Case

$$\begin{array}{l} s', \Sigma \longrightarrow \mathsf{Local}\langle \mathbf{nil}, \emptyset, e, \mathbf{nil}\rangle_l, s', \Sigma \\ \text{where } e \ \mathsf{obeys}\ s'(l) \\ \text{and all services } \sigma \text{ in } e \text{ are in } \Sigma \end{array}$$

This rule involves the generation of a new particle. The new particle $p = \mathsf{Local}\langle \mathbf{nil}, \emptyset, e, \mathbf{nil}\rangle_l$. There are two subcases:

- $l \in \mathsf{Con}$. Then the new particle can only be a valid routing update, and then $p \in \mathcal{P}_U(r, a_d, r')$ for some r, a_d, r' such that r, r' are neighbors.
- $l \notin \mathsf{Con}$. Then $l \notin K_\delta(s')$ and $l \notin K_{\kappa_n}(s')$ for all n. But then $\delta \notin s'(l)$, and $\kappa_n \notin s'(l)$, hence $\delta, \kappa_n \notin \mathsf{Local}\langle \mathbf{nil}, \emptyset, e, \mathbf{nil}\rangle_l$.

In both cases, $t'' = t' = t$ and $s'' = s'$ so the other conditions also hold true.

Case

$$\mathsf{Local}\langle a :: ch :: S, E, @ :: C, D\rangle_k \longrightarrow \mathsf{Local}\langle () :: S, E, C, D\rangle_k, \mathsf{Transit}\langle a, ch\rangle_k$$

This rule has two particles on the right hand side. There must be a $p' \in M'$ such that $p' = \mathsf{Local}\langle a :: ch :: S, E, @ :: C, D\rangle_k$, and $p'', p''' \in M''$ such that $p'' = \mathsf{Local}\langle () :: S, E, C, D\rangle_k$ and $p''' = \mathsf{Transit}\langle a, ch\rangle_k$. The three subcases are:

- $p' \in \mathcal{P}_L$.

  Then by definition of $\mathcal{P}_L$, $p'', p''' \in \mathcal{P}_L$. The $\mathsf{DataState}$ and $\mathsf{RouteState}$ do not change, so the other invariants are also satisfied.
- $p' \notin \mathcal{P}_L$, $p' \in \mathcal{P}_U(r, a_d, r')$ for some r, a_d and r'.

  Again, by definition, $p'' \in \mathcal{P}_U(r, a_d, r')$. Again, other invariants are also satisfied.
- $p' \notin \mathcal{P}_L$, $p' \notin \mathcal{P}_U(r, a_d, r')$ for any r, a_d and r'.

  Then $\delta \notin p'$ and for all n, $\kappa_n \notin p'$. By inspection of the rule, $\delta \notin p'', p'''$ and for all n, $\kappa_n \notin p'', p'''$. Again, the other invariants are also satisfied.

Case

$$\mathsf{Local}\langle v :: S, E, \mathbf{serv} :: \sigma :: C, D\rangle_l, s', t', z' \longrightarrow [\![\sigma]\!](v, l, s', t', z', (S, E, C, D))$$

This rule involves application of a service. $\sigma \in \Sigma = \{\mathsf{deliver}, \mathsf{routeUpdateP}\}$. Let us examine the three subcases:

- $p' \in \mathcal{P}_L$.
  Now, $\sigma = \mathsf{deliver}$. We know that

  $$[\![\mathsf{deliver}]\!](v, l, s', t', z', (S, E, C, D)) = \mathsf{Local}\langle () :: S, E, C, D\rangle_l, s'', t', z'$$

  where
  $$s'(l') = \begin{cases} s(l) \cup \{v\} & \text{if } l = l' \\ s(l) & \text{otherwise} \end{cases}$$

  Now, there must be $p' \in M' = \mathsf{Local}\langle v :: S, E, \mathbf{serv} :: \sigma :: C, D\rangle_l$, and $p'' \in M'' = \mathsf{Local}\langle () :: S, E, C, D\rangle_l$.
  Then l must be l_2 and v must be δ. By definition of $\mathcal{P}_L$, $p'' \in \mathcal{P}_L$. Also, $K(s') = K(s) \cup l_2$. $K(s) \subseteq \mathsf{Con}$ and $l_2 \in \mathsf{Con}$ implies that $K(s') \subseteq \mathsf{Con}$.
  Moreover, $t'' = t' = t$, and since $\kappa_n \notin p'$, $\kappa_n \in s''(l'')$ implies that $\kappa_n \in s'(l'')$. Hence the other invariants also hold.
- $p' \notin \mathcal{P}_L$, $p' \in \mathcal{P}_U(r, a_d, r')$ for some r, a_d and r'.
  Now, $\sigma = \mathsf{routeUpdateP}$. Again, there must be $p' \in M' = \mathsf{Local}\langle v :: S, E, \mathbf{serv} :: \sigma :: C, D\rangle_l$, and $p'' \in M'' = \mathsf{Local}\langle () :: S, E, C, D\rangle_l$. Then $l = r'$. Since t' contains shortest path routing tables , the application of $\mathsf{routeUpdateP}$ does not change the $\mathsf{RouteState}$. Moreover, this service does not change the $\mathsf{DataState}$ so the other invariants are also satisfied.
- $p' \notin \mathcal{P}_L$, $p' \notin \mathcal{P}_U(r, a_d, r')$ for any r, a_d and r'.
  Then $\delta \notin p'$ and for all n, $\kappa_n \notin p'$. By inspection of the rule, $\delta \notin p''$ and, for all n, $\kappa_n \notin p''$. So $K_\delta(s') = K_\delta(s) \subseteq \mathsf{Con}$ and $K_{\kappa_n}(s') = K_{\kappa_n}(s) \subseteq \mathsf{Con} \cap \mathsf{Topology}(n)$.

□

A.3 Theorems 8 and 9

Proof of Theorem 8. Assume the converse of the conclusion. That is, there is an $M' = \{t', s', z, \Sigma, p'_1, \ldots, p'_{k'}\}$ such that $M \longrightarrow^* M'$ and $\delta \in s'(l)$. Since $\delta \notin s(l)$, there must have been states M_1 and M'_1 in such that $M \twoheadrightarrow^* M_1 \overset{p}{\twoheadrightarrow} M'_1 \twoheadrightarrow^* M'$ where $p = \mathsf{Transit}\langle a, \mathsf{chunk}(x, v, E)\rangle_{l'}$ and $\delta \in v$ or $\delta \in E$, that caused l to learn δ. Obviously, $M_1 \not\sim_l M_1[\delta'/\delta]$, and by implication, $M \not\sim_l M[\delta'/\delta]$. □

Proof of Theorem 9. From the following two lemmas and by noting that $R = \{(M, M[\delta'/\delta]) \mid M$ is a valid machine state and $\delta' \notin M\}$ is an l-bisimulation. $\square$

Lemma 15. *If $M = \{t, s, z, \Sigma, p_1, \ldots, p_k\}$ and all services $\sigma \in \Sigma$ are secrecy-friendly, then for all secrets δ, δ' such that $\delta' \notin M$, $M \longrightarrow M'$ iff $M[\delta'/\delta] \longrightarrow M'[\delta'/\delta]$.*

Proof. Let us first prove the statement in the forward direction. We will examine two cases, the first being representative of all transition rules except one, which we consider separately:

Case

$$\mathsf{Local}\langle v :: S, E, \mathbf{ch} :: \sigma :: C, D\rangle_k \longrightarrow \mathsf{Local}\langle \mathsf{chunk}(\sigma, v, E) :: S, E, C, D\rangle_k$$

The truth of the statement is obvious, given that $\delta' \notin M$ implies that $\delta' \notin v, S, E, C, D$.

Case

$$\mathsf{Local}\langle v :: S, E, \mathbf{serv} :: \sigma :: C, D\rangle_l, s, t, z \longrightarrow [\![\sigma]\!](v, l, s, t, z, (S, E, C, D))$$

In this case, suppose that $[\![\sigma]\!](v, l, s, t, z, (S, E, C, D)) = s', t', z', p_1, \ldots, p_k$. Now, given the secrecy-friendly property we know that

$$[\![\sigma]\!]((v, l, s, t, a, (S, E, C, D)))[\delta'/\delta]) = \\ (s', t', a', p_1, \ldots, p_i)[\delta'/\delta]$$

and the result follows directly from that.

For the converse result, note simply that if $\delta' \notin M$, then $M[\delta'/\delta][\delta/\delta'] = M$, and $\delta \notin M[\delta'/\delta]$. $\square$

Lemma 16. *If $M = \{t, s, z, \Sigma, p_1, \ldots, p_k\}$ is such that all services $\sigma \in \Sigma$ are secrecy-friendly, and M provides δ-secrecy against l, and $\delta' \notin M$, then $M \twoheadrightarrow^* \overset{p}{\twoheadrightarrow} \twoheadrightarrow^* M'$ iff $M[\delta'/\delta] \twoheadrightarrow^* \overset{p}{\twoheadrightarrow} \twoheadrightarrow^* M'[\delta'/\delta]$*

Proof. Proof in the forward direction is from the previous lemma, and by noting that if $p = \mathsf{Transit}\langle a, \mathsf{chunk}(x, v, E)\rangle_l$ then the secrecy of δ requires that $\delta \notin v, E$.

For the converse, let $M[\delta'/\delta] \twoheadrightarrow^* \overset{p}{\twoheadrightarrow} \twoheadrightarrow^* M'[\delta'/\delta]$.

Now, $M[\delta'/\delta][\delta/\delta'] \twoheadrightarrow^* \overset{p[\delta/\delta']}{\twoheadrightarrow} \twoheadrightarrow^* M'[\delta'/\delta][\delta/\delta']$

i.e., $M \twoheadrightarrow^* \overset{p[\delta/\delta']}{\twoheadrightarrow} \twoheadrightarrow^* M'$

Since M provides $\delta-$secrecy against l, $\delta \notin p[\delta/\delta']$, i.e., $\delta' \notin p$.

Then, $p = p[\delta/\delta']$, and $M \twoheadrightarrow^* \overset{p}{\twoheadrightarrow} \twoheadrightarrow^* M'$. $\square$

A.4 Observation 11

In order to prove this observation, we first define the set $\mathcal{P}$ to be the set of all particles that appear in the computation of a labeled routing particle, $p = \mathsf{Local}\langle \mathbf{nil}, \emptyset, e_L, \mathbf{nil}\rangle_{\mathsf{Alice}}$, in the following way: let

$$\mathcal{P}^1 = \{p' \mid \{t, s, z, \Sigma, p\} \longrightarrow^* \{t, s', z, \Sigma, p'\}\}$$

and

$$\mathcal{P}^2 = \{p' \mid \{t, s, z, \Sigma, p\} \longrightarrow^* \{t, s', z, \Sigma, p', p''\}\},$$

and let $\mathcal{P} = \mathcal{P}^1 \cup \mathcal{P}^2$.

We can provide a detailed description of an exact computation starting from the machine state $\{t, s, z, \Sigma, p\}$, following the example of the proofs of Theorems 4 and 7. It turns out however that the computation here would span several pages. We therefore provide in Table 8 only the first few steps in the evaluation of particle p. Notice that the last machine state in the table looks almost exactly like the fourth one, with three differences: Alice has been replaced with $r1$, the environment in the only $\mathsf{Local}\langle \ldots \rangle$ particle has some extra entries which will not get used, and the $\mathsf{DataState}$ has been augmented so that all locations on $n1$ now know the secrets δ and `lab`. The rest of this computation will repeat the pattern shown in the table, once at $r1$ and then at $r2$, and then will end at Bob.

We now prove the Observation.

Proof of Observation 11. Suppose $\{t, s, z, \Sigma, p\} \longrightarrow^* M = \{t', s', z', \Sigma, p_1, \ldots, p_k\}$. Then all of the following must be true:

- $\delta, lab \notin s'(\mathsf{Eve})$.
- $z'(l)(lab) = z(l)(lab)$ for all $l \in \{\mathsf{Alice}, r1, r2, \mathsf{Bob}\}$, and $z'(l)(\delta') \neq lab, \delta$ for all $l \in \mathsf{Loc}$ and all $\delta' \in \mathsf{Data}$.
- For all $p \in M$, one of the following must hold:
  - $p \in \mathcal{P}$.
  - $\delta, lab \notin p$.

We prove this by induction on the length of the computation $\{t, s, z, \Sigma, p\} \longrightarrow^* M$.

Base Case: If the length of the computation is 0, then $M = \{t, s, z, \Sigma, p\}$. The results follow directly from the assumptions in this case.

Induction Step: Let the hypothesis be true for a computation of j steps. Now, let us consider a computation involving $j+1$ steps, and in particular look at the last transition.

Let $\{t, s, z, \Sigma, p\} \longrightarrow^* M' \longrightarrow M''$ where

$$\begin{array}{rcl} M' & = & \{t', s', z', \Sigma, p'_1, \ldots, p'_m\} \text{ and} \\ M'' & = & \{t'', s'', z'', \Sigma, p''_1, \ldots, p''_n\}. \end{array}$$

Table 8: Initial evaluation of particle p

In this table:

$$
\begin{array}{rcl}
E_1 & = & \emptyset[\mathsf{close}(\mathtt{labelRoute}, (\mathtt{d}, \mathtt{l}, \mathsf{c}, \mathtt{dest}), (\mathsf{c} = \mathtt{dest}) \,?\, \ldots, \emptyset)/\mathtt{labelRoute}] \\
E_2 & = & E_1[\delta/\mathtt{d}][\mathtt{lab}/\mathtt{l}][\mathsf{Alice}/\mathsf{c}][\mathsf{Bob}/\mathtt{dest}] \\
D_1 & = & \langle \mathbf{nil}, \emptyset, \mathbf{nil}, \mathbf{nil} \rangle \\
D_2 & = & \langle \mathbf{nil}, E_1, \mathbf{nil}, D_1 \rangle
\end{array}
$$

$$\{t, s, z, \Sigma, p\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, \emptyset, \mathsf{fun}\mathtt{labelRoute}\ldots\ \mathsf{in}\ \mathtt{labelRoute}(\delta, \mathtt{lab}, \mathsf{Alice}, \mathsf{Bob}), \mathbf{nil}\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, \emptyset, \mathsf{fun}\ \mathtt{labelRoute}\ldots :: \mathtt{labelRoute}(\delta, \mathtt{lab}, \mathsf{Alice}, \mathsf{Bob}), D_1\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, E_1, \mathtt{labelRoute}(\delta, \mathtt{lab}, \mathsf{Alice}, \mathsf{Bob}), D_1\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, E_1, (\delta, \mathtt{lab}, \mathsf{Alice}, \mathsf{Bob}) :: \mathtt{labelRoute} :: \mathbf{ap}, D_1\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, E_1, \delta :: \mathtt{lab} :: \mathsf{Alice} :: \mathsf{Bob} :: \mathbf{tup}_4 :: \mathtt{labelRoute} :: \mathbf{ap}, D_1\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \delta, E_1, \mathtt{lab} :: \mathsf{Alice} :: \mathsf{Bob} :: \mathbf{tup}_4 :: \mathtt{labelRoute} :: \mathbf{ap}, D_1\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathtt{lab} :: \delta, E_1, \mathsf{Alice} :: \mathsf{Bob} :: \mathbf{tup}_4 :: \mathtt{labelRoute} :: \mathbf{ap}, D_1\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathsf{Alice} :: \mathtt{lab} :: \delta, E_1, \mathsf{Bob} :: \mathbf{tup}_4 :: \mathtt{labelRoute} :: \mathbf{ap}, D_1\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathsf{Bob} :: \mathsf{Alice} :: \mathtt{lab} :: \delta, E_1, \mathbf{tup}_4 :: \mathtt{labelRoute} :: \mathbf{ap}, D_1\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle (\delta, \mathtt{lab}, \mathsf{Alice}, \mathsf{Bob}), E_1, \mathtt{labelRoute} :: \mathbf{ap}, D_1\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathsf{close}(\mathtt{labelRoute}, \ldots) :: (\delta, \mathtt{lab}, \mathsf{Alice}, \mathsf{Bob}), E_1, \mathbf{ap}, D_1\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, E_2, (\mathsf{c} = \mathtt{dest}) \,?\, \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, E_2, (\mathsf{c} = \mathtt{dest}) :: \,?\, :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, E_2, \mathsf{c} :: \mathtt{dest} ::{=}:: \mathbf{binary} :: \,?\, :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathsf{Alice}, E_2, \mathtt{dest} ::{=}:: \mathbf{binary} :: \,?\, :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathsf{Bob} :: \mathsf{Alice}, E_2, {=}:: \mathbf{binary} :: \,?\, :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathsf{false}, E_2, \,?\, :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, E_2, (|\mathtt{labelRoute}|(\mathtt{d}, \mathtt{l}, \#2(\mathsf{get}(\mathtt{l})), \mathtt{dest}))\ @\ (\mathsf{get}(\mathtt{l})), D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, E_2, |\mathtt{labelRoute}|(\ldots) :: \#2(\mathsf{get}(\mathtt{l})) ::\ @\ , D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, E_2, (\mathtt{d}, \mathtt{l}, \#2(\mathsf{get}(\mathtt{l})), \mathtt{dest}) :: \mathbf{ch} :: \mathtt{labelRoute} :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, E_2, \mathtt{d} :: \mathtt{l} :: \#2(\mathsf{get}(\mathtt{l})) :: \mathtt{dest} :: \mathbf{tup}_4 :: \mathbf{ch} :: \mathtt{labelRoute} :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \delta, E_2, \mathtt{l} :: \#2(\mathsf{get}(\mathtt{l})) :: \mathtt{dest} :: \mathbf{tup}_4 :: \mathbf{ch} :: \mathtt{labelRoute} :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathtt{lab} :: \delta, E_2, \#2(\mathsf{get}(\mathtt{l})) :: \mathtt{dest} :: \mathbf{tup}_4 :: \mathbf{ch} :: \mathtt{labelRoute} :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathtt{lab} :: \delta, E_2, \mathsf{get}(\mathtt{l}) :: \mathbf{proj}_2 :: \mathtt{dest} :: \mathbf{tup}_4 :: \mathbf{ch} :: \mathtt{labelRoute} :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathtt{lab} :: \delta, E_2, \mathtt{l} :: \mathbf{serv} :: \mathsf{get} :: \mathbf{proj}_2 :: \mathtt{dest} :: \mathbf{tup}_4 :: \mathbf{ch} :: \mathtt{labelRoute} :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathtt{lab} :: \mathtt{lab} :: \delta, E_2, \mathbf{serv} :: \mathsf{get} :: \mathbf{proj}_2 :: \mathtt{dest} :: \mathbf{tup}_4 :: \mathbf{ch} :: \mathtt{labelRoute} :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle (\mathsf{n1{,}r1}) :: \mathtt{lab} :: \delta, E_2, \mathbf{proj}_2 :: \mathtt{dest} :: \mathbf{tup}_4 :: \mathbf{ch} :: \mathtt{labelRoute} :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle r1 :: \mathtt{lab} :: \delta, E_2, \mathtt{dest} :: \mathbf{tup}_4 :: \mathbf{ch} :: \mathtt{labelRoute} :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathsf{Bob} :: r1 :: \mathtt{lab} :: \delta, E_2, \mathbf{tup}_4 :: \mathbf{ch} :: \mathtt{labelRoute} :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle (\delta, \mathtt{lab}, r1, \mathsf{Bob}), E_2, \mathbf{ch} :: \mathtt{labelRoute} :: \ldots, D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathsf{chunk}(\mathtt{labelRoute}, (\delta, \mathtt{lab}, r1, \mathsf{Bob}), E_2), E_2, \mathsf{get}(\mathtt{l}) ::\ @\ , D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathsf{chunk}(\mathtt{labelRoute}, (\delta, \mathtt{lab}, r1, \mathsf{Bob}), E_2), E_2, \mathtt{l} :: \mathbf{serv} :: \mathsf{get} ::\ @\ , D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle \mathtt{lab} :: \mathsf{chunk}(\mathtt{labelRoute}, (\delta, \mathtt{lab}, r1, \mathsf{Bob}), E_2), E_2, \mathbf{serv} :: \mathsf{get} ::\ @\ , D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle (\mathsf{n1{,}r1}) :: \mathsf{chunk}(\mathtt{labelRoute}, (\delta, \mathtt{lab}, r1, \mathsf{Bob}), E_2), E_2,\ @\ , D_2\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle (), E_2, \mathbf{nil}, D_2\rangle_{\mathsf{Alice}}, \mathsf{Transit}\langle (\mathsf{n1{,}r1}), \mathsf{chunk}(\mathtt{labelRoute}, (\delta, \mathtt{lab}, r1, \mathsf{Bob}), E_2)\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle (), E_1, \mathbf{nil}, D_1\rangle_{\mathsf{Alice}}, \mathsf{Transit}\langle (\mathsf{n1{,}r1}), \mathsf{chunk}(\mathtt{labelRoute}, (\delta, \mathtt{lab}, r1, \mathsf{Bob}), E_2)\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Local}\langle (), \emptyset, \mathbf{nil}, \mathbf{nil}\rangle_{\mathsf{Alice}}, \mathsf{Transit}\langle (\mathsf{n1{,}r1}), \mathsf{chunk}(\mathtt{labelRoute}, (\delta, \mathtt{lab}, r1, \mathsf{Bob}), E_2)\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s, z, \Sigma, \mathsf{Transit}\langle (\mathsf{n1{,}r1}), \mathsf{chunk}(\mathtt{labelRoute}, (\delta, \mathtt{lab}, r1, \mathsf{Bob}), E_2)\rangle_{\mathsf{Alice}}\}\longrightarrow$$
$$\{t, s', z, \Sigma, \mathsf{Transit}\langle (\mathsf{n1{,}r1}), \mathsf{chunk}(\mathtt{labelRoute}, (\delta, \mathtt{lab}, r1, \mathsf{Bob}), E_2)\rangle_{r1}\}\longrightarrow$$
$$\{t, s', z, \Sigma, \mathsf{Local}\langle \mathbf{nil}, E_2, \mathtt{labelRoute}(\delta, \mathtt{lab}, r1, \mathsf{Bob}), \mathbf{nil}\rangle_{r1}\}$$

We now need to verify that the induction hypothesis holds for each possible transition rule used in the last transition. We will not describe every case here, rather we will pick a representative set of rules. We thus examine five cases on the last transition, and claim that all rules except four are similar to the first case while the next four cases are special:

Case

$$\mathsf{Local}\langle v :: S, E, \mathbf{ch} :: \sigma :: C, D\rangle_k \longrightarrow \mathsf{Local}\langle \mathsf{chunk}(\sigma, v, E) :: S, E, C, D\rangle_k$$

Thus, there is some $p' \in M'$ and some $p'' \in M''$ such that $p' = \mathsf{Local}\langle v :: S, E, \mathbf{ch} :: \sigma :: C, D\rangle_k$ and $p'' = \mathsf{Local}\langle \mathsf{chunk}(\sigma, v, E) :: S, E, C, D\rangle_k$. Now there are two subcases:

- $p' \in \mathcal{P}$.
  From the definition of $\mathcal{P}$, $\{t, s, z, \Sigma, p_A\} \longrightarrow^* \{t, s', z, \Sigma, p'\}$. Since $\{t, s', z, \Sigma, p'\} \longrightarrow^* \{t, s', z, \Sigma, p''\}$, we must have $p'' \in \mathcal{P}$.
- $p' \notin \mathcal{P}$.
  From the induction hypothesis then $\delta, lab \notin p'$, and by an inspection of the rule, $\delta, lab \notin p''$.

In both subcases, this rule does not change either the $\mathsf{DataState}$ or the $\mathsf{Dictionary}$, thus $z'' = z'$ and $s'' = s'$ therefore the other invariants also hold.

Case

$$\begin{array}{l}\mathsf{Transit}\langle a, ch\rangle_l, t', s' \longrightarrow \mathsf{Transit}\langle a, ch\rangle_{l'}, t', s'' \\ \text{where } ch = \mathsf{chunk}(x, v, E) \text{ or } \mathsf{chunk}(\sigma, v, E) \\ \text{and}((n, l'), i) = t'(l)(a) \\ \text{and } s''(l'') = \left\{\begin{array}{l} s'(l'') \text{ if } l'' \notin \mathsf{Topology}(n) \\ s'(l'') \cup \{\delta | \delta \in v \text{ or } \delta \in E\} \text{ otherwise}\end{array}\right.\end{array}$$

This rule changes the $\mathsf{DataState}$. Now, there must be some $p' \in M'$ and some $p'' \in M''$ such that $p' = \mathsf{Transit}\langle a, ch\rangle_{l_1}$ and $p'' = \mathsf{Transit}\langle a, ch\rangle_{l_2}$. Notice that the rule does not change the $\mathsf{Dictionary}$, hence $z'' = z'$, satisfying the second invariant. Again, there are two subcases:

- $p' \in \mathcal{P}$.
  By definition of P, $p'' \in \mathcal{P}$. Also $l \in \{\mathsf{Alice}, r1, r2, \mathsf{Bob}\}$ and $n \in \{n1, n2, n3\}$. Since $\mathsf{Eve} \notin \mathsf{Topology}(n)$, and $\delta, lab \notin s'(\mathsf{Eve})$, we must have $\delta, lab \notin s''(\mathsf{Eve})$.
- $p' \notin \mathcal{P}$.
  Then $\delta, lab \notin p'$, so $\delta, lab \in s''(l'')$ only if $\delta \in s'(l'')$, but then $l'' \neq \mathsf{Eve}$. Also, by inspection of the rule, $\delta, lab \notin p''$.

Case

$$\begin{array}{l} s', \Sigma \longrightarrow \mathsf{Local}\langle \mathbf{nil}, \emptyset, e, \mathbf{nil}\rangle_l, s', \Sigma \\ \text{where } e \text{ } \mathsf{obeys} \text{ } s'(l) \\ \text{and all services } \sigma \text{ in } e \text{ are in } \Sigma \end{array}$$

This rule involves the generation of a new particle. The new particle is $\mathsf{Local}\langle \mathbf{nil}, \emptyset, e, \mathbf{nil}\rangle_l$. Since controlled nodes can't generate packets, $l = \mathsf{Eve}$, so $\delta, lab \notin s'(l)$. But $\delta, lab \notin \mathsf{Local}\langle \mathbf{nil}, \emptyset, e, \mathbf{nil}\rangle_l$. Moreover $z'' = z'$ and $s'' = s'$ so the other invariants also hold.

Case

$$\mathsf{Local}\langle a :: ch :: S, E, @ :: C, D\rangle_k \longrightarrow \mathsf{Local}\langle () :: S, E, C, D\rangle_k, \mathsf{Transit}\langle a, ch\rangle_k$$

This rule has two particles on the right hand side. There must be a $p' \in M'$ such that $p' = \mathsf{Local}\langle a :: ch :: S, E, @ :: C, D\rangle_k$, and $p'', p''' \in M''$ such that $p'' = \mathsf{Local}\langle () :: S, E, C, D\rangle_k$ and $p''' = \mathsf{Transit}\langle a, ch\rangle_k$. The two subcases are:

- $p' \in \mathcal{P}$.

  Then by definition of $\mathcal{P}$, $p'', p''' \in \mathcal{P}$. The $\mathsf{DataState}$ and $\mathsf{Dictionary}$ do not change, so the other invariants are also satisfied.

- $p' \notin \mathcal{P}$.

  Then $\delta, lab \notin p'$. By inspection of the rule, $\delta, lab \notin p'', p'''$. Again, the other invariants are also satisfied.

Case

$$\mathsf{Local}\langle v :: S, E, \mathbf{serv} :: \sigma :: C, D\rangle_l, s', t', z' \longrightarrow [\![\sigma]\!](v, l, s', t', z', (S, E, C, D))$$

This rule involves application of a service. $\sigma \in \Sigma = \{\mathsf{set}, \mathsf{get}, \mathsf{deliver}\}$. $[\![\sigma]\!](v, l, s', t', z', (S, E, C, D)) = p'', s'', t'', z''$. Let us examine the two subcases:

- $p' \in \mathcal{P}$.

  Then σ is either get or $\mathsf{deliver}$.

  - If $\sigma = \mathsf{get}$, $z'' = z'$ and $s'' = s'$.
  - If $\sigma = \mathsf{deliver}$, we must have $l = \mathsf{Bob}$. Now, $z'' = z'$, and $\delta, lab \in s''(l'')$ only if $\delta, lab \in s'(l'')$ or $l'' = \mathsf{Bob}$, thus $\delta, lab \notin s''(\mathsf{Eve})$.

  Also, by definition of $\mathcal{P}$, $p'' \in \mathcal{P}$.

- $p' \notin \mathcal{P}$.

  Then $\delta, lab \notin p'$. Let us examine three subcases on σ:

  - If $\sigma = \mathsf{get}$, $z'' = z'$ and $s'' = s'$. Since $z'(l)(\delta') \neq \delta, lab$ for all $l \in \mathsf{Loc}$ and all $\delta' \in \mathsf{Data}$, so $\delta, lab \notin p''$.

- If $\sigma = \mathsf{set}$, $s'' = s'$. Since $\delta, lab \notin p'$, $z''(l)(lab) = z'(l)(lab)$ for all l, and $z''(l)(\delta') \neq \delta, lab$ for all $l \in \mathsf{Loc}$ and all $\delta' \in \mathsf{Data}$. By inspection of the definition of set, $\delta, lab \notin p''$.
- If $\sigma = \mathsf{deliver}$, $z'' = z'$. Moreover, $\delta, lab \notin p'$ and $\delta, lab \notin s'(\mathsf{Eve})$ imply that $\delta, lab \notin s''(\mathsf{Eve})$. Again, by inspection of the definition of $\mathsf{deliver}$, $\delta, lab \notin p''$.

□

A.5 Theorem 12

Again, we define the set of particles $\mathcal{P}_L$ to be the set of particles arising during the computation of the particle p_L in the following way: let

$$\mathcal{P}_L^1 = \{p' \mid \{t, s, z, \Sigma, p_L\} \longrightarrow^* \{t, s', z, \Sigma, p'\}\}$$

and

$$\mathcal{P}_L^2 = \{p' \mid \{t, s, z, \Sigma, p_L\} \longrightarrow^* \{t, s', z, \Sigma, p', p''\}\},$$

and let $\mathcal{P}_L = \mathcal{P}_L^1 \cup \mathcal{P}_L^2$.

The evaluation of p_L will be similar to the evaluation in the last proof, hence we don't provide a detailed table here. The computation will involve a series of $\mathsf{Local}\langle \ldots \rangle$ and $\mathsf{Transit}\langle \ldots \rangle$ particles, with the former evaluating on each successive location l_i in the labeled route, and the latter being one-hop movements from one location in the route to the next.

We now prove the theorem. In the following, let $K_\delta(s) = \{l \mid \delta \in s(l)\}$, and let $K_{lab}(s) = \{l \mid lab \in s(l)\}$.

Proof of Theorem 12. Suppose $\{t, s, z, \Sigma, p_L\} \longrightarrow^* M = \{t', s', z', \Sigma, p_1, \ldots, p_k\}$. Then all of the following must be true:

- $K_\delta(s') \subseteq \mathsf{Con}$.
- $K_{lab}(s') \subseteq \mathsf{Con}$.
- $z'(l_i)(lab) = z(l_i)(lab)$ and $z'(l)(\delta') \neq lab, \delta$ for all $l \in \mathsf{Loc}$ and all $\delta' \in \mathsf{Data}$.
- For all $p \in M$, one of the following must hold:
  - $p \in \mathcal{P}_L$.
  - $\delta, lab \notin p$.

We prove this by induction on the length of the computation $\{t, s, z, \Sigma, p_L\} \longrightarrow^* M$.

Base Case: If the length of the computation is 0, then $M = \{t, s, z, \Sigma, p_L\}$. The results follow directly from the assumptions in this case.

Induction Step: Let the hypothesis be true for a computation of j steps. Now, let us consider a computation involving $j+1$ steps, and in particular look at the last transition.

Let $\{t, s, z, \Sigma, p_L\} \longrightarrow^* M' \longrightarrow M''$ where

$$\begin{array}{rcl} M' & = & \{t', s', z', \Sigma, p'_1, \ldots, p'_m\} \text{ and} \\ M'' & = & \{t'', s'', z'', \Sigma, p''_1, \ldots, p''_n\}. \end{array}$$

We now need to verify that the induction hypothesis holds for each possible transition rule used in the last transition. We will not describe every case here, rather we will pick a representative set of rules. We thus examine five cases on the last transition, and claim that all rules except four are similar to the first case while the next four cases are special:

Case

$$\mathsf{Local}\langle v :: S, E, \mathbf{ch} :: \sigma :: C, D\rangle_k \longrightarrow \mathsf{Local}\langle \mathsf{chunk}(\sigma, v, E) :: S, E, C, D\rangle_k$$

Thus, there is some $p' \in M'$ and some $p'' \in M''$ such that $p' = \mathsf{Local}\langle v :: S, E, \mathbf{ch} :: \sigma :: C, D\rangle_k$ and $p'' = \mathsf{Local}\langle \mathsf{chunk}(\sigma, v, E) :: S, E, C, D\rangle_k$. Now there are two subcases:

- $p' \in \mathcal{P}_L$.
  From the definition of $\mathcal{P}_L$, $\{t, s, z, \Sigma, p_L\} \longrightarrow^* \{t, s', z, \Sigma, p'\}$. Since $\{t, s', z, \Sigma, p'\} \longrightarrow^* \{t, s', z, \Sigma, p''\}$, we must have $p'' \in \mathcal{P}_L$.
- $p' \notin \mathcal{P}_L$.
  From the induction hypothesis then $\delta, \mathit{lab} \notin p'$, and by an inspection of the rule, $\delta, \mathit{lab} \notin p''$.

In both subcases, this rule does not change either the $\mathsf{DataState}$ or the $\mathsf{Dictionary}$, thus $z'' = z'$ and $s'' = s'$ therefore the other invariants also hold.

Case

$$\begin{array}{l} \mathsf{Transit}\langle a, ch\rangle_l, t', s' \longrightarrow \mathsf{Transit}\langle a, ch\rangle_{l'}, t', s'' \\ \text{where } ch = \mathsf{chunk}(x, v, E) \text{ or } \mathsf{chunk}(\sigma, v, E) \\ \text{and}((n, l'), i) = t'(l)(a) \\ \text{and } s''(l'') = \left\{ \begin{array}{l} s'(l'') \text{ if } l'' \notin \mathsf{Topology}(n) \\ s'(l'') \cup \{\delta \mid \delta \in v \text{ or } \delta \in E\} \text{ otherwise} \end{array} \right. \end{array}$$

This rule changes the $\mathsf{DataState}$. Now, there must be some $p' \in M'$ and some $p'' \in M''$ such that $p' = \mathsf{Transit}\langle a, ch\rangle_{l'}$ and $p'' = \mathsf{Transit}\langle a, ch\rangle_{l''}$. Notice that the rule does not change the $\mathsf{Dictionary}$, hence $z'' = z'$, satisfying the third invariant. Again, there are two subcases:

- $p' \in \mathcal{P}_L$.
  By definition of $\mathcal{P}_L$, $l = l_i$ for some i, and $a = (n_{i+1}, l_{i+1}, 0)$. Since $l_i \in \mathsf{Topology}(n_{i+1})$ and t' is a shortest path routing table, $t'(l)(a) =$

a. Thus $n = n_{i+1}$ and $l' = l_{i+1}$, thus $p'' \in \mathcal{P}_L$. Also $\mathsf{Topology}(n) \subseteq \mathsf{Con}$. Consider some $l'' \notin \mathsf{Con}$. Since $l'' \notin \mathsf{Topology}(n)$, and $\delta, lab \notin s'(l'')$, we must have $\delta, lab \notin s''(l'')$, so that $K_\delta(s''), K_{lab}(s'') \subseteq \mathsf{Con}$.

- $p' \notin \mathcal{P}_L$.
  Then $\delta, lab \notin p'$, so $\delta, lab \in s''(l'')$ only if $\delta \in s'(l'')$, but then $l'' \in \mathsf{Con}$, so that $K_\delta(s''), K_{lab}(s'') \subseteq \mathsf{Con}$. Also, by inspection of the rule, $\delta, lab \notin p''$.

Case

$$\begin{array}{l} s', \Sigma \longrightarrow \mathsf{Local}\langle \mathbf{nil}, \emptyset, e, \mathbf{nil}\rangle_l, s', \Sigma \\ \text{where } e \ \mathsf{obeys}\ s'(l) \\ \text{and all services } \sigma \text{ in } e \text{ are in } \Sigma \end{array}$$

This rule involves the generation of a new particle. The new particle is $\mathsf{Local}\langle \mathbf{nil}, \emptyset, e, \mathbf{nil}\rangle_l$. Since controlled nodes can't generate packets, $l \notin \mathsf{Con}$, so $\delta, lab \notin s'(l)$. But then $\delta, lab \notin \mathsf{Local}\langle \mathbf{nil}, \emptyset, e, \mathbf{nil}\rangle_l$. Moreover $z'' = z'$ and $s'' = s'$ so the other invariants also hold.

Case

$$\mathsf{Local}\langle a :: ch :: S, E, @ :: C, D\rangle_k \longrightarrow \mathsf{Local}\langle () :: S, E, C, D\rangle_k, \mathsf{Transit}\langle a, ch\rangle_k$$

This rule has two particles on the right hand side. There must be a $p' \in M'$ such that $p' = \mathsf{Local}\langle a :: ch :: S, E, @ :: C, D\rangle_k$, and $p'', p''' \in M''$ such that $p'' = \mathsf{Local}\langle () :: S, E, C, D\rangle_k$ and $p''' = \mathsf{Transit}\langle a, ch\rangle_k$. The two subcases are:

- $p' \in \mathcal{P}_L$.
  Then by definition of $\mathcal{P}_L$, $p'', p''' \in \mathcal{P}_L$. The $\mathsf{DataState}$ and $\mathsf{Dictionary}$ do not change, so the other invariants are also satisfied.
- $p' \notin \mathcal{P}_L$.
  Then $\delta, lab \notin p'$. By inspection of the rule, $\delta, lab \notin p'', p'''$. Again, the other invariants are also satisfied.

Case

$$\mathsf{Local}\langle v :: S, E, \mathbf{serv} :: \sigma :: C, D\rangle_l, s', t', z' \longrightarrow [\![\sigma]\!](v, l, s', t', z', (S, E, C, D))$$

This rule involves application of a service. $\sigma \in \Sigma = \{\mathsf{set}, \mathsf{get}, \mathsf{deliver}\}$. $[\![\sigma]\!](v, l, s', t', z', (S, E, C, D)) = p'', s'', t'', z''$. Let us examine the two subcases:

- $p' \in \mathcal{P}_L$.
  Then σ is either get or $\mathsf{deliver}$.
  - If $\sigma = \mathsf{get}$, $z'' = z'$ and $s'' = s'$.

  - If $\sigma = \mathsf{deliver}$, we must have $l = l_k$, so $l \in \mathsf{Con}$. Now, $z'' = z'$, and $\delta, lab \in s''(l'')$ only if $\delta, lab \in s'(l'')$ or $l'' = l_k$, and thus $K_\delta(s''), K_{lab}(s'') \subseteq \mathsf{Con}$.

  Also, by definition of $\mathcal{P}_L$, $p'' \in \mathcal{P}_L$.

- $p' \notin \mathcal{P}_L$.

  Then $\delta, lab \notin p'$. Let us examine three subcases on σ:

  - If $\sigma = \mathsf{get}$, $z'' = z'$ and $s'' = s'$. Since $z'(l)(\delta') \neq \delta, lab$ for all $l \in \mathsf{Loc}$ and all $\delta' \in \mathsf{Data}$, so $\delta, lab \notin p''$.
  - If $\sigma = \mathsf{set}$, $s'' = s'$. Since $\delta, lab \notin p'$, $z''(l)(lab) = z'(l)(lab)$ for all l, and $z''(l)(\delta') \neq \delta, lab$ for all $l \in \mathsf{Loc}$ and all $\delta' \in \mathsf{Data}$. By inspection of the definition of set, $\delta, lab \notin p''$.
  - If $\sigma = \mathsf{deliver}$, $z'' = z'$. Moreover, $\delta, lab \notin p'$ and $K_\delta(s'), K_{lab}(s') \subseteq \mathsf{Con}$ imply that $K_\delta(s''), K_{lab}(s'') \subseteq \mathsf{Con}$. Again, by inspection of the definition of $\mathsf{deliver}$, $\delta, lab \notin p''$.

$\square$

References

[1] Martín Abadi. Secrecy by typing in security protocols. In *Theoretical Aspects of Computer Software*, volume 1281 of *LNCS*, pages 611–638. Springer, September 1997.

[2] Martín Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999.

[3] Martín Abadi. Security protocols and specifications. In *Foundations of Software Science and Computation Structures: Second International Conference, FOSSACS '99*, pages 1–13. Springer-Verlag, March 1999.

[4] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[5] D. Scott Alexander, William A. Arbaugh, Angelos D. Keromyts, and Jonathan M. Smith. A secure active network environment architecture: Realization in SwitchWare. *IEEE Network Magazine*, 1998. To appear in the special issue on Active and Controllable Networks.

[6] D. Scott Alexander, Marianne Shaw, Scott M. Nettles, and Jonathan M. Smith. Active Bridging. In *Proceedings, 1997 SIGCOMM Conference*. ACM, 1997.

[7] Steven M. Bellovin. Using the domain name system for system break-ins. In *Proceedings of the Fifth USENIX UNIX Security Symposium*, June 1995.

[8] G. Berry and G. Boudol. The chemical abstract machine. In *Proceedings of the 17th Annual Symposium of Programming Languages*, pages 81–94, 1990.

[9] Gérard Berry and Gérard Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96:217–248, 1992.

[10] Dimitri P. Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, 1991.

[11] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Formal verification of standards for distance vector routing protocols, February 2000.

[12] Michael Hicks and Angelos D. Keromytis. A secure PLAN. In Stefan Covaci, editor, *Proceedings of the First International Working Conference on Active Networks*, volume 1653 of *Lecture Notes in Computer Science*, pages 307–314. Springer-Verlag, June 1999.

[13] Michael Hicks, Jonathan T. Moore, D. Scott Alexander, Carl A. Gunter, and Scott Nettles. PLANet: An active internetwork. In *Proceedings of the Eighteenth IEEE Computer and Communication Society Infocom Conference*, pages 1124–1133. IEEE Communication Society Press, March 1999.

[14] Mike Hicks, Pankaj Kakkar, Jonathan T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A packet language for active networks. In *Proceedings of the Third ACM SIGPLAN International Conference on Functional Programming Languages*, pages 86–93. ACM, 1998.

[15] P. Landin. The mechanical evaluation of expressions. *Comput. J.*, 6:308–320, 1964.

[16] P. Landin. An abstract machine for designers of computing languages. In *IFIP Congress*, pages 438–439. North-Holland, 1965.

[17] G. Malkin. *RIP Version 2 Applicability Statement.* IETF RFC 1722, November 1994.

[18] G. Malkin. *RIP Version 2 Carrying Additional Information.* IETF RFC 1723, November 1994.

[19] J. Moy. OSPF version 2. RFC 1583, IETF, March 1994.

[20] Jonathan M. Smith, Kenneth L. Calvert, Sandra L. Murphy, Hilarie K. Orman, and Larry L. Peterson. Activating networks: A progress report. *IEEE Computer*, 32(4):32–41, April 1999.

[21] David L. Tennenhouse, Jonathan M. Smith, W. David Sincoskie, David J. Wetherall, and Gary J. Minden. A survey of active network research. *IEEE Communications Magazine*, 35(1):80–86, January 1997.

[22] Bow-Yaw Wang, José Meseguer, and Carl A. Gunter. Specification and formal verification of a PLAN algorithm in Maude. In Tenh Lai, editor, *Proceedings of the 2000 ICDCS Workshop on Distributed System Validation and Verification*, pages E:49–E:56. IEEE Computer Society, April 2000.

[23] David J. Wetherall, John Guttag, and David L. Tennenhouse. ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols. In *IEEE OPENARCH*, April 1998.