Formal Prototyping in Early Stages of Protocol Design

Alwyn Goodloe
University of Pennsylvania

Carl A. Gunter
University of Illinois at Urbana-Champaign

Mark-Oliver Stehr
University of Illinois at Urbana-Champaign

ABSTRACT

Network protocol design is usually an informal process where debugging is based on successive iterations of a prototype implementation. The feedback provided by a prototype can be indispensable since the requirements are often incomplete at the start. A drawback of this technique is that errors in protocols can be notoriously difficult to detect by testing alone. Applying formal methods such as theorem proving can greatly increase one's confidence that the protocol is correct. However, formal methods can be tedious to use, rarely support successive design iterations and prototyping, are difficult to scale to entire designs, and typically require a clear understanding of requirements in advance. We investigate how formal simulation based on Maude executable specifications overcomes many of these hurdles. We apply this technique in the early stages of the design of a new security protocol, known as Layer 3 Accounting (L3A), aimed at protecting known vulnerabilities in the wireless accounting infrastructure. The protocol sets up a collection of IPsec security associations that provide the necessary protection. We demonstrate how formal simulation uncovered problems in several successive iterations of the L3A protocol design.

1. INTRODUCTION

Protocols usually begin their life on a blackboard. The requirements are rarely complete until late in the design process. In practice, the design and requirements often evolve together. Prototypes written in C or Java are frequently employed to provide feedback as the system evolves. Prototypes usually undergo many revisions before the requirements and design become stable. Because protocols must be highly reliable, it has become increasingly popular to subject them to formal analysis. Due to the complexity of even simple protocols, tool support in the form of an automated theorem prover or model checker [12] is usually required [6, 7]. Discrete event simulation has also been used to support the formal validation of protocols [4, 5]. Formal methods have been effective in exposing subtle design flaws, but the requirements usually need to be fairly well understood before they can be applied. Formal techniques that can be employed earlier in the process, before the design and requirements are stable, would clearly be desirable. Executable specification languages based on rewriting logic [27] and its membership equational sublogic [10] such as Maude [15, 14, 13] allow us to do just that. Instead of building our prototypes in general-purpose languages such as C or Java it is possible to construct a formal prototype that reflects both the design and the essential properties of the system. Although far more abstract than a C program, the model is executable and therefore can provide the same useful feedback as a traditional prototype. We often refer to executable specifications of system designs as *formal prototypes*. The Maude system can also perform an exhaustive search of the state space starting at an initial state and returning all possible results. This feature is useful when debugging a formal prototype. Together, we refer to the execution of formal prototypes and exhaustive search as *formal simulation*.

To the best of our knowledge formal simulation has been applied to existing protocols [16] and protocols late in their development [17], but not to the design of a new protocol. The purpose of this paper is to demonstrate the role of formal simulation in the design of a new protocol intended to secure the wireless accounting infrastructure at the network layer. We call our new protocol the Layer 3 Accounting (L3A) Protocol. It is accepted wisdom that removing errors as early as possible reduces both the cost and the effort expended to arrive at a correct solution. Our work on L3A has shown that protocols that set up a collection of security associations are subject to race conditions that result in control traffic getting stuck in partially set up associations. Eliminating these race conditions can require significant changes to the design. Formal simulation turned out to be an effective aid in this process.

The next section provides the reader with the necessary motivation and background material before we proceed with our discussion on the development of the L3A protocol. In Section 3 we present a skeleton of the protocol. This skeleton will be refined in successive sections of the paper into different versions of the protocol. We then describe the architecture of our formal prototype. The following sections of the paper illustrate how the protocol design evolves as formal simulation exposes problems that need to be corrected and new requirements are introduced that result in changes to the design, which in turn introduce new errors that need to be corrected. Finally, we briefly compare Maude with other formalisms and tools that have been used to model security protocols.

2. MOTIVATION AND BACKGROUND

In this section we provide background material needed in the rest of the paper. First, a brief survey of security issues related to wireless accounting motivates the need for a new protocol. We also arrive at a set of requirements that must be satisfied by our new protocol. This is followed by a short introduction to the IPsec security architecture. Finally, we present the X.509 key-exchange protocol and show how it can be modified to satisfy our requirements.

WITS'05, January 10, 2005, Long Beach, CA, USA.

2.1 Accounting

All commercial Internet access vendors charge their customers for service. While most conventional wired Internet service providers charge customers a flat monthly charge, the wireless links are sometimes deemed too valuable for such a flat service fee. Vendors typically prefer having the option of charging customers a flat fee or to bill based on the services actually used. Accounting devices are often embedded in the network infrastructure to enable wireless providers to track how much service a user consumes. Emerging protocols for wireless Internet access such as CDMA2000 [18] have accounting components. RADIUS servers [30] are designed to provide accounting services for protocols such as GPRS. Accounting information is reported to the billing system for computation of the user's charge [22]. It is important that the accounting infrastructure not be compromised otherwise a vendor may not be able to defend itself against a customer challenging his bill. This is a case where profits depend upon having sufficient security in place. The dependence on the integrity of the accounting infrastructure for billing makes it an inviting target for hackers.

Consider the 802.11 wireless network given in Figure 1. In this figure a client is attempting to securely communicate with a server via a wireless network. The network infrastructure, including some accounting device, is represented by an oval, and access to the Internet is provided by a Network Access Server (NAS). Note that we refer to traffic exiting the wireless network as *egress* traffic, and traffic entering the wireless network as *ingress* traffic. Egress traffic should be authenticated in order to prevent an attacker from spoofing the client and stealing service. In an 802.11 network, the link layer Wireless Encryption Protocol (WEP) is a popular mechanism for providing such protection. End-to-end security in such situations is often provided by SSL between the client and server. Security based on this combination of protocols is common today.

This combination of protocols leaves a vunerability in the authentication of ingress traffic, which is typically viewed as response traffic to communications established through the NAS to the Internet. However, this traffic is not explicitly authenticated, so, in certain circumstances, an adversary could supply false response packets. Since these are not detected as they traverse the NAS, they will be added to the accounting and charged to the wireless network user. There are essentially three kinds of attackers:

1. Attackers that are not on the communication path to the server.
2. Attackers other than the server that are on the communication path to the server.
3. A compromised or malicious server.

It is possible to address the first case in practical terms by establishing enough state on the NAS to make it difficult to guess the connection parameters well enough to spoof a response that will traverse the NAS. For instance, if there is a circuit firewall there, it may be necessary for the attacker to guess TCP sequence numbers; this may require substantial resources from the attacker if large numbers of incorrect guesses can be discarded quickly. The third case can be addressed only by preventing compromises in the first palce, avoiding or detecting malicious servers, or otherwise mitigating their effectiveness. The second case is the most interesting and is the focus of our attention in this paper. To address attackers on the communication path we propose establishing a tunnel between the NAS and the server to secure the ingress traffic as part of a protocol that coordinates the accounting with the end-to-end guarantees of confidentiality and integrity.

Examining the the vulnerabilities highlighted in the above discussion we derive four requirements that must be met in order to provide sufficient protection for the wireless accounting infrastructure. These requirements are as follows:

- The traffic traveling end-to-end should be encrypted and authenticated.
- All of the user's traffic should travel in DoS resistant security associations.
- Security associations should be set up using DoS resistant protocols.
- Both ingress and egress traffic should be authenticated.

We do not claim that attacks against the wireless accounting infrastructure are common at this time nor can we predict whether they will become a problem in the future, but it is a security vulnerability never the less. We propose a solution in the form of a new protocol that establishes a collection of security associations that provide sufficient protection to meet our requirements.

2.2 IPsec

Security based on composing link layer and application layer security protocols has vulnerabilities. Most of the existing link layer protocols have security flaws. That SSL is vulnerable to DoS attacks is well known. It seems that there should be a more cohesive approach to meeting our requirements than cobbling together a solution from different protocols. In fact, IPsec could be used to protect both the connection to the wireless access point as well as the end-to-end connection. Before we can present the details of the protocol some necessary background material on IPsec must be introduced.

lPsec is a security architecture for the Internet specified in RFC 2401 [21] 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 maintenance of virtual secure channels and policies governing the use of these channels. *Security associations* (SA) define a collection of cryptographic transforms that are applied to each IP packet belonging to that association and create a virtual secure channel. *Security policies* direct traffic into security associations based on selection criteria such as source and destination. Figure 2 illustrates a security association between a client and a server. The security association provides a secure communication path through the untrusted subnetwork. Both the server and the client must have security policies directing the flow of traffic between these two nodes into the proper association. IPsec does not provide a mechanism to set up associations and policies, but instead relies on external key-exchange protocols like IKE [20]. In this paper we will design a high-level protocol, L3A, that will use key-exchange protocols to set up a collection of IPsec security associations.

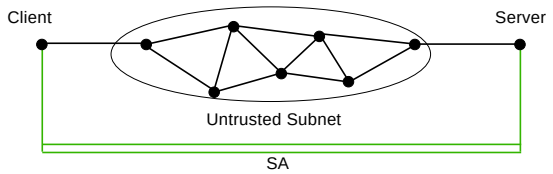

Figure 2: Security Associations

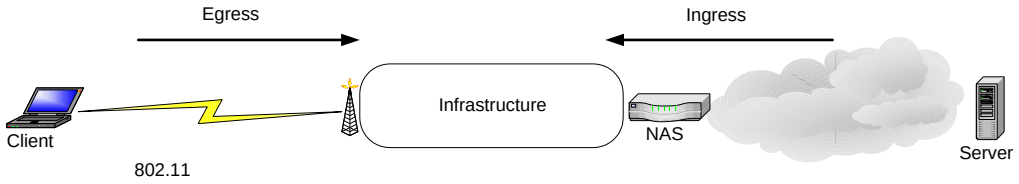

Figure 1: Example Network

An IP packet can be written as $ip(src, dest, data)$ where the header contains source (src) and destination ($dest$). IPsec enhances IP by adding authentication and encryption at the network layer. To this end, IPsec augments the IP header with an AH or an ESP 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. IPsec headers also contain a security policy index (SPI), which, together with the destination $dest$, uniquely identifies a security association. An IPsec packet is 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 $ip(src, dest, \ldots)$. A security association defines shared parameters between nodes (cryptoprotocol parameters and secret key). Each node maintains this information in its *security association database* (SADB). The decision about which security association to use for communication is determined by security policies. These policies are stored in a *security policy database* (SPDB) that is maintained at each node. To understand how these databases work, consider an outgoing packet undergoing IPsec processing. The policy database is checked to see if there is any policy governing the packet. If so, the security association database is consulted to obtain the proper SPI and cryptographic transforms to apply. The appropriate transforms are applied and the IPsec header is added. At the destination, the packet's SPI is used to look up in the SADB the correct key and transformations to apply to the packet in order to obtain the original message. The decision whether the packet, possibly secured by an entire bundle of security associations, can be accepted is made again on the basis of the SPDB. The correct and consistent maintenance of these databases is a nontrivial task for which the L3A protocol has been designed.

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 L3A uses UDP, we identify it with IP for the purpose of this paper, because port numbers and fragmentation are irrelevant for our analysis. Furthermore, L3A uses both *tunnel mode* and *transport mode*. Although both modes are used in the implementation, we uniformly use tunnel mode for simplicity sake. In tunnel mode IPsec encapsulates the original IP packet with IPsec headers and adds an outer IP header. Tunnel mode IPsec packets are of the form:

$$ip(src, dest, sec(spi, ip(src', dest', data))).$$

A typical use of tunnel mode is when security is provided by a node that did not initiate the packet as illustrated in Figure 3. In this case, the IPsec header $ip(src, dest, sec(spi, \ldots)))$ is added and removed by the two security gateways.

IPsec security associations may be composed to yield different patterns of protection than those provided by a single association.

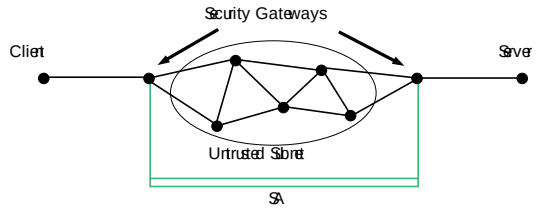

Figure 3: Tunneling

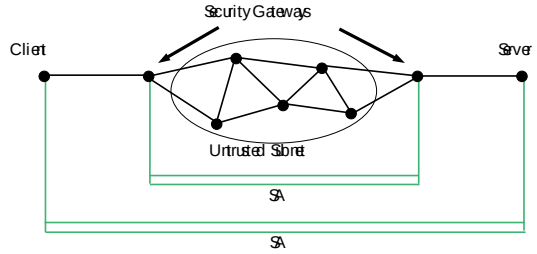

Figure 4: Nesting

The first form of association composition is *concatenation*. The concatenation of associations is an appropriate configuration when a user needs to securely send data to a particular destination, but several organizations on the path have policies requiring that they inspect all traffic entering or exiting their domain. The second form of composing associations is *nesting*. This is illustrated in Figure 4 where a client has an association with a server and the security gateways of each organization also have an association between them. The association between the gateways is nested inside the end-to-end association. 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 $ip(src', dest', sec(spi', \ldots)))$. L3A will make use of both forms of security association composition.

IPsec security associations can be set up by negotiating the security parameters (secret key, etc.) on the basis of a *Public Key Infrastructure* (PKI). Hence, a security association 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. L3A will use the trust relation induced by the PKI.

2.3 Key Exchange

To avoid manual key configuration, the symmetric key used by IPsec is usually established by a key-exchange protocol. The IETF designed the Internet Key Exchange (IKE) [20] protocol for just this purpose. The IKE protocol is rather complex in that there are several rounds of negotiation to support the most general use. Such a complexity is not needed for our purposes. A logical alternative

would be JFK [3], which aimed at simplifying the establishment of IPsec associations. Yet JFK has its own complexities such a subtle model of perfect forward secrecy. Moreover, we found the need to add additional support for IPsec tunnels into the key exchange. Hence we decided to design our own key-exchange protocol by starting with the X.509 three-pass protocol [26] and adding a few needed features. We called this protocol the Simplified Internet key-exchange (SIKE) protocol.

Before discussing our key-exchange protocol we shall present the version of the L3A three-pass protocol that served as a starting point for our work on a key-exchange protocol. We assume familiarity with digital certificates Γ and hope the notation is sufficiently self-explanatory. We assume that there are public and private keys associated with Γ denoted $pub(\Gamma)$ and $priv(\Gamma)$ respectively. Let S be a public key signature function, P a public key encryption function, and H be a one-way hash. To improve the readability of our presentation we define

$$S^*(k, M) \stackrel{\text{def}}{=} (M, S(k, H(M))),$$

to denote a message together with a public key signature.

We use a common informal notation for presenting protocols. For instance, we write

$$A \rightarrow B : S^*(\text{priv}(\Gamma_A), (ip_B, M))$$

to indicate that A sends to B a message containing the IP address of B, a piece of data M, and a signature. Principal B processes this message by first checking the signature using the public key in Γ_A. The the message is said to be *valid* if this condition is true. Otherwise, the message is discarded.

Protocol: X.509 three-pass

Initiation The protocol has two principals A (initiator) and B (responder); Principal A has the private key for a certifi cate Γ_A that is trusted by the other principals; Principal B has the private key for a certifi cate Γ_B. Principal A generates a nonce r_A and sends a message of the form

Msg 1 $\mathsf{A} \rightarrow \mathsf{B} : S^*(priv(\Gamma_\mathsf{A}), (\Gamma_\mathsf{A}, ip_\mathsf{B}, r_\mathsf{A}))$

If B gets a message of this form, it checks the signature using $\text{pub}(\Gamma_\mathsf{A})$. If the signature is valid, B extracts the nonce r_A. B then generates the key K that it will share with A and encrypts it using A's public key. Principal B then sends the message

Msg 2 $\mathsf{B} \rightarrow \mathsf{A} : S^*(\text{priv}(\Gamma_\mathsf{B}), (\Gamma_\mathsf{B}, r_\mathsf{B}, ip_\mathsf{A}, r_\mathsf{A}, P(\text{pub}(\Gamma_\mathsf{A}), K)))$

If A gets a message of this form, it checks the signature using $\text{pub}(\Gamma_\mathsf{B})$. If the signature is valid, the plain text identifi er ip_A is correct, and the nonce r_A matches the value sent in Msg 1, then A extracts the nonce r_B and sends the message:

Msg 3 $\mathsf{A} \rightarrow \mathsf{B} : S^*(\text{priv}(\Gamma_\mathsf{A}), (r_\mathsf{B}, ip_\mathsf{B})))$

If B gets a message of this form, the signature is checked using $\text{pub}(\Gamma_\mathsf{A})$. If signature is valid, the plain text identifi er ip_B is correct, and the nonce r_B matches the value sent in Msg 2, then message is verifi ed.

Due to encryption the key is secure when it arrives at principal A. The nonces protect the exchange from replay attacks. Unfortunately, the responder remains vulnerable to DoS attacks. In particular, attacks that exhaust state and CPU of the responder by flooding it with session initiation requests from forged IP addresses. It is possible to decrease the effectiveness of this attack by having the responder use minimal CPU resources and commit no state until it knows that the initiator can receive packets at the address from which it claims to be sending them. This is the fundamental idea behind a cookie. In order to thwart such attacks we extend the protocol given above by adding cookies.

The three-pass X.509 protocol given above terminates with a symmetric key shared between two parties, but this is not sufficient information to establish an association. In our discussion of IPsec, we described how each association must have a unique identifier called the SPI that allows the party receiving an IPsec packet to determine to which association the packet belongs. Without a unique identifier, the receiver of a packet would not have any idea what transforms to apply to a packet. The fact that the SPI must be unique to the destination of the association means that the destination must generate this unique identifier. In order to be as general as possible, each node in the exchange generates a SPI value. This accommodates either node being a destination of an association. We incorporate this feature into our key-exchange protocol.

The protocol that results from adding cookies and SPI generation to the X.509 three-pass protocol is given as follows:

Protocol: X.509 three-pass with SPI and cookies

Initiation The protocol has two principals A (initiator) and B (responder). Principal A generates a nonce r_A, a SPI value n for an association having destination A, and sends the message

Msg 1 $\mathsf{A} \rightarrow \mathsf{B} : r_\mathsf{A}, \text{SPI}(n, 0)$

If B gets a message of this form, it generates a nonce r_B, a SPI value m for an association having destination B, and a cookie

$$cookie_\mathsf{A} = version, H(r_\mathsf{A}, r_\mathsf{B}, ip_\mathsf{A}, secret)$$

where $secret$ is a private secret of the responder that is updated periodically; since the secret is periodically updated, each secret has an associated version identifi er $version$ included in the message; the address of A is denoted ip_A, and r_A and r_B are the nonces of A and B, respectively. Principal B sends the message

Msg 2 $\mathsf{B} \rightarrow \mathsf{A} : r_\mathsf{A}, r_\mathsf{B}, \text{SPI}(n, m), \Gamma_\mathsf{B}, cookie_\mathsf{A}$

If A receives a message of this form, it sends the message

Msg 3 $\mathsf{A} \rightarrow \mathsf{B} :$
$S^*(\text{priv}(\Gamma_\mathsf{A}), (\Gamma_\mathsf{A}, cookie_\mathsf{A}, r_\mathsf{A}, r_\mathsf{B}, ip_\mathsf{B}, \text{SPI}(n, m)))$

If B gets a message of this form, it checks the signature using $\text{pub}(\Gamma_\mathsf{A})$. If the signature is valid, the plain text identifi er ip_B is correct, and the nonce r_B matches the value sent in Msg 2, then node B extracts the nonce r_A. B then generates the key K that it will share with A and encrypts it using A's public key. Principal B then sends the message

Msg 4 $\mathsf{B} \rightarrow \mathsf{A} :$
$S^*(\text{priv}(\Gamma_\mathsf{B}), (r_\mathsf{A}, r_\mathsf{B}, ip_\mathsf{A}, \text{SPI}(n, m), P(\text{pub}(\Gamma_\mathsf{A}), K)))$

If A receives a message of this form, it checks the signature using $\text{pub}(\Gamma_\mathsf{B})$. If the signature is valid, the plain text identifi er ip_A is correct, and the nonce r_A matches the value sent in Msg 3, then node A decrypts the shared key.

IPsec associations are unidirectional so we have to ask if a single execution of our key exchange should establish a single unidirectional association that only protects the flow of information in one direction or should it establish two associations that protect the flow of information in both directions. We considered both possibilities in the course of our research. The protocol given above can be modified to accommodate either case. The first model of SIKE that we shall present will set up unidirectional associations while the second model of SIKE will set up bidirectional associations.

Whenever we create an association from principal A to principal C we must update the SADB with an $\mathsf{A} \rightarrow \mathsf{C}$ entry. We assume that an association being created from A to C will have an accompanying policy saying that all traffic flowing from A to C will go into the association $\mathsf{A} \rightarrow \mathsf{C}$. To support nested tunnels we enforce the following rule. Suppose that there is no association $\mathsf{A} \rightarrow \mathsf{C}$, but there exists a policy in the SPDB on node A that directs all traffic flowing from principal A to principal C into the tunnel $\mathsf{A} \rightarrow \mathsf{B}$. When creating an association $\mathsf{A} \rightarrow \mathsf{C}$, the existing policy on node

A is updated so that all traffic flowing from principal A to principal C is first be placed in the newly created A → C association and then in the A → B association. The resulting outgoing packet has the form

$$ip(ip_{\mathsf{A}}, ip_{\mathsf{B}}, sec(spi_{\mathsf{AB}}, ip(ip_{\mathsf{A}}, ip_{\mathsf{C}}, sec(spi_{\mathsf{AC}}, data)))).$$

It is left to the designer of the higher level protocol setting up the associations to ensure that all restrictions regarding the nesting of tunnels are obeyed.

We must decide when the SADB and SPDB are updated. One choice is to return information exchanged in the protocol and let the databases be modified at a higher level. This breaks modularity of the system and reduces the information that can be known by each side upon termination. Suppose both nodes participating in the key exchange simply updated their data bases after they had completed the four messages, then upon termination of the protocol neither party would be sure that an association has been established. Figure 5 shows a second option where the responder writes the information for the A → B association to the databases before sending the fourth message. As a consequence, the initiator can be assured that the A → B association is established when the protocol at the initiator terminates. A third option adds an additional acknowledgment from the initiator to the responder. This message is sent after the initiator has completed all of its database updates. Upon termination of the protocol, the initiator knows that the A → B association is established and the responder knows that both associations are established. (The initiator does not know whether the responder has completed updating its databases for the B → A association.) Adding a second acknowledgment from the responder to the initiator results in both parties knowing all security associations are established upon termination. We selected the *second* option, because it provides some knowledge about the state of of the databases, and at the same time provides the weakest guarantee of the three acceptable choices with a minumum number of messages. A protocol specification that was correct using this choice would remain correct if the weaker key exchange protocol were replaced by one offering stronger guarantees.

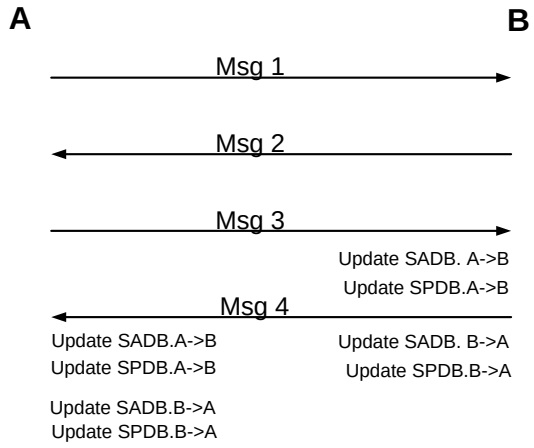

Figure 5: Write after message three.

3. PROTOCOL

Now that the necessary background has been established we turn our attention to the development of the L3A protocol. We start by elaborating what security associations are needed. A skeleton of the protocol is presented that will be refined in later sections. We then explain the architecture of the formal prototype used to model the protocol.

3.1 L3A

The L3A protocol sets up a collection of IPsec associations that are intended to protect the wireless accounting infrastructure as well as the user from a myriad of attacks. Recall our system configuration. A wireless client that needs to securely communicate with a server. The client gains access to the Internet via a Network Access Server (NAS) that belongs to a vendor that charges the client based on the volume of data going over the link. In this section we develop a general skeleton of the L3A protocol that will later be refined as we build successive prototypes of the model.

The first question to be answered in designing our protocol is what security associations are needed. The requirements tell us that there should be an association between the client and the server to protect traffic going from end to end. In practice, this would be an IPsec transport mode association providing both encryption and authentication. To simplify the formal model we will uniformly use tunnel mode associations. Another of our requirements states that all egress and ingress traffic must be authenticated by the NAS. This dictates that there be an IPsec tunnel mode association from the client to the NAS performing authentication and a similar tunnel from the server to the NAS. The resulting situation is one where the end-to-end association must tunnel through the two authentication tunnels. This configuration of associations can be seen in Figure 6.

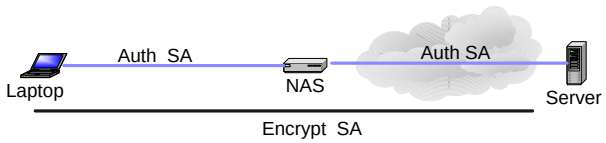

Figure 6: Base SA configuration

The client has a relationship with both the server and the NAS and must authenticate itself with each of these nodes during different stages of the protocol. We have already established that there will be an authentication association between the client and the NAS, an authentication association between the NAS and the server, and an association providing both authentication and encryption between the client and the server. It is clear that the client will authenticate itself to the NAS during the key exchange. Similarly, the client will authenticate to the server as part of a key exchange. It is less clear how the NAS will authenticate itself to the server, since it needs to communicate to the server that it is establishing an association on behalf of the client. We resolve this problem by having the client send the NAS a credential that the NAS will present to the server on behalf of the client. If the server verifies that the credential is from a valid user, then it allows the association between the NAS and server to be established.

We can now give an outline of the L3A protocol.

Skeleton of L3A Protocol

Client initiates protocol The client identifi es the server that it desires to communicate with and the NAS that will deliver access to the Internet. The client then invokes SIKE to establish an association between the client and the NAS. The client must pass to the NAS the credential that the NAS will present to the server as well as the address of the server.

Establish NAS-server SA Upon notifi cation that the client-NAS SIKE exchange is complete, the NAS gets the address of the server and the credential. The NAS then invokes SIKE to establish an association between the NAS and the server. The NAS presents the credential to

the server. If the credential is not valid, then the protocol is terminated.

Establish server-client SA Upon notification that the NAS-server SIKE exchange is complete, the server invokes a key exchange with the client.

3.2 Architecture of the Formal Model

We constructed formal models of the L3A and SIKE protocols early in the design phase. Maude supports the specification of complete designs in a modular way. We were able to model the IPsec and IP network layers in addition to the components of the L3A protocol. In modeling such a large system, it is important to keep in mind to only model those aspects of the system that are of interest. For example, we only model routing and message delivery in IP. Details such as fragmentation are ignored. Being able to construct such complete models is particularly useful when modeling protocols that rely on lower layer protocols since the interaction between the layers may be a source of errors. Modeling IPsec was necessary since the L3A and SIKE protocols can only be judged correct if they correctly set up the desired IPsec security associations and policies. In order to model IPsec, we needed to model IP to a certain degree. We now turn our attention towards the architecture of the formal model.

In keeping with good software engineering practices, the design of our formal model is modular and constructed as a hierarchy of abstractions reflecting the structure of the system being modeled. Figure 7 shows the components of our model and how they are

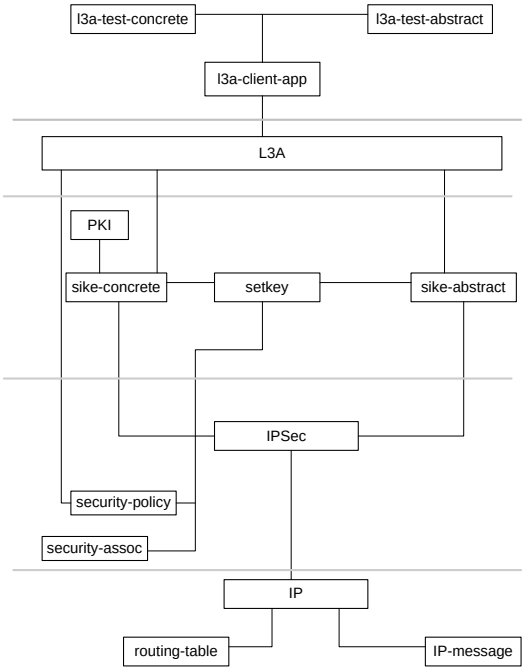

Figure 7: Architecture of the model

related. Note that the lightly shaded horizontal lines separate the different layers of the model hierarchy. The lowest layer models the sending and receiving of IP messages. Routing is modeled at this layer. At the next layer the `IPsec` module models a significant part of IPsec. It does not model IPsec transport mode since the differences between tunnel mode and transport mode are not significant enough to affect our analysis. We do not attempt to provide a concrete model of encryption since we are only concerned about ensuring that the proper headers are applied and not with the concrete cryptographic transformations that get applied to the packets once they arrive at their destination. On the other hand, we must model the IPsec databases, because their state defines the security associations and policies. At the level of the key exchange we include a module `setkey` that provides the SIKE modules with an interface to the IPsec databases. This module encapsulates the updating of both the SADB and SPDB databases. The `PKI` module provides a limited public key infrastructure. Two of the authors worked on building this model simultaneously during the early phases of this project. One author worked on modeling L3A while the other worked on SIKE. We constructed a stub key exchange module `SIKE-abstract` that allowed L3A development to progress in parallel with the development of SIKE. The full SIKE protocol is modeled by the module `SIKE-concrete`, which is a refinement of `SIKE-abstract`. The L3A protocol is formally modeled by the module `L3A`. Observing the diagram we see lines from both `L3A` and `setkey` to the SPDB, but only `setkey` has a connection to the SADB. This is because policies are created and modified at both the level of the key exchange and L3A, while the security associations are only created by SIKE. The `l3a-test` modules import all the other modules and define the system configuration. The abstract and concrete versions of `l3a-test` differ in that the abstract version imports the abstract version of SIKE, while the concrete version imports the actual SIKE protocol module. Constructing different test cases is facilitated by the module `l3a-client-app`. A bonus of the modular nature of our design is that we can reuse portions of the model when designing other protocols.

The SIKE protocol module is composed of two processes. The SIKE component handles all the processing on the initiator side and is invoked with parameters such as the address of the initiator and the address of the responder. The processing at the responder is performed by a daemon SIKED. Each node in the system must start its SIKED daemon before the protocol may be executed. The L3A protocol given above is decomposed into three processes—one process at each of the nodes (client, server, NAS). The protocol is started by some higher level (`l3a-client-app`) module at the client. The L3A processing at the NAS and server is performed by daemons. These daemons wait for notification from the SIKED daemon that a key exchange has completed. After receiving such a notification, the process executes its portion of the L3A protocol.

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 given in Figure 8. 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.

Since Maude specifications are executable, we were able to constantly test and refine the design in much the same way that we would debug a program. This immediate feedback allowed for quick turnaround as the design evolved. The design process went as follows. First, we coded our design in Maude; secondly, we would execute our model to check if there existed a correct solution (*i.e.* an expected execution of the protocol). After several iterations, we would arrive at a design that we deemed correct. We would then utilize Maude's search capabilities by executing `search initial =>! state : State`. The system performs an exhaustive search of the state space returning all possible solutions

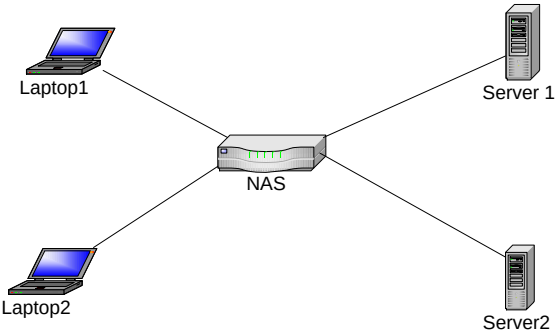

Figure 8: Model network topology.

(*i.e.* executions leading to a state in which no further protocol rules can be applied). Our protocols are deterministic so there should only be one solution when we run an exhaustive search on our test cases. In practice, it took several iterations to eliminate all the incorrect solutions. L3A is a distributed protocol running on three nodes and like most distributed systems the errors arose form unforeseen concurrent interleaving of statements. Even extensive testing often fails to replicate the errors that can be found through exhaustive search.

Representative samples of the Maude model can be found in the Appendix. For the full formal specification of each model we refer the reader to `http://formal.cs.uiuc.edu/stehr/l3a/l3a.html`.

4. L3A PROTOTYPES AND FORMAL SIMULATION

The previous sections have provided background material on IPsec and on the methodology employed during the development of our protocol. We have also presented the basic structure of both SIKE and L3A. In this section we refine the skeletal solutions given above into the first prototype of the protocol. We then demonstrate how formal simulation was used to debug the prototype. We then show how this process is repeated several more times before arriving at a stable design.

4.1 First Prototype

We now precede with the design of L3A and SIKE by refining the protocol skeleton given above. The first issue to be addressed is what traffic should the protected by the associations. This will determine if SIKE should establish a single unidirectional key exchange or establish a pair of associations protecting traffic flowing in both directions. We will also need to determine what if any additional data needs to be sent as part of the key exchange and address the possibility that additional messages need to be sent in L3A.

To ensure privacy between the client and the server we must have associations performing encryption and authentication going in both directions. It is not so clear what should be done in the case of the tunnels that ensure all traffic entering the NAS is authenticated. It is clear that there must be security associations flowing from the client to the NAS and flowing from the server to the NAS. Otherwise, the traffic is authenticated by the end-to-end association so there does not appear to be a need to have associations going the opposite direction. Since we do not always need security associations going in both directions, the key-exchange protocol should only set up a single association.

We have already established that the client must pass a credential as well as the address of the server to the NAS and the NAS must then pass this credential to the server. To accommodate this data we add a payload field to the third message of the SIKE protocol. Since the SIKE exchange will create a single association and this association need not flow from initiator to responder, the SIKE exchange needs to include a field indicting whether the security association flows from the initiator to responder or visa-versa. Another consequence of establishing only a single association is that each node only needs to update its SADB and SPDB once.

The first version of the SIKE protocol is given as follows:

SIKE V 1.0

Initiation The protocol has two principals A (initiator) and B (responder). Principal A generates a nonce r_{A}, a SPI value n for an association having destination A, and sends the message

Msg 1 $\mathsf{A} \rightarrow \mathsf{B} : r_{\mathsf{A}}, \mathrm{SPI}(n, 0)$

If B gets a message of this form, it generates a nonce r_{B}, a SPI value m for an association having destination B, and a cookie

$$cookie_{\mathsf{A}} = version, H(r_{\mathsf{A}}, r_{\mathsf{B}}, ip_{\mathsf{A}}, secret)$$

where $secret$ is a private secret of the responder that is updated periodically; since the secret is periodically updated, each secret has an associated version identifier $version$ included in the message; the address of A is denoted ip_{A}. The nonces of A and B are denoted r_{A} and r_{B} respectively. Principal B sends the message

Msg 2 $\mathsf{B} \rightarrow \mathsf{A} : r_{\mathsf{A}}, r_{\mathsf{B}}, \mathrm{SPI}(n, m), \Gamma_{\mathsf{B}}, cookie_{\mathsf{A}}$

If A receives a message of this form, it forms a pair (s, d) indicating the source s and destination d of the SA. A then sends the message

Msg 3 $\mathsf{A} \rightarrow \mathsf{B} : S^*(\mathrm{priv}(\Gamma_{\mathsf{A}}),$
$(\Gamma_{\mathsf{A}}, cookie_{\mathsf{A}}, r_{\mathsf{A}}, r_{\mathsf{B}}, ip_{\mathsf{B}}, \mathrm{SPI}(n, m), \mathrm{dir}(s, d), payload))$

If B gets a message of this form, it checks the signature using $\mathrm{pub}(\Gamma_{\mathsf{A}})$. If the signature is valid, the plain text identifier ip_{B} is correct, and the nonce r_{B} matches the value sent in Msg 2, then node B extracts the nonce r_{A} and the payload. If B is the destination of the SA, then the SADB and SPDB are updated. B then generates the key K that it will share with A and encrypts it using A's public key. Principal B then sends the message

Msg 4 $\mathsf{B} \rightarrow \mathsf{A} :$
$S^*(\mathrm{priv}(\Gamma_{\mathsf{B}}), (r_{\mathsf{A}}, r_{\mathsf{B}}, ip_{\mathsf{A}}, \mathrm{SPI}(n, m), P(\mathrm{pub}(\Gamma_{\mathsf{A}}), K)))$

If B is the source of the SA, then the SADB and SPDB are updated. If A receives a message of this form, it checks the signature using $\mathrm{pub}(\Gamma_{\mathsf{B}})$. If the signature is valid, the plain text identifier ip_{A} is correct, and the nonce r_{A} matches the value sent in Msg 3, then node A decrypts the shared key. The SADB and SPDB is updated according to the direction of the SA.

A message sequence chart representation of the SIKE protocol can be seen in Figure 9.

It is now possible to refine the basic skeleton of the L3A protocol into the first version of the protocol. The L3A protocol will use SIKE to establish the four security associations illustrated in Figure 10. As each node invokes the key-exchange protocol, it will have to pass as parameters the direction of the association as well as any payload. The protocol is always started at a client. The client calls SIKE to establish the $\mathsf{C} \rightarrow \mathsf{NAS}$ tunnel. The NAS then calls SIKE to establish the $\mathsf{S} \rightarrow \mathsf{NAS}$ tunnel. The server and client then create tunnels $\mathsf{S} \rightarrow \mathsf{C}$ and $\mathsf{C} \rightarrow \mathsf{S}$ respectively.

Version one of the L3A protocol follows and is illustrated by Figure 10.

L3A Protocol V 1.0

Client initiates protocol The client C identifies the server S that it desires to communicate with and the NAS that will provide access to the Internet. The client then invokes SIKE to establish an association from the client to the NAS with the parameters: $payload = cred, \mathsf{S}$ and $dir = \mathrm{dir}(\mathsf{C}, \mathsf{NAS})$. When the key exchange at the client has terminated it updates its SPDB with a policy saying that all traffic from the client to the server should flow through the $\mathsf{C} \rightarrow \mathsf{NAS}$ association.

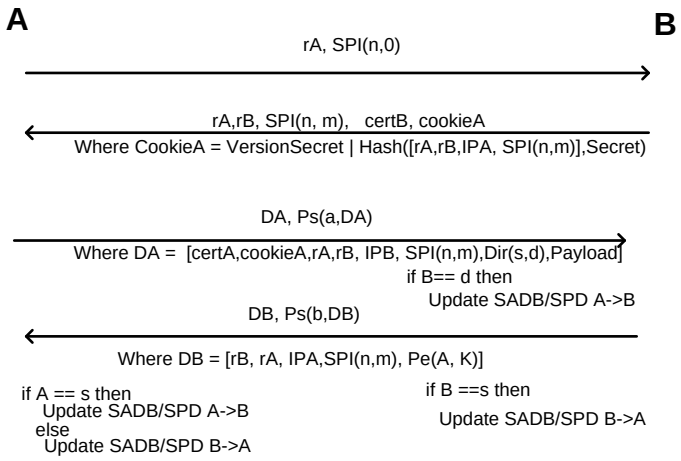

Figure 9: SIKE V 1.0

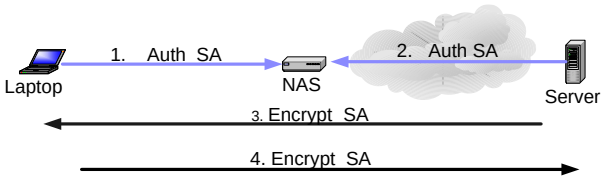

Figure 10: L3A V 1.0

Establish Server-to-NAS SA Upon notification that a key exchange with the client is complete, the NAS extracts the address of the server and the credential. It then it updates its SPDB with a policy saying that all traffic from the client to the server should flow through the $\mathsf{C} \rightarrow \mathsf{NAS}$ SA. The NAS then invokes SIKE to establish an SA from the NAS to the server. The two parameters to SIKE are: $payload = cred$ and $dir = \mathrm{dir}(\mathsf{S}, \mathsf{NAS})$. Upon termination of the key exchange, the NAS updates its policy database to reflect the fact that all traffic flowing from the server to the client should travel in the $\mathsf{S} \rightarrow \mathsf{NAS}$ association.

Establish Server-to-Client SA Upon notification that a key exchange with the NAS has occurred, the server extracts the credential that had been passed by the client. If the credential is valid, the server updates its SPDB with a policy saying that all traffic flowing from the server to the client should travel in the $\mathsf{S} \rightarrow \mathsf{NAS}$ association. The server then invokes SIKE to set up an association from the server to the client. The two parameters to SIKE are: $payload = \mathsf{emty}$ and $dir = \mathrm{dir}(\mathsf{S}, \mathsf{C})$.

Establish Client-to-Server SA Upon notification that the SIKE exchange with the server has occurred, the client invokes SIKE to create an association from the client to the server. The parameters to SIKE are: $payload = \mathsf{empty}$ and $dir = \mathrm{dir}(\mathsf{C}, \mathsf{S})$.

Given the simple topology in Figure 8 we ran a state space exploration that yields three different solutions. We now look at each of the solutions in more detail.

- Solution 1.

  The NAS SADB has no entry for the association $\mathsf{S} \rightarrow \mathsf{NAS}$ while the server SADB does have an association $\mathsf{S} \rightarrow \mathsf{NAS}$ entry. As a consequence the fourth message of the $\mathsf{S} \rightarrow \mathsf{NAS}$ SIKE exchange gets caught in the partially set up tunnel $\mathsf{S} \rightarrow \mathsf{NAS}$. The first message of the $\mathsf{S} \rightarrow \mathsf{C}$ SIKE exchange also gets caught in the partially set up $\mathsf{S} \rightarrow \mathsf{NAS}$ association.

- Solution 2.

  The first message of the $\mathsf{S} \rightarrow \mathsf{C}$ SIKE exchange gets caught in a partially set up $\mathsf{S} \rightarrow \mathsf{NAS}$ association. Both the policy databases and the association databases look correct, but further analysis shows that this was misleading. There is a concurrency problem. The SADB on the NAS was not updated until after the packet arrived.

- Solution 3. At each node the entries in the SADB and SPDB correspond to the associations and polices given in the requirements. For example, at the client there are entries in the SADB for the $\mathsf{C} \rightarrow \mathsf{NAS}$, $\mathsf{C} \rightarrow \mathsf{S}$, and $\mathsf{S} \rightarrow \mathsf{C}$ associations. There are policies that say that all traffic flowing from the client to the NAS travels in the $\mathsf{C} \rightarrow \mathsf{NAS}$ association; all traffic flowing from the client to the server travels in the $\mathsf{C} \rightarrow \mathsf{S}$ association, which is tunneled through the $\mathsf{C} \rightarrow \mathsf{NAS}$ association; all traffic flowing from the server to the client travels in the $\mathsf{S} \rightarrow \mathsf{C}$ association. The server and the NAS have similar entries reflecting their policies and the security polices active at each node. Since the SADB and SPDB at each node have the correct entries as dictated by the requirements, we consider the solution to be correct.

In the next section we shall revise the protocol in an attempt to correct the problems uncovered by the exhaustive search.

4.2 Revising the Prototype

The first version of our protocol exhibited several errors. The errors uncovered in formal analysis seemed to (at least partially) stem from the fact that the server has updated its SADB/SPDB before the NAS. The first solution produced by exhaustive search seemed to us rather vexing. The fourth SIKE message gets caught in a partially set up tunnel.

Analysis revealed that the problem was not caused by the protocol itself, but by the way we modeled communication. It turned out that our model of IP and IPsec semantics was too weak. The actual semantics for sending a message is that the send function does not return until the message is completely processed by IPsec. We checked the code! Our model allowed the sender of an IPsec message to continue processing as soon as the send call was made while the message was processed concurrently by IPsec. Consequently, the additional concurrency in our model resulted in cases

where the association is set up on the responder side before message 4 is actually out of the door. As a result of this behavior, the association is applied to the outgoing message, but there is no entry in the association initiator's SADB. We corrected our models of IP and IPsec to match the actual semantics. Given that the previous version of the protocol had been tested with an unrealistic model for IP and IPsec, it seems reasonable to limit our corrections to this problem to see if fixing the model solved all of our problems.

Performing an exhaustive search of the state space yields four solutions. It may seem like things are now worse since there are more solutions than the first version of the protocol. The first three solutions are all incorrect because the first message of the $S \rightarrow C$ SIKE exchange gets caught in a partially set up $S \rightarrow NAS$ association. The problem is due to the fact that the $S \rightarrow NAS$ is set up on the server side and the server goes ahead with the $S \rightarrow C$ SIKE exchange before the NAS has set up the association on its side. The fourth solution is correct.

4.3 Second Revision

The problem with the previous revision to the protocol seems due to the fact that the server can finish processing setting up the $S \rightarrow NAS$ and begin the $S \rightarrow C$ SIKE exchange before the NAS has set up the $S \rightarrow NAS$ association on its side. As a consequence, the first message of the $S \rightarrow C$ key exchange gets caught in a partially set up tunnel. The obvious solution to this problem is to force the server to wait until the NAS has completed its processing. We add a single message that the NAS sends to the server when it has completed its half of the SIKE protocol. The server waits to receive this message before it continues with any processing. The revised protocol is illustrated in Figure 11. Performing a search of the state space now yields only the correct solution.

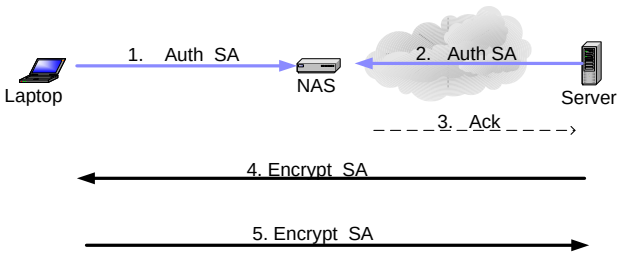

Figure 11: L3A V 2.0

5. FURTHER EVOLUTION

Formal simulation continued to be beneficial as the protocol underwent revision. The acknowledgment sent from the NAS to the server eliminated undesirable concurrent interleavings of the system, but the message does not travel in an authenticated DoS resistant security association. This violates one of the primary requirements of the protocol. Furthermore, it was also pointed out to us that the NAS would need to send maintenance traffic to the client and server. This traffic would need to travel in authenticated security associations as well. As a result of these requirements we revised our design. It now seemed obvious that the key exchange should establish a pair of security associations with one going in each direction. This would bring our protocol in line with IKE, which also sets up a pair of security associations.

As we modified SIKE and L3A we continued running logical simulations on a variety of scenarios. As we ran different test cases during development we found that the client could finish setting up the $C \leftrightarrow S$ association before the server. One possibility was to ignore this fact and assume that an L3A user would just back off and retry later if they discovered that the protocol was not yet set up. Given our previous problems with interleaving concurrency and the fact that the protocol was still evolving we decided to add a Fin message that would let the client know when the protocol had terminated. For consistency and uniformity we added an acknowledgment from the client to the NAS after the client had finished the $C \leftrightarrow NAS$ key exchange. This allowed us to remove the payload field from SIKE and include that information as part of the acknowledgments. The direction field is no longer needed since each SIKE invocation establishes a pair of associations. The new version of SIKE is illustrated in Figure 12.

In the updated L3A design, the client begins by performing a key exchange with the NAS. After the SIKE exchange between the client and the NAS has terminated, the client sends a Req message to the NAS containing the credential and server address. The NAS does not start the key exchange with the server until after receiving this message. Once the key exchange between the NAS and the server is complete, the NAS sends the server an Ack message with the credential. If the credential is valid, the server initiates a key exchange with the client to establish the two associations ($S \rightarrow C$ and $C \rightarrow S$). Upon termination of this key exchange, the server sends a Fin message to the client indicating that the protocol has terminated. The modified protocol is illustrated in Figure 13.

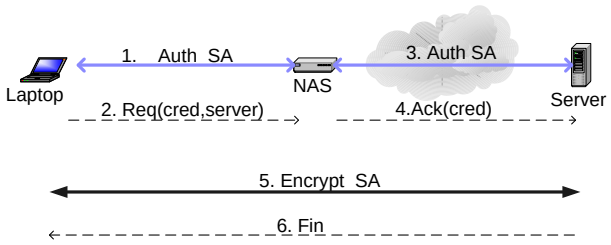

Figure 13: L3A V 3.0

The protocol underwent further modifications to accommodate reuse of security associations. For example, suppose several clients gain access to the network via the same NAS and all go to the same server. The $NAS \leftrightarrow S$ associations can be shared among the clients. Similarly, the $C \leftrightarrow NAS$ can be reused when a client connects to multiple servers. Getting the prototype right took a lot of debugging of tricky details that are typically ignored in mathematical models. At each node, processing for establishing security associations is now structured as a set of conditional blocks with each block handling a different case. For example, the NAS has to consider cases where its first action of the protocol is processing a Req message, because the client is reusing an association. Alternatively, the NAS first action could be the notification that a key exchange with the client has occurred. Sample Maude code for the client and NAS is included in the Appendix. This type of processing is not particularly difficult, but it is easy to miss a case, get the boolean conditions wrong, or to introduce new race conditions. Had the model not been executable we may have postponed the verification of such details to the implementation.

6. RELATED WORK

In this section we review several alternative formalisms and tools that we could have chosen instead of Maude. We briefly compare their suitability for use as a platform for prototyping the L3A protocol to Maude.

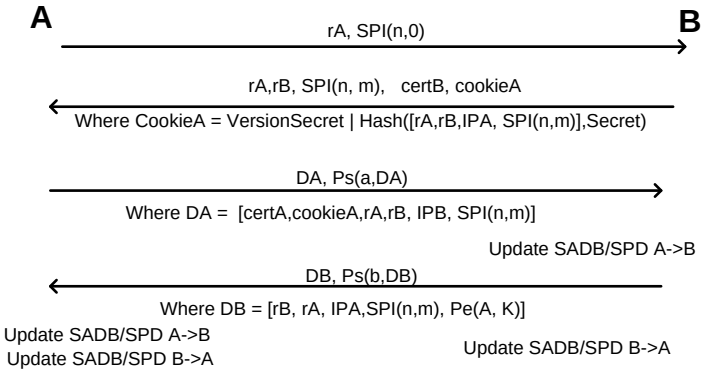

Figure 12: SIKE V 2.0

The methodology employed in this paper (executable specification and logical simulation) is not necessarily tied to Maude. Popular model checking tools such as SPIN [19] and SMV [24] could be tailored to support the same process. So one may wonder what advantage we felt was gained from using Maude. One advantage is that Maude's language of multiset rewriting with equational logic is somewhat more abstract than the languages usually employed in model checking tools. In order to sufficiently test our protocol, we had to model many of the underlying layers of the protocol stack that our system was dependent upon. In particular, we had to model the key exchange protocol, IPsec, and IP routing. A downside of such detail is the danger of state explosion. We made extensive use of Maude's equational logic in modeling these portions of the system. Maude's equational theories are assumed to be confluent and terminating. This helped keeping the size of the model in check. In contrast to purely operational approaches to protocol specification, *e.g.* based on variations of coventional programming languages, the logical nature of Maude not only provides a higher level of abstraction, but at the same time it reduces the gap between the executable model and its mathematical treatment, as witnessed by it's simple algebraic model-theoretic semantics.

Process algebras have a long history of being applied to the analysis of protocols and in the last decade they have become a popular vehicle for expressing and analyzing cryptographic protocols. Hoare's CSP [11, 31] and the associated FDR model checker were first used to discover an error in the Needham-Schroeder protocol a decade ago [23] and have continued to enjoy success in this application domain. More recently, several versions of the π-calculus have been developed to reason about cryptographic protocols [2, 8]. The ProVerif tool [9] provides sophisticated automated assistance to reason about secrecy properties of a protocol written in the π-calculus. This tool has been applied to the analysis of the JFK [1] key exchange protocol. Unfortunately, ProVerif lacks the kind of simulation capability that we found so useful for prototyping protocols. Given the success of applying process algebras to security protocols they may seem like a preferred alternative to rewriting logic. Yet L3A is a little different from traditional security protocols. The state of the databases at each node is a critical element of the protocol and must be part of any formal model. Although it would be possible to encode the databases within a process algebra, it would be quite a cumbersome exercise. We do not view this as a deficiency of process algebras, but as an indication of the need to match the formalism with the problem.

Logic-based tools such as automated theorem proving provide another alternative to Maude's term rewriting logic. Paulson has applied inductive techniques to a number of cryptographic protocols [28, 29]. The NRL protocol analyzer (NPA) [25] is a logic-based tool customized for analyzing security protocols. One verifies a protocols by feeding into NPA a state machine representation of the protocol, attack scenarios, and the machine starts at the attack state and executes the protocol backwards to see if an initial state can be reached. If so, the attack is successful. Logic based tools are an excellent choice when a protocol reaches a degree of stability to commit resources to apply heavy-weight formal methods. Formal simulation tools such as Maude allow the design to be debugged in a process that is similar to program debugging, but at a higher layer of abstraction. Now that we feel that we have relatively stable protocol, applying heavy-weight logic-based tools for a more through verification would be an appropriate next step.

7. CONCLUSION

We have used formal simulations to analyze a network layer accounting protocol from its earliest design stages to a mature specification. At the outset we were aware of requirements not met by existing protocols, such as the need to authenticate ingress traffic as well as egress traffic, but formal simulation revealed other issues of significant concern. In particular, we found that a naive protocol design included cases in which race conditions could create incomplete tunnels. There were also challenges in understanding the model, and our early versions of the IP model did not account for important but subtle properties. Our formulation led to a better understanding of these issues, their resolution, and the maintenance of the desired properties though a series of refactoring steps based on other changes in requirements and design. This case study shows the feasibility of using formal simulations to aid early stages of a protocol design. We are also able to contribute infrastructure for future protocols that concern the configuration of IPsec tunnels.

8. REFERENCES

[1] M. Abadi, B. Blanchet, and C. Fournet. Just Fast Keying in the Pi Calculus. In D. Schmidt, editor, *The European Symposium on Programming (ESOP)*, Lecture Notes inComputer Science 2618. Springer-Verlag, 2004.

[2] M. Abadi and A. Gordan. A Calculus for Cryptographic Protocols: The Spi Calculus. *Information and Computation*, 148(1):1–70, 1999.

[3] W. Aiello, S. Bellovin, M. Blaze, R. Caetti, J. Ioannidis, A. Keromytis, and O. Reingold. Just Fast Keying: Key Aggrement in a Hostile Internet. *ACM Transactions of Information System Security*, 7(2):242–273, 2004.

[4] K. Bhargavan, C. A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: Formal analysis of network simulations. In M. J. Harrold, editor, *ISSTA 00 Proceedings of the ACM SIGSOFT 2000 International Symposium on Software Testing and Analysis*, pages 2–13, Portland, OR, August 2000. ACM.

[5] K. Bhargavan, C. A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Transactions on Software Engineering*, 28(2):129–145, February 2002.

[6] K. Bhargavan, C. A. Gunter, and D. Obradovic. Routing information protocol in HOL/SPIN. In J. Harrison and M. Aagaard, editors, *Theorem Proving in Higher Order Logics: 13th International Conference, TPHOLs 2000*, volume 1869 of *Lecture Notes in Computer Science*, pages 53–72, Portland, Oregon, August 2000. Springer-Verlag.

[7] K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *Journal of the ACM*, 49(4):538–576, July 2002.

[8] B. Blanchet. From Secrecy to Authencity in Security Protocols. In *9th International Static Analysis Symposium (SAS'02)*, Lecture Notes In Computer Science 2477, pages 342–359. Springer-Verlag, 2002.

[9] B. Blanchet. Automatic Proof of Strong Secrecy for Security Protocols. In *IEEE Symposium on Security and Privacy*, pages 86–100, 2004.

[10] A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and Proof in Membership Equational Logic. *Theoretical Computer Science*, 236:35–132, 2000.

[11] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[12] E. Clark, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[13] 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.

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

[15] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude 2.0 Manual. June 2003, `http://maude.cs.uiuc.edu`.

[16] G. Denker, J. Meseguer, and C.Talctt. Protocol Specification and Analysis in Maude. In *Proc. of Workshop on Formal Methods and Security Protocols*, 1998.

[17] S. Gutierrez-Nolasco, N. Venkatasubramanian, M. Stehr, and C. Talcott. Exploring Adaptability of Secure Group Communication. University of California Technical Report, 2004.

[18] T. Hiller, P. Walsh, X. Chen, M. Munson, G. Dommety, S. Sivalingham, B. Lim, P. McCann, H. Shiino, B. Hirschman, S. Manning, R. Hsu, R. Hsu, M. Lipford, P. Calhoun, C. Lo, E. Jaques, E. Campbell, Y. Xu, S. Baba, T. Ayaki, T. Seki, and A. Hammed. CDMA2000 Wireless Data Reqirements for AAA. RFC 3141, IETF, 2001.

[19] G. Holzmann. *Design and Validation of Computer Protocols*. Prentice-Hall, 1991.

[20] C. Kaufman. Internet Key Exchange(IKE V2) Protocol. RFC 2407, IETF, 2004.

[21] S. Kent and R. Atkinson. Security Architecture for the Internet Protocol. RFC 2401, IETF, 1998.

[22] M. Koutsopoulou, A. Kaloxylos, A. Alonistioti, L. Merakos, and K. Kawamura. Charging, Accounting, and Biling Management Schemes in Mobile Telecommuncations Networks and the Internet. *IEEE Communications Surveys*, 6(1), 2004.

[23] G. Lowe. An Attack on the Needham-Schroeder Public-Key Auntentication Protocol. *Information Processing Letters*, 56(3):131–133, 1995.

[24] K. McMillan. The SMV Manual. November 2000, `http://www-2.cs.cmu.edu/~modelcheck/smv.html`.

[25] C. Meadows. The NRL Protocol Analyzer: An Overview. *Journal of Logic Programming*, 1994.

[26] A. J. Menezs, P. C. van Oorchot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

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

[28] L. Paulson. The Inductive Approach to Verifying Cryptographic Protocols. *J. Computer Security*, 6:85–128, 1998.

[29] L. Paulson. Inductive Analysis of the Internet Protocol TLS. *ACM Transactions on Computer and System Security*, 2(3):332–351, 1999.

[30] C. Rigney. RADIUS Accounting. RFC 2866, IETF, 2000.

[31] P. Ryan and S. Schneider. *Modeling and Analysis of Security Protocols*. Addison-Wesley, 2001.

APPENDIX

A. MAUDE SAMPLES

Maude [15, 14, 13] is an executable specification language based on rewriting logic [27] and its membership equational sublogic [10]. Maude also refers to its high-performance implementation, that is a rewriting engine that allows us to execute and analyze system specifications. The significance of rewriting logic is that distributed systems (and especially networking protocols) 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.

A specification in the basic version [27] of rewriting logic is a triple (Σ, E, R), where Σ is an algebraic signature, E is a set of equations over Σ and R is a set of rules over Σ. Typically, (Σ, E) is an equational specification that makes precise the static aspects of the system. This includes especially the algebraic structure of the state space, which in our case is a multiset, *i.e.* a commutative monoid, of local state elements. The dynamics of the system is then given by the rewrite rules R, which operate modulo the equations E, and in our case correspond to multiset rewrite rules. Hence, we can visualize the state of the distributed system as a 'soup' of local state elements which are transformed by local state transitions represented by rewrite rules.

The Maude formalization of the protocols given above is quite involved. We present several examples of how the informal descriptions are transformed into Maude. First, a note on notation. Maude allows two types of rewrite rules: basic and conditional. The basic rewrite rule has the form:

$$\texttt{rl}\ r_1 r_2 \dots r_n \Rightarrow l_2 l_2 \dots l_m.$$

When terms $r_1, r_2 \dots r_n$ are all present in the soup, the rule fires sending terms $l_1, l_2 \dots l_m$ into the soup. Conditional rewrite rules

take the following form:

$$\texttt{crl}\ r_1 r_2 \ldots r_n \Rightarrow l_2 l_2 \ldots l_m\ \texttt{if}\ B.$$

If the boolean condition B is true and the terms $r_1, r_2 \ldots r_n$ are all present in the soup, the rule fires sending terms $l_1, l_2 \ldots l_m$ into the soup.

A.1 PKI

In this section we give the Maude equations for the basic cryptographic operations encryption, decryption, digital signatures, and signature verification. As ordinary mathematical functions they can be naturally specified in terms of equational logic and executed by equational rewriting. We assume that the secret and public keys are maintained as a `keypair`. If data `d` was encrypted using the public key `pk`, then decrypting using the secret key `sk` should return the original data. Digital signatures are modeled as an ordered pair. The first element being the private key used to generate the signature and the second element being the data. The `data` operator acts as a projection function on the signature. The `verify` operator takes a public key and verifies that the signature was generated using the corresponding private key.

```
sort Data .
op encrypt : PubKey  Data  -> Data .
op decrypt : SecKey  Data ~> Data .
op keypair : SecKey  PubKey -> Prop .
var d : Data .

ceq decrypt(sk, encrypt(pk,d)) = d
    if keypair(sk,pk) : True .

op sign : SecKey Data -> Data .

op data : Data ~> Data .
eq data(sign(sk,d)) = d .

op verify : PubKey Data -> Prop .
cmb verify(pk, sign(sk,d)) : True
    if keypair(sk,pk)  : True .
```

A.2 SIKE Daemon

We now give the rules for the final version of the SIKE daemon. The SIKE daemon at each node waits for the first SIKE message from the initiator. The responder generates a nonce, SPI, and forms the cookie. This data is then sent to the initiator in the second message `msg2` of the SIKE exchange.

```
crl [SikedStart] :
   siked-start(nodeB)
   pki-info(nodeB, seckey', pubkey', cert')
   fresh(m)
   ipsec-delivered(nodeB, rinterB, attrs, sabundle,
       ip(addrA, addrB,  msg1(rA, spi(spi-A,0)))) =>
   fresh(s m)
   pki-info(nodeB, seckey', pubkey', cert')
   siked-phase-1(nodeB, addrB, cert', seckey',
             random(m),spi(spi-A,0), rA, addrA)
   if not(contains(attrs,discard)) .

rl [SikedPhase-1] :
   fresh(m)
   fresh-spi(nodeB, n')
   siked-phase-1(nodeB, addrB, cert', seckey',
          secret,   spi(spi-A,0), rA, addrA) =>
   fresh(s m)
   fresh-spi(nodeB, s n')
    siked-waiting(nodeB, addrB,  cert', seckey',
     secret, spi(spi-A, n'), rA, addrA, random(m))
   ipsec-send-req(siked(nodeB), nodeB,
    ip(addrB, addrA,
      msg2(rA, random(m), spi(spi-A,n'), cert',
          cookie(versec, hash(secret,
           cookiedat(rA, random(m), addrA))))) .
```

The daemon waits for the third message in the exchange. The cookie is first verified and if valid, the signature is checked. The SADB and SPDB are updated to include the A → B association.

```
crl [SikedWaiting] :
   ipsec-send-ack(siked(nodeB), nodeB)
   siked-waiting(nodeB, addrB,  cert', seckey',
                    secret, spi,  rA, addrA, rB)
   ipsec-delivered(nodeB, rinterB, attrs, sabundle,
         ip(addrA, addrB,
           msg3(cert, cookieb,signeddataA))) =>
   siked-phase-2(nodeB, addrB, cert', seckey',
              secret, spi, rA, addrA, rB, cert)
     if not(contains(attrs,discard)) /\
       cookieb == cookie(versec, hash(secret,
                  cookiedat(rA, rB, addrA))) /\
     verify(certkey(cert), signeddataA) : True  /\
    msg3data(rA, rB, addrB, spi) := data(signeddataA) .

rl [SikedPhase-2] :
   interfaces(nodeB, interfaces')
   fresh(m)
   siked-phase-2(nodeB, addrB,  cert', seckey',
               secret, spi, rA, addrA, rB, cert) =>
   interfaces(nodeB, interfaces')
   fresh(s m)
   siked-expect-setkey-term(nodeB, addrB, cert',
       seckey',  secret, spi,  rA, addrA, rB,
                      cert, sharedkey(random(m)))
   setkey-start(nodeB, addrA, addrB,
                       spival-resp(spi), IN-SPD) .
```

The fourth message is formed and sent to the initiator. Once processing has completed for that message, the SADB and SPDB are updated to include the B → A association.

```
rl [Siked-Send-MSG4] :
   siked-expect-setkey-term(nodeB, addrB, cert',
     seckey', secret, spi,rA, addrA, rB, cert,
                               sharedkey(random(m)))
   setkey-terminate(nodeB,addrA,addrB,spi-B,inout) =>
   siked-sent-msg4(nodeB, addrB,  cert', seckey',
        secret,spi, rA, addrA, rB, cert,
                               sharedkey(random(m))
     ipsec-send-req(siked(nodeB), nodeB,
      ip(addrB, addrA,
             msg4( sign(seckey',
              msg4data(rB, addrA, rA, spi,
                  encrypt(certkey(cert),
                  sharedkey(random(m))))))) .

rl  siked-sent-msg4(nodeB, addrB,  cert', seckey',
             secret, spi, rA, addrA, rB, cert,
                               sharedkey(random(m)))
              ipsec-send-ack(siked(nodeB), nodeB) =>
    siked-expect-setkey-term'(nodeB, addrB, cert',
       seckey', secret,spi, rA, addrA, rB, cert,
                                sharedkey(random(m)))
   setkey-start(nodeB, addrB, addrA,
                         spival-init(spi), OUT-SPD) .

 rl [sikedPhase3] :
   siked-expect-setkey-term'(nodeB, addrB,  cert',
        seckey',secret,  spi, rA, addrA, rB, cert,
                                 sharedkey(random(m)))
   setkey-terminate(nodeB,addrB,addrA,spi-A,inout) =>
   siked-term(nodeB, addrB, cert', seckey', secret,
        rA, addrA,  rB, cert, spi,
                               sharedkey(random(m)))
   siked-notify(nodeB, addrA, addrB, spi)
```

```
   siked-restart(nodeB, addrB, cert',seckey',secret,
    spi, rA, addrA, rB, cert, sharedkey(random(m))) .

crl [SikedRestart] : interfaces(nodeB, interfaces')
    siked-restart(nodeB, addrB, cert', seckey',
              secret, spi, rA, addrA, rB, cert,
              sharedkey(random(m))) =>
    interfaces(nodeB, interfaces')
    siked-start(nodeB)
       if contains(interfaces', addrB) .
```

A.3 L3A

The informal description of L3A says that after the server receives notification from SIKE that key exchange with the NAS is complete, it checks the credential and begins a key exchange with the client to establish the association between the server and client. Once this exchange is complete, the policy is updated and the Fin message is sent. This is formalized by the following rewrite rules, which are executed sequentially.

```
rl l3a-server-start(node)
  siked-notify(node, nas, server, server-nas-spi) =>
  l3a-server-expect-ack(node, nas,server,
                                server-nas-spi) .

crl l3a-server-expect-ack(node,nas,server,
                            server-nas-spi)
    spdb(node, spindb,spoutdb)
    ipsec-delivered(node,rinterface,attrs,sabundle,
      ip(nas,server,ack(ccert,client-cred,New) )) =>
   spdb(node,spindb, sSPList(sp(src-dest-pattern(
   server,certaddr(ccert)), sSAList(sa(server,nas
        ,spival-init(server-nas-spi)))))  spoutdb)
      l3a-server-check-cred(node,nas,server,
                client-cred, ccert, server-nas-spi)
          if not(contains(attrs,discard)) .

crl    l3a-server-check-cred(node,nas,server,
        client-cred, ccert, server-nas-spi)  =>
       l3a-server-activated(node,certaddr(ccert),
       nas,server,client-cred,ccert, server-nas-spi)
  if verify(certkey(ccert), client-cred) : True  /\
     reqdata(tlive,nas) := data(client-cred) .

rl  l3a-server-activated(node,client,nas,server,
      client-cred,ccert, server-nas-spi)
    pki-info(node, seckey, pubkey, cert)  =>
    l3a-server-expect-sike-ack(node,client,
  nas,server,client-cred,ccert, server-nas-spi)
    pki-info(node, seckey, pubkey, cert)
    sike-start(node,server,client,cert,seckey) .

rl  l3a-server-expect-sike-ack(node,client,nas,
     server,client-cred,ccert, server-nas-spi )
     sike-ack(node, server, client,
                          server-client-spi)  =>
        l3a-server-send-fin(node,client,nas,server,
 client-cred,ccert,server-nas-spi, server-client-spi)

rl  l3a-server-send-fin(node,client,nas,server,
           client-cred,ccert, server-nas-spi,
                           sderver-client-spi ) =>
    ipsec-send-req(server(node),node,
              ip(server,client,fin) )
    l3a-server-expect-ipsec-send-ack(
       node,client,nas,server,client-cred,
    ccert, server-nas-spi, server-client-spi) .
```

A.3.1 Reusing Tunnels

We shall now illustrate some of the rules involving tunnel reuse that were given informally in Section 5. The first time the client uses L3A the client invokes SIKE to set up a tunnel between the client and the NAS and then sends the Req message. The protocol assumes that all further interaction will be through the same NAS. So when the first invocation of L3A is finished, the protocol moves into the `l3a-client-restart` mode. If L3A is invoked in this state, the Req message is sent, but there is no SIKE exchange with the NAS. The client's rules are given as follows.

```
rl  l3a-client-start(node,client)
    l3a-client-req(node,client,nas,server)
    pki-info(node,seckey,pubkey,cert) =>
    l3a-client-expect-sike-ack-1(node,client
                                      ,nas,server)
    pki-info(node,seckey,pubkey,cert)
    sike-start(node, client, nas,cert, seckey) .

rl l3a-client-expect-sike-ack-1(node,client,nas,server)
   sike-ack(node, client, nas, client-nas-spi) =>
   l3a-client-send-req-new(node,client,nas,server,
                                    client-nas-spi) .
 .....
rl  l3a-client-restart(node,client,nas, client-nas-spi)
    l3a-client-req(node, client,nas, server)       =>
     l3a-client-send-req-reuse(node,client,nas,
                              server,client-nas-spi) .

rl l3a-client-send-req-reuse(node,client,nas,server,
                                     client-nas-spi)
   pki-info(node,seckey,pubkey,cert)
   spdb(node,spindb,spoutdb) =>
   pki-info(node,seckey,pubkey,cert)
   ipsec-send-req(client(node),node,ip(client,nas,
       req(cert,sign(seckey,reqdata(Timelive,nas)),
                                  server,Reused) ) )
  spdb(node, spindb, sSPList(sp(src-dest-pattern(
       client,server), sSAList(sa(client, nas,
       spival-resp(client-nas-spi) )))) spoutdb)
    l3a-client-expect-ipsec-send-ack(node,client,
                        nas,server,client-nas-spi).
```

If the NAS is in the start state and receives notification that a key exchange has taken place it waits for the Req message. On the other hand, there may already be a tunnel with the client so the NAS would receive a Req message instead of a key exchange occurring. In both cases, the policy database must be updated with a rule saying that all traffic from the client to the server involved in this L3A invocation must travel in the client-NAS tunnel.

```
rl l3a-nas-start(node)
  siked-notify(node,client,in-nas,client-nas-spi) =>
  l3a-nas-got-notify(node,client,in-nas,
                                    client-nas-spi) .

crl l3a-nas-got-notify(node,client,in-nas,
                                    client-nas-spi)
    routetab(node,routelist)
    spdb(node, spindb,spoutdb)
    ipsec-delivered(node,rinterface,attrs,sabundle,
         ip(client,in-nas,req(initiator-cert,
                        client-cred,server,New))) =>
   spdb(node, sSPList(sp(src-dest-pattern(
       client,server), sSAList(sa(client,in-nas,
     spival-resp(client-nas-spi))))) spindb, spoutdb)
   routetab(node,routelist)
   l3a-nas-phase2(node,client,in-nas,
        get-interface-to-dest(routelist,server),
   server,initiator-cert, client-cred, client-nas-spi)
    if not(contains(attrs,discard)) .

crl l3a-nas-start(node)
    sadb(node,sadb)
    ipsec-delivered(node,rinterface,attrs,sabundle,
    ip(client,in-nas,req(initiator-cert,client-cred,
                                  server,Reused))) =>
```

```
    sadb(node,sadb)
    l3a-nas-reuse-cn(node,client,in-nas,server,
    initiator-cert,client-cred, spi(getspi(sadb,
       in-nas,client),getspi(sadb,client,in-nas)))
    if not(contains(attrs,discard)) /\
         fetch(sadb,client,in-nas) =/= nullsa  /\
         fetch(sadb,in-nas,client) =/= nullsa .

rl spdb(node,spindb,spoutdb)
   routetab(node,routelist)
   l3a-nas-reuse-cn(node,client,in-nas,server,
       initiator-cert,client-cred,client-nas-spi) =>
  routetab(node,routelist)
  spdb(node,sSPList(sp(src-dest-pattern(
    client,server), sSAList(sa(client, in-nas,
    spival-init(client-nas-spi))))) spindb, spoutdb)
  l3a-nas-phase2(node,client,in-nas,
   get-interface-to-dest(routelist,server),
   server,initiator-cert,client-cred,client-nas-spi) .
```