

Cross-Layer Verification of Type Flaw Attacks on Security Protocols

Benjamin W. Long¹

Colin J. Fidge²

David A. Carrington¹

¹School of Information Technology and Electrical Engineering
The University of Queensland, Brisbane, Australia

²School of Software Engineering and Data Communications
Queensland University of Technology, Brisbane, Australia

Abstract

Security protocols are often specified at the *application layer*; however, application layer specifications give little detail regarding message data structures at the *presentation layer* upon which some implementation-dependent attacks rely. In this paper we present an approach to verifying security protocols in which both the application and presentation layers are modelled. Using the Group Domain of Interpretation protocol as an example, our application layer specification of the protocol is used as input to the AVISPA model checking tool for analysis. Two type flaw attacks are found via model checking which are then verified against the corresponding presentation layer specification, thus identifying the minimal requirements to prevent the attacks.

1 Introduction

Electronic communication is now the predominant means of interaction for commercial, industrial, and private use. In response to this trend, it is important to ensure that transmitted information is not compromised by malicious parties, especially in areas such as defence, medicine, and commerce, where information leakage and corruption could have catastrophic consequences. In order to shield ourselves against potential threats, communication messages are secured by application of cryptographic functions and sent as part of ordered message sequences called *security protocols*.

Informal narrations that mix natural language and ad hoc notations (Abadi 2000), such as the *standard notation* (Carlsen 1994), conveniently describe security protocols at the application layer — the level at which message content is determined. However, standard notation descriptions do not indicate precisely what internal actions are required by protocol agents implementing them, nor does it suggest desirable properties of message items or cryptographic functions used. Furthermore, the standard notation gives little detail regarding message data structures at the presentation layer — the level at which the low-level representation of messages is determined — upon which some implementation-dependent attacks such as *type flaw attacks* rely. A lack of precise details at either of these layers can lead to misunderstandings and disputes over protocol correctness (Boyd 1990, Boyd & Mao 1993).

Formal methods provide well-defined languages that allow precise specifications to be written and

subsequent rigorous verification procedures to be performed. Hence, the use of formal methods is advocated for the design and analysis of security protocols (Gollmann 2003). The *Common Criteria* (The Common Criteria Project Sponsoring Organisations 1999), an internationally recognised set of criteria for evaluating security critical products, demands strict application of formal methods to the development process for achieving the highest assurance levels. Therefore, we are interested in the use of formal methods for specifying precise details of security protocols, and for analysing specifications of security protocols for potential attacks.

Type flaw attacks (Boyd 1990) occur at the presentation layer when intruders send unexpected messages to protocol agents who subsequently misinterpret the bit string encoding of message item types. For example, a type flaw attack may occur when a single message item of one type is confused with a single item of another type, or when a single message item of one type is confused with the concatenation of two or more other items of varying types. Type flaw attacks can be prevented by the use of ‘tags’ (Heather, Lowe & Schneider 2000), but problems can arise when a protocol interacts with another protocol that does not use a tagging scheme, or tags data in a different way (Meadows, Syverson & Cervesato 2004) (such as the attack on the GDOI protocol described in Section 6).

Nevertheless, we seek to find the necessary and not merely the sufficient. Hence for a particular attack, we want to identify the specific weaknesses upon which an attack depends in order to minimise the effort required to prevent it. Over-protective tagging schemes can unnecessarily increase message sizes, complexity and other communication overheads. Battery-powered embedded systems such as PDAs, cell phones, networked sensors and smart cards require such overheads to be minimal (Potlapally, Ravi, Raghunathan & Jha 2003). In these systems, minimising the resources needed to prevent potential attacks is clearly beneficial.

Generally, security protocol analysis tools have not been very good at finding type flaw attacks (Meadows 2003). Although more advanced tools exist (Meadows et al. 2004, Armando et al. 2005) that are capable of finding type flaw attacks, their analyses are based on high-level application layer specifications. However, by relying on application layer specifications, proofs of correctness may not be accurate with respect to the implementation of the protocol at the lower presentation layer. For example, attacks are sometimes found that are actually prevented by implicit presentation layer behaviour, and sometimes presentation layer attacks are completely overlooked. Furthermore, it is difficult to add detailed presentation layer corrections to application layer specifications, due to the level of abstraction exhibited at the application layer; often

corrections are presented informally or the entire design is changed. Once a type flaw attack has been discovered at the application layer, we need a way of verifying whether the attack is possible given the presentation layer specification, and to confirm the required presentation layer correction for inclusion in the formal specification.

In this paper we bridge the gap between the application and presentation layers by providing a framework for specifying security protocols for formal analyses at both levels of abstraction. Conveniently, we produce a single formal specification in which the presentation layer is transparent but easily reasoned with when required, thus improving readability and simplifying translation for input to off-the-shelf analysis tools. We demonstrate our approach by providing a multi-layered formal specification of the Group Domain of Interpretation (GDOI) protocol (Baugher, Hardjono, Harney & Weis 2001) in Object-Z (Duke & Rose 2000). The application layer model is analysed using the AVISPA model checking tool (Armando et al. 2005) to find two type flaw attacks. The attacks are then verified against the presentation layer model using Object-Z’s schema calculus, and the particular presentation layer requirements required to prevent the attacks are formally derived for inclusion in the formal specification.

2 Related Work

In previous research Donovan et al. (1999) used CSP with the FDR model checker to analyse cryptographic protocols. They state that their approach is not good at finding type flaws. This is no doubt due to the lack of data structure support in the CSP model — message structure is captured in the event name.

Carlsen (1993) modified the logic of Communication, Knowledge and Time (CKT5) for analysis of cryptographic protocols, and demonstrated his approach by finding a type flaw attack on a version of the Neuman-Stubblebine protocol. The attack depends on confusion between a nonce¹ and a key. Carlsen found this attack by assuming that each item belonging to a particular type was distinct from items in all other types, and then by imposing a special requirement that nonces and keys were not necessarily distinct.

Bozzano and Delzanno (2002) used linear logic to discover that Millen’s *ffgg* protocol (Millen 1999) is vulnerable to an attack which also requires one item to be interpreted as another. They assumed informally that the attack requires all items to have the same length. Bozzano (2002) used the same method to discover the type flaw attack on the Otway-Rees protocol. To allow for several items to be read as one, he introduced an extra concatenation operator *cons* to “glue together” different items in the message.

Theya et al. (1998) have used *strand spaces* to analyse cryptographic protocols. In this formalism, protocol correctness claims are expressed in terms of the connections between sequences of legitimate and/or intruder actions. Type flaw attacks are catered for by allowing unification of single items or pairs of items with other single items.

Cervesato (2001) used the strongly-typed Multi-set Rewriting (MSR) language which is based on term rewriting, linear logic and type theory, to express type flaw attacks. He uses polymorphism to allow for ‘confusable types’; the user of the approach decides which types are confusable in order to find type flaw attacks.

The attacks considered in these approaches generally involve simple ‘type confusion’, in which message

items of one type are confused with items of another type, or in the more advanced models, in which an item of one type is confused with the concatenation of two or more other items of varying types. For instance, Figure 1 shows two identical bit string representations of a message at the presentation layer consisting of an agent identifier and a nonce, and for each, an alternative way of interpreting the message at the application layer. The first illustrates the scenario in which the single nonce item is interpreted as a key, and the second illustrates the scenario in which the concatenation of the agent identifier with the nonce is interpreted as a key.

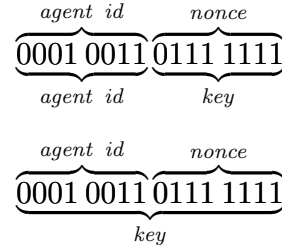


Figure 1: Misinterpretation of message items.

Meadows (2002) highlighted the further possibility of attacks in which sub-items of one type may be confused with sub-items of another type. This is particularly imaginable in the case where agents use different parsing algorithms for different protocols. In Figure 2, for instance, not only is the message interpreted at the application layer to consist of different types, but the item lengths are not even the same. Motivated by this, Meadows (2003) investigated a presentation layer procedure for determining the probability of any kind of type confusion occurring for any two pairs of protocol messages.

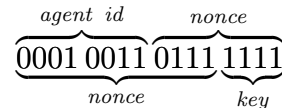


Figure 2: Misinterpretation of message sub-items.

Recently Meadows et al. (2004) analysed the GDOI protocol using the purpose-built NRL Protocol Analyzer to find a type flaw attack (Attack B) that takes advantage of the fact that GDOI is built on top of a protocol that does not tag message headers. Discovery of the attack also led to the manual discovery of a second type flaw attack (Attack A). Unfortunately, the tool could not have found Attack A since it lacked the ability to model associativity of message concatenation.

On the other hand, the AVISPA tool (Armando et al. 2005) handles associativity of message concatenation, so we were convinced it would successfully find both type flaw attacks on the GDOI protocol. Nevertheless, most protocol analysis tools (including AVISPA) are aimed at application layer models. Hence, it is difficult to include specific presentation layer requirements, for verification of the presentation layer specification.

Previously we demonstrated the value of using the Object-Z formal specification language for reasoning about the kind of type flaw attacks illustrated in Figures 1 and 2, enabling us to determine whether potential type flaw attacks are, or are not, actually possible for a given presentation layer specification of a security protocol (Long 2005). In this paper we bridge the

¹A nonce (Gong 1993) is a datum that is unique to each protocol instance.

1. $A \longrightarrow S : M, \{h(N_A, G), N_A, G\}_{K_{AS}}$
2. $S \longrightarrow A : M, \{h(N_A, N_S, SA), N_S, SA\}_{K_{AS}}$
3. $A \longrightarrow S : M, \{h(N_A, N_S, \{N_A, N_S\}_{K_A^{-1}}), \{N_A, N_S\}_{K_A^{-1}}\}_{K_{AS}}$
4. $S \longrightarrow A : M, \{h(N_A, N_S, Q, K, \{N_A, N_S\}_{K_S^{-1}}), Q, K, \{N_A, N_S\}_{K_S^{-1}}\}_{K_{AS}}$

Figure 3: Group Key Pull protocol.

gap between application and presentation layer verification by taking advantage of the AVISPA tool for automated analysis at the application layer followed by formal proof at the presentation layer.

3 The GDOI Protocol

The Group Domain of Interpretation (GDOI) protocol (Baugher et al. 2001) is proposed for group key distribution, in which each key is secret to a group of agents for secure communication amongst its members. Based on related work by Meadows et al. (2004), its two subprotocols relevant to our analysis are described below.

3.1 Group Key Pull Protocol

Agents wishing to join a group take part in the *Group Key Pull* protocol to learn the current group key (described using the standard notation in Figure 3).

In step 1, the new group member, Alice A , sends a message consisting of her nonce N_A , the name G of the group she wishes to join, and a hash of these items $h(N_A, G)$, all encrypted (denoted by ‘ $\{\}$ ’) using the key K_{AS} shared by her and the server, Sam S . A message header M is prepended to the message.

In step 2, Sam responds by sending Alice a message consisting of his nonce N_S , the security association SA , and a hash of these items accompanied by Alice’s nonce, again encrypted and prepended by the message header. The security association describes the security functions and policies used by the group.

In step 3, using her private key K_A^{-1} , Alice creates a signature from the two nonces. Her request for a key (message 3) contains this signature, and a hash of the signature accompanied by the original nonce values.

Like Alice, Sam creates a signature using his private key K_S^{-1} from the two nonces, and in step 4 sends a similar message to Alice containing the current sequence number Q , and the current group key K . (The sequence number must be known by all group members for use in the Group Key Push protocol.) Alice authenticates message 4 by ensuring the nonce values are the same as those she signed in message 3, and records the sequence number Q and key K for subsequent secure communication amongst the group.

3.2 Group Key Push Protocol

The group key is changed for the entire group by having all members act in Alice’s role for the *Group Key Push* protocol described in Figure 4.

5. $S \longrightarrow A : M, \{Q, SA, K^*, \{M, Q, SA, K^*\}_{K_S^{-1}}\}_K$

Figure 4: Group Key Push protocol.

In this single message protocol, Sam sends a message to Alice containing the sequence number, security association, and the new group key K^* , along with a signature taken over the entire content including the message header. The message is encrypted

using the current group key K . The sequence number is incremented by Sam each time the group key is changed so as to avoid the possibility of replay attacks on this protocol. Therefore, on receipt of this message, Alice checks that the sequence number is the one expected before accepting the key K^* as the new group key.

4 Specifying GDOI in Object-Z

Object-Z (Duke & Rose 2000) is an object-oriented formal specification language in which set theory and logic is used to describe the internal states of system classes and the behaviour of class operations on those states. In previous work (Long 2005) we presented Object-Z data types for modelling messages at the presentation layer. We reuse these structures for the foundation of this specification.

We assume a given set $ATOM$ of ‘atoms’ from which all messages are constructed at the presentation layer (for example, bits or bytes). Then the set of all messages MSG is the set of all possible sequences of atoms.

$$MSG == \text{seq } ATOM$$

Then we declare eight subsets of MSG for the different types of data items that exist at the application layer: header items HID , nonces NON , group identifiers GID , security associations SEC , sequence numbers SEQ , keys KEY , encrypted items ENS and hashed items HSH .

$$\left| \begin{array}{l} HDR, NON : \mathbb{P} MSG \\ GID, SEC : \mathbb{P} MSG \\ SEQ, KEY : \mathbb{P} MSG \\ ENC, HSH : \mathbb{P} MSG \end{array} \right.$$

The subsets are not necessarily disjoint which means that individual items may belong to one or more subsets of MSG . Keys are divided into another three subsets (symmetric keys SYM , public keys PUB and private keys PRV) enabling us to model both symmetric key and public key encryption.

$$\left| \begin{array}{l} SYM : \mathbb{P} KEY \\ PUB : \mathbb{P} KEY \\ PRV : \mathbb{P} KEY \end{array} \right.$$

To allow for analyses of type flaw attacks, our data structures let different agents interpret the same message as consisting of different sequences of typed items. However, when two agents ‘speak the same language’ or agree on the type structure of the message, there should be no ambiguity.

Instead of enforcing a specific correlation between the application and presentation layers, we assume the following global axiom which says that if two agents both interpret the initial part of a message to be a particular type of item, then the values they associate with this item are identical. (The sequence concatenation operator ‘ \wedge ’ forms a single message from the two supplied messages.)

$$\left| \begin{array}{l} (\forall m, n : HDR; o, p : MSG \bullet \\ m \hat{\wedge} o = n \hat{\wedge} p \Rightarrow m = n) \end{array} \right.$$

Thus, if two identical messages ‘ $m \hat{\wedge} o$ ’ and ‘ $n \hat{\wedge} p$ ’ begin with items m and n , both of which are interpreted to be of type *HDR*, then m and n must be the same message header. We have omitted the predicates constraining the other item types for the sake of brevity.

Our approach relies on unifying messages encrypted with the same key. To enable this, the encrypt function *enc* ensures that for every key there is a unique function that produces a unique encrypted item for each message.

$$\left| \text{enc} : KEY \mapsto (MSG \mapsto ENC) \right.$$

The hash function *hsh* used by the Group Key Pull protocol takes a key as input. The following specification ensures that for every key there is a unique function that produces at most one corresponding hash value for each message.

$$\left| \text{hsh} : KEY \mapsto (MSG \rightarrow HSH) \right.$$

Each public key is associated with a unique private key (its inverse) as specified by the function *inv*. In practice, this function is not one that would be globally available for use by protocol agents. However, it is convenient to have this association formalised. Then for the purpose of our specification we only need to allocate public keys to agents; private keys can be accessed using the *inv* function. In Section 5 we use a model checking tool for verification that provides a function *inv* associating public and private keys in the same way.

$$\left| \text{inv} : PUB \mapsto PRV \right.$$

4.1 Protocol Roles

Although protocol descriptions refer to agents by name, in reality these protocols can be run between any pair of agents. In fact, when we refer to agent Alice, Alice is the name we are giving, not to the agent participating in the interaction, but to the role the agent is participating in. We could model the protocol as a single system or as the interaction of specific agents. However, as in other approaches (Snekkenes 1992, Ryan, Schneider, Goldsmith, Lowe & Roscoe 2000), we choose to model the protocol in terms of protocol roles and the interactions between them.

Between the two subprotocols, there is a total of four roles agents can play: *Alice_{PULL}*, *Sam_{PULL}*, *Alice_{PUSH}* and *Sam_{PUSH}*. Each role is captured within a single Object-Z class specification, including only information and operations relevant to the role it is modelling.

The first class *Alice_{PULL}* corresponds to Alice’s role in the Group Key Pull protocol. All items used by Alice in this role are declared as state variables in the state schema. Alice is required to generate a ‘fresh’ value for her nonce N_A and the message header M for each protocol instance. To allow for this, Alice keeps a record *used* of previously used items.

<i>Alice_{PULL}</i>
$\begin{array}{l} M : HDR; N_A, N_S : NON; G : GID \\ Q : SEQ; SA : SEC; K_{AS}, K : SYM \\ K_A, K_S : PUB; used : \mathbb{P} MSG \end{array}$
<hr/> <i>initiate</i> $\begin{array}{l} \Delta(M, N_A, used) \\ msg! : MSG \end{array}$ <hr/> $\begin{array}{l} M' \notin used \wedge N_A' \notin used \\ used' = used \cup \{M', N_A'\} \\ msg! = M' \hat{\wedge} enc(K_{AS})(hsh(K_{AS})(N_A' \hat{\wedge} G) \hat{\wedge} N_A' \hat{\wedge} G) \end{array}$
<hr/> <i>requestKey</i> $\begin{array}{l} \Delta(N_S, SA) \\ msg?, msg! : MSG \end{array}$ <hr/> $\begin{array}{l} msg? = M \hat{\wedge} enc(K_{AS})(hsh(K_{AS})(N_A \hat{\wedge} N_S' \hat{\wedge} SA') \hat{\wedge} N_S' \hat{\wedge} SA') \\ msg! = M \hat{\wedge} enc(K_{AS})(hsh(K_{AS})(N_A \hat{\wedge} N_S' \hat{\wedge} enc(inv(K_A))(N_A \hat{\wedge} N_S')) \hat{\wedge} enc(inv(K_A))(N_A \hat{\wedge} N_S')) \end{array}$
<hr/> <i>pullKey</i> $\begin{array}{l} \Delta(K, Q) \\ msg? : MSG \end{array}$ <hr/> $\begin{array}{l} msg? = M \hat{\wedge} enc(K_{AS})(hsh(K_{AS})(N_A \hat{\wedge} N_S \hat{\wedge} Q' \hat{\wedge} K' \hat{\wedge} enc(inv(K_S))(N_A \hat{\wedge} N_S)) \hat{\wedge} Q' \hat{\wedge} K' \hat{\wedge} enc(inv(K_S))(N_A \hat{\wedge} N_S)) \end{array}$

Within operation schemas, pre-state variables are undecorated and denote the value before execution of the operation, whereas post-state variables are decorated with a prime ‘ $'$ ’ and denote the value after execution of the operation. The symbol ‘ Δ ’ declares pre-state and post-state variables for each of the named variables, indicating that they may be changed by the operation. Incoming and outgoing messages are specified using input and output variables, denoted by ‘?’ and ‘!’ respectively.

Operation *initiate* corresponds to step 1 of the Group Key Pull protocol. Alice chooses fresh values for her nonce N_A' and the message header M' by checking that they do not yet belong to the set *used* of previously used values. She updates her record of used items to include these values using the set union operator. Appropriate functions are applied to the items required to construct message 1 and the output variable *msg!* is updated with the resultant message.

Operation *requestKey* corresponds to step 2. Use of post-state variables N_S' and SA' in the description of the incoming message *msg?* models the way Alice learns these values for future use. Use of the function *inv* accesses Alice’s private key for creating her signature over the nonces.

Finally, *pullKey* corresponds to Alice receiving message 4 in which she learns the value of the key K' and the current sequence number Q' for the group.

The second class specified for the Group Key Pull protocol is for Sam’s role. Operation *respond* corresponds to step 2. Like Alice, Sam ensures his nonce N_S' has not been used previously. On receiving Alice’s nonce N_A' from the incoming message, he updates the message in transit with his fresh nonce and the security association SA . Operation *giveKey* corresponds to step 4 in which Sam receives Alice’s request for the key in message 3 and sends the group key K and sequence number Q in message 4.

Sam_{PULL}

$$M : HDR; N_A, N_S : NON; G : GID$$

$$Q : SEQ; SA : SEC; K_{AS}, K : SYM$$

$$K_S, K_A : PUB; used : \mathbb{P} MSG$$
respond

$$\Delta(M, N_S, N_A, used); msg?, msg! : MSG$$

$$N_S' \notin used \wedge used' = used \cup \{N_S'\}$$

$$msg? = M' \frown enc(K_{AS})(hsh(K_{AS})(N_A' \frown G) \frown N_A' \frown G)$$

$$msg! = M' \frown enc(K_{AS})(hsh(K_{AS})(N_A' \frown N_S' \frown SA) \frown N_S' \frown SA)$$
giveKey

$$msg?, msg! : MSG$$

$$msg? = M \frown enc(K_{AS})(hsh(K_{AS})(N_A \frown N_S \frown enc(inv(K_A))(N_A \frown N_S)) \frown enc(inv(K_A))(N_A \frown N_S))$$

$$msg! = M \frown enc(K_{AS})(hsh(K_{AS})(N_A \frown N_S \frown Q \frown K \frown enc(inv(K_S))(N_A \frown N_S)) \frown Q \frown K \frown enc(inv(K_S))(N_A \frown N_S))$$

Sam's role in the Group Key Push protocol has one operation *pushKey*, in which he distributes the new group key K^* along with the current group sequence number Q and security association SA , all encrypted with the previous group key K . Group keys, sequence numbers and security associations are not renewed in each protocol instance since they must remain the same for each group member participating. So for our analyses, we assemble various scenarios by coordinating these values as required in the initialisation schema *Init* (in Section 6). Therefore, there is no need in either of Sam's roles to ensure they are fresh.

Sam_{PUSH}

$$M : HDR; Q : SEQ; SA : SEC$$

$$K, K^* : SYM; K_S : PUB; used : \mathbb{P} MSG$$
pushKey

$$\Delta(M, used); msg! : MSG$$

$$M' \notin used \wedge used' = used \cup \{M'\}$$

$$msg! = M' \frown enc(K)(Q \frown SA \frown K^* \frown enc(inv(K_S))(M' \frown Q \frown SA \frown K^*))$$

Finally, Alice has one operation *getKey* for receiving the Group Key Push message. Alice updates her value of the group key by using the post-state variable K' in the description of the incoming message. Earlier we mentioned that Alice checks the sequence number to avoid replay attacks. The value she expects is based on the value she received previously in the Group Key Pull protocol. However, at this level we are focusing on independent roles, hence the value she uses is hard-coded in the initialisation predicate when assembling particular scenarios we are interested in.

Alice_{PUSH}

$$M : HDR; Q : SEQ; SA : SEC$$

$$K : SYM; K_S : PUB$$
getKey

$$\Delta(M, SA, K); msg? : MSG$$

$$msg? = M' \frown enc(K)(Q \frown SA' \frown K' \frown enc(inv(K_S))(M' \frown Q \frown SA' \frown K'))$$

5 Modelling GDOI in AVISPA

AVISPA (Armando et al. 2005) is a tool for the Automated Validation of Internet Security-sensitive Protocols and Applications. Protocol roles are modelled in the High-Level Protocol Specification Language (HLP_{SL}) as state transition systems in a similar way to our Object-Z specification, thus making the translation straightforward. The tool translates HLP_{SL} specifications into an Intermediate Format (IF) describing a single infinite-state transition system for analysis by any of four back-end tools. Two of the back-end tools, the On-the-fly Model-Checker (OFMC) and the Constraint-Logic-based Attack Searcher (CL-AtSe), support type flaw detection.

As an example, the following HLP_{SL} role *sam_{push}* corresponds to the Object-Z *Sam_{PUSH}* class. In Object-Z, we equate values used by multiple roles in particular scenarios in the initialisation schema. However, in AVISPA we equate such values by using parameters, and instantiating roles with particular values at the top-level *environment* role. Therefore, those values we wish to control are declared as parameters, whereas, values used within a single role only are declared as local variables. Declaring the channel with attribute 'dy' indicates the presence of a Dolev-Yao (Dolev & Yao 1983) intruder who has complete control over messages sent between agents.

```

role sam_push(
  A, S : agent, Q, SA : text,
  K, K2 : symmetric_key,
  Ks : public_key,
  MSG : channel(dy))
played_by S def=
local
  M : text
transition
  pushKey.
    MSG(start) =|>
    M' := new() /\
    MSG(M'.{Q.SA.K2.
      {M'.Q.SA.K2}_inv(Ks)}_K) /\
    witness(S,A,alice_server_k,K2)
end role

```

Operation *pushKey* corresponds to the Object-Z *pushKey* operation. In the Object-Z specification we ensured the new value of the message header was fresh by checking it was not in the set of *used* values. Conveniently, the HLP_{SL} provides a special operator *new()* that achieves a similar effect.

The tool analyses authentication properties by attempting to match pairs of *witness* and *request* events. In the above operation, the witness event indicates that Sam intends to agree on key K_2 with Alice. (A *protocol id* *alice_server_k* is used to identify the pair.) The matching request event is made by Alice in the following HLP_{SL} role where she learns the new group key K' .

```

role alice_push(
  A, S : agent, Q : text,
  K : symmetric_key,
  Ks : public_key,
  MSG : channel(dy))
played_by A def=
local
  M, SA : text
transition
  getKey.
    MSG(M'.{Q.SA'.K'}).

```

```

    {M'.Q.SA'.K'}_inv(Ks)}_K) =|>
    request(A,S,alice_server_k,K')
end role

```

HLPSL provides a generic authentication property `authentication_on` which, given a protocol id, will ensure that the request event for that id is always preceded by the matching witness event. If any of the supporting tools find a trace in which the request event is not preceded by the corresponding witness event, an attack will have been found and will be presented to the user. Therefore, to ensure the value Alice accepts for the key is the same as the value Sam intended for the key, we state ‘`authentication_on alice_server_k`’ as the goal. AVISPA allows us to check more requirements, however, we require only this one to find the two type flaw attacks.

6 Verifying Attack A

Using the AVISPA model of the GDOI protocol, AVISPA’s CL-AtSe tool successfully finds the type flaw attack Meadows et al. (2004) were unable to find using the NRL Protocol Analyzer. The attack (Attack A shown in Figure 5) occurs in the scenario where the Group Key Pull protocol is run between a dishonest member, Carl, and the key server Sam, followed by the Group Key Push protocol between an honest member, Alice, and Carl who is posing as Sam.

Being a (dishonest) member of the group, we assume Carl already knows the current sequence number Q . Since all members of a group have the means to calculate the next sequence number for identifying replay attacks on the Group Key Push protocol, he can produce the next sequence number Q^* that the group members are expecting to accompany the new group key.

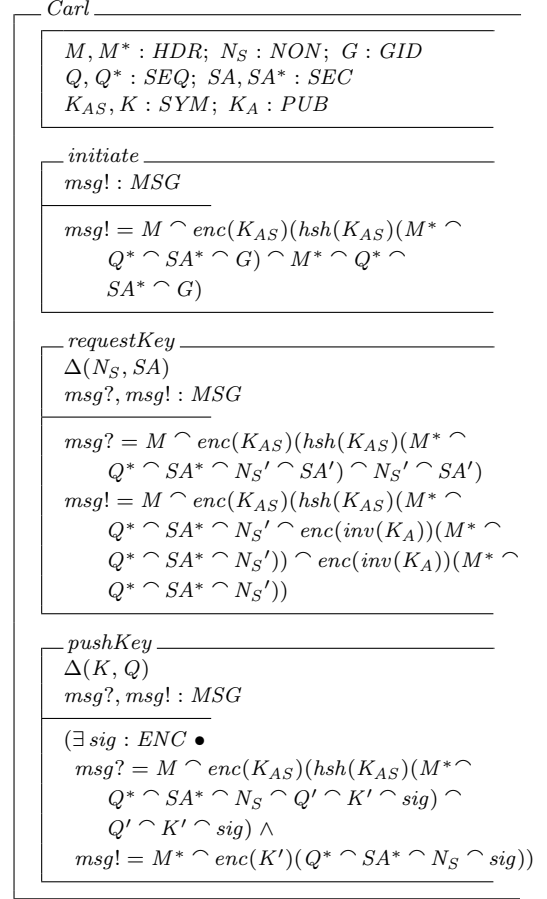
Carl initiates the Group Key Pull protocol using as his nonce N_C , the concatenation of a message header M^* , the next value of the sequence number Q^* and a security association SA^* . Interpreting the composition of these fields as Carl’s nonce, Sam responds appropriately with his nonce N_S and the security association SA in step 2. Continuing the protocol, Carl sends message 3 in which the fake nonce and Sam’s nonce are signed with his private key. In step 4 Sam sends the current sequence number Q and group key K to Carl, together with his own signature over Carl’s fake nonce and Sam’s nonce.

Interestingly, if this signature is sent to Alice as part of a Group Key Push message, she may interpret the concatenation of values M^* , Q^* , and SA^* , that Carl chose for his nonce, hence interpreting N_S as the new group key. Knowing this, Carl initiates an instance of this protocol (step 5) with Alice using Sam’s signature from the previous protocol. Thus, Carl will have successfully masqueraded as the trusted group key server and tricked Alice into believing that Sam’s nonce is the new group key. Carl can initiate the Group Key Push protocol with all members of the group resulting in group communication with an unauthorised key. Furthermore, future communication amongst the group will be completely disrupted for new members and for the group key server.

When this scenario is executed in the AVISPA tool, on accepting Sam’s nonce as the new group key, Alice will signal a `request` event for the value N_S . However, since there is no corresponding `witness` event for this value, the CL-AtSe attack searcher will know that the value Alice received for the key is not the correct one and that the authentication goal has been violated.

In order to formally prove the existence of this attack at the presentation layer and to identify the

specific requirements that suppress it, we first derive a specification of Carl’s involvement in both instances as follows.



The first two operations, *initiate* and *requestKey*, are similar to the operations of the same name in Alice’s role for the Group Key Pull protocol, only the nonce N_A is replaced by the concatenation of values, M^* , Q^* and SA^* . In operation *pushKey*, Carl uses the signature *sig*, created by the server in response to his dummy request, to create the fake Group Key Push message.

An instance of each role required to simulate the scenario in which the attack is present is declared below.

```

carl : Carl; sam_L : Sam_PULL
sam_S : Sam_PUSH; alice : Alice_PUSH

```

First we specify the initial conditions that must hold for the attack to be possible in the following initialisation schema *Init*. We identify which variables belong to each agent instance by prefixing them with the instance name.

Initially, agents will have the same values for certain state variables such as keys, to ensure they can communicate successfully. (The group identifier and sequence number used are also coordinated in this way.) This information is specified in the initialisation schema *Init*.

<i>Init</i>
$carl.G = sam_L.G \wedge carl.K_{AS} = sam_L.K_{AS}$ $sam_L.K_S = alice.K_S \wedge carl.K_A = sam_L.K_A$ $carl.Q^* = alice.Q \wedge sam_L.K = alice.K$

Using Object-Z’s sequential composition operator ‘*g*’, the single operation *attack_A* specifies the complete

1. $C \longrightarrow S : M, \{h(N_C, G), N_C, G\}_{K_{CS}} (N_C = M^*, Q^*, SA^*)$
2. $S \longrightarrow C : M, \{h(N_C, N_S, SA), N_S, SA\}_{K_{CS}}$
3. $C \longrightarrow S : M, \{h(N_C, N_S, \{N_C, N_S\}_{K_C^{-1}}), \{N_C, N_S\}_{K_C^{-1}}\}_{K_{CS}}$
4. $S \longrightarrow C : M, \{h(N_C, N_S, Q, K, \{N_C, N_S\}_{K_S^{-1}}), Q, K, \{N_C, N_S\}_{K_S^{-1}}\}_{K_{CS}}$
5. $C(S) \longrightarrow A : M^*, \{Q^*, SA^*, N_S, \{M^*, Q^*, SA^*, N_S\}_{K_S^{-1}}\}_K$

Figure 5: Attack A on the GDOI protocol.

sequence of operations that lead to the attack after initialisation.

$$\begin{aligned} \text{attack}_A \hat{=} & \text{Init} \wedge \text{carl.initiate} \wp \\ & \text{sam}_L.\text{respond} \wp \text{carl.requestKey} \wp \\ & \text{sam}_L.\text{giveKey} \wp \text{carl.pushKey} \wp \\ & \text{alice.getKey} \end{aligned}$$

Sequential composition is achieved by taking the union of the variables in the Δ -list and the conjunction of both operation predicates, where we assume the existence of an *intermediate state* in which the primed variables from the first operation are equated with the unprimed variables of the same name in the second operation. An intermediate state is introduced by declaring an *intermediate variable* (identified by a double-prime) for each state variable. Additionally, those output variables from the first operation with the same base name as input variables of the second operation are equated and hidden. We do this by introducing an existentially quantified variable to replace them in the same way as we do for intermediate state variables. For example, composition of Carl's *initiate* operation with *sam_L.respond* produces an intermediate variable *msg''* as follows.

$$\begin{aligned} (\exists \text{msg}'' : \text{MSG} \bullet \\ \text{msg}'' = \text{carl}.M \wedge \\ \text{enc}(\text{carl}.K_{AS})(\text{hsh}(\text{carl}.K_{AS})(\text{carl}.M^* \wedge \\ \text{carl}.Q^* \wedge \text{carl}.SA^* \wedge \text{carl}.G) \wedge \text{carl}.M^* \wedge \\ \text{carl}.Q^* \wedge \text{carl}.SA^* \wedge \text{carl}.G) \wedge \\ \text{msg}'' = \text{sam}_L.M' \wedge \\ \text{enc}(\text{sam}_L.K_{AS})(\text{hsh}(\text{sam}_L.K_{AS})(\text{sam}_L.N_A' \wedge \\ \text{sam}_L.G) \wedge \text{sam}_L.N_A' \wedge \text{sam}_L.G)) \end{aligned}$$

Since we know $\text{carl}.G = \text{sam}_L.G$ and $\text{carl}.K_{AS} = \text{sam}_L.K_{AS}$ from initialisation, and due to the constraints placed on presentation layer data types in Section 4, we can derive the equalities $\text{sam}_L.M' = \text{carl}.M$ and (more importantly) $\text{sam}_L.N_A' = \text{carl}.M^* \wedge \text{carl}.Q^* \wedge \text{carl}.SA^*$, allowing us to reason about type flaw attacks in more detail.

Composition of the entire sequence of operations defining the attack trace attack_A results in the following schema.

$\begin{aligned} & \text{attack}_A \\ \Delta & (\text{sam}_L.M, \text{sam}_L.N_S, \text{sam}_L.N_A, \text{alice}.SA \\ & \text{sam}_L.\text{used}, \text{carl}.N_S, \text{carl}.SA, \\ & \text{carl}.K, \text{carl}.Q, \text{alice}.M, \text{alice}.K) \end{aligned}$
$\begin{aligned} \text{carl}.G &= \text{sam}_L.G \wedge \text{carl}.K_{AS} = \text{sam}_L.K_{AS} \\ \text{carl}.K_A &= \text{sam}_L.K_A \wedge \text{carl}.Q^* = \text{alice}.Q \\ \text{sam}_L.K &= \text{alice}.K \wedge \text{sam}_L.K_S = \text{alice}.K_S \\ \text{carl}.SA' &= \text{sam}_L.SA \wedge \text{carl}.N_S' = \text{sam}_L.N_S' \\ \text{sam}_L.N_S' &\notin \text{sam}_L.\text{used} \\ \text{sam}_L.\text{used}' &= \text{sam}_L.\text{used} \cup \{\text{sam}_L.N_S'\} \\ \text{sam}_L.M' &= \text{carl}.M \wedge \text{alice}.K' = \text{sam}_L.N_S' \\ \text{sam}_L.N_A' &= \text{carl}.M^* \wedge \text{carl}.Q^* \wedge \text{carl}.SA^* \\ \text{carl}.K' &= \text{sam}_L.K \wedge \text{carl}.Q' = \text{sam}_L.Q \\ \text{alice}.M' &= \text{carl}.M^* \wedge \text{alice}.SA' = \text{carl}.SA^* \end{aligned}$

Based on the authentication requirement analysed by the tool, a suitable requirement for our verification is that by the end of the Group Key Push protocol, the values that Sam and Alice have for the new group key are identical. Therefore, insecurity of this protocol is proven by demonstrating that after the attack, these values are not equal.

$$\text{attack}_A \Rightarrow \text{sam}_S.K^* \neq \text{alice}.K'$$

We know that after the attack, Alice's value for the group key is equal to the value of Sam's nonce ($\text{alice}.K' = \text{sam}_L.N_S'$). It is reasonable to assume that the value for Sam's nonce is not equal to the value Sam has for the new group key ($\text{sam}_L.N_S' \neq \text{sam}_S.K^*$), and with this assumption we can conclude that the above predicate holds, thus verifying insecurity of the protocol.

The attack operation also reveals the following two conditions that must be preserved by the presentation layer for the attack to succeed.

$$\begin{aligned} (\exists \text{sam}_L.N_A' : \text{NON} \bullet \\ \text{sam}_L.N_A' = \text{carl}.M^* \wedge \text{carl}.Q^* \wedge \text{carl}.SA^*) \\ (\exists \text{alice}.K' : \text{KEY}; \text{sam}_L.N_S' : \text{NON} \bullet \\ \text{alice}.K' = \text{sam}_L.N_S') \end{aligned}$$

The first condition states that a nonce can be constructed from items $\text{carl}.M^*$, $\text{carl}.Q^*$ and $\text{carl}.SA^*$. If this condition is not met, Sam will not accept message 1 from Carl and the protocol will be aborted. The second condition states that Alice can interpret a nonce as a key. If this condition is not met, she will not accept message 5 and again the protocol will be aborted.

These results are somewhat expected; however, we now know for certain that the attack will be prevented by avoiding at least one of these conditions. Thus, the negation of these conditions forms a special presentation layer requirement, providing an Object-Z specification of the protocol secure against this attack, without having to make any additional changes to the application layer design. Either of the following two conditions must hold to prevent the attack.

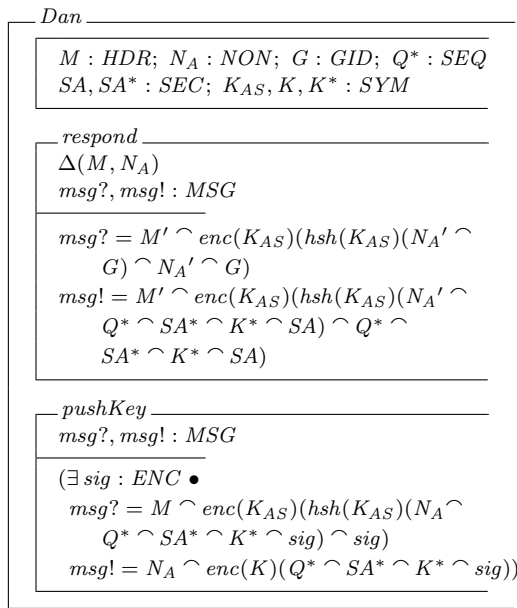
$$\begin{aligned} \neg(\exists \text{sam}_L.N_A' : \text{NON} \bullet \\ \text{sam}_L.N_A' = \text{carl}.M^* \wedge \text{carl}.Q^* \wedge \text{carl}.SA^*) \\ \neg(\exists \text{alice}.K' : \text{KEY}; \text{sam}_L.N_S' : \text{NON} \bullet \\ \text{alice}.K' = \text{sam}_L.N_S') \end{aligned}$$

7 Verifying Attack B

The CL-AtSe tool also successfully finds the type flaw attack (Attack B shown in Figure 6) Meadows et al. (2004) found using the NRL Protocol Analyzer, this time in the scenario where the intruder, Dan *D*, has discovered the key K_{AS} shared between Alice and Sam, and hence also the group key K and sequence number. Additionally, it assumes Alice may play the

part of both a member and key server of the same group. Firstly, the Group Key Pull protocol is run between member Alice and Dan, whom she believes is the key server, Sam. Then the Group Key Push protocol is run between Dan (posing as Alice in her key server role) and group member Bob.

Alice initiates a Group Key Pull protocol with the key server, Sam. However, Dan intercepts message 1 and impersonates Sam by sending message 2 to Alice using the concatenation of the next value of the sequence number Q^* , a security association SA^* , and a key K^* as the value for the nonce N_D . Following the protocol, Alice creates a signature from her nonce and the value she believes is Sam's nonce. Once again, this signature could be passed off as the signature used in the Group Key Push protocol, this time where Alice's nonce is interpreted as the message header. Dan can use this signature and the group key he already knows to create a Group Key Push message (step 5). Thus, Dan will have successfully tricked Alice into accepting his chosen key K^* for secure communication amongst the group. Formalisation of Dan's behaviour follows.



Operation *respond* is similar to the operation of the same name in Sam's role where Sam's nonce is replaced by the concatenation of the three decorated values chosen by Dan. In operation *pushKey*, Dan accepts Alice's request for the key and sends a Group Key Push message using the signature *sig* created by Alice in order to pose as her.

Once again, the required roles are coordinated to simulate the scenario in which the attack is found. Alice is playing the part of both a group member in the Group Key Pull protocol $alice_L : Alice_{PULL}$ and a server in the Group Key Push protocol $alice_S : Sam_{PUSH}$.

$$alice_L : Alice_{PULL} \wedge dan : Dan$$

$$bob : Alice_{PUSH} \wedge alice_S : Sam_{PUSH}$$

The initialisation predicate is specified below coordinating state variables belonging to each of the participating roles. Since the attack depends on Dan having discovered the key K_{AS} shared between Alice and Sam, we specify that $dan.K_{AS} = alice_L.K_{AS}$. Additionally, Dan knows the group for which Alice is establishing a key ($dan.G = alice_L.G$) and the key K used for the group ($dan.K = bob.K$). He also knows the current sequence number which means that the next value Q^* of the sequence number is the one

Bob is expecting in the Group Key Push protocol ($dan.Q^* = bob.Q$). Alice plays the role of the server in the Group Key Push protocol with Bob which means that her public key is the key Bob has for the server ($alice_S.K_A = bob.K_S$). It is also important to state that Alice's public key is the same in both of her roles ($alice_L.K_A = alice_S.K_A$).

<i>Init</i>
$dan.G = alice_L.G \wedge dan.K_{AS} = alice_L.K_{AS}$ $alice_L.K_A = alice_S.K_A \wedge dan.Q^* = bob.Q$ $dan.K = bob.K \wedge alice_L.K_A = bob.K_S$

We simulate the attack sequence by evaluating the sequential composition of the operations leading to the attack after initialisation.

$$attack_B \hat{=} Init \wedge alice_L.initiate \ ; \ dan.respond \ ;$$

$$alice_L.requestKey \ ; \ dan.pushKey \ ;$$

$$bob.getKey$$

Composition of the operations defining the attack trace $attack_B$ results in the following schema.

<i>attack_B</i>
$\Delta(alice_L.M, alice_L.N_A, alice_L.used, alice_L.N_S,$ $alice_L.SA, dan.M, dan.N_A, bob.M, bob.SA,$ $bob.K)$
$dan.G = alice_L.G \wedge dan.K_{AS} = alice_L.K_{AS}$ $dan.N_A' = alice_L.N_A' \wedge dan.Q^* = bob.Q$ $dan.K = bob.K \wedge alice_L.K_A = bob.K_S$ $alice_L.K_A = alice_S.K_A \wedge dan.M' = alice_L.M'$ $\{alice_L.M', alice_L.N_A'\} \cap alice_L.used = \emptyset$ $alice_L.used' = alice_L.used \cup \{alice_L.M', alice_L.N_A'\}$ $alice_L.N_S' = dan.Q^* \wedge dan.SA^* \wedge dan.K^*$ $alice_L.SA = dan.SA \wedge bob.M' = alice_L.N_A'$ $bob.SA' = dan.SA^* \wedge bob.K' = dan.K^*$

This time, insecurity of this protocol is proven by demonstrating that after the attack, the value Bob has for the new group key is not identical to the one distributed by Alice in her server role.

$$attack_B \Rightarrow alice_S.K^{*'} \neq bob.K'$$

From the equalities formed in $attack_B$, we know that Bob's value of the new key is the same as the value Dan has chosen for the new group key ($bob.K' = dan.K^*$). It is reasonable to assume that the value he chose is not equivalent to the value Alice had chosen for the new group key ($alice_S.K^* \neq dan.K^*$). With this additional assumption the theorem above is true. Hence, we have verified Attack B.

Again we notice two conditions in the attack schema that must be preserved by the presentation layer for Attack B to succeed, and again, negating these produces the following two preventative conditions, one of which must hold to secure the protocol against the attack.

$$\neg(\exists alice_L.N_S' : NON \bullet$$

$$dan.Q^* \wedge dan.SA^* \wedge dan.K^*)$$

$$\neg(\exists bob.M' : HDR; dan.N_A' : NON \bullet$$

$$bob.M' = dan.N_A')$$

In practice, the GDOI specification now contains an informal presentation layer requirement that each signed message must contain a single tag, identifying whether it belongs to the Group Key Pull or Group Key Push protocol (Meadows et al. 2004). This is formally specified in the following predicate ensuring the

1. $A \longrightarrow D(S) : M, \{h(N_A, G), N_A, G\}_{K_{AS}}$
2. $D(S) \longrightarrow A : M, \{h(N_A, N_D, SA), N_D, SA\}_{K_{AS}} (N_D = Q^*, SA^*, K^*)$
3. $A \longrightarrow D(S) : M, \{h(N_A, N_D, \{N_A, N_D\}_{K_A^{-1}}), \{N_A, N_D\}_{K_A^{-1}}\}_{K_{AS}}$
5. $D(A) \longrightarrow B : N_A, \{Q^*, SA^*, K^*, \{N_A, Q^*, SA^*, K^*\}_{K_A^{-1}}\}_K$

Figure 6: Attack B on the GDOI protocol.

content type of the two signatures is distinguishable for all possible combinations of items. We know from our formal proofs that this requirement successfully secures the protocol from the two type flaw attacks since it implies both of the derived requirements.

$$\begin{aligned}
&(\forall N_A, N_S : NON; M : HDR; Q : SEQ; \\
&SA : SEC; K : KEY \bullet \\
&N_A \wedge N_S \neq M \wedge Q \wedge SA \wedge K)
\end{aligned}$$

8 Conclusion

In this paper we provided a formal specification of the Group Domain of Interpretation protocol using Object-Z, in which both the application and presentation layers were present. We were required to specify presentation layer constraints on the data types, allowing agents to interpret messages correctly when they agree on the type structure of the message, but allowing multiple potential interpretations otherwise. Since the two models of differing detail are encapsulated within the one specification, there was no need for complex transformations from one to the other.

We combined the relative strengths of model checking and theorem proving to verify the specification and to secure it against two type flaw attacks. The application layer model was analysed using the AVISPA model checking tool to find the attacks. Subsequently, we verified the attacks against the presentation layer model using the Object-Z schema calculus, confirming the assumptions that allow the attacks and deriving the presentation layer requirements that prevent them. Furthermore, we confirmed that the approach taken by GDOI to distinguish between the content of signatures is sufficient to secure the protocol against both attacks, independent of other tagging schemes used.

Acknowledgements

We would like to thank Catherine Meadows for help in understanding the GDOI protocol and the anonymous referees for their comments. This research was supported in part by the Defence Signals Directorate and the Australian Research Council via Linkage-Projects Grant LP0347620, Formally-Based Security Evaluation Procedures.

References

Abadi, M. (2000), Security protocols and their properties, in ‘Foundations of Secure Computation’, NATO Science Series, IOS Press, pp. 39–60.

Armando et al. (2005), The AVISPA tool for the automated validation of internet security protocols and applications, in ‘Proceedings of the 17th International Conference on Computer-Aided Verification (CAV’05)’, Vol. 3576 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 281–285.

Baughner, M., Hardjono, T., Harney, H. & Weis, B. (2001), ‘Group domain of interpretation for ISAKMP’. Archived at <http://www.watersprings.org/pub/id/draft-irtf-smug-gdoi-01.txt>, January 2001.

Boyd, C. (1990), ‘Hidden assumptions in cryptographic protocols’, *IEE Proceedings, Part E* pp. 433–436.

Boyd, C. & Mao, W. (1993), On a limitation of BAN logic, in T. Helleseht, ed., ‘Advances in Cryptology (EUROCRYPT’93)’, Vol. 1055 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 240–246.

Bozzano, M. (2002), A Logic-Based Approach to Model Checking of Parameterized and Infinite-State Systems, PhD thesis, DISI, University of Genova. <http://www.disi.unige.it/person/BozzanoM/publications.html>.

Bozzano, M. & Delzanno, G. (2002), Automated protocol verification in linear logic, in ‘Proceedings of the Fourth ACM SIGPLAN Conference on Principles and Practice of Declarative Programming’, ACM Press, pp. 38–49.

Carlsen, U. (1993), Using logics to detect implementation-dependent flaws, in ‘Proceedings of the Ninth Annual Computer Security Applications Conference’, IEEE Computer Society Press, pp. 64–73.

Carlsen, U. (1994), Generating formal cryptographic protocol specifications, in ‘Proceedings of the 1994 IEEE Computer Society Symposium on Research in Security and Privacy’, IEEE Computer Society Press, pp. 137–146.

Cervesato, I. (2001), ‘Expressing type-flaw attacks in a strongly typed language’. Second Workshop on Foundations for Secure/Survivable Systems and Networks, Tokyo, Japan, 27 October 2001.

Dolev, D. & Yao, A. C. (1983), ‘On the security of public key protocols’, *IEEE Transactions on Information Theory* **29**(2), 198–208.

Donovan, B., Norris, P. & Lowe, G. (1999), Analyzing a library of security protocols using Casper and FDR, in ‘Proceedings of the Workshop on Formal Methods and Security Protocols’, Trento, Italy.

Duke, R. & Rose, G. (2000), *Formal Object-Oriented Specification Using Object-Z*, Cornerstones of Computing, Macmillan Press Limited, UK.

Gollmann, D. (2003), Analysing security protocols, in ‘Formal Aspects of Security’, Vol. 2629 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 71–80.

Gong, L. (1993), Variations on the themes of message freshness and replay — or the difficulty of devising formal methods to analyze cryptographic

- protocols, *in* 'Proceedings of the Computer Security Foundations Workshop VI', IEEE Computer Society Press, pp. 131–136.
- Heather, J., Lowe, G. & Schneider, S. (2000), How to prevent type flaw attacks on security protocols, *in* 'Proceedings of 13th IEEE Computer Security Foundations Workshop (CSFW'00)', IEEE Computer Society Press, pp. 32–43.
- Long, B. W. (2005), Formal verification of a type flaw attack on a security protocol using Object-Z, *in* '4th International Conference of B and Z Users, ZB 2005', Vol. 3455 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 319–333.
- Meadows, C. (2002), Identifying potential type confusion in authenticated messages, *in* 'Proceedings of Workshop on Foundation of Computer Security (FCS'02)', pp. 75–84. Published as a joint DIKU technical report, <http://www.diku.dk/>.
- Meadows, C. (2003), A procedure for verifying security against type confusion attacks, *in* 'Proceedings of 16th IEEE Computer Security Foundations Workshop', IEEE Computer Society Press, pp. 62–72.
- Meadows, C., Syverson, P. & Cervesato, I. (2004), 'Formal specification and analysis of the Group Domain of Interpretation Protocol using NPA-TRL and the NRL Protocol Analyzer', *Journal of Computer Security* **12**(6), 893–931.
- Millen, J. K. (1999), A necessarily parallel attack, *in* 'Proceedings of the Workshop on Formal Methods and Security Protocols', Trento, Italy. <http://www.cs.bell-labs.com/who/nch/fmsp99/program.html>.
- Potlapally, N. R., Ravi, S., Raghunathan, A. & Jha, N. K. (2003), Analyzing the energy consumption of security protocols, *in* 'Proceedings of the 2003 international symposium on Low power electronics and design', ACM Press, pp. 30–35.
- Ryan, P., Schneider, S., Goldsmith, M., Lowe, G. & Roscoe, B. (2000), *The Modelling and Analysis of Security Protocols: The CSP Approach*, Addison-Wesley.
- Snekkenes, E. (1992), Roles in cryptographic protocols, *in* 'Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy', IEEE Computer Society Press, pp. 105–119.
- Thayer Fábrega, F. J., Herzog, J. C. & Guttman, J. D. (1998), Strand spaces: Why is a security protocol correct?, *in* 'Proceedings of the 1998 IEEE Symposium on Security and Privacy', IEEE Computer Society Press, pp. 160–171.
- The Common Criteria Project Sponsoring Organizations (1999), *Common Criteria for Information Technology Security Evaluation*, 2.1 edn. ISO/IEC Standard 15408.