Design and Analysis of Sectrace: A Protocol to Set up Security Associations and Policies in IPSec Networks[*]

Alwyn Goodloe
University of Pennsylvania
agoodloe@seas.upenn.edu

Michael McDougall
University of Pennsylvania
mmcdouga@saul.cis.upenn.edu

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

Mark-Oliver Stehr
University of Illinois
at Urbana Champaign
stehr@uiuc.edu

September 26, 2004

Abstract

To provide encryption and authentication services IPSec assumes the existence of suitable security associations and policies. Setting up such associations and policies is not a trivial task, especially in the presence of nested channels and concatenated channels involving several security gateways. Since IPSec does not address this problem, the sectrace protocol has be designed. In this paper we have developed an executable formal specification of the protocol and performed some preliminary analysis. Our analysis indicates that the solutions found by the protocol are not always optimal, because certain possibilities to set up correct security associations are missed. It also shows that that concurrent runs of the protocol can cause undesirable interference effects. As a result of this analysis we are currently investigating formal prototypes of alternative protocol designs. Two of the these are discussed in this paper.

1 Introduction

IPSec is a Security Architecture for the Internet Protocol specified in RFC 2401 [KA98] that provides *authentication* and *encryption* for IP packets. Loosely speaking, IPSec is implemented between network and transport layers of the protocol stack, but the detailed interaction with the IP stack is slightly more complex. The operation of IPSec critically depends on the existence and maintenence of security associations, that is virtual secure channels, and policies, governing the use of these channels. IPSec does not provide a mechanism to set up these entities, but instead relies on external protocols like IKE, IKEv2, or JFK. In this paper we investigate sectrace [GGM02, GGM03], a simple protocol for security gateway discovery and for setting up security associations and policies in networks with security gateways which go beyond the simple two-party case supported by the abovementioned protocols.

An IPSec packet can be written as $ip(src, dest, sec(spi, data))$, meaning that it consists of (possibly encrypted) application data $data$ equipped with an AH or ESP header $sec(spi, \ldots)$, and this piece of information is in turn equipped with an IP header $ip(src, dest, \ldots)$. We refer to

[*]Supported by the Office of Naval Research under MURI Research Contract N00014-02-1-0715.

$ip(src, dest, sec(spi, \ldots))$ as an IPSec header. AH and ESP stand for authentication header and encapsulated security payload, respectively, the former providing authentication only and the later providing authentication with optional encryption, but in this paper we are concerned with the authentication aspect only. The IP header contains source (*src*) and destination (*dest*) address, and the AH/ESP header contains *spi*, a *security parameter index* (SPI), which together with the destination *dest* uniquely identifies a security association. A *security association* (SA) defines shared parameters between nodes (cryptoprotocol parameters and secret key). To this end, each node maintains a local *security association database* (SAD). The decision about which security association to use for the communication is determined by *security policies*. The information about these policies is stored in a local *security policy database* (SPD) which is maintained at each node. The correct and consistent maintenence of these databases is a nontrivial task for which the sectrace protocol has been designed.

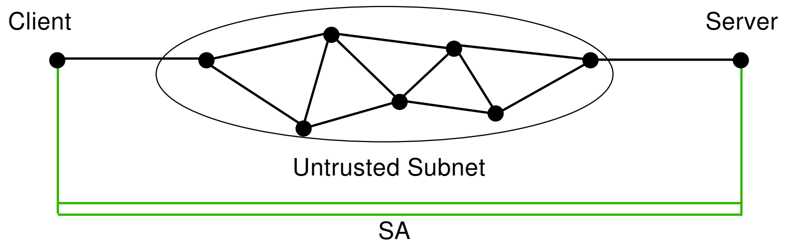

Figure 1: Security Associations

We slightly simplify the presentation of IPSec by not considering the UDP layer, which is placed above the IPSec layer in the protocol stack. Although in practice sectrace uses UDP, we identify it with IP for the purpose of this paper, because port numbers and fragmentation are irrelevant for our analysis. Furthermore, sectrace consistently uses IPSec in *tunnel mode* (as opposed to *transport mode*), where the encapsulated data again has the form of an IP packet. In this mode, full IPSec packets are of the form: $ip(src, dest, sec(spi, ip(src', dest', data)))$. A typical use of tunnel mode is to transmit information through an untrusted subnet, as illustrated in Fig. 2. The IPSec header $ip(src, dest, sec(spi, \ldots)))$ is added and removed by the two security gateways.

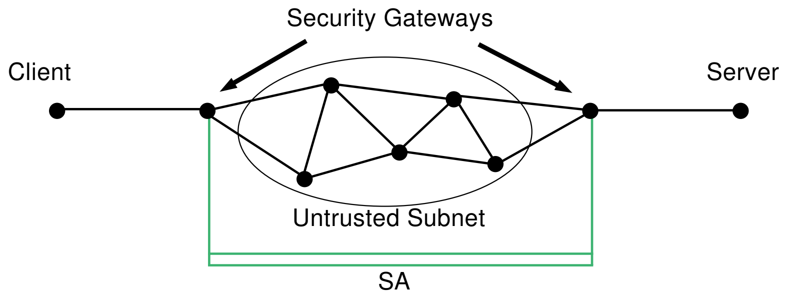

Figure 2: Tunneling

In addition to tunnels and their *concatenation*, sectrace will also support *nesting* of tunnels as illustrated in Fig. 3. In the case of nesting the full IPSec packet has the form $ip(src, dest, sec(spi, ip(src', dest', sec(spi', data))))$. Here the client and server both act as additional security gateways, which will be responsible for adding and removing the inner IPSec header $ip(src', dest', sec(spi', \ldots))))$.

In all existing protocols, security associations are set up by negotiating the security parameters (secret key, etc.) on the basis of a *public key infrastructure* (PKI). Hence, a security association

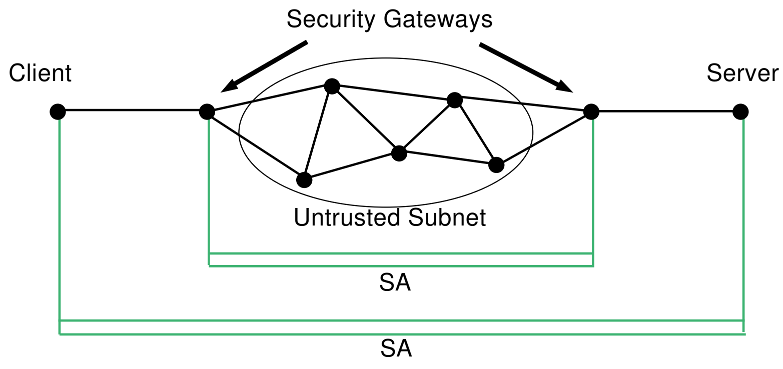

Figure 3: Nesting

from A to B can be set up if A is trusted by B, i.e. the root certificate authority of A is among those trusted by B. Sectrace will use the trust relation induced by the PKI, but as discussed in the conclusions other trust relations could be used instead.

2 Methodology

In contrast to traditional approaches to system and software engineering, we use formal methods early in the design process. We especially use light-weight tools developed for Maude [CDE$^+$, CDE$^+$02, CDE$^+$00], an executable specification language based on rewriting logic [Mes92] and its membership equational sublogic [BJM00]. Maude is an executable specification language together with a high-performance implementation, that is as a language that allows us to execute and analyze system specifications *before* they become implementations. Therefore, we often refer to such executable specifications of system designs as *formal prototypes*.

The significance of rewriting logic is that distributed systems (and especially networks) perform local state transitions, which can be naturally presented as rewrite rules that operate locally on a symbolic and mathematically precise representation of the system state. The generality of rewriting logic is witnessed by its broad applicability as a uniform semantic and logical framework in areas ranging from concurrent object-oriented programming [Mes93] to networking [MÖST02, ST02].

This accumulated experience from earlier projects has shown that the design of new network systems and protocols is a challenging task, because network designers using only traditional design tools such as simulators have limited *predictive power* to know if their designs will work in all critical scenarios. Our lightweight formal methods and tools, based on formal *executable* models, have proved very effective in: (1) *accelerating* the design process by giving very valuable feedback at design time, before implementation, about how protocols will behave, before implementation, and by allowing easy and fast experimentation and detailed analysis with many design alternatives; and (2) *catching very subtle errors*, which were not uncovered by traditional simulation *and* testing methods for already designed protocols.

Since modeling is an abstraction process there are two key questions that need to be addressed in every attempt to formally analyze a real-world system:

1. Does the formal model adequately capture intended model ?
2. Does the formal model have the desired properties ?

The spectrum of tools that can help us to answer these questions includes *light-weight techniques* such as symbolic execution, state space exploration, and model checking, as well as *heavy-weight* *techniques* such as informal mathematical proofs or rigorous formal proofs.

Similar to software engineering, the specification style has a major impact on the effort that goes into modeling and formal analysis. Qualities that we consider important are:

1. small representational distance (minimizes gap between formal model and reality),
2. abstract from all nonessential aspects (reduces explosion of complexity),
3. simplicity, mathematical elegance (enables better understanding),
4. modularity (increases maintainability),
5. executability (allows use of light-weight formal methods),

A typical development cycle of a formal specification consists of the use of light-weight methods (symbolic execution and state space exploration in this paper) to validate the specification against the informal description and to check if the system has the desired properties. The results should then be used to confirm the expectations or otherwise lead to a revision of the design. Once a sufficient degree of maturity is reached more heavy-weight analysis can be employed. Again new problems may be detected at this stage leading to another revision and iteration of the development cycle.

3 Protocol Objective and Assumptions

Security Gateways (SGs) are network elements that aim to protect nodes from unwanted communications. They accomplish this by analyzing packets addressed to the nodes they protect and dropping certain kinds of packets if they might lead to unwanted communications. Secure traceroute (sectrace) is a protocol that discovers security gateways and configures Security Associations to enable security gateways to operate efficiently and flexibly by classifying packets as authenticated or unauthenticated. The analysis of authenticated packets can be based on the packet's origin. The analysis of unathenticated packets must be based on the packet's shape.

There are two general modes in which traffic is analyzed by security gateways. In *filter* mode, packets are reviewed based on their shape. For example, many firewalls filter packets based on port numbers, return addresses (ingress and egress filtering), or protocol-specific flags (SYN filtering). In *authentication* mode, the packets are reviewed based on some cryptographic authentication criteria such as a Message Authentication Code (MAC). A *dual mode* security gateway is one that combines these two modes. For instance, such a security gateway may place packets into an authenticated tunnel only if they have a certain shape. We will focus on security gateways of this kind. We assume that security gateways are able to authenticate traffic in an IPSec tunnel and also filter traffic based on various protocol-specific criteria.

The aim of a security gateway is to ensure that a class of what we call *initiation* traffic is routed only if it is authenticated. This class of traffic is contrasted with *response* traffic. The precise definition of initiation and response traffic determine the low-level protections provided by a security gateway. We will classify sectrace traffic as initiation or response traffic and assume that it is also possible to classify IPSec negotiation traffic in this way. The functionality of a security gateway for other protocols depends on this choice; in particular, a security gateway may classify a TCP SYN as an initiation packet and other TCP packets as responses.

It is assumed that routing determines a unique path from source to destination for each pair of nodes. Security associations are assumed to be asymmetric; that is, when initiator A negotiates a security association with responder B, then this security association applies only to packets sent by A to B and not to those sent by B to A. Each node is assumed to have a public key certificate and a valid certificate chain from a root key. Each node is also assumed to have a set of root keys it trusts. For simplicity we assume that any request that is authenticated is also authorized: in practice, this could be much more complicated, but much of this complexity is orthogonal to the initial challenge of authentication.

With these assumptions, objective of sectrace can be stated more precisely as follows: the protocol uses PKI, routing, and IPSec to ensure that all initiation traffic is efficiently authenticated to its destination and all of the security gateways that must be traversed to reach it.

The mechanism of sectrace is similar to that of traceroute. An IP node uses the traceroute protocol to determine the routing path between itself and a chosen destination node. This is accomplished by sending a sequence of messages toward the destination with increasing time-to-live (TTL) values and reconstructing the path from ICMP responses generated by exhausted TTLs. Sectrace works by sending a sequence of requests toward a destination and processing responses from security gateways that intercept the requests. The protocol establishes a security association *route* consisting of a collection of security associations that authenticate packets to security gateways. This authentication may be with the originator or with any security gateway on the routing path so long as the recipient trusts the signature of the authenticating node.

4 Example

An example illustrates the basic strategy. Assume the following configuration:

```
C ----- SG1 ----- SG2 ----- S
```

where the dashes indicate the routing path from client node C to server node S. $SG1$ and $SG2$ are two security gateways that lie on this path; these nodes aim to protect C and S from unauthenticated initiation traffic. The goal is for C to establish a TCP connection to S. To accomplish this, C must gain the approval of $SG1$, $SG2$, and S using IPSec authentication based on its certificate and those of $SG1$ and $SG2$.

Sectrace initiates path setup by sending a sectrace Route Request (*RReq*) toward S. This is intercepted by $SG1$, which replies to C with a sectrace Route Reply (*RRep*) containing a list of root keys it will accept for authentication. C then creates an IPSec security association with $SG1$ (if it can) and sends a new *RReq* toward S, via $SG1$. The new *RReq* is encapsulated in the security association between C and $SG1$ established in the previous stage; $SG1$ decapsulates the *RReq* and forwards it toward S. This *RReq* is intercepted by $SG2$, which responds to it with an *RRep*. We do not assume that paths are symmetric, so this *RRep* may or may not pass through $SG1$, but any security gateways that it encounters will allow it to pass back to C. There are three possibilities at this point: (1) $SG2$ trusts the root of $SG1$, (2) $SG2$ trusts the root of C but not $SG1$, and (3) $SG2$ trusts neither root.

1. When it receives the *RRep* from $SG2$, the client C compares the trusted root information in it to what it received from $SG1$. If $SG2$ trusts the root of $SG1$, then C sends a sectrace Security Association Request (*SAReq*) to $SG1$ requesting that $SG1$ establish an SA with $SG2$. After this is negotiated by $SG1$ and $SG2$, security gateway $SG1$ sends a positive Security Association Reply (*SARep*) to C declaring that the necessary security association exists. C then sends another *RReq* toward S using the security association with $SG1$; this *RReq* is then forwarded by $SG1$ toward S using its SA with $SG2$.

2. If $SG2$ trusts the root of C but not the root of $SG1$, then C establishes an SA with $SG2$ using its SA with $SG1$. Once this is done, C sends an *RReq* toward S encapsulated in its (nested) security associations with $SG1$ and $SG2$.

3. If neither C nor $SG1$ have roots that $SG2$ trusts, then an authenticated path from C to S cannot be established and the protocol terminates unsuccessfully.

When S finally receives an *RReq* from C, it responds with an *RRep* that indicates the roots it trusts. C uses this to establish a security association with S based on a similar technique to the one used for $SG1$ and $SG2$. After C completes its sectrace, there will be three security associations, one for each security gateway and the server, so that packets sent from the client to the server will always be authenticated at the server and each security gateway. This security association route can be used to transmit initiation packets from C to S.

The sectrace protocol aims to amortize the cost of security association setup across multiple communications, so, when a node receives an *SAReq*, it checks to see if the requested security association already exists and immediately sends a positive response if it does. Security associations are assumed to expire after a period of time. We make no special provision for this; if a communication link is broken by a security association expiration, then sectrace is run again to renew the necessary security association. We assume that security gateways have some means of notifying a node that sectrace is needed, but have not specified that here. It would be reasonable to configure security associations so that they are automatically renewed if they are used, with new session keys being created as appropriate.

5 Informal Description of Sectrace

We split the problem of discovering and configuring a security association route into two parts. First there is a sectrace *signaling protocol* that sends a sequence of messages to discover security gateways, learn their trust relations, and request that security associations be established to authenticate future communications. Second, there is a sectrace security association *selection protocol* that determines which security associations to request based on existing security associations and newly-discovered security gateways and their trust relations. We describe the signaling protocol assuming the existence of a security association selection protocol and then describe the chosen security association selection protocol. We then provide an extended example of the two protocols on a network with four security gateways. We then provide a discussion of tradeoffs and limitations in sectrace and conclude by mentioning some related work.

5.1 Signaling Protocol

An instance of the protocol is created for each client/server pair. Only the client maintains state aside from what is needed to establish security associations and the security association database entries that result. Messages are sent via UDP so it is assumed that they include a source and destination. We use a dot notation to indicate information in messages. For instance, *RReq.src* and *RReq.dst* are the source and destination attributes of *RReq*.

5.1.1 Message Types

The signaling protocol involves four kinds of messages: *RReq*, *RRep*, *SAReq*, *SARep*. We indicate fields other than the source and destination field and whether the message is classified as an initiation or a response.

RReq	Route Request, initiation
RRep	Route Reply, response
RRep.tr	List of trusted roots
RRep.done	Positive or negative
RRep.server	Server that was the destination of the corresponding *RReq*
RRep.cert	Certificate of the sender
SAReq	Security Association Request, initiation
SAReq.resp	Requested responder
SARep	Security Association Reply, response
SARep.resp	Requested responder
SAReq.done	Positive or negative

Note that for *SAReq* no additional field is needed for the requested initiator, because it is *SAReq.dst*, the destination node.

Messages *RReq* and *RRep* are said to *match* if $RReq.dst = RRep.server$ and $RReq.src = RRep.dst$. Messages *SAReq* and *SARep* are said to *match* if $SAReq.dst = SARep.src$ and $SAReq.src = SARep.dst$.

The protocol uses the following timeouts:

RReqTmt	Route Request Timeout
SAReqTmt	SA Request Timeout

At the client the following state is maintained:

RReqTmtCntr	*RReq* timeout counter: integer
SAReqTmtCntr	*SAReq* timeout counter: integer
RRepLst	*RRep* list: list of prior *RRep* messages
OReq	outstanding request: *RReq* or *SAReq*

To avoid repetition, the done flag on an *RRep* is negative unless otherwise indicated.

5.1.2 Establishing a Security Association

A security association is established between an initiator I and a responder R; it is negotiated starting with a message from the former to the latter. To achieve authentication, the sectrace protocol employs IPSec with *encapsulated security payload* (ESP) using null encryption and authentication via a MAC. The security association is established in tunnel mode and entered into the security association database (SAD). In sectrace, a security association is established with respect to a server S and the security association is applicable to packets ultimately addressed to S. A packet is ultimately addressed to S if the packet obtained by removing all ESP authentication headers is addressed to S. So, if a packet ultimately addressed to S is processed by I, it will add an ESP header authenticating the packet to R, append a new IP header for R, and dispatch the packet to R. Responder R will remove the new IP header, check and remove the ESP header and trailer, and process the packet. The policy for applicable security associations is stored in the security policy database (SPD).

5.1.3 Client Actions

Client initiates protocol The client initiates the protocol by sending an *RReq* toward a server S. When it does so it sets the *RReqTmt* timeout and sets *OReq* to be *RReq*.

Client receives RRep If the client receives an *RRep* message, then it first checks whether it matches the outstanding request *OReq*. If it does not, then the *RRep* is ignored. If it does, then the client clears the *RReqTmt* timer. It inspects the collection of roots in the message and the *RRep* list to choose an initiator using the sectrace security association selection protocol (see Section 5.3). The client creates an *SAReq* with the chosen initiator (*SAReq.dst*) and uses the source of the *RRep* as the requested responder (*SAReq.resp* = *RRep.src*). This *SAReq* is entered as the outstanding request. The *SAReq* and *RRep* messages are added to the list *RRepLst* of prior *RRep* messages. The client stores the certificate given in the *RRep.cert* field as the certificate of the sender; this certificate will be used if subsequent *RRep* messages are received from gateways further along the path to the server.

If the chosen initiator is the client itself, then the client attempts to establish a security association with the responder (that is, the sender of the *RRep* message). If this succeeds, and the server is the responder, then the protocol terminates successfully. If it succeeds and the server is not the responder, then the client sets the *RReqTmt* timeout, sets the counter *RReqTmtCntr* to zero, sets the outstanding request to *RReq*, and sends a new *RReq* toward the server.

If the initiator is not the client, then it sends an *SAReq* to the initiator indicating the desired responder. The client sets the *SAReqTmt* timeout, sets the *SAReqTmtCntr* counter to zero, and enters the *SAReq* as the outstanding request.

Client RReqTmt expires If the *RReqTmt* expires, then the client checks to see if the *RReqTmt* counter is greater than or equal to three. If it is, the protocol terminates unsuccessfully. Otherwise the *RReqTmt* counter is incremented, the *RReqTmt* timeout is reset, and the client resends the outstanding *RReq*.

Client receives SARep If the client receives an *SARep* message, then it first checks whether it matches the outstanding request *OReq*. If it does not, then the *SARep* is ignored. If it does match the *OReq* the *SAReqTmt* is cleared. If the *SARep* is positive and the responder *SARep.resp* is the server, then the protocol ends successfully. If the *SARep* is positive and the responder is not the server, then the client sends a new *RReq* message toward S after setting it as the outstanding request, setting the *RReqTmt* timeout, and setting the *RReqTmt* counter to zero. If the *SARep* is negative, then the protocol ends unsuccessfully.

Client SAReqTmt expires If the *SAReqTmt* expires, then the client checks to see if the *SAReqTmt* counter is greater than or equal to three. If it is, the protocol terminates unsuccessfully. Otherwise the counter is incremented and the client resends the outstanding *SAReq*.

5.1.4 Security Gateway Actions

Security gateway receives RReq If the security gateway receives an authenticated *RReq* (that is, receives the *RReq* in a security association), then it routes the *RReq* on toward its destination using any applicable security associations. If the *RReq* is unauthenticated, then the security gateway sends an *RRep* with its trusted roots and its own certificate toward the *RReq* sender. It drops the *RReq*.

Security gateway receives RRep If the security gateway receives a *RRep* it routes it toward its destination using any applicable security association.

Security gateway receives SAReq If a security gateway receives an unauthenticated *SAReq*, then it drops it. If a security gateway receives an authenticated *SARep* addressed to another node, then it forwards it. If it receives an authenticated *SAReq* addressed to itself, then it checks to see if it has a policy in the security policy database and a security association in the security association database for the server in the request. If it does, then the security gateway sends a positive *SARep* to the client. If there is no applicable policy in place, then the security gateway attempts to establish a security association with the specified responder. If this is successful, it sends a positive *SARep* to the sender and updates its security association and policy databases to indicate that packets for the given server should use the newly established security association. If this is not successful, it sends a negative *SARep* to the sender.

Security gateway receives SARep If the security gateway receives an *RRep* it routes it forwards it toward its destination using any applicable security association.

Security gateway receives IPSec Negotiation Packets Security gateways should relay IPSec security association negotiation response packets without authentication.

Security gateway receives other packets In general, an SG relays a packet iff it is authenticated or it is a response packet.

5.1.5 Server Actions

Server gets RReq If the server gets an unauthenticated *RReq* for itself, it sends an *RRep* with its trusted roots. If it receives an authenticated *RRep*, it sends an *RRep* with a positive done flag.

5.2 Properties of the Protocol

Here are a few properties expected of the protocol.

1. The timeout *SAReqTmt* never expires when *OReq* is an *RReq*.
2. The timeout *RReqTmt* never expires when *OReq* is an *SAReq*.
3. If the client sends an *SAReq* to a node, then it will arrive there over an authenticated channel. That is, it will be within a security association.
4. *RReqTmtCtr* and *SAReqTmtCtr* are never greater than 3.

An *RReq* is never forwarded by a security gateway unless it is authenticated. In particular, an unauthenticated *RReq* cannot be used to gather information through a security association.

5.3 Security Association Selection

Given a collection of *RReq*s, a client must select the next security association to request. The sectrace protocol does this by selecting the intermediate initiator that provides the "shortest" security association to the given responder. In general, there is no guarantee that a security association route from the client to the server exists. However, if any such route does exist, then the sectrace protocol will find one.

5.3.1 Example

A node A is *trusted* by a node B if the root of A is one of the trusted roots of B. If A is trusted by B we write $A \ll B$. The relation $\ll$ is called the *trust relation*. Consider again the example from Section 4:

```
    C ----- SG1 ----- SG2 ----- S
```

For case (1) from Section 4, we have $C \ll SG1 \ll SG2$, so the protocol establishes associations from C to $SG1$ and from $SG1$ to $SG2$. However, in case (2) we have only $C \ll SG1$ and $C \ll SG2$ so the associations are from C to $SG1$ and from C to $SG2$. Let us assume below that $SG2 \ll S$.

In treating this mathematically, it is helpful to view security associations as determined by a function on an initial segment of the numbers. The example is then rendered as follows:

```
    0 -----  1  -----  2  ----- 3
```

and there is a function f from the non-zero numbers $1, 2, 3$ into the numbers $0, 1, 2, 3$ such that, for each n, $f(n)$ is the node that authenticates packets to n. That is, $f(n) < n$ and $f(n) \ll n$ for each n. In case (1) of the example, we have $f(1) = 0$, $f(2) = 1$, $f(3) = 2$. But in case (2) we have $f(1) = 0$, $f(2) = 0$, $f(3) = 2$. In case (3) of the example, there is no function f that satisfies the criterion that $f(n) \ll n$ for each n.

5.3.2 Theory

Let $\mathbb{N}^+$ denote non-zero numbers and $\mathbb{N}$ be the natural numbers including 0. A trust relation $\ll$ is a binary predicate on $\mathbb{N}$. To simplify matters, we assume that $m \ll n$ implies $m < n$, although this is not important to the formalization. The element 0 corresponds to the client and each number to an SG or the server. For instance, if there are two SGs then we are concerned with trust relations between $0, 1, 2, 3$, where 3 is the server and $1, 2$ are SGs. In general, we need to find a function $f : \mathbb{N}^+ \to \mathbb{N}$ with the following properties:

Trusted: $f(x) \ll x$

Encapsulation: If $x < y$ then $f(y) \leq f(x)$ or $x \leq f(y)$

Let us say that a partial function $f : \mathbb{N}^+ \to \mathbb{N}$ is a *trusted encapsulation* for n if it is defined on $1, \ldots, n$ and is an encapsulation on this initial segment of numbers. For a given relation $\ll$ and number n, there may be no trusted encapsulation for n.

We define a partial function *fs* by induction as follows:

Base Case *fs*$(1) = 0$ if $0 \ll 1$, otherwise it is undefined.

Inductive Case Assume that *fs* is defined for $1, \ldots, n$ and it is a trusted encapsulation on these this initial segment. Let *fs*$(n+1) =$ the largest value m such that *fs* restricted to $0, \ldots, n+1$ is a trusted encapsulation, if there exists such an m, otherwise let $f(n+1)$ be undefined. If *fs* is undefined on any of the values $1, \ldots, n$, then *fs*$(n+1)$ is also undefined.

We say that *fs* is the *shortest security association function* induced by the relation $\ll$.

Lemma If f is a trusted encapsulation for n,
then *fs* is a trusted encapsulation for n and $f(k) \leq$ *fs*(k) for each $k \leq n$.

Proof This proceeds by induction on n. Suppose that $n = 1$. Then $f(1) = 0$ and *fs*$(1) = 0$ so there is no problem. Suppose that the lemma holds for n and f is a trusted encapsulation for

$n+1$. In this case, f is also a trusted encapsulation for n, so fs is a trusted encapsulation for n and $f(k) \leq fs(k)$ for each $k \leq n$. Now, we need to show that $fs(n+1)$ exists and that it is $\geq f(n+1)$. Since $fs(n+1)$ is defined to be the largest value such that fs is an trusted encapsulation, given the values of fs on $1, \ldots, n$, it suffices to show that fs' is a trusted encapsulation where fs' is equal to fs on $1, \ldots, n$ and $fs'(n+1) = f(n+1)$. Clearly $fs'(n+1) \ll n+1$ since $f(n+1) \ll n+1$. Suppose $x < y$ for some $x, y < n+1$. Then the encapsulation property is satisfied by fs on x, y by induction, and fs equals fs' on these values. Suppose, however, that $y = n+1$. We need to show that

$$fs'(n+1) \leq fs'(x) \text{ or } x \leq fs'(n+1).$$

This is the same as showing that

$$fs'(n+1) > fs'(x) \text{ implies } x \leq fs'(n+1)$$

Suppose $fs'(n+1) > fs'(x)$. Then $f(n+1) > fs(x)$. By induction $fs(x) \geq f(x)$, so $f(n+1) > f(x)$. Thus $x \leq f(n+1)$ because f is an encapsulation and $f(n+1) = fs'(n+1)$ by definition.

Theorem There exists a trusted encapsulation for n iff fs is a trusted encapsulation for n.

Proof Necessity is obvious and sufficiency is a corollary of the previous lemma.

5.3.3 Implementation

Section 5.3 describes the security association selection algorithm in formal terms. This section describes the algorithm in a form more amenable to implementation. This algorithm assumes that the client does not know any of the gateways on the path to the server when the protocol starts.

The client maintains an ordered list of gateways

$$g_0,\ g_1,\ g_2,\ \cdot\cdot\ ,\ g_i\ ,\ \cdot\cdot\ ,\ g_n$$

that are available on the path to S (the client itself is considered a gateway for the purposes of this algorithm). At the start of the algorithm, the list contains only the client, which is the g_0 element of the list. Given a gateway g (not in the list) and a list of roots that g trusts, the client chooses an IPSec tunnel endpoint by selecting g_i such that g_i is the rightmost element that has a certificate that g trusts (in other words, g_i such that g_i is trusted by g and g_j is not trusted by g for all $j > i$). The client can then instruct g_i to establish a tunnel with g.

The client must then remove $g_{i+1}, g_{i+2}, .., g_n$ from the list because these gateways are not available as tunnel endpoints because a tunnel from, say, g_{i+1} to some other gateway g' that is beyond g would lead to an overlapping tunnel like:

```
g_i  -----  g_i+1  --..---  g  --..--  g'
     ===================
                    =================
```

After the removal of the unavailable gateways the list looks like

$$g_0,\ g_1,\ g_2,\ \cdot\cdot\ ,\ g_i$$

Finally, the client can add the newly discovered gateway, g, so that it can be used to establish tunnels with gateways further along the path. The list looks like

$$g_0,\ g_1,\ g_2,\ \cdot\cdot\ ,\ g_i,\ g$$

after the addition of g.

6 Extended Example

Consider the following configuration of two clients, as server, and four security gateways:

```
               C2
                |
               SG1
                |
               SG2
                |
C1 ----- SG3 ----- SG4 ---- S
```

Assume

- $C1 \ll SG2, SG3, S$
- $C2 \ll SG1, SG3, S$
- $SG1 \ll SG2$
- $SG3 \ll SG4, S$
- $SG4 \ll S$

Intuitively, $SG1$, $SG2$ are gateways in one autonomous system and $SG3$, $SG4$ are gateways in another. The nodes $C1$ and $C2$ have relations with gateways in each system and are currently situated within the system protected by $SG1$, $SG2$. All this information is depicted in Fig. 4, which corresponds to the initial state of the protocol.

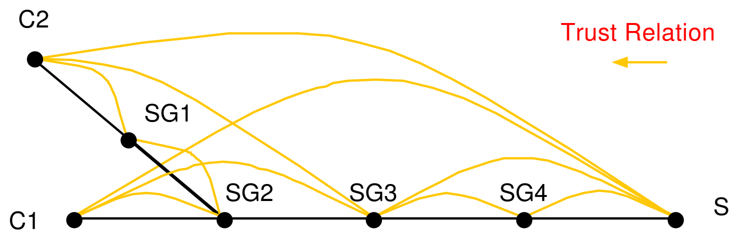

Figure 4: Typical Scenario

In the following we walk through an execution of the protocol, in which we assume that $C1$ invokes the protocol first. Some intermediate states of this part of the protocol execution are depicted in Fig. 5 and the resulting state is depicted in Fig. 6.

1. $C1$ sends an $RReq$ toward S
2. $SG2$ gets this $RReq$ and sends an $RRep$ to $C1$
3. $C1$ gets this $RRep$ and establishes an SA with $SG2$
4. $C1$ sends an $RReq$ toward S using the SA with $SG2$
5. $SG2$ gets this authenticated $RReq$ and forwards it toward S
6. $SG3$ gets this $RReq$ from $C1$ but it is not authenticated

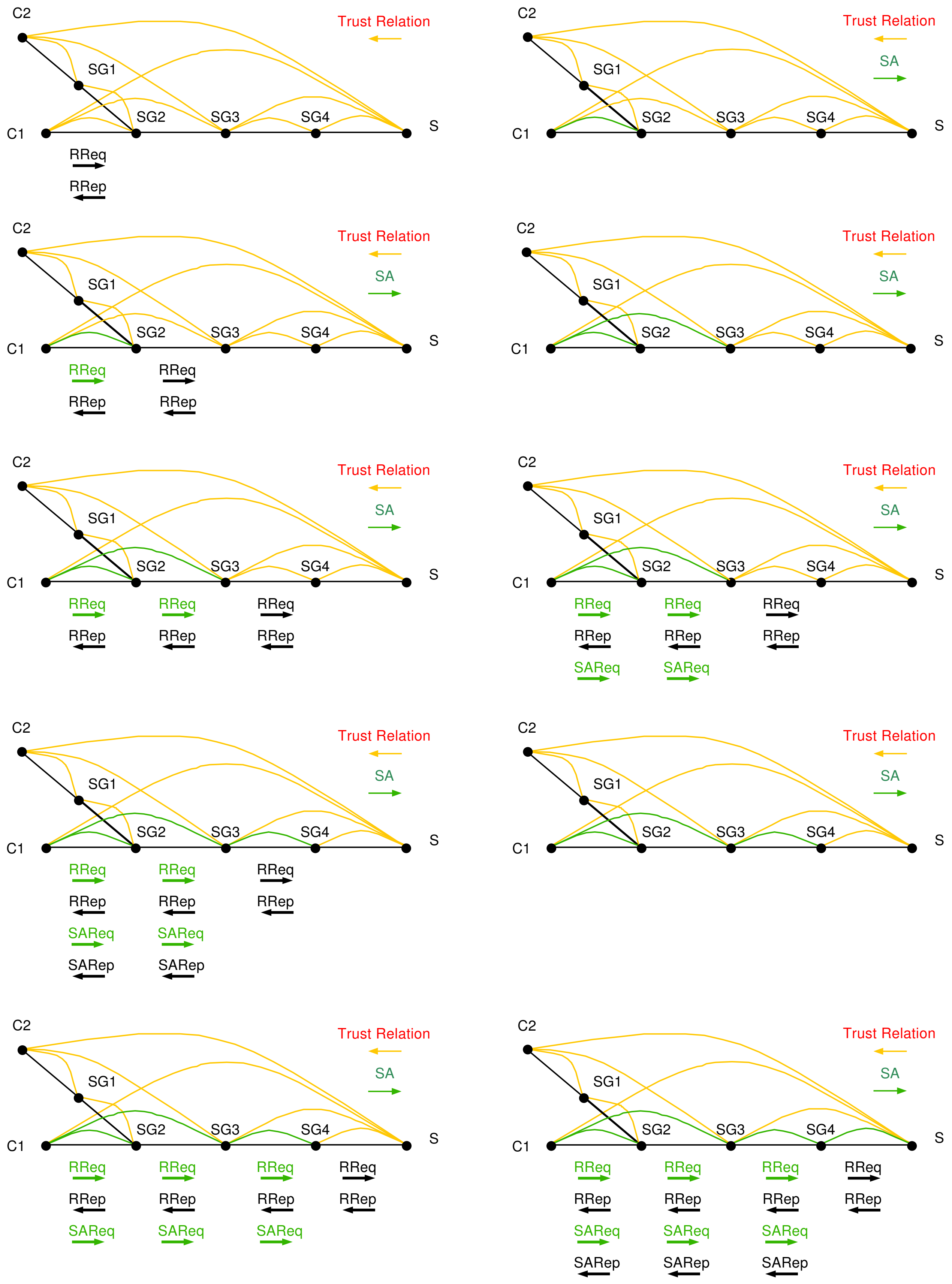

Figure 5: Protocol Execution for C1

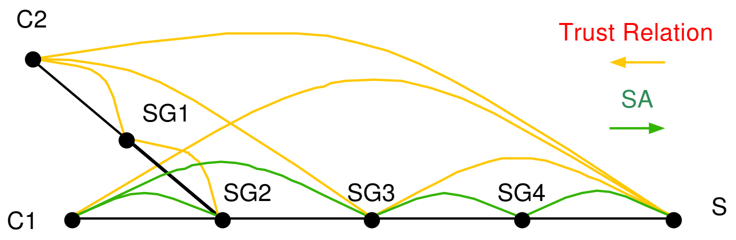

Figure 6: Result of Protocol Execution for C1

7. $SG3$ sends an $RRep$ to $C1$
8. $C1$ gets the $RRep$ from $SG3$
9. $C1$ establishes an SA with $SG3$
10. $C1$ sends an $RReq$ toward S using the SAs for $SG2$ and $SG3$
11. $SG2$ and $SG3$ receive this $RReq$ and forward it toward S
12. $SG4$ receives the $RReq$ and responds to $C1$ with an $RRep$
13. $C1$ sends an $SAReq$ to $SG3$, which establishes an SA with $SG4$
14. $SG3$ sends an $SARep$ to $C1$
15. $C1$ sends an $RReq$ toward S, which is authenticated though $SG2$, $SG3$ and $SG4$
16. S gets an $RReq$ from $C1$ and responds with an $RRep$
17. $C1$ sends an $SAReq$ to $SG4$
18. $SG4$ gets the $SAReq$ from $C1$ and establishes an SA with S
19. $SG4$ sends an $SARep$ to $C1$
20. $C1$ is now able to send authenticated packets to S

    The security associations now look like in Fig. 6, which we also write as:

```
C1 ----- SG2 ----- SG3 ----- SG4 ----- S
   =====           =====     =====
   ===============
```

Now $C2$ becomes active an invokes the protocol as well. Again some intermediate states of this part of the protocol execution are depicted in Fig. 7.

21. $C2$ sends an $RReq$ toward S
22. $SG1$ gets this $RReq$ and sends an $RRep$ to $C2$
23. $C2$ establishes an SA with $SG1$
24. $C2$ sends an $RReq$ toward S

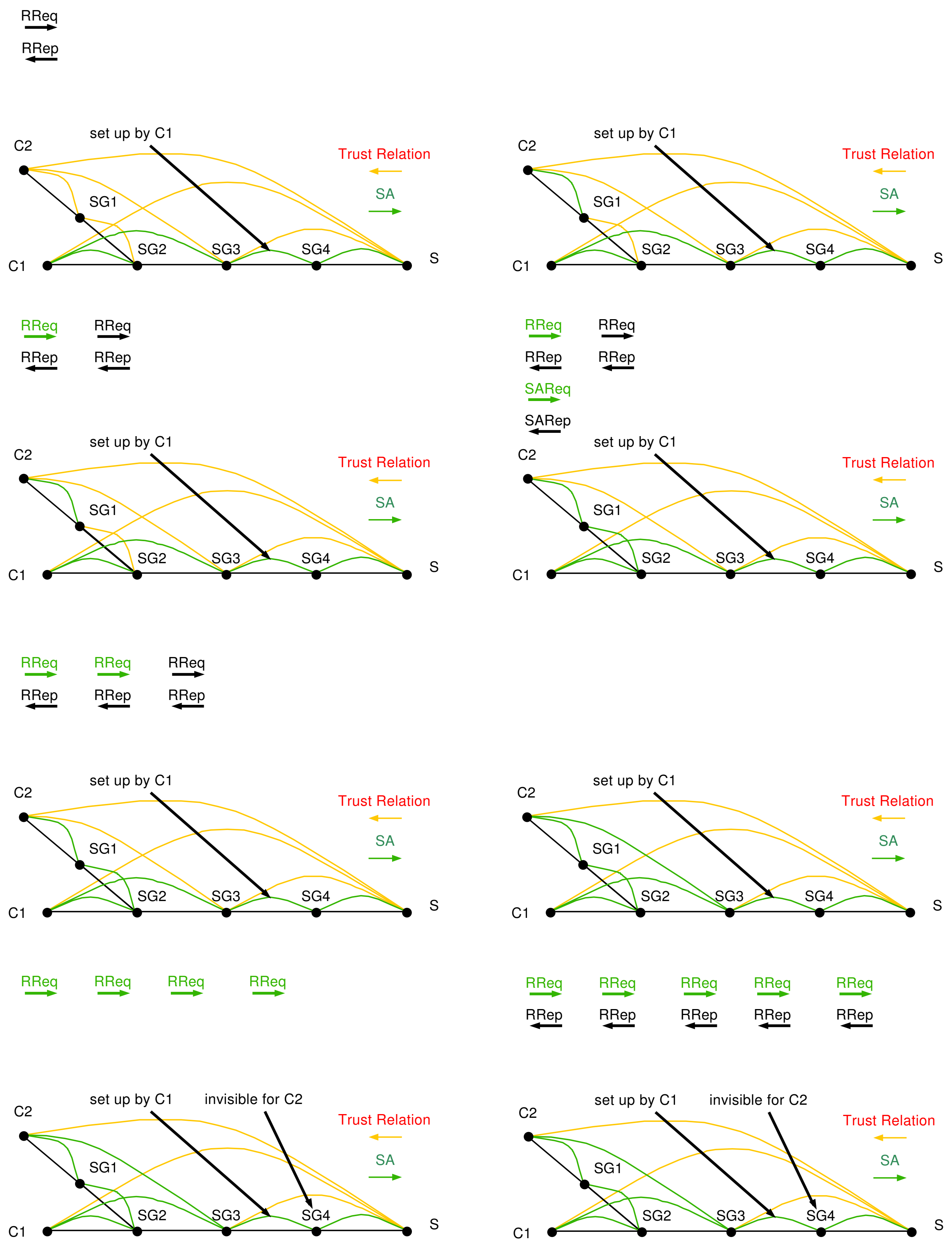

Figure 7: Protocol Execution for C2

25. $SG2$ gets this $RReq$ and sends an $RRep$ to $C2$

26. $C2$ sends an $SAReq$ to $SG1$, which establishes an SA with $SG2$ and sends an $SARep$ to $C2$

27. $C2$ sends an $RReq$ toward S

28. $SG3$ gets the $RReq$ from $C2$ and sends an $RRep$

29. $C2$ establishes an SA with $SG3$

30. $C2$ sends an $RReq$ toward S within SAs for $SG1$ and $SG3$

31. $SG1$ forwards this $RReq$ within an SA between itself and $SG2$

32. $SG3$ receives this $RReq$ and forwards it to $SG4$ using the SA established by $C1$ between $SG3$ and $SG4$

33. $SG4$ forwards this $RReq$ to S using the SA between $SG4$ and S

34. S sends an $RRep$ with a positive done flag to $C2$

35. $C2$ is now able to send authenticated packets to S

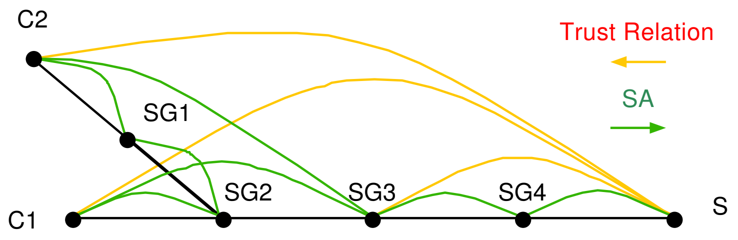

Figure 8: Result of Protocol Execution for C1 and C2

The final result for $C2$ looks as follows (see also Fig. 8):

```
C2 ----- SG1 ----- SG2 ----- SG3 ----- SG4 ----- S
   =====     =====                =====     =====
   =========================
```

Note that the associations $SG3$ ===== $SG4$ ===== S were "invisible" to $C2$ because they had already been established by $C1$.

7 Formal Specification

The formal specification of the sectrace protocol in Maude closely follows the informal description [GGM02, GGM03] given previously. In the following we just give a flavor of how the specification looks, by giving the formalization of some protocol rules in Maude without giving the details about the underlying algebraic signature and equational theory. For the full formal specification and detailed analysis results we refer to `http://formal.cs.uiuc.edu/stehr/sectrace_eng.html`.

Recall the informal description from Section 5.1.3:

> The client initiates the protocol by sending an $RReq$ toward a server S. When it does so it sets ... and sets $OReq$ to be $RReq$.

This is formalized by the following multiset rewriting rule:

```
rl  sectrace-start(node,client,server) =>
    sectrace-rreq(node,client,server,
                  sMessageList(rreq(client,server)),
                  eMessageList)
    ipsec-send(node,ip(client,server,rreq(client,server))) .
```

The terms `sectrace-start(node,...)` and `sectrace-rreq(node,...)` represent local states of the sectrace client process running at `node`. The term `ipsec-send(node,...)` is part of the IPSec API and represents a service request that will be processed at `node` by the IPSec layer, which is specified independently.

Similarly, recall the informal specification from the same section:

> If the client receives an *RRep* message, then it first checks whether it matches the outstanding request *OReq*. If it does not, then the *RRep* is ignored. ... It inspects the collection of roots in the message and the *RRep* list to choose an initiator using the sectrace security association selection protocol. The client creates an *SAReq* with the chosen initiator *SAReq.dst* and uses the source of the *RRep* as the requested responder (*SAReq.resp* = *RRep.src*). This *SAReq* is entered as the outstanding request. The *SAReq* and *RRep* messages are added to the list *RRepLst* of prior *RRep* messages.

This is formalized by two conditional multiset rewriting rules, which are executed sequentially.

```
crl sectrace-rreq(node,client,server,
                  sMessageList(rreq(client,server)),
                  rreplist)
    rootinfo(node,myroot,mytrustedroots)
    ipsec-delivered(node,rinterface,attrs,sabundle,message) =>
    sectrace-rrep3(node,client,server,
                   sMessageList(sareq(client,server,initiator,responder)),
                   (rreplist sMessageList(message)))
    rootinfo(node,myroot,mytrustedroots)
    if not(contains(rreplist,message)) /\
       ip(responder,dest,rrep(client,server,root,trustedroots,done)) := message /\
       done == false /\
       initiator := select(client,myroot,
                           (rreplist sMessageList(message)),responder) /\
       initiator =/= client .

crl sectrace-rrep3(node,client,server,
                   sMessageList(sareq(client,server,initiator,responder)),
                   (rreplist sMessageList(message)))
    rootinfo(node,myroot,mytrustedroots) =>
    sectrace-sareq(node,client,server,
                   sMessageList(sareq(client,server,initiator,responder)),
                   (rreplist sMessageList(message)))
    rootinfo(node,myroot,mytrustedroots)
    ipsec-send(node,message')
    if message' := ip(client,initiator,sareq(client,server,initiator,responder)) .
```

The terms `sectrace-rreq(node,...)`, `sectrace-rrep3(node,...)`, and `sectrace-sareq(node,...)` represent local states of the sectrace client running at `node`, which in the last two components carry the information about *OReq* and *RRepLst*. The sequential execution of the two rules is achieved by introducing the intermediate state `sectrace-rrep3(node,...)`. The term `ipsec-delivered(node,...)` is part of the IPSec API, and delivers a message to the sectrace client at `node` after it has been received and processed by IPSec. The term `rootinfo(node,...)` represents the local knowledge of the sectrace client about its assigned root CA and all root CAs it trusts.

8 Formal Analysis

To analyze the specification using Maude we need to specify a concrete initial state, which contains information about the nodes (which can act as hosts or security gateways) and an enumeration of all subnets, representing the network topology. We use the network topology from the example of Section 6. Furthermore, the initial state contains for each node its network interfaces, its routing table, information about trusted certificate authorities, the initial security association database and the initial security policy database. All this constitutes a multiset that is defined as follows:

```
eq network =

    shost(node("C1")) shost(node("C2")) ...

    sgw(node("SG1")) sgw(node("SG2")) ...

    sectraced(node("C1")) sectraced(node("C2")) ...

    subnet(sAddrSet(addr("C1a")) sAddrSet(addr("SG2b")))
    subnet(sAddrSet(addr("C2a")) sAddrSet(addr("SG1a"))) ...

    interfaces(node("C1"),sAddrSet(addr("C1a")))
    interfaces(node("C2"),sAddrSet(addr("C2a"))) ...

    routetab(node("C1"),
       sRouteList(route(addr("C1a"), addr("C1a"), addr("C1a")))
       sRouteList(route(addr("C2a"), addr("C1a"), addr("SG2b"))) ...)
    routetab(node("C2"),
       sRouteList(route(addr("C1a"), addr("C2a"), addr("SG1a")))
       sRouteList(route(addr("C2a"), addr("C2a"), addr("C2a"))) ...)

    rootinfo(node("C1"), addr("CAC1"), sAddrSet(addr("CAC1")))
    rootinfo(node("C2"), addr("CAC2"), sAddrSet(addr("CAC2"))) ...

    sadb(node("C1"),eSASet)
    sadb(node("C2"),eSASet)
    ...
    spdb(node("C1"),
          eSPList,
          sSPList(sp(isinitiation,eSAList)) sSPList(sp(isresponse,eSAList)))
```

```
   spdb(node("C2"),
         eSPList,
         sSPList(sp(isinitiation,eSAList)) sSPList(sp(isresponse,eSAList)))
   ...
```

The next ingredient for formal analysis is the environment model, which in this case defines how sectrace is invoked. We begin with a *sequential execution plan* which invokes sectrace sequentially at two different clients $C1$ and $C2$. This can be expressed again by multiset rewrite rules, which control the execution of sectrace using intermediate states represented by `start`, `next`, and `terminated`.

```
rl start =>
   sectrace-start(node("C1"), addr("C1a"), addr("Sa")) .

rl sectrace-terminated(node("C1"), addr("C1a"), addr("Sa"))
   => next .

rl next =>
   sectrace-start(node("C2"), addr("C2a"), addr("Sa")) .

rl sectrace-terminated(node("C2"), addr("C2a"), addr("Sa"))
   => terminated .
```

The execution from the initial state consisting of the initial state of the network and the initial state of the execution plan can be invoked using the `rew` command of Maude, which gives the following result:

```
rew network start .
rewrites: 270753 in 160ms cpu (160ms real) (1692206 rewrites/second)
result State:
terminated
...
sadb(node("C1"),  sSASet(sa(addr("C1a"),  addr("SG2b"), 0))
                  sSASet(sa(addr("C1a"),  addr("SG3a"), 0)))
sadb(node("C2"),  sSASet(sa(addr("C2a"),  addr("SG1a"), 0))
                  sSASet(sa(addr("C2a"),  addr("SG3a"), 0)))
sadb(node("S"),   sSASet(sa(addr("SG4a"), addr("Sa"), 0)))
sadb(node("SG1"), sSASet(sa(addr("C2a"),  addr("SG1a"), 0))
                  sSASet(sa(addr("SG1a"), addr("SG2a"), 0)))
sadb(node("SG2"), sSASet(sa(addr("C1a"),  addr("SG2b"), 0))
                  sSASet(sa(addr("SG1a"), addr("SG2a"), 0)))
sadb(node("SG3"), sSASet(sa(addr("C1a"),  addr("SG3a"), 0))
                  sSASet(sa(addr("C2a"),  addr("SG3a"), 0))
                  sSASet(sa(addr("SG3a"), addr("SG4a"), 0)))
sadb(node("SG4"), sSASet(sa(addr("SG3a"), addr("SG4a"), 0))
                  sSASet(sa(addr("SG4a"), addr("Sa"), 0)))
...
spdb(node("C1"),
  eSPList,
  sSPList(sp(towards(addr("Sa")),
    sSAList(sa(addr("C1a"), addr("SG3a"), 0))
    sSAList(sa(addr("C1a"), addr("SG2b"), 0))))
  sSPList(sp(isinitiation, eSAList))
  sSPList(sp(isresponse, eSAList)))
spdb(node("SG2"),
  sSPList(sp(towards(addr("Sa")),
    sSAList(sa(addr("C1a"), addr("SG2b"), 0))))
  sSPList(sp(towards(addr("Sa")),
    sSAList(sa(addr("SG1a"), addr("SG2a"), 0)))),
  sSPList(sp(isinitiation, eSAList))
  sSPList(sp(isresponse, eSAList)))
...
```

The above execution only shows the existence of a terminating execution of the protocol under the given execution plan. In order to check if all executions satisfying the sequential execution plan are terminating we perform a state space exploration using the `search` command of Maude:

```
search start =>! state:State .
Solution 1 (state 304)
states: 305  rewrites: 434933 in 370ms cpu (370ms real) (1175494 rewrites/second)
state:State -->
terminated
...
sadb(node("C1"), sSASet(sa(addr("C1a"), addr("SG2b"), 0))
                 sSASet(sa(addr("C1a"), addr("SG3a"), 0)))
sadb(node("C2"), sSASet(sa(addr("C2a"), addr("SG1a"), 0))
                 sSASet(sa(addr("C2a"), addr("SG3a"), 0)))
sadb(node("S"),  sSASet(sa(addr("SG4a"), addr("Sa"), 0)))
sadb(node("SG1"), sSASet(sa(addr("C2a"), addr("SG1a"), 0))
                  sSASet(sa(addr("SG1a"), addr("SG2a"), 0)))
sadb(node("SG2"), sSASet(sa(addr("C1a"), addr("SG2b"), 0))
                  sSASet(sa(addr("SG1a"), addr("SG2a"), 0)))
sadb(node("SG3"), sSASet(sa(addr("C1a"), addr("SG3a"), 0))
                  sSASet(sa(addr("C2a"), addr("SG3a"), 0))
                  sSASet(sa(addr("SG3a"), addr("SG4a"), 0)))
sadb(node("SG4"), sSASet(sa(addr("SG3a"), addr("SG4a"), 0))
                  sSASet(sa(addr("SG4a"), addr("Sa"), 0)))
...
spdb(node("C1"),
  eSPList,
  sSPList(sp(towards(addr("Sa")),
    sSAList(sa(addr("C1a"), addr("SG3a"), 0))
    sSAList(sa(addr("C1a"), addr("SG2b"), 0))))
  sSPList(sp(isinitiation, eSAList))
  sSPList(sp(isresponse, eSAList)))
...
No more solutions.
states: 305
```

```
rewrites: 434933 in 380ms cpu (1260ms real) (1144560 rewrites/second)
```

We conclude from this that all executions from the given initial state of the protocol terminate under the sequential execution plan with the above unique solution.

In reality, however, we cannot assume that the clients of sectrace synchronize and perform sequential invocations of sectrace. It is more realistic to assume that several clients can invoke sectrace at the same time. Therefore, we next investigate the *concurrent execution plan*, which can be represented as follows:

```
op start : -> State .
op terminated : -> State .

rl start =>
   sectrace-start(node("C1"), addr("C1a"), addr("Sa"))
   sectrace-start(node("C2"), addr("C2a"), addr("Sa")) .

rl sectrace-terminated(node("C1"), addr("C1a"), addr("Sa"))
   sectrace-terminated(node("C2"), addr("C2a"), addr("Sa")) =>
   terminated .
```

Now state space exploration yields 24 different solutions, some of them have terminated as before, but others consists of a deadlock state:

```
search start =>! state:State .

Solution 1 (state 139379) state:State -->
  ... terminated ...
Solution 2 (state 139423) state:State -->
  ... terminated ...
...

Solution 9 (state 155255) state:State -->
  ...
  sectrace-terminated(node("C2"), addr("C2a"), addr("Sa"))
  ipsec-received(node("S"), addr("Sa"), eAttrSet,
    sSAList(sa(addr("SG3a"), addr("Sa"), 0)),
    ip(addr("C1a"), addr("SG4a"),
       sareq(addr("C1a"), addr("Sa"), addr("SG4a"), addr("Sa"))))
  sectrace-sareq(node("C1"), addr("C1a"), addr("Sa"), ...) ...
Solution 10 (state 155543) state:State -->
  ...
  sectrace-terminated(node("C2"), addr("C2a"), addr("Sa"))
  ipsec-received(node("S"), addr("Sa"), eAttrSet,
    sSAList(sa(addr("SG3a"), addr("Sa"), 0)),
    ip(addr("C1a"), addr("SG4a"),
       sareq(addr("C1a"), addr("Sa"), addr("SG4a"), addr("Sa"))))
  sectrace-sareq(node("C1"), addr("C1a"), addr("Sa"), ...) ...
...
```

```
Solution 17 (state 165315) state:State -->
  ... terminated ...
...
Solution 24 (state 165783) state:State -->
  ... terminated ...

No more solutions.
states: 185271
rewrites: 556616699 in 3311100ms cpu (3342860ms real) (168106 rewrites/second)
```

It is not surprising that the state space is much bigger than with the sequential execution plan, namely 185271 states versus 305 states. An inspection of all the 24 solutions shows that there are inessential differences in the order of database entries (the security association and policy entries are required to be ordered in IPSec). We found that the solutions can be classified into four different types (corresponding to equivalence classes modulo the above differences), which we explain in the following.

8.1 Type 1 Solution

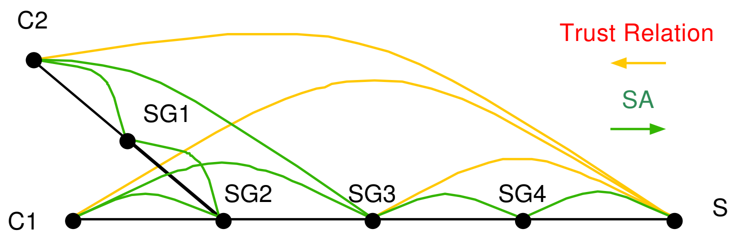

Figure 9: Type 1 Solution

In solutions of Type 1 (see Fig. 9) both clients have terminated successfully and have set up shortest security associations. This is the expected type of solution that has already been obtained under the sequential execution plan. The essential part including the security association database looks as follows:

```
terminated
sadb(node("C1"),  sSASet(sa(addr("C1a"),  addr("SG2b"), 0))
                  sSASet(sa(addr("C1a"),  addr("SG3a"), 0)))
sadb(node("C2"),  sSASet(sa(addr("C2a"),  addr("SG1a"), 0))
                  sSASet(sa(addr("C2a"),  addr("SG3a"), 0)))
sadb(node("SG1"), sSASet(sa(addr("C2a"),  addr("SG1a"), 0))
                  sSASet(sa(addr("SG1a"), addr("SG2a"), 0)))
sadb(node("SG2"), sSASet(sa(addr("C1a"),  addr("SG2b"), 0))
                  sSASet(sa(addr("SG1a"), addr("SG2a"), 0)))
sadb(node("SG3"), sSASet(sa(addr("C1a"),  addr("SG3a"), 0))
                  sSASet(sa(addr("C2a"),  addr("SG3a"), 0))
                  sSASet(sa(addr("SG3a"), addr("SG4a"), 0)))
sadb(node("SG4"), sSASet(sa(addr("SG3a"), addr("SG4a"), 0))
                  sSASet(sa(addr("SG4a"), addr("Sa"), 0)))
sadb(node("S"),   sSASet(sa(addr("SG4a"), addr("Sa"), 0)))
```

8.2 Type 2 Solution

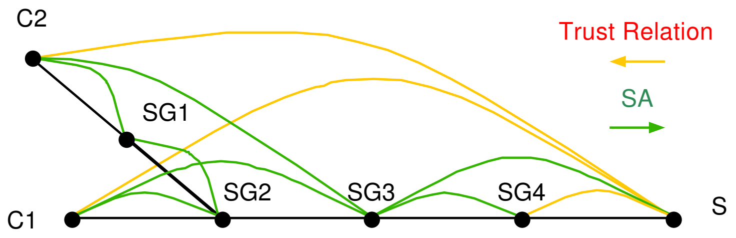

Figure 10: Type 2 Solution

In solutions of Type 2 (see Fig. 10) both clients terminate successfully, but the security association between $SG3$ and S is not the shortest one. The essential part of the solution looks as follows:

```
terminated
sadb(node("C1"),  sSASet(sa(addr("C1a"),  addr("SG2b"), 0))
                  sSASet(sa(addr("C1a"),  addr("SG3a"), 0)))
sadb(node("C2"),  sSASet(sa(addr("C2a"),  addr("SG1a"), 0))
                  sSASet(sa(addr("C2a"),  addr("SG3a"), 0)))
sadb(node("SG1"), sSASet(sa(addr("C2a"),  addr("SG1a"), 0))
                  sSASet(sa(addr("SG1a"), addr("SG2a"), 0)))
sadb(node("SG2"), sSASet(sa(addr("C1a"),  addr("SG2b"), 0))
                  sSASet(sa(addr("SG1a"), addr("SG2a"), 0)))
sadb(node("SG3"), sSASet(sa(addr("C1a"),  addr("SG3a"), 0))
                  sSASet(sa(addr("C2a"),  addr("SG3a"), 0))
                  sSASet(sa(addr("SG3a"), addr("SG4a"), 0))
                  sSASet(sa(addr("SG3a"), addr("Sa"), 0)))
sadb(node("SG4"), sSASet(sa(addr("SG3a"), addr("SG4a"), 0)))
sadb(node("S"),   sSASet(sa(addr("SG3a"), addr("Sa"), 0)))
```

8.3 Type 3 Solution

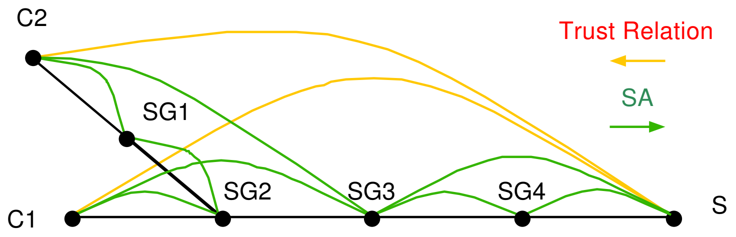

Figure 11: Type 3 Solution

In solutions of Type 3 (see Fig. 11) both clients terminate successfully, but one security association is redundant. Again we give the essential part of the solution:

```
terminated
sadb(node("C1"),  sSASet(sa(addr("C1a"),  addr("SG2b"), 0))
                  sSASet(sa(addr("C1a"),  addr("SG3a"), 0)))
```

```
sadb(node("C2"),  sSASet(sa(addr("C2a"),  addr("SG1a"), 0))
                  sSASet(sa(addr("C2a"),  addr("SG3a"), 0)))
sadb(node("S"),   sSASet(sa(addr("SG4a"), addr("Sa"), 0))
                  sSASet(sa(addr("SG3a"), addr("Sa"), 0)))
sadb(node("SG1"), sSASet(sa(addr("C2a"),  addr("SG1a"), 0))
                  sSASet(sa(addr("SG1a"), addr("SG2a"), 0)))
sadb(node("SG2"), sSASet(sa(addr("C1a"),  addr("SG2b"), 0))
                  sSASet(sa(addr("SG1a"), addr("SG2a"), 0)))
sadb(node("SG3"), sSASet(sa(addr("C2a"),  addr("SG3a"), 0))
                  sSASet(sa(addr("SG3a"), addr("SG4a"), 0))
                  sSASet(sa(addr("C1a"),  addr("SG3a"),0))
                  sSASet(sa(addr("SG3a"), addr("Sa"), 0)))
sadb(node("SG4"), sSASet(sa(addr("SG3a"), addr("SG4a"), 0))
                  sSASet(sa(addr("SG4a"), addr("Sa"), 0)))
```

8.4 Type 4 Solution

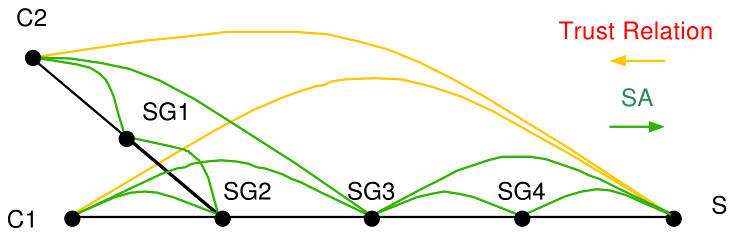

Figure 12: Type 4 Solution

In solutions of Type 4 (see Fig. 12) one client does not terminate, because an *SAReq* message is misguided by the security policy. The essential part of the solution looks as follows:

```
sectrace-terminated(node("C2"), addr("C2a"), addr("Sa"))

ipsec-received(node("S"), addr("Sa"), eAttrSet,
  sSAList(sa(addr("SG3a"), addr("Sa"), 0)),
  ip(addr("C1a"), addr("SG4a"),
      sareq(addr("C1a"), addr("Sa"), addr("SG4a"), addr("Sa"))))

sectrace-sareq(node("C1"), addr("C1a"), addr("Sa"), ...) ...
```

8.5 Traces

For each type of solution we can extract a trace, which leads to the given final state. In the case of the Type 1 solution we obtain precisely the trace detailed in Section 6.

8.5.1 Type 2 Trace

In the case of the Type 2 solution the trace contains the intermediate states depicted in Fig. 13. The problem with this solution is that is that the resulting security association from $SG3$ to S is not the shortest one. The trace shows that this is caused by an interference between the sectrace protocol executions of clients $C1$ and $C2$. The first state in Fig. 13 show a security association

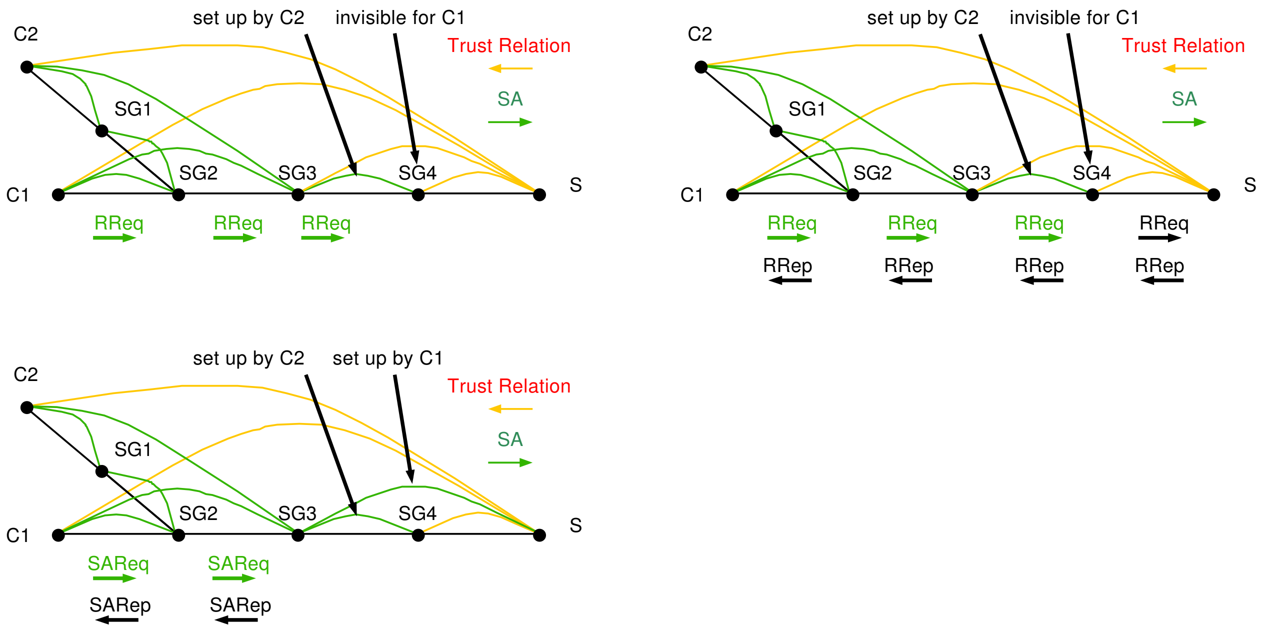

Figure 13: Type 2 Trace

from $SG3$ to $SG4$ that has been established in an earlier part of the trace (not depicted). Now $SG4$ is invisible from the viewpoint from $C1$, because the *RReq* message is received at $SG4$ and forwarded (without authentication) to S, and all this is done by IPSec without any interaction with the sectrace daemon running on $SG4$. As a consequence, $SG4$ is not recognized as a possible endpoint for a security association to S, and the next best choice $SG3$ is selected by $C1$.

8.5.2 Type 3 Trace

Some intermediate states of a trace for a Type 3 solution are depicted in Fig. 14. The problem with this solution is that the protocol sets up security associations from $SG4$ to S and from $SG3$ to S, one of them being redundant. Again this is caused by the interference between the two invocations of sectrace by $C1$ and $C2$. The first state in Fig. 14 shows a situation where the security association between $SG3$ and $SG4$ has already been set up by $C1$, which makes the potential endpoint $SG4$ invisible for $C2$. The trace illustrated by the remaining states in Fig. 14 shows that both $C1$ and $C2$ send out *RReq* messages *concurrently*, and then set up the two security associations in question on the basis of the information gathered by *RRep* messages.

8.5.3 Type 4 Trace

Two states of the trace leading to a Type 4 solution are depicted in Fig. 15.

On the left of Fig. 15 we see a situation where $C1$ sends an *SAReq* to $SG4a$, because $C1$ wants to establish a security association from $SG4a$ to Sa:

```
ipsec-send(node("SG3"),
  ip(addr("C1a"), addr("SG4a"),
      sareq(addr("C1a"), addr("Sa"), addr("SG4a"), addr("Sa"))))
```

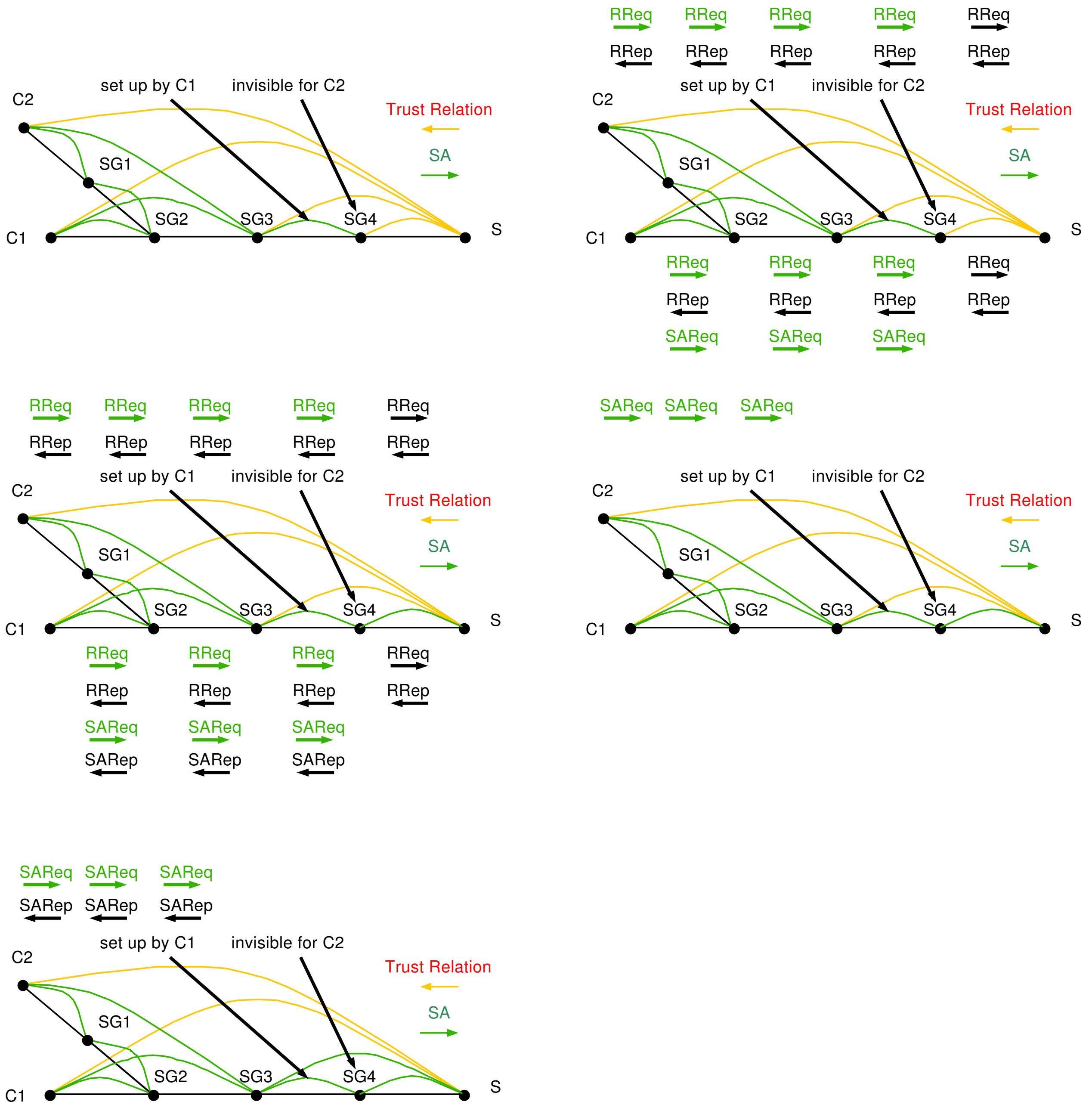

Figure 14: Type 3 Trace

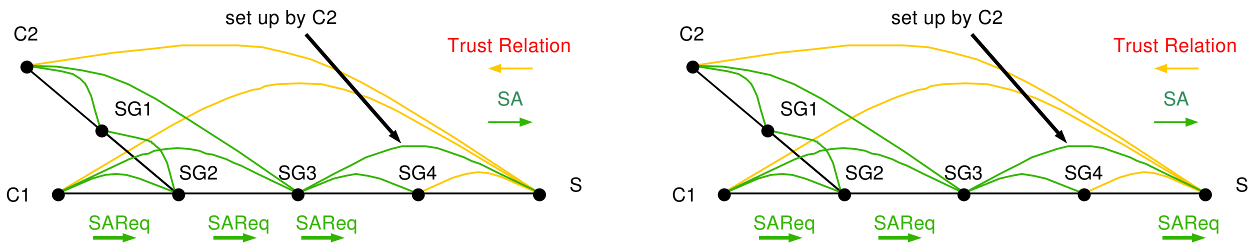

Figure 15: Type 4 Trace

In the final state which is depicted on the right of Fig. 15 we see that instead of $SG4$ the server S has received this $SAReq$, but it cannot be delivered because its destination is $SG4a$:

```
ipsec-received(node("S"), addr("Sa"), eAttrSet,
  sSAList(sa(addr("SG3a"), addr("Sa"), 0)),
  ip(addr("C1a"), addr("SG4a"),
      sareq(addr("C1a"), addr("Sa"), addr("SG4a"), addr("Sa"))))
```

The reason for this confusion is that before the $SAReq$ is received by $SG4$ (left hand side of Fig. 15), the client $C2$ has added a new security association from $SG3$ to S (similar to the Type 2 trace) and has changed the policy at $SG3$ accordingly to direct all traffic towards S into a (nested) tunnel that makes it impossible for $SG4$ to receive the $SAReq$.

9 Discussion

Does the formal model adequately capture the intended model? After several revisions using light-weight methods for validation we are confident that this is the case, with the exception that failures/retries are not formalized to study the protocol in its pure form (without modeling message loss). The experience obtained from the formal specification has led to a revised version of the informal specification, but the degree of detail of the formal specification is still beyond that of the informal one, an example being the precise modeling of IPSec and its security association and policy databases (which involves a notion of security association bundles) and their maintenence by sectrace. Gaps that have been filled by the formal specification include:

1. Choice of security policy patterns and entries. We introduced three patterns: `initiation`, `response`, `towards(server)`.

2. Protocol used to negotiate security associations and to modify the databases. We have currently modeled this in the most abstract way.

3. API between sectrace and IPSec. There are some subtle issues, for instance, sectrace must be aware of unauthenticated $RReq$ packets (which are discarded by IPSec).

Does the formal model have the desired properties? Our analysis shows that the protocol successfully sets up security associations in a various scenarios. Under the sequential execution plan a unique solution is obtained as expected. Under the concurrent execution plan additional solutions arise due to interference between different protocol instantiations. Type 2 and 3 solutions are not globally optimal, but these effects of interference are not surprising. The Type 4 solution was entirely unexpected. The retry mechanism (which is currently not modeled) ensures that the protocol succeeds in this case as well, but relying on timeouts under normal behavior does not seem to be a satisfactory approach.

It is important to point out that even for the case of sequential execution our analysis does not constitute a proof of correctness of the protocol in general, because the analysis is based on inexpensive light-weight formal methods which require a concrete choice of the initial state, that is a particular network with application processes. The light-weight analysis however helped us to discover various forms of unexpected behavior, which can be all traced back to the single problem that certain potential endpoints for security associations are invisible if the setup of security associations and policies is not entirely controlled by a single sectrace client. The following section will discuss this problem and suggests modifications to the protocol design. Since such iterations in the design process are very common, it is good engineering practice to start out with light-weight methods to discover as many bugs in the design as possible and move to more expensive heavy-weight formal methods only if the design has reached a degree of maturity that makes fundamental changes unlikely.

Another independent problem that we address in the next section is the implementability of policies based on the ultimate destination (`towards(server)` in our current specification). This does not seem to be difficult, once a hierarchical structure of subnets protected by security gateways is assumed.

10 Design Alternatives

The problems that have been detected in the previous section can all be reduced to the following simple scenario:

```
  ===
C --- SG1 --- S
```

Assume a security association C === $SG1$ that is already in place with a suitable policy. Now sectrace is started at C, the first *RReq* is sent out, and arrives at $SG1$. Since the sectrace daemon at $SG1$ needs to send out unauthenticated *RReq* packets, there must be at least an outbound policy entry for this case (with an empty security association bundle). Now the IPSec layer has received *RReq*, finds the outbound policy, and *RReq* is forwarded by IPSec as an unencapsulated packet. This forwarding again happens at the IPSec level, so that the sectrace daemon running on $SG1$ cannot notice it. The effect is that the security gateway at the end of the tunnel, in this case $SG1$, is invisible to C, and will not be recognized as a possible endpoint for a security association towards S.

A similar problem can occur with security gateways that are not at the end of a concatenated tunnel, as for instance $SG1$ in the following scenario:

```
  ===     ===
C --- SG1 --- SG2 --- S
```

Assume the security associations C === $SG1$ and $SG1$ === $SG2$ are already there, and a suitable policy is in place. They could have been set up manually, using sectrace, or using some other protocol. Now sectrace is executed at C. The problem here is that during the *RReq*/*RRep* gathering phase $SG1$ is invisible for C, because $SG1$ immediately forwards *RReq* to $SG2$ without sending a *RRep*. Again this happens without any chance for the sectrace demon running at $SG1$ to interact or even know about this. Again, the effect is that $SG1$ will not be recognized as a possible endpoint for a security association towards S.

In both scenarios, if $SG1$ is the only security gateway trusted by S, this would make it impossible to establish a secure communication path. We have considered the following two design alternatives as potential solutions to the above problems. We discuss the first alternative, which leads to several difficulties for which we do not have a good solution at present. We then move on to the second design alternative, which appears to be much simpler and hence in the spirit of the original sectrace ideas.

10.1 First Design Alternative

The fist design alternative introduces two changes to the current sectrace protocol: (1) Taking the source into account in the policy entries, and (2) to replace the selection based on ultimate destination by the immediate destination (assuming a certain hierarchical structure of the network). Item (2) is necessary to make sectrace implementable with standard IPSec implementations, which do not allow policy selection based on ultimate destination.

1. We modified sectrace by using a policy selector which takes into account not only the ultimate destination but also the ultimate source. This works well, because all interference related problems disappear, so that unique and shortest security associations are obtained.
2. In the next step, we replaced the ultimate destination by an immediate address range, represented as a set of addresses. Correspondingly, the selection based on ultimate source is replaced by an address range, to make it implementable.

The use of source information in addition to the destination seems to help but only in the case where the preexisting security associations are governed by a policy with a different ultimate source (e.g. because they have been set up by another client).

Unfortunately, as for the destination selector, IPSec cannot select policy entries based on ultimate source but can just take the immediate source into account. This leads to complications in the case of nested security associations, as illustrated by the subsequent scenario:

```
   ===     ===     ===
C --- SG1 --- SG2 --- SG3 --- S
```

Assume sectrace has already set up three security associations and their policies as above. Hence, the outbound policy database of $SG2$ should contain an entry saying that traffic from immediate source C to immediate destination S should use the SA between $SG2$ and $SG3$.

```
           ===================
   ===     ===     ===
C --- SG1 --- SG2 --- SG3 --- S
```

Now assume that a new security associations is being set up by C from $SG1$ (initiator) to S (responder). Now the immediate source of packets arriving at $SG2$ can only be $SG1$ rather than C, as shown above. In other words, because of nesting $SG2$ cannot “see” C anymore. Therefore the old policy which uses C in the source selector becomes obsolete/unusable.

There seem to be at least three potential solutions to this subproblem:

1. When the security association from $SG1$ to S is added, the policy of $SG2$ is modified. The source selector C is replaced by $SG1$. This solution does not look very elegant, because (1) policies need to be modified, and (2) the modification is required at nodes different from the initiator and the responder.

2. One could make the policy at $SG2$ more general in the first place, i.e. already at the time when the security association from $SG2$ to $SG3$ is set up. For instance, only could include in the selector all trusted nodes between $SG2$ and C (but no other clients which could be connected to $SG1/SG2$ as well). Unfortunately, this path does not usually coincide with an address range, and hence would lead to an explosion of policy entries.

3. One could be even more general, that is to include everything that is protected by $SG2$ in the source selector for entries at $SG2$.

In all three approaches, policies are not client-specific anymore, which was our original reason to introduce policy selection based on source. In summary, the impossibility to access the ultimate source leads again to the inference effects that we discovered in the old version of the protocol, because now packets coming from the same security gateway share the same policy, which we just tried to avoid by taking extra source information into account.

Other solutions are conceivable, e.g. the abuse of port numbers as ultimate source/destination (they behave very much like that), but this seems to be impractical because port numbers serve already a different purpose. There is also an undocumented userid selector [KA98], which might be used for a similar purpose.

10.2 Second Design Alternative

In view of these difficulties, we considered a second alternative design based on the use of the TTL field, which would allow us to bound the number of logical hops, i.e. hops over (top-level) security associations, so that by sending out *RReq* with increasing TTL, C can be sure to get *RRep*-feedback from all security gateways which are not already buried under some (applicable) security association.

The alternative design uses the TTL field without using a source selector. Policy selection is solely based on immediate destination. IPSec maintains the TTL field as one would expect, i.e. it is decreased at each logical hop in case of forwarded IPSec traffic. Informally, the changes to the sectrace protocol are as follows:

Sectrace starts to explore the network and gather information using successive *RReps* with increasing TTL numbers (starting with 1).

- When the client receives an *RRep* with $done = true$ which does not come from the server (this case was not used in the old protocol), this means that the responder has received the packet in authenticated form but did not forward it because of insufficient TTL.

- When the client receives an *RRep* with $done = false$ which does not come from the server, this means that the responder has received the packet in unauthenticated form. This case is treated just as in the old protocol, i.e. a security association has to be set up.

After the security association is set up, the new protocol just forgets *all the information* it accumulated and goes back into the initial state, where it again tries to gather new security gateway information using successive *RReps* with increasing TTL (again starting from 1).

We have developed a formal specification of this revised protocol (using the destination address ranges), and the analysis shows that it generates the unique expected shortest solution in our example scenarios, even under the concurrent execution plan.

It is also interesting to note that the selection algorithm becomes simpler, because after setting up a security association the protocol forgets everything and only (top level) nodes which are potential initiators are "visible" in the *RReq*/*RRep* gathering phase. So the selection algorithm boils down to choosing the rightmost trusted initiator from the *RRepLst*.

One drawback of this variant of sectrace is that more messages are exchanged in the gathering phase. One might consider optimizations by keeping more info about previous *RRep* messages, but then the simplicity and elegance might suffer.

11 Related Work

The IETF working group IPSP has a charter that describes the need for protocols to discover and configure authentication and authorization through security gateways. Opportunistic encryption has a number of points of commonality with sectrace. The IETF working group midbox is concerned with configuration of network elements that include NAT boxes as well as security gateways like firewalls and monitoring systems like Intrusion Detection Systems (IDSs).

12 Conclusions

Policy at SGs is based on a packet's destination address. In practice, this is the address of a server. Thus the security policy database needs to find a policy for an arriving packet based on its ultimate destination. If a packet has a tunnel header for an SG, this entails searching into the packet to find the ultimate destination. Since this is not supported by IPSec we have in the formal specifications of the design alternatives (Section 10) assumed a hierarchical structure based on address ranges protected by security gateways.

It may be desirable for *RRep* messages to be signed by security gateways. This does not introduce any significant burden on the security gateway, since the signature only needs to be created when roots change.

The current protocol is based on trusted roots, but a more likely foundation for the trust relation is delegated authorization. That is, $X \ll Y$ means that Y is willing to delegate traversal authorization to X. If the trust relation $\ll$ holds between every pair of nodes, then there is no need for nested tunnels.

The current protocol assumes that all nodes will use the same rules for establishing security association routes, namely shortest security associations. Another extreme is to let each node configure longest security associations in which each client authenticates to each security gateway and the server. Two points are unclear: (1) whether different clients can use different security association selection policies in a practical way and (2) whether it is possible to enforce the use of the same configuration policy on all nodes. One idea is to allow clients to configure policies but prove that this does not cause problems with shared security associations. Another alternative is to cause security gateways to set up configurations on the grounds that they are more trusted than client endpoints. There are several ways to do this. For instance, the first security gateway could do the entire path setup, or each security gateway could take responsibility for setting up the next security gateway. The latter approach has the disadvantage of moving lists of roots from one security gateway to the next, but this is not likely to be a major burden. By contrast, it has the advantage of moving policy configuration of security gateways that are close to the server to those security gateways themselves; this may reduce the risk of conflicting policy approaches.

The second design alternative (Section 10.2) seems to be a reasonably elegant solutions and our formal modeling and analysis did not detect any problems with sequential or concurrent protocol executions. Moreover, in view of the use of TTL fields, this design alternative is even closer to the way traceroute works than the original sectrace proposal. A practical issue, however, is the implementability using standard IPSec protocol stacks. In order to set the done flag in *RRep* as explained above, a gateway needs to be aware of and distinguish discarded packets of two kinds: the case where TTL is insufficient to forward the packet, and the case where no inbound policy applies (and the packet is treated as unauthenticated). The implementation of this variant of sectrace would require an IPSec implementation that provides this information.

References

[BJM00] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and proof in membership equational logic. *Theoretical Computer Science*, 236:35–132, 2000.

[CDE+] M. Clavel, F. Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. Maude 2.0 Manual. June 2003, `http://maude.cs.uiuc.edu`.

[CDE+00] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and J. Quesada. A tutorial on Maude. `http://maude.csl.sri.com`, March 2000.

[CDE+02] M. Clavel, F. Durán, S. Eker, P. Lincoln, . Martí-Oliet, J. Meseguer, and J. Quesada. Maude: specification and programming in rewriting logic. *Theoretical Computer Science*, 285:187–243, 2002.

[GGM02] Carl A. Gunter, Alwyn Goodloe, , and Michael McDougall. Secure traceroute (sectrace). Draft 0, December 2002.

[GGM03] Carl A. Gunter, Alwyn Goodloe, , and Michael McDougall. Secure traceroute (sectrace). Draft 1, March 2003.

[KA98] S. Kent and R. Atkinson. Security architecture for the internet protocol. Request for Comments 2401, November 1998.

[Mes92] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96:73–155, 1992.

[Mes93] J. Meseguer. A logical theory of concurrent objects and its realization in the Maude language. In G. Agha, P. Wegner, and A. Yonezawa, editors, *Research Directions in Concurrent Object-Oriented Programming.* MIT Press, 1993.

[MÖST02] J. Meseguer, P. C. Ölveczky, M.-O. Stehr, and C. L. Talcott. Maude as a wide-spectrum framework for formal modeling and analysis of active networks. In *DARPA Active Networks Conference and Exposition (DANCE), San Francisco*, May 2002. `http://schafercorp-ballston.com/dance2002/`.

[ST02] M.-O. Stehr and C. L. Talcott. PLAN in Maude: Specifying an active network programming language. In F. Gadducci and U. Montanari, editors, *The 4th International Workshop on Rewriting Logic and its Applications, Pisa, Italy, September 19–21, 2002,* *Proceedings*, volume 71 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2002. `http://www.elsevier.nl/locate/entcs/volume71.html`.