On the Automated Correction of Security Protocols Susceptible to a Replay Attack*

Juan C. Lopez P.[1], Raúl Monroy[1], and Dieter Hutter[2]

[1] Computer Science Department
Tecnológico de Monterrey, Campus Estado de México
Carretera al lago de Guadalupe, Km 3.5, Atizapán, 52926, Mexico
{juan.pimentel,raulm}@itesm.mx
[2] DFKI, Stuhlsatzenhausweg 3, D-66123 Saarbrücken, Germany
hutter@dfki.de

Abstract. Although there exist informal design guidelines and formal development support, security protocol development is time-consuming because design is error-prone. In this paper, we introduce Shrimp, a mechanism that aims to speed up the development cycle by adding automated aid for protocol diagnosis and repair. Shrimp relies on existing verification tools both to analyse an intermediate protocol and to compute attacks if the protocol is flawed. Then it analyses such attacks to pinpoint the source of the failure and synthesises appropriate patches, using Abadi and Needham's principles for protocol design. We have translated some of these principles into formal requirements on (sets of) protocol steps. For each requirement, there is a collection of rules that transform a set of protocol steps violating the requirement into a set conforming it. We have successfully tested our mechanism on 36 faulty protocols, getting a repair rate of 90%.

1 Introduction

Although there exists formal development support, as well as informal design guidelines, a lot of protocols, whether recent or not, are faulty. Further aid for protocol development is thus required. In this paper, we introduce Shrimp, a Smart metHod for Repairing IMperfect security Protocols. Shrimp aims at speeding up the formal software development cycle, bridging the gap between design and analysis by means of diagnosis and repair. It offers benefits to practising security engineers, including getting a better insight into a protocol flaw and enabling incremental protocol design. These features are all of interest because nowadays protocols are more complicated than just 3—5 steps (e.g. the SET protocol) and their various parts are intertwined, making it hard for a human to cope with all the subtle dependencies.

* This paper has largely benefited from numerous discussions with Alan Bundy and Graham Steel. It is based on research Monroy conducted while on sabbatical leave at Edinburgh University and DFKI, Saarbrücken. Monroy's stay in Germany was funded by DAAD A/06/32941 and CONACyT J110.382/06.

Shrimp relies on existing state-of-the-art tools both to analyse an (intermediate) protocol and, if the protocol is flawed, to find one or more of protocol runs violating a given security requirement, called an *attack*. It then analyses the protocol and the attack[1] to pinpoint the faulty steps of the protocol and synthesises appropriate changes to fix them. This yields an improved version of the protocol that should be analysed and potentially patched again until no further flaws can be detected.

To identify and patch a protocol flaw, we have translated some of the informal principles for the design of security protocols of Abadi and Needham [1] into formal requirements on sets of protocol steps. For each requirement, there is a collection of rules that transform a set of protocol steps violating the requirement into a set conforming it. The correction of security protocols incorporates the use of several of these rules. However, the patches are not independent and the application of a rule requires preconditions to be applicable and should guarantee postconditions once it has been applied. As a general framework to organise the application of such rules, we have adopted the proof planning methodology [4], developed to automate inductive theorem proving.

We have hitherto focused on automatically fixing protocols subject to a replay attack,[2] since many known faulty protocols fail to resist it.[3] This paper introduces two patching methods which, together with a generalisation of that presented in [10], almost deal with the full class of replay attacks proposed by Syverson [14] (the only exception being the type flaw subclass.[4]) We have successfully tested Shrimp on 36 protocols, 21 out of which were borrowed from the Clark and Jacob library, obtaining a repair rate of 90%. Since our approach to protocol repair requires an attack to reason about, to analyse a protocol and to get an attack whenever it was faulty, we used AVISPA, `http://www.avispa-project.org/`. So throughout this paper we shall refer to AVISPA's hierarchy of authentication.

The rest of this paper is organised as follows: §2 describes the types of flaws we want to automatically patch and describes the theoretical framework underlying our approach to protocol repair. Shrimp is presented in §3. We recapitulate the experimental results found throughout our investigation in §4 and discuss related work in §5. Conclusions and indications for further work appear in §6.

2 Fixing Faulty Security Protocols

2.1 Abadi and Needham's Principles

Abadi and Needham postulated 11 principles for the prudent design of security protocols [1] after noticing common features amongst protocols they found hard

[1] Our experiments show that it is not necessary to explicitly consider the property the protocol fails to satisfy; this might be attributable to that such a property is already implicit in the attack.

[2] A *replay attack* is a form of attack where a data transmission is repeated or delayed.

[3] Most of the attacks reported in the Clark-Jacob library [7] are of type replay.

[4] A *type flaw attack* is an attack where a participant confuses a (field of a) message containing data of one type with a message data of another.

to analyse. We use some of these principles to diagnose the cause of an attack. In particular, SHRIMP takes care of replay attacks where the message being reused is a cypher-text, dealing with the following protocol flaws:

1. Two or more different cypher-texts of the same protocol cannot be distinguished from one another. This flaw violates principle 10, *recognising messages and encodings*, which prescribes being careful about the format of a message: principals should be able to associate which step or which run a message corresponds to, regardless of whatever protocol they are running;
2. The originator/recipients of a cypher-text in one message of the protocol cannot be distinguished. This flaw violates principle 3, *agent naming*, which prescribes that the agent names relevant for a message should all be derivable either from the format of a message or from its content; and
3. Two or more different runs of the same protocol cannot be distinguished from one another. Upon reception, a participant cannot separate which run the message belongs to. This flaw violates principle 10, since the message cannot be bound to a particular run of the protocol. It also violates principles 6—8, since the protocol does not guarantee association or temporal succession.

2.2 Strand Spaces

In order to reason about non-trivial messages and their intended role in a protocol, SHRIMP uses a formalisation of individual message notations. Most this formalisation has already been developed for protocol verification (e.g. strand spaces [15] or Paulson's logic [12]). SHRIMP's constructs might be accommodated within any logic. Here we choose strand spaces, because the method for fixing a protocol without a proper message encoding can be theoretically justified using authentication tests [9]. In the sequel, we assume knowledge of strands spaces, though notation and authentication tests are recalled below.

Messages, ranged over by $M_1, M_2, \ldots$, are also called terms. The set of terms, A, is freely generated from two disjoint sets, the set of texts (T) and the set of keys (K), by means of concatenation, $M_1; M_2$, and encryption, $\{\!|M|\!\}_K$ ($K \in \mathsf{K}$). T contains nonces, $N_a, N_b, \ldots$, timestamps, $T_a, T_b, \ldots$, agent names, $A, B, \ldots$, and tags, $\ell_a, \ell_b, \ldots$. There are two functions, one maps principals, $A, B, \ldots$, to their public keys, $K_a^+, K_b^+, \ldots$, and the other a pair of principals, $\langle A, B\rangle$, to their symmetric shared key, K_{ab}. K comes with an inverse operator mapping each member of a key pair for an asymmetric cryptosystem to the other, $(K_a^+)^{-1} = K_a^-$, and each symmetric key to itself, $(K_{ab})^{-1} = K_{ab}$. Let Safe denote the set of keys that are safe and Safe_a denote the set of keys known by a regular, non-compromised agent A.

The subterm relation, $\sqsubseteq$, is the least relation such that $M \sqsubseteq M$, $M \sqsubseteq \{\!|M_1|\!\}_K$ if $M \sqsubseteq M_1$, and $M \sqsubseteq M_1; M_2$ if $M \sqsubseteq M_1$ or $M \sqsubseteq M_2$. Notice that, for any $K \in \mathsf{K}$, $K \sqsubseteq \{\!|M|\!\}_K$ only if $K \sqsubseteq M$. A message is atomic if it is not an encrypted term or a concatenated one. A message M_0 is a component of M if $M_0 \sqsubseteq M$, M_0 is not a concatenated term, and for every $M_1 \neq M_0$ such that $M_0 \sqsubseteq M_1 \sqsubseteq M$ implies that M_1 is a concatenated term.

A *strand* is a sequence of nodes, each denotes a communicating event, where transmission (respectively reception) of a term M is denoted as $+M$ (respectively $-M$). So a node is either positive or negative. Let s be a strand and let $\langle s, i\rangle$ and $\langle s, i+1\rangle$ denote the i-th and the $i+1$-th nodes of s. Then $\langle s, i\rangle \Rightarrow \langle s, i+1\rangle$. $\Rightarrow^+$ and $\Rightarrow^*$ are used to respectively denote the transitive and the transitive-reflexive closure of $\Rightarrow$. $n \rightarrow n'$ denotes inter-strand communication; it requires the nodes to be complementary one another, in terms of polarity, $\mathsf{sign}(n) = + \neq \mathsf{sign}(n')$, and matching, in terms of the message being exchanged, $\mathsf{msg}(n) = \mathsf{msg}(n')$. A *strand space* Σ is a set of strands, where $\Rightarrow$ and $\rightarrow$ impose a graph structure on the nodes of Σ. Each strand represents a protocol run from the local perspective of a participant. If the participant is honest, the strand, as well as the strand nodes, is said to be regular and penetrator otherwise. A term M *originates* at a node n if $\mathsf{sign}(n) = +$; $M \sqsubseteq \mathsf{msg}(n)$; and $M \not\sqsubseteq \mathsf{msg}(n')$, for every $n' \Rightarrow^+ n$. M is said to be *uniquely originating* if it originates on only one node in the strand space. unique_s is the set of terms uniquely generated at strand s.

A finite, acyclic graph, $\mathcal{B} = \langle \mathcal{N}, (\rightarrow \cup \Rightarrow)\rangle$, is a *bundle* if for every $n_2 \in \mathcal{N}$, i) if $\mathsf{sign}(n_2) = -$, then there is a unique $n_1 \in \mathcal{N}$ with $n_1 \rightarrow n_2$; and ii) if $n_1 \Rightarrow n_2$ then $n_1 \in \mathcal{N}$ and $n_1 = \langle s, i\rangle$ and $n_2 = \langle s, i+1\rangle$. Let $\mathcal{B}$ be a bundle, then $\prec_{\mathcal{B}}$ and $\preceq_{\mathcal{B}}$ denote respectively the transitive and the transitive-reflexive closure of $(\rightarrow \cup \Rightarrow)$.

For brevity, protocols will be specified by a sequence of steps, each of the form $q.\ A \rightarrow B : M$, meaning that, at step q, agent A sends message M to agent B, which B receives. Similarly, an attack will be given as a sequence of steps, each affixed with their session: $s : q.\ A \rightarrow B$ stands for the q-th step of the s-th run of a protocol. We find it convenient to respectively use S and Spy to refer to the server and the penetrator. So $A \rightarrow \mathsf{Spy}(B) : M$ and $\mathsf{Spy}(A) \rightarrow B$ respectively denote interception of M and impersonation of A. We will use K_a as an abbreviation for $K_{a\mathsf{s}}$.

Suppose that A is a participant in a protocol. Suppose that at node n_0 A creates a new term N, builds $M = \{\!|M'|\!\}_K$, such that $N \sqsubseteq M$ and M is a component of $\mathsf{msg}(n_0)$, and then transmits $\mathsf{msg}(n_0)$. Suppose that N is uniquely generated, that M is not a subterm of a component of any regular node in the protocol and that the decryption key is safe, $K^{-1} \in \mathsf{Safe}$. If N is later received, at node n_1, outside the form $\{\!|M'|\!\}_K$, then only a honest participant, not the penetrator, must have been responsible for N to have gone out of this form. The edge $n_0 \Rightarrow^+ n_1$ is an *outgoing test for* N *in* M. If, instead, N is sent possibly in clear and it later is received in encrypted form $\{\!|\ldots; N; \ldots|\!\}_{K'}$, where $K' \in \mathsf{safe}$, then only a honest participant, not the penetrator, must have been responsible for N to have entered to this form. The edge $n_0 \Rightarrow^+ n_1$ is an *incoming test for* N *in* $\{\!|\ldots; N; \ldots|\!\}_{K'}$.

2.3 Shrimp's Meta-logic

Shrimp analyses the encoding of a message to determine when a cypher-text may be used to arm a replay and how it should be changed to prevent a replay. In the following we introduce a notion of similarity of messages and how messages can be modified to break a given similarity. Let $\Box$ be a special symbol that is not an atomic message and S be a set of keys. The *pattern* [2] p of a message M visible wrt. S is given by:

$$\begin{array}{ll} p_S(M) = \{\!|p_S(M)|\!\}_K \text{ if } M = \{\!|M|\!\}_K \wedge K^{-1} \in S & p_S(M) = M \text{ if } M \text{ is atomic} \\ p_S(M) = p_S(M_1, S); p_S(M_2) \text{ if } M = M_1; M_2 & p_S(M) = \Box \text{ otherwise} \end{array}$$

Definition 1 (Similarity). *Two messages M and M' are* similar *wrt. S, written $M \sim_S M'$, iff there is a bijective replacement α on patterns (with $\Box\alpha = \Box$), mapping symbols from M to M', such that $p_S(M)\alpha = p_{S\alpha}(M')$ holds.*

Definition 2 (Visible Content). *The* visible content *$ct_S(M)$ of a message M wrt. a set of keys S is given by:*

$$\begin{array}{ll} ct_S(M) = \{M\} \quad \textit{if } M \textit{ is atomic} & ct_S(M_1; M_2) = ct_S(M_1) \cup ct_S(M_2) \\ ct_S(\{\!|M|\!\}_K) = ct_S(M) \quad \textit{if } K^{-1} \in S & ct_S(M) = \emptyset \quad \textit{if } K^{-1} \notin S \end{array}$$

Two messages M and M' are equivalent under rearrangement*, $M \equiv M'$ for short, iff $ct_S(M) = ct_S(M')$ for all sets of keys S.*

Shrimp also includes symbols that allow us to compute the name of the agents involved in the exchange of a message and to reason about the name of the agents that can be inferred from the encoding of a message. Given a bundle $\mathcal{B}$, A is the *originator* (respectively *recipient*) of a term M, $A \rhd_{\mathcal{B}} M$, (respectively $A \lhd_{\mathcal{B}} M$) iff there is a positive (respectively negative) node n in a strand played by A of $\mathcal{B}$ such that M originates at n (respectively $M \in ct_S(\mathsf{msg}(n))$, S being the keys known to A.)

Definition 3 (Correspondents). *Let $\mathcal{B}$ be a bundle. The participants involved in the exchange of M in $\mathcal{B}$, called the* correspondents of M, Partners$([M])_{\mathcal{B}}$*, is given by $\{A : A \rhd_{\mathcal{B}} M \vee A \lhd_{\mathcal{B}} M\}$.*

Definition 4. *The names of the agents that can be (safely) deduced from the encoding of a message is given by:*

$$\begin{array}{rll} \text{Agents}(\{\!|M|\!\}_K) &= \{A\} \cup \text{Agents}(M) & \text{if } K = K_a^+ \vee K = K_a^- \\ \text{Agents}(\{\!|M|\!\}_{K_{ab}}) &= \text{Agents}(M) & \\ \text{Agents}(M_1; M_2) &= \text{Agents}(M_1) \cup \text{Agents}(M_2) & \\ \text{Agents}(M) &= \{M\} & \text{if } M \text{ is a name} \\ \text{Agents}(M) &= \{\,\} & \text{otherwise} \end{array}$$

Notice that we do not infer names from symmetric encryption, because the flow of the message cannot be determined.

Shrimp finds bugs in protocols by analysing a bundle containing a penetrator strand. Often this analysis suggests a change in the structure of a message and in that case Shrimp identifies the node originating such message considering a bundle denoting an intended protocol run. With this, we compute the changes to be done in one such a regular bundle, which we then propagate to the protocol description. The following definitions aim at a meta-theory to allow for tracing the consequences of changing all the nodes in the set of strands depending on changes to a particular node.

Let $\mathrm{Pos}(M)$ be the set of all positions π in M. Elements in Pos are sequences and $\cdot$ denotes sequence concatenation. Two positions π, π' are independent if there is no π'' that either $\pi = \pi' \cdot \pi''$ or $\pi' = \pi \cdot \pi''$ holds. $M_{|\pi}$ is the submessage of M at position π and $M[\pi \leftarrow M']$ is the message obtained by replacing $M_{|\pi}$ in M by M'.

Definition 5 (Explicit Replacement). *A set* $\zeta = \{(\pi_1, M_1), \dots, (\pi_n, M_n)\}$ *is an* explicit replacement *iff for all* $1 \leq i \leq n : \pi_i \in \mathrm{Pos}$ *and* M_i *are messages and for all* $i \neq j$*:* π_i *and* π_j *are independent.* $\mathcal{ER}$ *denotes the set of all explicit replacements.*

$\zeta = \{(\pi_1, M_1), \dots, (\pi_n, M_n)\} \in \mathcal{ER}$ *is an* explicit replacement for a message M *iff* $\pi_i \in \mathrm{Pos}(M)$ *for all* $1 \leq i \leq n$*. The application of* ζ *on a message* M *is given by* $\zeta(M) = M[\pi_1 \leftarrow M_1, \dots, \pi_n \leftarrow M_n]$*.*

Changing a message that is sent from A to B changes the knowledge of B and thus its abilities to construct consecutive messages. If B should forward an encrypted messages coming from A and this latter message is changed in the protocol, then we also have to change the message forwarded by B. Hence we need a mapping between an old encrypted (sub-)message and the corresponding new message:

Definition 6 (P-Assignment). *A position* π *of a message* M *is* protected *wrt. a set of keys* S *iff* $M_{|\pi}$ *has the form* $\{\!|M'|\!\}_K$ *and* $K^{-1} \notin S$*. A protected position is* minimal *iff all smaller positions are not protected wrt.* S*.* $Prot_S(M)$ *denotes the set of all submessages of* M *at minimal protected positions of* M *wrt.* S*.*

Two messages M *and* M' *are* p-assigned *wrt.* S *iff for each* $\{\!|N|\!\}_K \in Prot_S(M)$ *there is a unique message* N' *with* $\{\!|N'|\!\}_K \in Prot_S(M')$*. The* p-assignment $\Delta_{M,M',S}$ *of* M *and* M' *is the set of pairs* $(\{\!|N|\!\}_K, \{\!|N'|\!\}_K)$ *of corresponding terms at minimal protected positions of* M *and* M'*.*

In many cases, a protocol change will simply enrich the messages exchanged by the principals with additional information. We capture this property as follows:

Definition 7 (Monotonicity). *Let* M, M' *be messages and let* $\mathsf{A}_0 \subseteq \mathsf{A}$*.* $M \leq_{\mathsf{A}_0} M'$ *iff* $ct_S(M) \subseteq ct_S(M')$ *and* $ct_S(M') \subseteq ct_S(M) \cup ct_S(\mathsf{A}_0)$ *for all* $S \subseteq \mathsf{K}$*.*

An explicit replacement ζ *is* monotone *for* M *wrt.* A_0 *iff* $M \leq_{\mathsf{A}_0} \zeta(M)$ *holds.* ζ *is* strongly *monotone iff for all* $S \subseteq \mathsf{K}$ *and all* $(\{\!|N|\!\}_K, \{\!|N'|\!\}_K) \in \Delta_{M,M',S}$*:* $N \leq_{\mathsf{A}_0} N'$ *holds.*

Definition 8 (Collision Freeness). *Let* S *be a set of keys,* M *a message and let* $\mathsf{A}_0 \subseteq \mathsf{A}$ *be a set of messages.* M *is* collision free *to* A_0 *iff* $\forall M' \in ct_S(M).\ \forall M'' \in \mathsf{A}_0.\ (M' \not\sim_S M'')$*.*

Given a set of keys S, a set $\Delta_{M,M',S} = \{(M_1, N_1), \dots (M_n, N_n)\}$ denotes also a term replacement function which maps messages to messages. Given a message

N, we obtain $\Delta_{M,M',S}(N)$ by replacing each occurrence of M_i in a minimal protected position of N by N_i.

Definition 9 (Admissibility). *An explicit replacement ζ for M is* admissible *for M wrt. a set of keys S iff M and $\zeta(M)$ are p-assigned. Let ζ be admissible for M then $\Delta_{M,\zeta(M),S}$ is the corresponding term replacement function of ζ wrt. M and S.*

Given a term replacement function Δ, a set of keys S and a node n with $M = \mathsf{msg}(n)$ then $\Delta^{n,S} = \{(\pi_1, \Delta(M_{|\pi_1})), \ldots, (\pi_n, \Delta(M_{|\pi_n}))\}$ with $\{\pi_1, \ldots \pi_n\}$ being all minimal protected positions π of M wrt. S with $\Delta(M_{|\pi}) \neq M_{|\pi}$.

Definition 10 (Representativeness). *A bundle $\mathcal{B} = \langle \mathcal{N}_\mathcal{B}, (\rightarrow_\mathcal{B} \cup \Rightarrow_\mathcal{B})\rangle$ over a protocol is* representative *iff it is regular and for every node $\langle s, i\rangle \in \mathcal{B}$ and $s = r\alpha_s$ the replacement application operator α_s is injective and $r\alpha \subseteq \mathcal{N}_\mathcal{B}$.*

Definition 11 (Adaption). *Let $\mathcal{B} = \langle \mathcal{N}_\mathcal{B}, (\rightarrow_\mathcal{B} \cup \Rightarrow_\mathcal{B})\rangle$ be a representive bundle over a protocol. The* adaption *ζ of $\mathcal{B}$ for an explicit replacement ζ_0 in a positive node $n_0 \in \mathcal{N}_\mathcal{B}$ maps each node of the bundle to an explicit replacement and is defined as follows:*

$$\begin{array}{llll} \zeta[n] = \emptyset & \textit{if } n \preceq_\mathcal{B} n_0 \textit{ and } n \neq n_0 & \zeta[n] = \zeta_0 & \textit{if } n = n_0 \\ \zeta[n] = \zeta[n'] & \textit{if } n' \rightarrow n & \zeta[n] = \Delta^{n,S} & \textit{if } n_0 \prec_\mathcal{B} n \textit{ and } n \textit{ is positive} \end{array}$$

where $\Delta = \bigcup_{n' \in \boldsymbol{N}} \Delta_{\mathsf{msg}(n'),\zeta[n'](\mathsf{msg}(n')),S}$ with $\boldsymbol{N}$ being the set of all negative nodes n' with $n' \Rightarrow n$ and S is the set of keys known by the principal playing the role associated with the strand where n lies, at the moment of sending $\mathsf{msg}(n)$. ζ is admissible on $\mathcal{B}$ iff, for all $n \in \mathcal{B}$, $\zeta[n]$ is admissible on $\mathsf{msg}(n)$.

Given a representative bundle and an injective function α_s mapping a strand s of the protocol to nodes of the bundle we can easily use the adaption of these nodes in order to change the strand s itself by applying $\zeta[\langle s, i\rangle]\alpha_s^{-1}$ to each node $\langle s, i\rangle$ of s. Given an adaption ζ on a representive bundle for P we write $\zeta(P)$ to denote the protocol that resuls after the application of $\zeta[\langle s, i\rangle]\alpha_s^{-1}$.

Using this procedure, Shrimp is able to modify a protocol description, starting from the node originating the message that suggested the enhancement to all the successor nodes where the changes endure. This yields a path, we call a *change enduring path*. A change enduring path is guaranteed to be finite and acyclic both because bundles are also finite and acyclic and because changes are propagated considering only inter-strand transitions.

Proposition 1 (Monotonicity of Bundles). *Let $\mathcal{B} = \langle \mathcal{N}_\mathcal{B}, (\rightarrow_\mathcal{B} \cup \Rightarrow_\mathcal{B})\rangle$ be a representive bundle over a protocol and ζ be an adaption for an explicit replacement ζ_0 in a positive node $n_0 \in \mathcal{N}_\mathcal{B}$. If ζ_0 is strongly monotone for $\mathsf{msg}(n_0)$ wrt. a set of messages A_0 then $\zeta[n]$ is strongly monotone for $\mathsf{msg}(n)$ wrt. A_0 for all $n \in \mathcal{N}_\mathcal{B}$.*

Proof (Outline). The proof is done by induction on the length of $\prec_\mathcal{B}$. As one base case we assume $n \prec_\mathcal{B} n_0$. Thus $\zeta[n] = \emptyset$ and the proposition holds trivially.

As another base case consider $n_0 = n$ then again the proposition holds trivially due to the given assumptions on ζ_0. Suppose n is negative and $\exists n'.\ n' \rightarrow n$. As an induction hypothesis we assume that $\zeta[n']$ is strongly monotone for $\mathsf{msg}(n')$ and S wrt. A. Since $\mathsf{msg}(n') = \mathsf{msg}(n)$ we conclude that $\zeta[n] = \zeta[n']$ is strongly monotone for $\mathsf{msg}(n)$ and S wrt. A. Now suppose n is positive and different from n_0. As an induction hypothesis we assume that $\zeta[n']$ is strongly monotone for all negative nodes n' with $n' \Rightarrow^+ n$. Thus, $N \leq_{\mathsf{A}_0} N'$ for all $(\{\!|N|\!\}_K, \{\!|N'|\!\}_K) \in \Delta_{\mathsf{msg}(n'),\zeta[n'](\mathsf{msg}(n')),S}$ and arbitrary $S \in \mathsf{K}$. Hence, $N \leq_{\mathsf{A}_0} N'$ for all $(\{\!|N|\!\}_K, \{\!|N'|\!\}_K) \in \Delta$ holds and $\zeta[n] = \Delta^{n,S}$ is strongly monotone wrt. A_0. $\square$

2.4 Patch Planning Faulty Security Protocols

A *patch method* is a 5-tuple (name, input, preconditions, patch, effect). The first component is the *name* of the method. The second component is the *input*, which is often the description of a faulty protocol P, a bundle $\mathcal{B}_A$ describing the attack, and a representive bundle $\mathcal{B}_R$ describing the intended run of the protocol. The third component is the *preconditions*, a formula written in a meta-logic that the input objects must satisfy. Shrimp uses these preconditions to predict whether the associated patch will make the protocol no longer susceptible to the attack. The fourth component is the *patch*, a procedure specifying how to mend the input protocol. Finally, the fifth component is the *effects*, a formula specifying required properties of the newer version of the protocol.

Methods can be compound by invoking other methods using *methodicals* (functions that link methods together to control search). A *compound method* is a 4-tuple (name, input, preconditions, method). It involves the name of the compound method, the input, the preconditions, and then the method build from methodicals, mostly in our case *orelse_meth*. *orelse_meth* $meth_1$ $meth_2$ tries $meth_1$ and if that fails tries $meth_2$.

3 Fixing Faulty Protocols Subject to a Replay Attack

The chief method of Shrimp is the *replay* compound method, see Fig. 1. It invokes three sub-methods: *message_encoding*, *agent_naming* and *session_binding*. The order in which methods are attempted is important as it imposes a hierarchy in terms of the complexity of implementing the patch. *message_encoding* is more viable because when modifying a protocol message it may not introduce additional components. *agent_naming* is more viable than *session_binding* because it only modifies protocol messages. By contrast, *session_binding* involves the insertion of protocol steps.

3.1 Patching Protocols Violating Principle 10

The *message_encoding* method, see Fig. 2, repairs a faulty protocol that portrays two or more cypher-texts that are different one another but have similar structure and so violates principle 10. The adversary may exploit this vulnerability by

```
Name: replay
Input: $P \in \Sigma$, $\mathcal{B}_A$, $\mathcal{B}_R$
Preconditions:
    % Spy reuses cypher-text $\{\!|M|\!\}_K$:
    $\exists i, j, k, M, K.$
      $\langle s_r, i\rangle \preceq_{\mathcal{B}} \langle s_p, j\rangle \prec_{\mathcal{B}} \langle s_q, k\rangle \wedge \{\!|M|\!\}_K \sqsubseteq \mathsf{msg}(\langle s_r, i\rangle) \wedge \{\!|M|\!\}_K \sqsubseteq \mathsf{msg}(\langle s_p, j\rangle)$

    % $s_p$ is the penetrator while $s_q$ the principal being deceived
    $\wedge \langle s_r, i\rangle$ and $\langle s_q, k\rangle$ are regular but $\langle s_p, j\rangle$ is not
    $\wedge\, \mathsf{sign}(\langle s_r, i\rangle) = + = \mathsf{sign}(\langle s_p, j\rangle)$ but $\mathsf{sign}(\langle s_q, k\rangle) = -$
Method:
    orelse_meth message_encoding($P, \mathcal{B}_A, \mathcal{B}_R, M, K, \langle s_q, k\rangle$)
    orelse_meth agent_naming($P, \mathcal{B}_A, \mathcal{B}_R, M, K, \langle s_q, k\rangle$)
                session_binding($P, \mathcal{B}_A, \mathcal{B}_R, M, K, \langle s_q, k\rangle$)
```

Fig. 1. The *replay* compound method

making one cypher-text play the role of the other. An example faulty protocol with this vulnerability is Wide-Mouth Frog (WMF): (part of) the initiator's first message can be reused to mimic the result of another request the server has acted upon. AVISPA proves WMF fails to guarantee weak authentication of B to A, yielding:

WMF	Attack	
1. $A \to \mathsf{S} : A; \{\!|B; T_a; K_{ab}|\!\}_{K_a}$	1:1. $A \to \mathsf{Spy(S)}$	$: A; \{\!|B; T_a; K_{ab}|\!\}_{K_a}$
2. $\mathsf{S} \to B : \{\!|A; T_a{+}d; K_{ab}|\!\}_{K_b}$	2:2. $\mathsf{Spy(S)} \to A$	$: \{\!|B; T_a; K_{ab}|\!\}_{K_a}$

To remove the protocol flaw, it suffices to break this similarity. When input WMF and the attack above, *message_encoding* successfully repairs it returning:[5]

$$1.\ A \to \mathsf{S} : A; \{\!|\ \boxed{T_a; B}\ ; K_{ab}|\!\}_{K_a} \qquad 2.\ \mathsf{S} \to B : \{\!|A; T_a + d; K_{ab}|\!\}_{K_b} \tag{1}$$

Proposition 2. *Let ζ be the adaption of the representive bundle $\mathcal{B}_R$ as given in Fig. 2. and $P' = \zeta(P)$ be the corresponding revised protocol. Then $\zeta(\{\!|M|\!\}_K)$ cannot be used to arm a message encoding replay attack on P'.*

Proof (outline). $\zeta(\{\!|M|\!\}_K)$ is not similar to any other component. Then, only the execution of a specific step in P' may cause $\zeta(\{\!|M|\!\}_K)$ to appear on the traffic, if ever. If P satisfies security property ϕ then so will P', because extra elements in $\zeta(\{\!|M|\!\}_K)$, if any, are all innocuous tags. □

3.2 Patching Protocols Violating Principles 6—10

The *session_binding* method deals with faulty protocols that contain a message which cannot be associated with a particular protocol run. An attack exploiting this flaw, called *replay protection*, causes an agent to consider that another is

[5] Protocol changes are enclosed within a solid box to ease reference.

Name: *message_encoding*
Input: $P \in \Sigma$, $\mathcal{B}_A$, $\mathcal{B}_R$, $n \in \mathcal{B}_A \cap \mathcal{B}_R, \pi \in Pos(\mathsf{msg}(n))$
    `%` n `lies on strand of deceived agent where` $\mathsf{msg}(n)_{|\pi} = \{|M|\}_K$ `is the`
    `% message used to elaborate replay`
Preconditions:
    `% different cypher-texts cannot be distinguished:`
    Let $S = \mathsf{Safe}_r$
        $\exists n' \in \mathcal{B}_A, \exists M', K'$.
            $(\{|M'|\}_{K'} \in \mathsf{msg}(n') \wedge \{|M'|\}_{K'} \sim_S \{|M|\}_K \wedge \{|M'|\}_{K'} \neq \{|M|\}_K)$
Patch:
    `% Break similarity of` $\{|M|\}_K$`:`
    select M'' such that $M \leq_{\mathsf{A}_0} M''$, A_0 is a minimal set of tags, and
    $\{|M''|\}_K$ is collision free with $L = \{\{|M'|\}_{K'} : \{|M'|\}_{K'} \sqsubseteq \mathsf{msg}(n), n \in \mathcal{B}_R\}$
    With $\zeta_0 = \{(\pi, \{|M''|\}_K)\}$ and ζ be the adaption of $\mathcal{B}$ for ζ_0 in n
    let $P' = \zeta(P)$.

Fig. 2. The *message_encoding* method

trying to set up a simultaneous session, when he is not [11]. Two example protocols subject to this type of attack, because none satisfies *strong authentication* of B to A, are (1) and the Denning-Sacco Shared Key (DSSK) protocol. The DSSK protocol and the attack that AVISPA finds are as follows:

DSSK		Attack
1. $A \rightarrow \mathsf{S} : A; B$	1:1.	$A \rightarrow \mathsf{S} : A; B$
2. $\mathsf{S} \rightarrow A : \{|B; K_{ab}; T_s; \{|B; K_{ab}; A; T_s|\}_{K_b}|\}_{K_a}$	1:2.	$\mathsf{S} \rightarrow A : \{|B; K_{ab}; T_s; \{|B; K_{ab}; A; T_s|\}_{K_b}|\}_{K_a}$
3. $A \rightarrow B : \{|B; K_{ab}; A; T_s|\}_{K_b}$	1:3.	$A \rightarrow B : \{|B; K_{ab}; A; T_s|\}_{K_b}$
	2:3.	$\mathsf{Spy}(A) \rightarrow B : \{|B; K_{ab}; A; T_s|\}_{K_b}$

Notice that both the DSSK and the WMF protocol prescribe the responder, B, to react upon an *unsolicited test* [9].

Shrimp is equipped with a repair method that introduces a nonce-flow requirement to fix this flaw [1,13,11,9] (c.f. principle 7.) This requirement is realised by transforming the unsolicited test into an authentication one. Let $A \xrightarrow{\boxed{M}} B$ denote the step at which the replay is realised and let $A \xrightarrow{M_1} B \xrightarrow{M_2} C$ abbreviate two consecutive protocol steps: $q.\ A \rightarrow B :\ M_1$ and $q+1.\ B \rightarrow C :\ M_2$. Then, the nonce-flow requirement is introduced via the following transformation rules, called $\mathsf{nonce\_flow}$ and tried to be applied in the order of appearance:

$$A \xrightarrow{M_1} B \xrightarrow{\boxed{M_2}} C \rightsquigarrow A \xrightarrow{M_1} \boxed{C} \xrightarrow{M_1 \boxed{; \{|C; N_c, h(M_1)|\}_{K_b^+}}} B \xrightarrow{M_2 \boxed{; \{|C; N_c; h(M_2)|\}_{K_b^-}}} C$$

$$B \xrightarrow{\boxed{M_1}} C \rightsquigarrow B \xrightarrow{M_1} C \xrightarrow{\boxed{\{|C; N_c; h(M_1)|\}_{K_b^+}}} \boxed{B} \xrightarrow{\boxed{\{|C; N_c|\}_{K_b^-}}} \boxed{C}$$

where $h(M)$ denotes a one-way, collision-resistant hash function, which is used to tie each test component to the current run. Notice that if $A = C$ then the first two steps of the right-hand side of the first rule merge. Also notice that if $A \xrightarrow{M} B \rightsquigarrow A \xrightarrow{M'} \boxed{C} \xrightarrow{M''} B$, the strands ought to be modified as follows: i) for A and B, $\exists n_a \in s_a, n_b \in s_b.\ \mathsf{msg}(n_a) = \mathsf{msg}(n_b) = M$, so we shall have $\mathsf{msg}(n_a) = M'$ and $\mathsf{msg}(n_b) = M''$; and ii) for C, two nodes, n_{c_1} and n_{c_2}, are inserted such that $\mathsf{msg}(n_{c_1}) = M'$ and $\mathsf{msg}(n_{c_2}) = M''$ with $\mathsf{sign}(n_{c_1}) = - \neq \mathsf{sign}(n_{c_2})$. The rule identifies where these nodes are to be inserted: c.f. C's participation, previous to the attack. Any other transformation is handled similarly.

We now introduce two refinements to this transformation. First, notice that applying $\mathsf{nonce\_flow}$ without considering the structure of M_1 or M_2 may add unnecessary components. We shrink the message $M_1; \{\!|C; N_c; h(M_1)|\!\}_{K_b^+}$ (respectively $M_2; \{\!|C; N_c; h(M_2)|\!\}_{K_b^-}$) as follows:

$$\begin{array}{ll} \mathrm{shrnk}(M, M') = \mathrm{shrnk}_0(M, M') & \text{if } \mathrm{shrnk}_0(M, M') \neq M \\ \mathrm{shrnk}(M, M') = M; M' & \text{otherwise} \end{array}$$

where

$$\begin{aligned} \mathrm{shrnk}_0(A, M) &= A \qquad\qquad \text{if } A \text{ is atomic} \\ \mathrm{shrnk}_0(M_1; M_2, M') &= \mathrm{shrnk}_0(M_1, M'); \mathrm{shrnk}_0(M_2, M') \\ \mathrm{shrnk}_0(\{\!|M|\!\}_K, \{\!|C; N_c; M'|\!\}_K) &= \{\!|C; N_c; M|\!\}_K \\ \mathrm{shrnk}_0(\{\!|M|\!\}_{K_c^-}, \{\!|C; N_c; M'|\!\}_{K_b^+}) &= \{\!|\mathrm{shrnk}_0(M, \{\!|C; N_c; h(M)|\!\}_{K_b^+})|\!\}_{K_c^-} \\ \mathrm{shrnk}_0(\{\!|M|\!\}_{K_c^+}, \{\!|C; N_c; M'|\!\}_{K_b^-}) &= \{\!|\mathrm{shrnk}_0(M, \{\!|C; N_c; h(M)|\!\}_{K_b^-})|\!\}_{K_c^+} \end{aligned}$$

where we assume that C originates the message $M_1; \{\!|C; N_c; h(M_1)|\!\}_{K_b^+}$ (respectively $M_2; \{\!|C; N_c; h(M_2)|\!\}_{K_b^-}$), the challenger, and B is the recipient, the champion. Second, notice that applying $\mathsf{nonce\_flow}$ when the server is involved in the replay may yield a clumsy protocol. This is because the protocol would involve too many server participations and provide guarantees to the server rather than to the participants. We get around this situation by applying a very specific patching strategy, due to Lowe [11], which consists of making the participants handshake. The handshake messages are cyphered using the session key, similar to a key confirmation step. $\mathsf{nonce\_flow}$ is thus attempted only if the following rules, called $\mathsf{handshake}$, are *not* applicable:

$$\mathsf{S} \xrightarrow{\boxed{M(K)}} X_i \rightsquigarrow \mathsf{S} \xrightarrow{M(K)\boxed{;T}} X_i \xrightarrow{\boxed{\{\!|X_i; X_j; N_j|\!\}_K}} X_j \xrightarrow{\boxed{\{\!|X_j; N_j|\!\}_K}} X_i$$

$$X_k \xrightarrow{\boxed{M'}} \mathsf{S} \xrightarrow{M(K)} X_i \rightsquigarrow X_k \xrightarrow{M'} \mathsf{S} \xrightarrow{M(K)\boxed{;T}} X_i \xrightarrow{\boxed{\{\!|X_i; X_j; N_j|\!\}_K}} X_j \xrightarrow{\boxed{\{\!|X_j; N_j|\!\}_K}} X_i$$

where $T = \{\!|T'; h(M(K))|\!\}_{K_i}$ and where the last two steps of the protocol structure on the right-hand side are applied for all $j \neq i$ and so are actually rounds of messages. Notice that for $\mathsf{handshake}$ to be applicable the message M should carry a session key. This patching strategy is known to be susceptible to a known-key

```
Name: session_binding
Input: $P \in \Sigma, \mathcal{B}, \mathcal{B}_R, M, K, n = \langle s, k \rangle$
Preconditions:
        % The deceived participant, associated to strand $s$,
        % receives an unsolicited test:
        $\forall n_0 \in \mathcal{B}$ if $n_0 \prec_{\mathcal{B}} n$ then $\forall M' \sqsubseteq \mathsf{msg}(n_0). (M' \not\sqsubseteq M \vee M' \notin \mathsf{unique}_s)$
Method:
        % Introduce nonce-flow requirement, transforming unsolicited test
        % into an authentication one:
        orelse_meth handshake nonce_flow
```

Fig. 3. The *session_binding* method

attack,[6] but the insertion of the timestamp, T', makes it very difficult for an adversary to timely realise the replay.

Two consecutive applications of *session_binding*, Fig. 3, on input DSSK yield:

1. $A \rightarrow B : A; B$
2. $B \rightarrow \mathsf{S} : A; B; \boxed{\{\!|N_b; A|\!\}_{K_b}}$
3. $\mathsf{S} \rightarrow A : \{\!|B; K_{ab}; \boxed{T_s}; \{\!|\boxed{N_b}; B; K_{ab}; A; \boxed{T_s}|\!\}_{K_b}|\!\}_{K_a}$
4. $A \rightarrow B : \{\!|\boxed{N_b}; B; K_{ab}; A; \boxed{T_s}|\!\}_{K_b}; \boxed{\{\!|A; B; Na|\!\}_{K_{ab}}}$
5. $B \rightarrow A : \boxed{\{\!|B; Na|\!\}_{K_{ab}}}$

Step 2., together with N_b, is inserted in the first application, preventing a replay protection attack on B, while the handshake à la Lowe is inserted in the second one, preventing a replay protection attack on S.

Proposition 3. *Let P' be the revised version of the protocol and let N be the nonce introduced by an application of* nonce_flow *on the replay of M. If $N \in$* unique_s *then M cannot be used to elaborate a replay.*

Proof (outline). Take the first rule, so $M = \boxed{M_2}$. Let $node_a^+(M')$ (respectively $node_a^-(M')$) denote the positive (respectively negative) node of strand A at which message M' is sent (respectively received), then if K_b^- is safe:

$$node_c^+(M_1\boxed{; \{\!|C; N_c; h(M_1)|\!\}_{K_b^+}}) \Rightarrow^+ node_c^-(M_2\boxed{; \{\!|C; N_c; h(M_2)|\!\}_{K_b^-}})$$

is an outgoing test for N_c in $\{\!|C; N_c, h(M_1)|\!\}_{K_b^+}$. Then by proposition 19 of [9] only a regular participant must have been responsible for N to exit the cyphertext $\{\!|C; N_c; h(M_1)|\!\}_{K_b^+}$ and then enter to $\{\!|C; N_c; h(M_2)|\!\}_{K_b^-}$. Any occurrence of M_i $(i = 1, 2)$ is thus tied via $h(M_i)$ to a unique test and so the result follows. The proof for the other rule is similar. □

[6] A *known-key attack* is an attack whereby, once getting knowledge of a session key, the adversary is able, if passive, to compromise keys of other sessions or, if active, to impersonate one of the (honest) protocol parties.

3.3 Patching Protocols Violating Principle 3

In this section, we generalise the method introduced in [10], designed to fix a faulty protocol containing a message without proper naming. Our generalisation consists of introducing the names of the correspondents of the reused message, rather than the names of the initiator or the responder in the protocol.

When input the NSPK protocol, *agent_naming*, Fig. 4, will add the name of the agent originating message 2, B in this case, arriving at Lowe's fix:

$$1.\ A \rightarrow B : \{\!|N_a, A|\!\}_{K_b^+} \quad 2.\ B \rightarrow A : \{\!|\boxed{B}, N_a, N_b|\!\}_{K_a^+} \quad 3.\ A \rightarrow B : \{\!|N_b|\!\}_{K_b^+}$$

Name: *agent_naming*
Input: $P, \mathcal{B}_A, \mathcal{B}_R, n \in \mathcal{B}_A \cap \mathcal{B}_R$ with $\mathsf{msg}(n)_{|\pi} = \{\!|M|\!\}_K$
Preconditions:

$$\bigvee_{A \in \text{Partners}([\{\!|M|\!\}_K])} A \notin \text{Agents}(\{\!|M|\!\}_K)$$

Patch:

Select M'' such that:
$M \leq_I M''$ with $I = \text{Partners}([\{\!|M|\!\}_K]) \setminus \text{Agents}(\{\!|M|\!\}_K)$
$\{\!|M''|\!\}_K$ is collision free with $\{\{\!|M'|\!\}_{K'} : \{\!|M'|\!\}_{K'} \sqsubseteq \mathsf{msg}(n), n \in \mathcal{B}_R\}$ wrt. Safe_r
With $\zeta_0 = \{(\pi, \{\!|M''|\!\}_K)\}$ and ζ be the adaption of $\mathcal{B}$ for ζ_0 in n, let $P' = \zeta(P)$.

Effects:

$\text{Partners}([\zeta(\{\!|M|\!\}_K)]) = \text{Agents}(\zeta(\{\!|M|\!\}_K))$

Fig. 4. The *agent_naming* method

Proposition 4. *Let P' and $\zeta(\{\!|M|\!\}_K)$ be the revised protocol and cypher-text. Then, $\zeta(\{\!|M|\!\}_K)$ may not be used to arm a naming replay attack.*

Proof (outline). Let $P(R, \vec{x})$ denote the set of strands of role R in P instantiated with parameters $\vec{x}$. Effects guarantee that $\forall R \in \text{Partners}([\zeta(\{\!|M|\!\}_K)]).P'(R, \vec{x}) \subseteq P(R, \vec{x})$, because the parameters *agree* at least on the associated names of the correspondents of $\zeta(\{\!|M|\!\}_K)$. It follows that the cypher-text cannot be used to arm a naming replay. □

4 Results

Table 1 summarises our results. We considered 36 experiments, of which 20 involve protocols borrowed from the Clark-Jacob library;[7] 4 are variants of some of these protocols (annotated with $\star$); and 12 are protocols output by SHRIMP, a next-generation of an input protocol. Next-generation protocols are shown in a separate row within the associated entry.

[7] The Clark-Jacob library comprehends 50 protocols, 26 out of which are known to be faulty. So our validation test set contains all but 6 of these security protocols. The faulty protocols that were left out are not susceptible to a replay attack.

Each row displays the result of testing a protocol against a (hierarchical) collection of properties: secrecy, s, weak authentication of the initiator, wa_i (respectively responder, wa_r) and strong authentication of the initiator, sa_i (respectively responder sa_r), where $wa_i < sa_i$ (respectively $wa_r < sa_r$.) The table separates the verification results for the original protocol, *before*, and the mended protocol, *after*, as output by Shrimp. The field value that exists at the intersection between a protocol P and a property ϕ might be either T, meaning P satisfies ϕ, F, meaning P does not satisfy ϕ, or X, meaning this property was not tested (because P was not expected to satisfy it.) Column M specifies the method that was applied to modify each faulty protocol: message (E)ncoding, agent (N)aming or session (B)inding. For the latter method, B_1 refers to rule nonce_flow and B_2 to rules handshake. In all our experiments, the application of a patch method yielded a revised protocol able to satisfy the security property that the original one did not. Whenever applicable, each mended protocol was then further requested to satisfy the remaining, stronger properties in the hierarchy, thus explaining why some entries have several runs. Note that in the discovery of some attacks we had to specify the possibility of losing a session key (annotated with †.)

Table 1. Experimental results

	before						*after*				
Protocol	s	wa_i	sa_i	wa_r	sa_r	M	s	wa_i	sa_i	wa_r	sa_r
ASRPC	T	T	T	T	F	B_1	T	T	T	T	T
BAN ASRPC	T	F	X	X	X	N	T	T	T	T	T
CCITTX.509(1)	T	F	X	X	X	N	T	T	X	X	X
	T	T	F	X	X	B_1	T	T	T	T	T
CCITTX.509(3)	T	F	X	T	T	N	T	T	T	T	T
DSSK	T	T	F	X	X	B_1	T	T	T	T	X
	T	T	T	T	F	B_2	T	T	T	T	T
NSSK	T	T	X	T	F	B_1	T	T	X	T	T
	T	T	F†	T	T	B_2	T	T	T	T	T
DSPK	T	F	X	X	X	N	T	T	X	X	X
	T	T	F	X	X	B_2	T	T	T	X	X
Kao Chow A. v1	T	T	F†	T	T	B_2	T	T	T	T	T
KSL	T	F	X	X	X	E	T	T	T	T	T
NSPK	F	X	X	T	T	N	T	T	T	T	T
BAN OR	T	F	X	X	X	N	T	T	X	X	X
Splice/AS	T	X	X	F	X	N	T	T	X	T	X
	T	T	F	T	X	B_2	T	T	T	T	T
CJ Splice	T	F	X	T	T	B_2	T	T	T	T	T
HC Splice	T	X	X	F	X	N	T	X	X	T	X
WMF	T	F	X	X	X	E	T	T	X	X	X
	T	T	F	X	X	B_2	T	T	T	T	T
WMF++ ⋆	T	T	F	X	X	B_2	T	T	T	T	T
ASRPC prune ⋆	T	F	X	X	X	N	T	T	X	X	X
	T	T	X	F	X	N	T	T	X	T	X
	T	T	X	T	F	B_1	T	T	X	T	T
	T	T	F	T	T	B_1	T	T	T	T	T
WLM	T	F	X	X	X	E	T	T	T	T	T
BAN Yahalom	T	T	T	F	X	E	T	T	T	T	T
A. DH ⋆	T	X	X	F	X	N	T	X	X	T	X
	T	X	X	T	F	B_1	T	X	X	T	T
2steps SK ⋆	T	X	X	F	X	N	T	X	X	T	X
	T	X	X	T	F	B_1	T	T	X	T	T
	T	T	F	T	T	B_1	T	T	T	T	T

Shrimp is thus able to identify a flaw and a successful candidate patch in 33 faulty protocols out of 36. Of these experiments, it applied 12 times *agent_naming*, 4 times *message_encoding*, 9 times rules nonce_flow and 8 times handshake. The protocols Shrimp fails to fix, namely: Neumann-Stubblebine, Otway-Rees and Woo-Lam Pi, are all susceptible to the type flaw subclass of replay attack. It would be misleading to dismiss message encoding on account of the few protocols it patched. This is because while applying the other methods we use it to ensure the patch did not incur in an infringement of principle 10.

We have recently made Shrimp try to patch the IKEv2-DS protocol, which is part of AVISPA's library and an abstraction of IKEv2. We found that if we abstract out the equational issues inherent to the AVISPA attack, Shrimp successfully identifies a violation to a good practice for protocol design: the omission of principal names. While the revised protocol is up to satisfy strong authentication on the session key, this patch may be subject to a criticism because IKEv2 was deliberately designed so that no principal should mention the name of its corresponding one. We then deleted *agent_naming* and re-ran our experiment; this time Shrimp applied handshake suggesting a protocol similar to IKEv2-DSx, which also is part of AVISPA's library and attack-free.

5 Comparison to Related Work

R. Choo [6] has also looked at the problem of automated protocol repair. His development framework applies a model checker to perform state-space analysis on an (indistinguishableness-based) model of a protocol, encoded using asynchronous product automata. If the protocol is faulty, Choo's approach automatically repairs it. A fair comparison between Choo's approach and ours cannot be conducted mainly because so far there is not an archival publication of [6].

When considering an automatic protocol repair mechanism like ours, one wonders whether there is an upper bound as to the information that every message should include to avoid a replay. If there is one, we could simply ensure that every message conforms it previous to any verification attempt. Carlsen [5] has looked into this upper bound. He suggested that to avoid replays every message should include five pieces of information: protocol-id, session-id, step-id, message subcomponent-id and primitive type of data items. In a similar vein, Aura [3] suggested one should also use several cryptalgorithms in one protocol and hash any authentication message and any session key. Protocol designers, however, find including all these elements cumbersome. By comparison, Shrimp only inserts selected pieces of information considering the attack at hand but may add steps to the protocol if necessary to fulfil a stronger security property.

Complementary to ours is the work of Perrig and Song [13], who have developed a system, called APG, for the synthesis of security protocols. The synthesis process, though automated, is generate and test: APG generates (extends) a protocol step by step, taking into account the security requirements, and then discards those protocols that do not satisfy them. APG is limited to generate only 3-party protocols (two principals and one server). As a reduction technique, it uses an impersonation attack and so rules out protocols that (trivially) fail to provide authenticity. The main problem to this tool is the combinatorial explosion (the search space is of the order 10^{12} according to the authors).

6 Conclusions and Further Work

In this paper, we have presented Shrimp, a method for automatically repairing faulty security protocols. Shrimp consists of a collection of patch planning methods. Each patch method is designed to transform a set of protocol steps violating a design principle into one conforming it, while ruling out the possibility of violating other principles. We have carried out a large number of experiments to validate Shrimp, finding that it successfully deals with the class of replay attacks, except for the subclass type flaw. Ongoing research thus is concerned with extending Shrimp to cope with this type of attack. Ongoing research also is concerned with extending Shrimp to account for more cryptographic primitives, including the equational properties thereof, and a greater variety of protocols. Patch methods are independent of the global security requirements required from the security protocol under investigation and the local change of one or two protocol steps may cause flaws in combination with other protocol steps which have been not considered at that point. Therefore, ongoing research involves considering global rules that may change a protocol depending on these global requirements to be achieved, controlling the local patch process.

References

1. Abadi, M., Needham, R.: Prudent engineering practice for cryptographic protocols. IEEE Transactions on Software Engineering 22(1), 6–15 (1996)
2. Abadi, M., Rogaway, P.: Reconciling two views of cryptography (the computational soundness of formal encryption). Journal of Cryptology 15(2), 103–127 (2002)
3. Aura, T.: Strategies against replay attacks. In: CSFW'97. Proceedings of the 10th IEEE Computer Security Foundations Workshop, pp. 59–69. IEEE Computer Society Press, Los Alamitos (1997)
4. Bundy, A.: The use of explicit plans to guide inductive proofs. In: Lusk, E.R., Overbeek, R. (eds.) 9th International Conference on Automated Deduction. LNCS, vol. 310, pp. 111–120. Springer, Heidelberg (1988)
5. Carlsen, U.: Cryptographic protocols flaws. In: Computer Security Foundations Workshop, CSFW'94. Proceedings of the 7th IEEE Computer Security Foundations Workshop, pp. 192–200. IEEE Computer Society, Los Alamitos (1994)
6. Raymond Choo, K.-K.: An integrative framework to protocol analysis and repair: Indistinguishability-based model + planning + model checker. In: Proceedings of Five-minute Talks at CSFW'06. Computer Security Foundations Workshop (2006)
7. Clark, J., Jacob, J.: A survey of authentication protocol literature: v1.0. Technical report, Department of Computer Science, University of York (November 1997)
8. Proceedings of the 7th IEEE Computer Security Foundations Workshop. In: CSFW'94. IEEE Computer Society (1994)
9. Guttman, J.-D., Thayer, F.-J.: Authentication tests and the structure of bundles. Theoretical Computer Science 283(2), 333–380 (2002)
10. López-Pimentel, J.C., Monroy, R., Hutter, D.: A method for patching interleaving-replay attacks in faulty security protocols. Electronic Notes in Theoretical Computer Science 174, 117–130 (2007)
11. Lowe, G.: A family of attacks upon authentication protocols. Technical Report 1997/5, Department of Mathematics and Computer Science, University of Leicester (1997)
12. Paulson, L.C.: The inductive approach to verifying cryptographic protocols. Journal in Computer Security 6(1-2), 85–128 (1998)
13. Perrig, A., Song, D.: Looking for diamonds in the desert — extending automatic protocol generation to three-party authentication and key agreement protocols. In: Proceedings of the 13th CSFW'00, pp. 64–76. IEEE Computer Society Press, Los Alamitos (2000)
14. Syverson, P.: A taxonomy of replay attacks. In: Computer Security Foundations Workshop, CSFW'94. Proceedings of the 7th IEEE Computer Security Foundations Workshop, pp. 187–191. IEEE Computer Society, Los Alamitos (1994)
15. Thayer Fabrega, F.J., Herzog, J.C., Guttman, J.D.: Strand spaces: Why is a security protocol correct? In: Proceedings Symposium on Security and Privacy, pp. 160–171. IEEE Computer Society Press, Los Alamitos (1998)