Strand-Based Approach to Patch Security Protocols

Dieter Hutter
German Research Center for Artificial Intelligence
Enrique Schmidt Str. 5
D-28359 Bremen, Germany
Tel. +49(421)218 64 277
Fax. +49(421)218 9864 277
hutter@dfki.de

Raúl Monroy
Tecnológico de Monterrey
Lago de Guadalupe Km. 3.5
52926, Atizapán, Mexico
Tel. +52(55)5864 5316
Fax. +52(55)5864 5651
raulm@itesm.mx

Abstract

In this paper, we introduce a mechanism that aims to speed up the development cycle of security protocols, by adding automated aid for diagnosis and repair. Our mechanism relies on existing verification tools analyzing intermediate protocols and synthesizing potential attacks if the protocol is flawed. The analysis of these attacks (including type flaw attacks) pinpoints the source of the failure and controls the synthesis of appropriate patches to the protocol. Using strand spaces [39], we have developed general guidelines for protocol repair, and captured them 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 implemented our mechanism into a tool, called SHRIMP. We have successfully tested SHRIMP on numerous faulty protocols, all of which were successfully repaired, fully automatically.

1 Introduction

A security protocol is a protocol that aims to establish one or more security goals, often a combination of authentication, confidentiality or non-repudiation. Security protocols are critical applications, because they are crucial to provide key Internet services, such as electronic-banking. So, security protocols are thoroughly studied to guarantee that there does not exist an interleaving of protocol runs violating a security goal, called an attack. Designing a security protocol is, however, error-prone. Although security protocols may be extremely simple, consisting of only a few steps, they are difficult to get right. Formal methods have been successfully used to identify subtle assumptions underlying a number of faulty protocols, yielding novel attacks.

In this paper, we introduce a mechanism that aims to speed up the development cycle of security protocols, by adding automated support for diagnosis and repair. Our mechanism has been especially designed to fix protocols that are susceptible to an attack of the full class replay [37]. This attack class includes a number of sub-classes; some are well known, such as reflection, and unknown key share, but others have passed slightly ignored, like type flaw.

Overview of Approach for Comparison

There exist several approaches to security protocol development, ranging from the systematic generation of a protocol (called protocol synthesis) to the transformation of a given protocol into one that is stronger, up to some specified properties (called protocol compiling). Protocol synthesis (blindly) generates a candidate protocol, and then tests it to see if it complies with the intended security requirements. By contrast, protocol compiling (blindly) transforms an input protocol, by wrapping it with explicit security constructs; thus,

---

1 A replay attack is one where a valid message is maliciously repeated in other (not necessarily different) session; so, the active participation of a penetrator is required (c.f. the Dolev-Yao penetrator model [37]).

2 A type flaw attack is one where a participant confuses a (field of a) message containing data of one type with a message data of another.
protocol compiling barely depends on the security guarantees provided by the input protocol, if any. Our method is in between, and follows the formal approach to software development: it is applied after a failed verification attempt, in order to spot a protocol flaw, and suggest a candidate patch.

To compare the relative value of our approach against that of all these competitors, we suggest to use efficiency of a given output security protocol. There are three common criteria for measuring this dimension [23]:

**Round complexity**, the number of rounds until the protocol terminates. A round is a collection of messages that can be simultaneously sent by parties, assuming that the adversary delivers all these messages intact, immediately, and to the corresponding party.

**Message complexity**, the maximum number of messages sent by any single party.

**Communication complexity**, the maximum number of bits sent by any single party.

We shall have more to say about this later on in the text, when comparing our method against rival techniques (see Section 8.)

**Contributions of Paper** In this paper, we make four key contributions. First, we provide a fully automated repair mechanism, which supersedes and surpasses our previous methods (see [25, 21].) The new mechanism has been entirely developed within strand spaces [39], and it is no longer heuristic-based. We have used strand spaces to provide a full characterization of a wide class of protocol attacks, and our repair strategies, in a way amenable to mechanization. Our mechanism for protocol repair has been successfully tested on a number of faulty protocols collected from the literature. Second, we provide a handful of basic principles for protocol repair, stemmed from our insights on our strand spaces characterization of a failed verification attempt. Our repair principles are simple and intuitive, and follow from a straight formalization of an interleaving of partial protocol runs, and the way messages are used to achieve an attack. Third, we provide an extension of the theory of strand spaces, which encapsulates issues about the implementation of messages, hence, providing a neat border between reasoning about the representation of a message, from reasoning about its actual form (a byte stream). Our extension consists of an equivalence relation over messages, and a distinctive notion of message origination. With it, we have been able to capture type flaw attacks, using the constructions of the strand spaces calculus. What is more, our abstraction of messages enables the transparent application of our strategies for protocol repair; that is, we can fix protocols that are subject to a replay attack because a protocol message can be mixed up with other one, without considering whether message confusion exploits a type flaw or not. Fourth, we provide a comparison of protocol repair against other prominent approaches for protocol development, concluding that our approach gives the designer a better insight for protocol development, guiding the conception for the rôle of messages and their composition.

**Overview of Paper** The remaining of the paper is organized as follows: first, we overview our approach to protocol repair (see Section 2). Then, we introduce notation and key, preliminary concepts (see Section 3). Next, we introduce and discuss the soundness of our extended version of strand spaces, suitable to capture type flaw attacks (see Section 4). Then, after describing the kinds of flaws we want to automatically repair (see Section 5), we provide a method for automatic protocol repair (see Section 6), and the results obtained from an experimental test (see Section 7). We conclude the paper, after comparing related work and giving directions to further work (see Sections 8 and 9).

## 2 Protocol Repair: General Approach

Our approach to protocol repair involves the use of two tools (see Fig. 1). One tool is a protocol verifier (e.g. OFMC [6, 5], ProVerif [8], or Cl-Atse [40]), which is used to analyze the protocol at hand and to yield an attack on that protocol, if the protocol is faulty. The other tool, called SHRIMP, embodies our mechanism for protocol repair. SHRIMP compares the attack against a run of the protocol where the penetrator does not participate, called a regular run. This comparison
usually spots differences that indicate where the protocol went wrong. Depending on these differences, SHRIMP offers various candidate patches, each of which modifies the protocol in a specific way. Some patches may rearrange parts of an individual message or enrich it with additional information, while others may alter the flow of messages in the protocol.

Our approach to protocol repair is iterative. Since we proceed from a local perspective to fix a problem that has become apparent during attack inspection, we cannot guarantee that a fix will remove all flaws from the protocol. So, the mended protocol is sent back to the verification tool for reanalysis. If this tool still detects a bug, either because this other bug was already present in the original protocol (as often is) or because it was introduced by our method (as has never happened so far), we iterate, applying our repair strategies on the mended protocol.

Our patches change the protocol in a very restricted and conservative way, since we have no explicit representation of the intended purpose of a protocol. For example, suppose that to prevent a replay attack we have to change the structure of an individual message in a protocol so that it uniquely determines the protocol run, as well as the protocol step it originates from. To preserve the semantics of that message, we modify it only in such a way that the new message preserves both encryption (it is encrypted in the same way) and information (it contains all information present in the original message), with respect to the knowledge of some keys.

Attack analysis and protocol repair are driven by a handful of informal guidelines for protocol design. We have translated 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. The correction of security protocols incorporates the use of several of these rules. However, protocol patches are not independent; so a rule requires preconditions to be applicable and should guarantee postconditions once it has been applied.

We have developed our mechanism within (our extension to) the theory of strand spaces [39]. Thus, the specification of a protocol and of one of its associated attack have both to be given using the strand space notation. Extra machinery is required to translate a given protocol specification so it is suitable for the chosen verification tool (see Fig. 1). Also, extra machinery is required to translate the attack output by that verification tool, if the protocol is faulty, so that the attack can be used by SHRIMP. We will see later on in the text that since a protocol specification and a protocol attack must each form a bundle, this translation is straightforward. However, care needs to be taken when the attack to the protocol is type-flaw, since in this case the translation needs to deal with the introduction of special construction blocks that justify the type flaw attack. We call these construction blocks implementation traces, as shall be seen in Section 4. A preliminary version of SHRIMP is available at [http://homepage.cem.itesm.mx/raulm/pub/shrimp/](http://homepage.cem.itesm.mx/raulm/pub/shrimp/).
3 Formal Preliminaries

We now recall standard concepts and notations about security protocols, in general, and from strands spaces [38, 39, 18], in particular. For a thorough introduction to the strand spaces, though, readers are referred to [39].

As usual, we consider messages, $A$, as the set of terms, which is freely generated from the sets of atomic messages, $TK$, consisting of nonces, $\text{Nonce}$, timestamps, $\text{Timestamp}$, agent names, $\text{Agent}$, tags, $\text{Tag}$, and keys, $K$, using concatenation, $m_1; m_2$, and encryption, $\mathbf{\{m\}}_k$, with $k \in K$ and $m, m_1, m_2 \in A$.

We assume two functions. One maps principals, $a, b, \ldots$, to their public keys, $k_a, k_b, \ldots$, and the other a pair of principals, $(a, b)$, to their symmetric, shared key, $k_{ab}$. The set of keys, $K$, comes with an inverse operator, mapping either each member of a key pair for an asymmetric cryptosystem to the other, $(k_a)^{-1} = k_a^-$, or each symmetric key to itself, $(k_{ab})^{-1} = k_{ab}$.

$\text{Parts}_K$ maps a message $t \in A$ to the set containing all the subterms of $t$ that are accessible knowing the set of keys, $K$; i.e. it is the least set satisfying: $t \in \text{Parts}_K(t)$, $\text{Parts}_K(t_i) \subseteq \text{Parts}_K(t_1; t_2)$, for $i \in \{1, 2\}$, and $\text{Parts}_K(t) \subseteq \text{Parts}_K(\{t\}_k)$, if $k^{-1} \in K$. $\text{Parts}(t) = \text{Parts}_K(t)$ defines the set of all subterms. We write $t \subseteq_K t'$ (and $t \sqsubset t'$, respectively) to denote $t \in \text{Parts}_K(t')$ (and $t \in \text{Parts}(t')$, respectively).

Let $t$ and $K$ be a message and a set of keys, respectively. Further, let $M_0, M_1, \ldots$ be a sequence of sets of messages and $K_0, K_1, \ldots$ be a sequence of sets of keys, such that $K_0 = K$, $M_0 = \text{Parts}_K(t)$, $K_{i+1} = K_i \cup (K \cap M_i)$, and $M_{i+1} = \bigcup_{t \in M_i} \text{Parts}_{K_{i+1}}(t')$. Since a message contains only finitely many keys, there is an $i_0$ with $K_j = K_{j+1}$, for all $j \geq i_0$. Therefore, $M_0, M_1, \ldots$ has a fixed-point, $M_\infty$, which we call $\text{Analz}_K(t)$. Let $M$ be a set of messages; then $\text{Synthz}(M)$ is the smallest set such that: $M \subseteq \text{Synthz}(M)$; if $m_1, m_2 \in \text{Synthz}(M)$, then $m_1; m_2 \in \text{Synthz}(M)$; and if $m \in \text{Synthz}(M)$ and $k \in \text{Synthz}(M)$, then $\mathbf{\{m\}}_k \in \text{Synthz}(M)$.

Let $\mathbf{fl}(t)$ be the set of all messages in a message $t$ obtained by flattening the top-level concatenations; i.e. $\mathbf{fl}(t) = \{t' \in \text{Parts}_K(t) | t' \in TK \text{ or } \exists m, k, t' = \mathbf{\{m\}}_k\}$. Whenever $m \in \mathbf{fl}(t)$, $m$ is said to be a component of $t$. In an abuse of notation, we write $TK(m)$ for the set of atomic messages in $m$; i.e. $TK(m) = \{t \in TK| t \in \text{Parts}_K(m)\}$. We also extend $TK(\cdot)$ to a homomorphism over sets of terms in the expected manner.

A strand represents a principal’s local view on a protocol. Thus, a strand $s$ is a sequence of directed messages (called nodes), $\pm t_1 \Rightarrow_s \pm t_2, \ldots \Rightarrow_s \pm t_n$, each of which contains the information whether the message is received from the outside (indicated by the sign “-”), or sent by the principal (indicated by “+”). Accordingly, we call a node either positive or negative. The relation $\Rightarrow_s$ connects consecutive messages in strand $s$. $\Rightarrow^+_s$ and $\Rightarrow^*_s$ are respectively used to denote the transitive, and the transitive-reflexive closure of $\Rightarrow_s$. We write $\Rightarrow_T$ for the union of $\Rightarrow_s$, for all $s \in T$. Notice, however, that, when understood from the context, we shall simply write $\Rightarrow$ to refer to $\Rightarrow_s$.

Let $v = \langle s, i \rangle$ denote the $i$-th node of a strand $s$. Then, we respectively use $\text{msg}(v)$ and $\text{sign}(v)$ to denote the message, and the event, sending or reception, associated with node $v$. We write $\#(s)$ for $\#(s)$ to stand for the length of a strand $s$; thus, $1 \leq i \leq \#(s)$, whenever $\langle s, i \rangle$ denotes the $i$-th node of strand $s$.

The abilities of a Dolev-Yao penetrator are characterized by means of a set of building blocks, called penetrator strands. There are eight penetrator strands: (K)ey: $\langle +k \rangle$, provided that $k \in K_0$, with $K_0$ being the set of keys initially known to the penetrator; (T)ee: $\langle -m, +m, +m \rangle$; (F)lush: $\langle -m \rangle$; (M)essage: $\langle +m \rangle$, provided that $m \in \text{Agent}$ or $m \in \text{Nonce}$; (C)oncatenation: $\langle -m_1, -m_2, +m_1, +m_2 \rangle$; (S)eparation: $\langle -m_1, m_2, +m_1, +m_2 \rangle$; (E)ncryption: $\langle -k, -m, +\mathbf{\{m\}}_k \rangle$; and finally (D)ecryption: $\langle -\mathbf{\{m\}}_k, +k, +m \rangle$.

A strand $s$ denotes a set of strands. Given two strands, $s$ and $s'$, with $s \neq s'$, $\langle s, i \rangle \rightarrow \langle s', j \rangle$ represents inter-strand communication from $s$ to $s'$. It requires the nodes messages to be equal, i.e. $\text{msg}(\langle s, i \rangle) = \text{msg}(\langle s', j \rangle)$ but having complementary signs, i.e. $\text{sign}(\langle s, i \rangle) = +$ and $\text{sign}(\langle s', j \rangle) = -$.

A bundle is a composition of (possibly incomplete) strands and penetrator traces, hooked together via inter-strand communication. Formally, it is a finite, acyclic graph $B = \langle V, (\rightarrow \cup \Rightarrow) \rangle$, \"}
such that for every \( v_2 \in V \), the two following conditions hold: i) if \( \text{sign}(v_2) = - \), then there is a unique \( v_1 \in V \) with \( v_1 \rightarrow v_2 \); and ii) if \( v_1 \Rightarrow v_2 \), then \( v_1 \in V \) and \( v_1 = (s, i) \) and \( v_2 = (s, i + 1) \).

\( \prec_B \) and \( \preceq_B \) denote respectively the transitive and the transitive-reflexive closure of \( (\rightarrow \cup \Rightarrow) \).

Let \( s_a \) be the strand of participant \( a \). If \( a \) is honest, then the strand \( s_a \), as well as each individual strand node, \( (s_a, i) \), is said to be regular. Otherwise, it is said to be penetrator. Let \( v = (s_a, j) \) be some node in the strand of participant \( a \); then, we use \( \text{Agent}(v) = a \) to denote the strand participant name.

The pair \( (\Sigma, \mathcal{P}) \) is said to be an infiltrated strand space, whenever \( \Sigma \) is a strand space and \( \mathcal{P} \subseteq \Sigma \), such that every \( p \in \mathcal{P} \) is a penetrator strand. In an infiltrated strand space, \( (\Sigma, \mathcal{P}) \), the penetrator can build masquerading messages using \( M, K, F, T, C, S, E, \) and \( D \), only. Notice that penetrator traces of type \( M \) cannot suddenly output an unguessable nonce, which are modelled using origination assumptions.

A bundle is said to be regular, if it contains no penetrator strands, and it is said to be penetrator, otherwise. A message \( m \) is said to be a component of a node \( v \) if it is a component of \( \text{msg}(v) \). A node \( v \) is said to be a \( M, K, \ldots \) node, if it lies on a penetrator strand with a trace of kind \( M, K, \ldots \).

The following section is entirely devoted to the extension of strand spaces that allows the capture of type-attack flaws.

## 4 Extending Strand Spaces to Capture Type Flaw Attacks

The strand space approach, see above, introduces messages as a freely generated datatype, built up on atomic messages, like nonces, names, or keys, with the help of the two constructors, namely: concatenation and encryption. Hence, messages are considered to be different if their syntactical representations (as constructor ground terms) are different. However, in practice, messages are typically implemented as byte-streams, and their parsing as messages may not be unique: there might be two different messages that share the same implementation as a byte-stream. Thus, the expectation of the receiver about a byte-stream controls the interpretation it gives to the message. Put differently, a penetrator can fake a message (or parts thereof) by (re-)using messages of the wrong type but with suitable implementation.

**Example 1** For example, consider the Woo Lam π₁ protocol \([4]\), given as a bundle below (there, and henceforth, \( \text{Init} \), \( \text{Resp} \), and \( \text{Serv} \) stand for the initiator, responder, and the server, respectively):

\[
\begin{align*}
+a & \rightarrow & -a \\
\downarrow & & \downarrow \\
-n_b & \leftarrow & +n_b \\
\downarrow & & \downarrow \\
+\{a; b; n_b\}_k & \rightarrow & -\{a; b; n_b\}_k \\
\downarrow & & \downarrow \\
+\{a; b; \{a; b; n_b\}_k\}_k & \rightarrow & -\{a; b; \{a; b; n_b\}_k\}_k \\
-\{a; b; n_b\}_k & \leftarrow & +\{a; b; n_b\}_k \\
\end{align*}
\]

\( \text{Init} \) \hspace{2cm} \text{Resp} \hspace{2cm} \text{Serv} \]

This protocol is vulnerable to a type flaw attack, when the implementation of nonces can be confused with the implementation of encrypted messages, giving rise to the following attack:

1. \( \text{spy}(a) \rightarrow b : a \)
2. \( b \rightarrow \text{spy}(a) : n_b \)
3. \( \text{spy}(a) \rightarrow b : n_b \)
4. \( b \rightarrow \text{spy(srv)} : \{a; b; n_b\}_k \)
5. \( \text{spy(srv)} \rightarrow b : \{a; b; n_b\}_k \)
That $n_b$ and $\{a; b; n_0\}_{k_a}$ can be confused is not unreasonable, since the receiver, $b$, does not know the key, $k_{as}$, shared between $a$ and $s$, and it is therefore not able to realize the hidden structure of the message under encryption.  

Strand spaces cannot represent the attack above, since terms are freely generated. Hence any guarantees we might obtain from this theory could make use of the freeness axioms, and so could not be transferred to a situation where they do not hold. So, in what follows, we will extend the strand space notation to deal with type flaw attacks.

### 4.1 Theories of Message Implementation

In a first step, we have to specify the implementation of messages, which we use to represent a run of the protocol in practice. We assume that there is some translation function $\llbracket \cdot \rrbracket$ that maps any message $t$ (given in the freely generated datatype) to its implementation $\llbracket t \rrbracket$. Furthermore, we assume also that there are implementations $\llbracket \cdot \rrbracket^1$ and $\llbracket \cdot \rrbracket^2$ for concatenation and encryption on the implementation level, such that $\llbracket \cdot \rrbracket$ is a homomorphism from messages to their implementations. We also assume that $\llbracket \cdot \rrbracket$ denotes a faithful representation on individual message types, i.e. we can distinguish the implementations of, for instance, two agents or two nonces (while still a nonce and a key can share a common implementation). The following definition formalizes this idea and provides the minimal requirements to an implementation.

**Definition 1 (Implementation, $\approx$)** Let $R$ be a set and let $\llbracket \cdot \rrbracket$ be a mapping from $A$ to $R$. $(\llbracket \cdot \rrbracket, R)$ is an implementation of $A$ iff the following holds.

\begin{align}
\forall x, y \in A. & \llbracket (x; y) \rrbracket = \llbracket x \rrbracket \llbracket y \rrbracket \\
\forall x, y \in A. & \llbracket (x) \rrbracket = \llbracket x \rrbracket \\
\forall x, y \in \text{Agent}. & \llbracket x \rrbracket = \llbracket y \rrbracket \rightarrow x = y \\
\forall x, y \in K. & \llbracket x \rrbracket = \llbracket y \rrbracket \rightarrow x = y \\
\forall x, y \in \text{Nonce}. & \llbracket x \rrbracket = \llbracket y \rrbracket \rightarrow x = y \\
\forall x, y \in \text{Timestamp.} & \llbracket x \rrbracket = \llbracket y \rrbracket \rightarrow x = y
\end{align}

where $x$ and $y$ are all meta-variables. We write $m_1 \approx m_2$ iff $\llbracket m_1 \rrbracket = \llbracket m_2 \rrbracket$. If $\llbracket \cdot \rrbracket$ is fixed by the environment we simply write $m_1 \approx m_2$, instead of $m_1 \approx m_2$.

For a typical setting, we might need to extend Definition 1 with additional axioms specifying in more detail how messages and their operations are implemented. For example, one might like to formalize an implementation theory which copes with message length, i.e. one which assumes that all types of atomic messages have a specific length. Then, reasoning about such a theory requires arithmetic (and, in the worst case, properties of least common multiples). Another example extension of Definition 1 is the use of Meadow’s probabilistic approach [30] to reason about the equality of the implementations of two messages.

### 4.2 Implementation Traces

We now introduce the new ability of the penetrator that enables it to reinterpret a message $m$ as another message $m'$, as long as they share a common implementation. Since the Dolev-Yao-like abilities of a penetrator are formalized by the set of penetrator traces (which it can use to analyze, reuse and synthesize messages), we enlarge this set by an additional rule, we call $I$-trace.

**Definition 2 (Penetrator Trace)** Let $K_p$ denote the keys initially known to the penetrator. Then, a penetrator trace of the extended strand space is either a penetrator strand, as given in Section 3, or the following:

---

3Recall that, for all $m_1, m_2 \in A$, $m_1 = m_2$ implies $m_1 \approx m_2$; though, clearly, this implication does not reverse.
(I)Implementation: \( \langle -m, +m' \rangle \), provided that \( m \approx m' \), but \( m \neq m' \).

We call the messages \( m \) and \( m' \) camouflaged and spoofed, respectively.

**Example 2** Consider the example flawed protocol Woo Lam \( \alpha_1 \), introduced in Example 1. To capture the attack to this protocol, we first formalize the theory for the implementation of messages. So, besides the axioms given in Definition 1 (which hold for all implementations), we require that the implementation of nonces can be confused with the implementation of encrypted messages: hence, we add the following axiom:

\[
\forall z \in \text{Nonce}. \exists y \in K, x \in A. \quad I(z) = I(\langle x \rangle_y)
\]

(7)

Then, we build the penetrator bundle describing the type flaw attack, as follows:

\[\begin{align*}
\text{spy} & \quad b \\
+ a & \quad \longrightarrow \quad - a \\
\downarrow & \quad \downarrow \\
I-\text{trace} & \quad \downarrow \\
+ \langle m \rangle_k & \quad \longleftrightarrow \quad - \langle m \rangle_k \\
\downarrow & \quad \downarrow \\
- \langle a; b; \langle m \rangle_k \rangle_{k_a} & \quad \longleftrightarrow \quad + \langle a; b; \langle m \rangle_k \rangle_{k_a} \\
\downarrow & \quad \downarrow \\
I-\text{trace} & \quad \downarrow \\
+ \langle a; b; \langle m \rangle_k \rangle_{k_a} & \quad \longrightarrow \quad - \langle a; b; \langle m \rangle_k \rangle_{k_a}
\end{align*}\]

This bundle contains two \( I \)-trace instances. The first instance is used to reinterpret the nonce \( n_b \) as an encrypted message \( \langle m \rangle_k \). To justify the application of the trace, both messages have to agree on their implementation, i.e. \( I(n_b) = I(\langle m \rangle_k) \). Both, \( m \) and \( k \), appear for the first time at the second node of this \( I \)-trace, which basically means that the penetrator is free to choose \( m \) and \( k \) such that \( I(n_b) = I(\langle m \rangle_k) \) holds. Axiom 7 guarantees that the penetrator will always find appropriate values for \( m \) and \( k \), and justifies the application of the \( I \)-trace rule. The second instance of an \( I \)-trace rule is used to revert the reinterpretation and replaces \( \langle m \rangle_k \) by \( n_b \) inside a message. The justification of this application is rather simple: the penetrator has chosen \( m \) and \( k \) such that \( I(n_b) = I(\langle m \rangle_k) \) holds. Applying axioms 4 and 5, we can deduce that then also \( I(\langle a; b; n_b \rangle_{k_a}) = I(\langle a; b; \langle m \rangle_k \rangle_{k_a}) \) holds, which guarantees the applicability of the \( I \)-trace rule also in this case. ■

In what follows, we will abstract from this example and formalize the justification of \( I \)-trace rule applications in general.

### 4.3 Originating Terms and Implementation Equivalences

The application of the \( I \)-trace rule demands that two messages \( m \) and \( m' \) share the same implementation. The proof for this precondition is done within the underlying implementation theory. In the example above we have used the fact that the penetrator is free to choose \( m \) and \( k \) appropriately. Therefore, to justify the rule application we have to prove that:

\[
\forall n_b \in \text{Nonce}. \exists k \in K. \exists m \in A. \quad I(n_b) = I(\langle m \rangle_k)
\]

(8)

which is an easy consequence of axiom 7.

Notice that the proof obligation would change if the penetrator were not free to choose \( m \) and \( k \), because both messages have occurred on regular strands before applying any \( I \)-trace rule. In such a case, we would have to prove:

\[
\forall n_b \in \text{Nonce}. \forall k \in K. \forall m \in A. \quad I(n_b) = I(\langle m \rangle_k)
\]

(9)

The notion of message origination is an important one, and, hence, prompts adequate formalization; we shall do so next, see Section 4.3.
to ensure that the penetrator can always camouflage $\{m\}_k$ by $n_k$ regardless of how $m$ and $k$ have been chosen beforehand. As a consequence all nonces and encrypted terms would share the same implementation. Furthermore, due to the injectivity (5) of the implementation for nonces, (9) implies that the set of messages can only contain a single nonce and a single encrypted term.

In practice, the proof obligation (9) is too strong and might prevent to reveal an attack since a honest principal could have unintentionally chosen $m$ and $k$ in such a way that actually $I(n_k) = I(\{m\}_k)$ holds. However, the probability for this to happen is similar to the probability that two principals will independently choose the same nonce in order, for instance, to use it as a challenge. This actually justifies our use of (8), and also illustrates the abstraction we have made in order to avoid probability calculations, like they are done in [30, 31].

The question of selecting either (8) or (9) for justifying the first I-trace in Example 2 depends on the fact whether $m$ and $k$ originate at the second node of the I-trace or not. The notion of a node originating a term (e.g. a nonce) comprises the principal’s freedom to choose an appropriate value for the term (e.g. the nonce). In the original approach, the notion of a node originating a term is a syntactical property of strands [39]:

“Let $B = \langle V, (\rightarrow \cup \Rightarrow) \rangle$ be a bundle. An unsigned term $m$ originates at a node $v \in V$, if $\text{sign}(v) = +$; $m \subseteq \text{msg}(v)$; and $m \nsubseteq \text{msg}(v')$, for every $v' \Rightarrow v$. $m$ is said to be uniquely originating if it originates on a unique $v \in V$.”

Any information that a strand, or its respective principal learns, occurs syntactically within some received message. Hence, a nonce originates in a particular node of a strand if it does not occur syntactically in any previous node of this strand. In our setting, the situation is different, since an information might be semantically included in a message but camouflaged by using I-traces.

For example, consider the second I-trace, $(\text{spy}, 4) \Rightarrow (\text{spy}, 5)$, in the attack given in Example 2. Although $n_k$ occurs in $(\text{spy}, 5)$, it has been introduced in the bundle already in $(\text{spy}, 2)$, and was received by the I-trace camouflaged as $\{m\}_k$ in $(\text{spy}, 4)$.

To guarantee that a term originates also in terms of its implementation at a specific node, c.f. $n_k$, we have to make sure that it was never subject to a camouflage. Therefore, given a node $v$ of a bundle, we collect all camouflages introduced by I-traces occurring in front of $v$ in the set of equalational constraints of $v$:

**Definition 3 (Equalational constraints)** Let $B = \langle V, (\rightarrow \cup \Rightarrow) \rangle$ be a bundle and let $v \in V$. Then, the equalational constraints of $v$ wrt. $B$, written $\mathcal{E}Q_B(v)$, is a set of pairs given by:

$$\mathcal{E}Q_B(v) = \{ \langle \text{msg}(v_1), \text{msg}(v_2) \rangle : v_1 \Rightarrow I\text{-trace} v_2 \text{ and } v_2 \preceq_B v \}$$

where $v_1 \Rightarrow I\text{-trace} v_2$ denotes both that $v_1$ and $v_2$ both lie on a penetrator strand, and that they are interconnected via a trace of kind $I$.

**Example 3** Going back to our running example, we find that $\mathcal{E}Q((\text{spy}, 4)) = \{ \langle n_k, \{m\}_k \rangle \}$ and $\mathcal{E}Q((\text{spy}, 5)) = \{ \langle n_k, \{m\}_k \rangle, \{\{m\}_k, n_k\} \}$. Using Definition 3, we can adapt the notion of messages originating in a node to our settings.

**Definition 4 (Originating nodes)** Let $B = \langle V, (\rightarrow \cup \Rightarrow) \rangle$ be a bundle and let $v \in V$ be a node in $B$. An atomic term $m$ originates at $v$ (in $B$) iff:

- $\text{sign}(v) = +$;
- $m \subseteq \text{msg}(v)$;
- $m \nsubseteq \text{msg}(v')$ for every $v' \Rightarrow v$; and
- $\forall (m', m'') \in \mathcal{E}Q_B(v)$. ($m \nsubseteq m'$ and $m \nsubseteq m''$).

In an abuse of notation, we extend the above definition also to encrypted messages. Hence, an encrypted message $\{m'\}_k$ originates at a node $v$ (in $B$) iff $\{m'\}_k$ and $v$ satisfy the conditions of the atomic message $m$ and node $v$ in Definition 4. In our example of the Woo Lam $\pi_1$ protocol, both $m$ and $k$ originate at $(\text{spy}, 3)$.
Figure 2: Bundle of the KP-protocol in [36]
Example 4 (The KP Protocol) Another example protocol that is subject to a type flaw attack is KP (see Fig. 2). Snekkenes [36] discusses this protocol and illustrates the attack. The type flaw attack to KP is based on the facts that:

1. On step 3, the responder, b, extracts \( \{ a; b; n; k_s \}_{k_{bs}} \) from the component encrypted under \( k_{bs} \), and, then, on step 4, sends this extracted message to a;
2. Keys, agents and encrypted share common implementations (c.f. the first component of the fourth message of the responder); and
3. Therefore, the implementation of message 3 can be misinterpreted (in a different run) to be the implementation of the first component of message 4.

Similar to the previous example, we build up an appropriate message theory reflecting the assumptions laid on before. Besides axioms (1)—(6), we require additional axioms specifying the assumption that keys (generated dynamically by the server) and agents (or encrypted messages, respectively) potentially share the same implementation:

\[
\forall x \in \text{Agent}. \exists w \in K. \ \mathbb{I}(x) = \mathbb{I}(w) \tag{10}
\]

\[
\forall w \in K. \exists w' \in K, y \in A. \ \mathbb{I}(w) = \mathbb{I}(\{y\}_{w'}) \tag{11}
\]

Fig. 3 illustrates the resulting attack in the strand space notation using our notion of \( \mathbb{I} \)-traces. This example penetrator bundle contains three \( \mathbb{I} \)-traces. Node (spy, 7) originates the key \( k'_s \) on the penetrator strand. Since this key originates at (spy, 7), the penetrator is free to choose its value, and axiom (11) guarantees that there is an appropriate key \( k'_s \) possessing the same implementation as a.

Similar arguments hold for (spy, 7) originating \( m \) and \( k \) by using axiom (11). For (spy, 9), it is the case that \( \mathcal{E}_{\mathcal{Q}_B}(\langle \text{spy}, 9 \rangle) = \{ \langle a, k'_s \rangle, (k_s, \{m\}_{k_s}) \} \), which justifies the application of the \( \mathbb{I} \)-trace rule application, since it implies trivially that \( \mathbb{I}(\{m\}_{k_s}) = \mathbb{I}(k_s) \) holds. \[\blacksquare\]
Figure 3: Type flaw attack in the KP-protocol in [36]
5 Protocol Flaws

In this section, we will analyze the different types of flaws resulting in an insecure protocol. The main source of such flaws in security protocols is that typically a protocol is not run in isolation, but that there are, simultaneously, multiple run instances of the same protocol that might interfere with each other. Typically, the penetrator carries out an attack on a protocol run, by reusing information gathered from another run (or even from the same one). Hence, patching security protocols is mostly concerned with disambiguating individual protocol messages in order to avoid that a message in some protocol run can be misinterpreted either as another message at a different protocol step, or as a message of the same protocol step but on a different protocol run. We start with an investigation of how to find the steps in a protocol that cause the misuse of messages and then we will discuss possible changes in the shape of such messages to avoid the confusion.

5.1 Locating the Reuse of Messages

In order to fix a protocol, we must analyze the input attack bundle of the protocol with the aim of identifying the positions where messages, eavesdropped from one or more runs, are fed into some others. So, in this section, we will provide the notation that is necessary to split an attack bundle into strand sets representing the different protocol runs involved therein. We will also define the notion of canonical bundle, which represents an ideal protocol run imposed by a set of given honest protocols. Comparing the message flow in the canonical bundle versus the attack bundle enables us to determine the crucial steps in the protocol which cause the flawed behavior.

5.1.1 Roles and Protocol Participation

We start with the formal introduction of roles as a sort of generic strands. Roles are instantiated to strands, by renaming generic atomic messages to specific elements in \( \text{TK} \). We also consider penetrator traces as roles.

**Definition 5 (Roles)** A role \( r \) is a pair \((s, M)\) of a strand \( s \) and \( M \subseteq \text{TK} \) of atomic messages occurring in \( s \). Intuitively, \( r \) denotes a strand \( s \) parameterized in \( M \).

In abuse of notation we identify a role \( r \) with its strand \( s \) if \( M \) is the set of all atomic messages occurring in \( s \).

**Definition 6 (I-th Step Execution of a Strand)** Let \( r = (s, M) \) be a role and let \( 1 \leq i \leq \#(s) \). Then, a strand \( s' \) is an instance of \( r \), up to the execution of step \( i \), iff there is a renaming \( \alpha \) of \( M \), such that \( s' = \alpha(s, 1) \Rightarrow \ldots \Rightarrow \alpha(s, i) \). We denote the strand instance \( s' \) of \( r \) up to step \( i \) by \( r[i, \alpha] \). Let \( s' = r[i, \alpha] \); whenever \( i = \#(s) \), then \( s' \) is said to be complete; otherwise, it is said to be partial.

Given a set \( \mathcal{R} \) of roles, then \( \mathcal{R}_H \) denotes the regular roles in \( \mathcal{R} \), and \( \mathcal{R}_P \) those of the penetrator. Likewise, let \( \mathcal{T} \) be a set of strand instances, then \( \mathcal{T}_H \) denotes all regular, possibly partial, strand instances, and \( \mathcal{T}_P \) all the penetrator ones.

**Example 5** The classical Needham-Schroeder Public Key (NSPK) protocol consists of two roles:

\[
\begin{align*}
\text{initiator role } r_{\text{Init}} & : + \langle a; n \rangle_{k_b} \Rightarrow - \langle n; n' \rangle_{k_a} \Rightarrow + \langle n' \rangle_{k_b} & (12) \\
\text{responder role } r_{\text{Resp}} & : - \langle a; n \rangle_{k_b} \Rightarrow + \langle n; n' \rangle_{k_a} \Rightarrow - \langle n' \rangle_{k_b} & (13)
\end{align*}
\]

Then, from these roles, we may obtain the following strand instances:

\[
\begin{align*}
\ r_{\text{Init}}[2, \{ a \leftarrow a, n \leftarrow n, k_a \leftarrow k_a, k_b \leftarrow k_b \}] & : + \langle a; n \rangle_{k_b} \Rightarrow - \langle n; n' \rangle_{k_a} \\
\ r_{\text{Resp}}[1, \{ a \leftarrow a, n \leftarrow n, k_b \leftarrow k_c \}] & : - \langle a; n \rangle_{k_c}
\end{align*}
\]
A bundle $B$ is constructed from a set of roles, by combining various role instances with the help of inter-strand communication, $\rightarrow$. Hence, stripping off inter-strand communication from a bundle, $B$, we end up with a multiset $T$ of isolated instances of some roles.

**Definition 7 (R-Bundles)** Let $R$ be a set of roles and $T$ be a multiset of (possibly partial) strand instances of roles in $R$. Then, $B$ is a $R$-bundle out of $T$ iff (i) each node in a strand of $T$ is also a node in $B$ and vice versa, and (ii) $n \Rightarrow n'$ iff $n \Rightarrow_B n'$ for all nodes $n, n'$ of $B$ (and $T$ respectively).

We define a canonical bundle to be a bundle representing an intended run of a protocol. In a canonical bundle, each role is instantiated by a strand exactly once, and each strand instance is complete. Furthermore, the strands of a canonical bundle, and thereby the associated roles, are all regular.

**Definition 8 (Canonical Bundles)** An $R$-bundle $B$ out of $T$ is canonical iff all roles in $R$ are regular and there is a symbol renaming $\alpha$ that is injective wrt. each role $r \in R$ such that $T = \{ r[#(r), \alpha] \mid r \in R \}$

Given a canonical $R$-bundle $B$ we can easily retrieve $R$ (up to isomorphism) from $B$ (or $T$, respectively).

Furthermore, we can consider each finite, acyclic graph $B = \langle V, (\rightarrow \cup \Rightarrow) \rangle$ as a canonical $R$-bundle if for all $v_2 \in V \, \mathrm{sign}(v_2) = -$ implies that there is a unique $v_1 \in V$ with $v_1 \rightarrow v_2$, and that $R$ is the set of all roles extracted from $B$. Later on, we use this property in order to formulate changes of a protocol (i.e. changes of the roles of a protocol) simply by changing the original canonical protocol and interpret the resulting graph as the canonical bundle of the new protocol comprising also its new roles (up to isomorphism).

In what follows, we will use sans serif fonts to indicate messages in canonical bundles. Furthermore, we use the superscript $c$ to denote a canonical bundle $B^c$, and the roles $R^c$ or set of strand instances $T^c$ of a canonical bundle. In the examples, we also assume that the honest roles of a canonical bundle are normalized so that $\alpha = \{\}$, and then write $(r,i)$, instead of $(r[#(r),\{\}],i)$.

To illustrate our ideas and definitions, we will make use of various protocols as running examples. We first present canonical bundles of these protocols.

**Example 6** We start with the NSPK protocol. Given the roles $R^c_{\text{NSPK}} = \{\text{Init, Resp}\}$ of $\text{NSPK}$, (12) and (13), we obtain the following canonical bundle, $B^c$:

\[ 
\begin{array}{c}
+ \{a; n\}_{k_b} \quad \longrightarrow \quad - \{a; n\}_{k_b} \\
\downarrow \\
- \{n; n'\}_{k_a} \\
\downarrow \\
+ \{n'; k\}_{k_b} \\
\downarrow \\
\text{Init} \quad \text{Resp}
\end{array} 
\]

**Example 7** As another example, consider the Wide-Mouth-Frog (WMF) protocol. With $R^c_{\text{WMF}} = \{\text{Init, Serv, Resp}\}$, the canonical bundle of this protocol is as follows:

\[ 
\begin{array}{c}
+ a, \{b; t_2; k\}_{k_a} \quad \longrightarrow \quad - a, \{b; t_2; k\}_{k_a} \\
\downarrow \\
+ \{a; t_2+d; k\}_{k_b} \\
\quad \rightarrow \quad - \{a; t_2+d; k\}_{k_b} \\
\text{Init} \quad \text{Serv} \quad \text{Resp}
\end{array} 
\]
Example 8 As a third example, look into the Woo Lam authentication protocol:

\[
\begin{array}{c}
+ x, n \\
\downarrow \\
- b, n' \\
\downarrow \\
+ \{ x; b; n; n' \}_{kas} \\
\downarrow \\
- \{ x; b; n; n' \}_{kas} \\
\downarrow \\
\end{array}
\]

Example 9 Our last example is the Denning-Sacco-Shared Key (DSSK) protocol.

\[
\begin{array}{c}
+ x, b \\
\downarrow \\
- \{ x; b; k_{ab}; t_s; \{ x; b; k_{ab}; x; t_s \}_{kas} \}_{kas} \\
\downarrow \\
+ \{ x; b; k_{ab}; x; t_s \}_{kas} \\
\downarrow \\
\end{array}
\]

5.1.2 Characterizing the Perspective of each Participant in a Protocol Attack

Attacks detected by some security protocol analyzer typically involve different (partial) protocol runs. Messages generated in one run are misused to fake messages in other runs. Since honest principals only see the messages of a protocol that they receive or send, each principal develops his own view on the protocol run. In the attack of the NSPK protocol, see Example 10, Alice assumes she is communicating with Charly, while Bob thinks he is communicating with Alice. However, both views are not compatible. In the following definition, we truss compatible strands of honest principals to so-called protocol sections. While in a canonical bundle all honest strands belong to a single protocol section, the attack bundle of NSPK consists of two protocol sections denoting the (faked) protocol runs of Alice with Charly and Alice with Bob, respectively.

Definition 9 (Protocol Section) Let \( B^c \) be a canonical \( R^c \)-bundle out of some \( T^c \) and let \( B \) be an \( R \)-bundle out of some \( T \).

A set \( T' \subseteq T \) is a protocol section wrt. \( B^c \) iff there is a renaming \( \beta \) such that, for all \( r[i, \alpha'] \in T' \), there is one, and only one, \( r[\#(r), \alpha] \in T^c \), with \( \alpha' \) being a function denoting the composition of \( \beta \) and \( \alpha \) and subject to the domain \( \text{TK}(r) \). We call \( \beta \) the canonical renaming for \( T' \) wrt. \( B^c \).

We call the mapping that maps each node of a regular strand \( \langle r[i, \alpha], j \rangle \) in \( T' \) to the node \( \langle r[i, \alpha'], j \rangle \) in \( B^c \), for all \( r[i, \alpha'] \in T' \) and \( 1 \leq j \leq i \), the canonical mapping of \( T' \) to \( B^c \).

\[\text{Fig. 4: Relations of role instances}\]

Fig. 4 illustrates the relations between roles and their instances in the canonical bundle and a protocol section, respectively.
Example 10 To exemplify all these notions, let us study again the attack on NSPK, illustrated in the bundle shown below.

Besides the two roles of $R_{\text{NSPK}}$, this bundle contains the penetrator traces $(K)$, $(E)$, and $(D)$. It is therefore an $R$-bundle, with $R = \{ (K), (E), (D) \} \cup R_{\text{NSPK}}$. $T$ consists of the six individual strands occurring in the bundle.

Using the canonical bundle $B^c$ given in Example 6, this attack bundle contains an instance $r_{\text{init}}[3, \beta]$ with $\beta = \{ a \leftarrow a, b \leftarrow c, n \leftarrow n, n' \leftarrow n', k_a \leftarrow k_a, k_b \leftarrow k_b \}$ and another instance $r_{\text{resp}}[3, \beta']$ with $\beta' = \{ a \leftarrow a, b \leftarrow b, n \leftarrow n', n' \leftarrow n', k_a \leftarrow k_a, k_b \leftarrow k_b \}$. Both form individual protocol sections $T_1 = \{ r_{\text{init}}[3, \beta] \}$ and $T_2 = \{ r_{\text{resp}}[3, \beta'] \}$. However, $T_3 = T_1 \cup T_2$ is not a protocol section, because $\beta$ and $\beta'$ are incompatible.

Example 11 Concerning the WMF protocol (see Example 7) AVISPA [3] has found that it fails to guarantee weak authentication of the responder, Bob, to the initiator, Alice, yielding the attack bundle shown below:

$+a, \langle b; t_a; k \rangle_{k_a} \rightarrow -a, \langle b; t_a; k \rangle_{k_a}$

Alice(Init)

Comparing this attack bundle against the canonical one, $B^c$, shown in Example 6, we notice that it contains instances for the initiator and responder strands, while the server is impersonated by the penetrator: $r_{\text{init}}[1, \beta]$ with $\beta = \{ a \leftarrow a, b \leftarrow b, t_a \leftarrow t_a, k_a \leftarrow k_a \}$ and $r_{\text{resp}}[1, \beta']$ with $\beta' = \{ a \leftarrow b, b \leftarrow a, t_a \leftarrow t_a, k_a \leftarrow k_a \}$. Both form individual protocol sections, $T_1 = \{ r_{\text{init}}[1, \beta] \}$ and $T_2 = \{ r_{\text{resp}}[1, \beta'] \}$. Again, $T_1 \cup T_2$ is not a protocol section, because $\beta_{\text{init}}$ and $\beta_{\text{resp}}$ are not compatible.

Example 12 DSSK is vulnerable to a so-called multiplicity attack (see Section 6.3).
Let \( \text{Definition 11 (Acceptable Messages)} \)
which a honest principal will accept a received message, because it satisfies his expectations, as by misusing an observed encrypted message. In a first step, we characterize the conditions under which the penetrator confuses an honest principal, in what follows, we characterize situations in which the penetrator confuses an honest principal, operating under the same assumptions. This is the rationale that enables us to compute the minimal set of protocol sections which, together with the penetrator’s activities, constitutes a given attack bundle.

\[ \text{Definition 10 (Coverage)} \]
Let \( \mathcal{B} \) be an \( \mathcal{R} \)-bundle out of some \( \mathcal{T} \) and let \( \mathcal{B}^c \) be a canonical \( \mathcal{R}^c \)-bundle with \( \mathcal{R}_H \subseteq \mathcal{R}^c \). A partition \( T_1, \ldots, T_k \) of \( \mathcal{T} \) is a coverage of \( \mathcal{T} \) wrt. \( \mathcal{B}^c \) iff each \( T_i \) is a protocol section wrt. \( \mathcal{B}^c \). A coverage is optimal iff it is not a refinement of any other coverage. The canonical mapping of \( \mathcal{B} \) to \( \mathcal{B}^c \) is the composition of the canonical mappings of \( T_1, \ldots, T_k \) to \( \mathcal{B}^c \).

\[ \text{Example 13} \]
Let us illustrate coverage through two examples. Go back to Example 10 there, \( \{T_1, T_2\} \) is a coverage of \( \mathcal{T} \); it is also optimal since \( T_2 \) is not a protocol section. Now go back to the attack bundle given in Example 12 then, we obtain an optimal coverage consisting of two protocol sections: the first covers an instance of the entire canonical bundle, while the second consists only of the second copy of the responder strand.

\[ \text{5.1.3 The Misuse of Messages} \]
In what follows, we characterize situations in which the penetrator confuses an honest principal, by misusing an observed encrypted message. In a first step, we characterize the conditions under which a honest principal will accept a received message, because it satisfies his expectations, as formulated in the definition of the protocol.

\[ \text{Definition 11 (Acceptable Messages)} \]
Let \( m, m' \) be messages, \( K \) a set of keys and \( \text{Vars} \subseteq \text{TK}(m) \) be a set of atomic messages in \( m \). Then, \( m' \) is accepted for \( m \) wrt. a renaming \( \sigma \) with \( \text{DOM}(\sigma) \subseteq \text{Vars} \) and keys \( K \), iff \( \vdash_K (m, m', \sigma) \) according to the following rules:

\[ +a; b \quad \vdash \quad -a; b \]

\[ \text{Init} \quad \text{Serv} \]

\[ \begin{align*}
\vdash_K (m, m, \langle \rangle) & \quad \text{(Id)} \\
\vdash_K (m, m', \sigma) & \quad \text{if } \text{Dom}(\sigma) \subseteq \text{Vars} \cap \text{TK}(m) \quad \text{I}(\sigma(m)) = \text{I}(m'), \text{ and } m' \in \text{Vars} \text{ or } m = (\text{Sub}) \\
\vdash_K (m_1, m'_1, \sigma_1) & \quad \text{if } \sigma_1, \sigma_2 \text{ compatible and } \text{I}(m') = \text{I}(m'_1; m'_2) (\text{Seq}) \\
\vdash_K (m_1; m_2, m', \sigma_1 \circ \sigma_2) & \quad \text{if } \sigma_1, \sigma_2 \text{ compatible, and } \text{I}(m') = \text{I}(\{m\}_{m_2}; m_2) \text{ with } k^{-1} \in K (\text{Enc})
\end{align*} \]

We say \( m' \) is accepted for \( m \) wrt. atoms \( \text{Vars} \) and keys \( K \) if there is a renaming \( \sigma \) with \( \text{DOM}(\sigma) \subseteq \text{Vars} \) such that \( m' \) is accepted for \( m \) wrt. \( \sigma \) and \( K \).
Example 14 For example, consider again the attack to the NSPK protocol (see Example 10). For the sake of simplicity, assume that $\llbracket$ is the identity function. In the second step of her strand, Alice, $a$, expects a message $m = \llbracket n_a; n_b \rrbracket_{k_a}$. Suppose she receives $m' = \llbracket n_a; n'_b \rrbracket_{k_a}$, then we can derive:

\[
\begin{array}{c}
\vdash K (k_a, \langle \rangle) \\
\vdash K (n_a, n_a, \langle \rangle) \quad \text{Id} \\
\vdash K (n_b, n_c, \langle n_b \leftarrow n_c \rangle) \quad \text{Seq} \\
\vdash K (n_a; n_b, n_a; n_c, \langle n_b \leftarrow n_c \rangle) \quad \text{Enc} \\
\vdash K (\llbracket n_a; n_b \rrbracket_{k_a}, \llbracket n_a; n_c \rrbracket_{k_a}, \langle n_b \leftarrow n_c \rangle)
\end{array}
\]

Example 15 As a second example, consider now the type flaw attack on the flawed Woo Lam $\pi_1$ protocol, given in Example 2 (see Example 2). In step 3 of his strand, Bob, $b$, expects a message of the form $\llbracket a; b; n_b \rrbracket_{k_a}$, but receives $n_b$ instead. According to our definition, $b$ will accept $n_b$ provided that there is an instantiation $\sigma$ for $n_b$ and $k_{as}$ such that $\llbracket \sigma(\llbracket a; b; n_b \rrbracket_{k_a}) \rrbracket = \llbracket n_b \rrbracket$. Assuming $b$ takes the key $k_{as}$ to be $k$, then we would infer the following conditions as those under which $b$ accepts $n_b$ as a legal message:

\[
\begin{array}{c}
\vdash K (k, \langle \rangle) \\
\vdash K (a, a, \langle \rangle) \quad \text{Id} \\
\vdash K (b, b, \langle \rangle) \quad \text{Id} \\
\vdash K (a; b, a; b, \langle n_b \leftarrow n'_b \rangle) \quad \text{Seq} \\
\vdash K (a; b, a; b; m'_3, \langle n_b \leftarrow m'_3 \rangle) \quad \text{Enc} \\
\vdash K (a; b; n_b, a; b; m'_3, \langle n_b \leftarrow m'_3 \rangle)
\end{array}
\]

which holds, provided that there are messages $m', m'_1, m'_2, m'_3$ such that $\llbracket a \rrbracket = \llbracket m_1 \rrbracket$, $\llbracket b \rrbracket = \llbracket m'_2 \rrbracket$, and $\llbracket (m') \rrbracket = \llbracket (m'_1; m'_2; m'_3) \rrbracket$, $\llbracket (m') \rrbracket = \llbracket (a; b; m'_3) \rrbracket_k$ and $\llbracket (n_b) \rrbracket = \llbracket (m'_3) \rrbracket_k$ holds. $\blacksquare$

Now, an obvious misuse is that the penetrator sees an encrypted message in a protocol run and reuses it in another run, but in the same step. We can easily detect this case, by comparing the protocol sections of the node where the message is originating and the node where the message is received by the honest principal. We call such a situation a cross-protocol-confusion. Another misuse is to use some encrypted message observed in a run to camouflage another message of a different step (but possibly in the same run), because they coincide in their structure. In order to detect this new situation, we compare the attack bundle with the canonical bundle and check whether the encrypted message in the attack bundle and its counterpart in the canonical bundle originate at corresponding message positions. We call this alternative situation message confusion.

The definition below captures our intuition of the misuse of messages; it makes use of the standard notion of message position, which is an address represented by a string. Roughly, the subterm of $m$, at position $\pi$, $m|\pi$, is captured as follows: $m|\pi = m, f(m_1, \ldots, m_k)|i \cdot p = m_i|p$, where $e$ and $\cdot$ respectively stand for the empty string, and string concatenation. We use $m|\pi \rightarrow m'$ to denote the message that results from replacing $m|\pi$ with $m'$ in $m$.

Definition 12 (Confusion) Let $\mathcal{B}$ be an $\mathcal{R}$-bundle out of some $\mathcal{T}$, $\mathcal{B}^c$ be a canonical $\mathcal{R}^c$-bundle with $\mathcal{R}_H \subseteq \mathcal{R}^c$. Let $\mathcal{C}$ be an optimal coverage of $\mathcal{T}$ wrt $\mathcal{B}^c$ and $\theta$ be the corresponding canonical mapping. Then, there is a confusion in a honest node $v \in \mathcal{B}$ iff there is a position $\pi$ of $v$ with $\llbracket m \rrbracket_k = \text{msg}(v)|\pi$, originating in an honest node $v' \in \mathcal{B}$, at some position $\pi'$ of $\text{msg}(v')$, such that either:

- $v$ and $v'$ are located on strands of different protocol sections, giving rise to a cross-protocol confusion and $v \in \mathcal{B}$ causing the cross-protocol confusion with $\llbracket m \rrbracket_k$, or
- $\text{msg}(\theta(v))|\pi$ does not originate in $\theta(v')$ or $\mathcal{E}_{Q_{\mathcal{B}^c}}(\theta(v')) \not\ni \llbracket \text{msg}(\theta(v'))|\pi' \rrbracket = \llbracket \text{msg}(\theta(v))|\pi \rrbracket$, giving rise to a message confusion and $v \in \mathcal{B}$ causing the message confusion with $\llbracket m \rrbracket_k$.

Example 16 In the NSPK attack bundle (see Example 10), there is only one node that gives rise to a confusion. While both $\llbracket a; n \rrbracket_{k_a}$ and $\llbracket n'; n \rrbracket_{k_a}$ originate on a penetrator strand, $\llbracket n, n' \rrbracket_{k_a}$
originates on $\langle r_{\text{resp}}[3, \beta'], 2 \rangle$. This latter message is later received in $\langle r_{\text{init}}[3, \beta], 2 \rangle$, which belongs to a different protocol section. Hence, there is a cross-protocol confusion in $\langle r_{\text{init}}[3, \beta], 2 \rangle$. There is not a message confusion, because $\{n, n'\}_{k_a}$ originates in $\theta(\langle r_{\text{resp}}[3, \beta'], 2 \rangle) = \langle r_{\text{resp}}, 2 \rangle$ of the canonical bundle at position root. Notice also that $\emptyset \vdash \{n, n'\}_{k_a} = \{n, n'\}_{k_a}$ holds. ■

**Example 17** In the Woo Lam $\pi_1$ attack bundle (see Example [2]), the principal $b$ receives an encrypted message in $\langle r_{\text{resp}}[5, \beta], 5 \rangle$, which originates in $\langle r_{\text{resp}}[5, \beta], 4 \rangle$. This is different to the canonical bundle, where $\text{msg}(\langle r_{\text{resp}}, 5 \rangle)$ originates at $\langle r_{\text{serv}}, 2 \rangle$. Thus, there is a message confusion in $\langle r_{\text{resp}}[5, \beta], 5 \rangle$. ■

**Example 18** In the WMF protocol attack bundle (see Example [11]), there is both a message confusion and a cross-protocol confusion in $\langle r_{\text{resp}}[1, \beta'], 1 \rangle$. ■
Figure 5: The attack bundle for the Woo Lam authentication protocol
Example 19 In the attack bundle to the Woo Lam authentication protocol (see Fig. 3), there are several nodes in which honest principals receive encrypted messages. There is no confusion in \( r_{s,\text{serv}}[2, \beta], 1 \), because (i) \( \{a; n; k_c\}_{k_c} \) originates on a penetrator strand, and (ii) \( \{c; n; n'; k_{ac}\}_{k_{ac}} \) originates in \( r_{\text{Init}}[5, \beta], 3 \) in the same protocol section, and analogously to the canonical bundle. Similar arguments hold for \( r_{\text{Init}}[5, \beta], 4 \) and \( r_{\text{Resp}}[7, \beta'], 4 \).

A cross-protocol confusion occurs in \( r_{\text{Resp}}[7, \beta'], 5 \), because the encrypted message \( \{c; n; n'; k_{ac}\}_{k_{ac}} \) originates in \( r_{s,\text{serv}}[2, \beta], 2 \), which is situated in a different protocol section. It is also a message confusion. Although the encrypted message \( \{c; n; n'; k_{ac}\}_{k_{ac}} \) originates in the attack bundle at \( r_{s,\text{serv}}[2, \beta], 2 \), which corresponds to \( r_{\text{serv}}, 2 \) in the canonical bundle, the positions in which they occur differ.

5.2 Limits of Changing Messages to Protocol Fix

We now deal with the situation in which an attacker has already been able to trick an honest principal into accepting a message \( m' \), instead of an expected message, \( m \), for a given protocol. Therefore, we want to change the protocol in such a way that the attacker is no longer able either to construct \( m' \) from a collection of eavesdropped messages, or to pass \( m' \) as a camouflaged message for \( m \). To formalize this idea, we first introduce the following definitions.

Definition 13 (Agents Knowledge) Let \( s \) be a strand of a bundle \( B \). Further, let \( TK_{\text{Init}} \) and \( K_{\text{Init}} \) respectively be the set of atomic messages in \( s \) and the set of keys, initially known to the agent of \( s \). In particular, \( TK_{\text{Init}} \) includes all atomic messages that are originating in \( s \).

Given a node \( (s, i) \) of \( s \) then \( \text{pred}(s, i) \) is the closest negative predecessor node of \( (s, i) \), if any, and it is undefined, otherwise; i.e. if \( \forall 1 \leq j < i. \text{sign}(s, j) = + \).

The set of known keys \( K(s, i) \) in a negative node \( s, i \), with \( i \leq \#(s) \), is defined by:

\[
K(s, i) = \begin{cases} 
K_{\text{Init}} & \text{if } \text{pred}(s, i) \text{ undefined}, \\
K_{\text{pred}}(s, i) \cup (\text{analz}_{K_{\text{pred}}(s, i)}((s, i))) & \text{otherwise.}
\end{cases}
\]

and the set of known atomic messages \( TK(s, i) \) in a negative node \( s, i \), with \( i \leq \#(s) \), by:

\[
TK(s, i) = \begin{cases} 
TK_{\text{Init}} & \text{if } \text{pred}(s, i) \text{ undefined}, \\
TK_{\text{pred}}(s, i) \cup TK(\text{analz}_{K_{\text{pred}}(s, i)}((\{\text{msg}(s, i)\}))) & \text{otherwise.}
\end{cases}
\]

Now, if a protocol has to be changed in order to avoid confusions, there are two opposing requirements. On the one hand, we would like to add additional information into individual messages to allow the recipient to distinguish the original from the fake message. But, on the other hand, we would not like to change the intended semantics of the protocol. Our approach is not based on any explicit semantics of the protocols under consideration, but the semantics is only implicitly given by the syntactical design of the protocol. When changing steps of the protocol syntactically, we ought to make sure that the implicit semantics of the protocol is not changed either. We try to achieve this goal by changing individual protocol steps only using conservative extensions and reshuffles of messages.

Suppose, for example, that to patch a protocol, we need to modify an encrypted term, \( t \). Clearly, upon the reception of \( t \), a principal can decompose it into the subterms that are visible under some known keys, \( K \). When patching a protocol by modifying \( t \) into \( t' \), we further require that, under \( K \), \( t' \) exposes at least the same information to the principal as \( t \).

Definition 14 (Message Substitution) A message substitution \( \sigma \) is a finite list \( \langle \{t_1\}_{k_1} \leftarrow \{t'_1\}_{k'_1}, \ldots, \{t_n\}_{k_n} \leftarrow \{t'_n\}_{k'_n} \rangle \) of pairs of encrypted messages, such that \( \{t_i\}_{k_i} = \{t_j\}_{k_j} \implies i =
j. As expected, we define \( \text{Dom}(\sigma) = \{ \|t_1\|_{k_1}, \ldots, \|t_n\|_{k_n} \} \). Given \( t \in A \), \( \sigma(t) \) is defined by:

\[
\sigma(t) = \begin{cases} 
  t & \text{if } t \in \text{TK} \\
  \sigma(t_1); \sigma(t_2) & \text{if } t = t_1; t_2 \\
  \|\sigma(t')\|_{k_1} & \text{if } t = \|t_1\|_{k_1}, \|t_2\|_{k_1} \in \text{Dom}(\sigma) \\
  \|\sigma(t')\|_{k_2} & \text{if } t = \|t'\|_{k_2}, \|t''\|_{k_2} \notin \text{Dom}(\sigma)
\end{cases}
\]  

We extend message substitution \( \sigma \) also to sets \( M \) of messages by \( \sigma(M) = \{ \sigma(t) \mid t \in M \} \).

**Example 20** Let \( \sigma = \{ \|A; B\|_K \} : \|A; C\|_K ; \|B\|_K \leftarrow \|B; D\|_K \} \) then

\[
\sigma((\|A; B\|_K) ; \|B\|_K) = \{ \|A; C\|_K ; \|B; D\|_K \}
\]

**Definition 15** A message substitution \( \sigma \) is information enhancing wrt. \( M \) of messages iff \( \forall \|t_i\|_k \in \text{Dom}(\sigma). \exists M' \subseteq M. \text{fl}(t_i) = \text{fl}(t_i) \cup M' \). A message substitution \( \sigma \) is injective on a set \( M \) of messages iff \( \forall t, t' \in M. \sigma(t) = \sigma(t') \implies t = t' \).

**Example 21** The message substitution \( \sigma \) in Example 20 is information enhancing wrt. \( \{C, D\} \).

**Definition 16** Let \( \sigma_1, \sigma_2 \) be two message substitutions with \( \text{Dom}(\sigma_1) \cap \text{Dom}(\sigma_2) = \emptyset \). Then the co-substitutions \( \overline{\sigma}_1 \) and \( \overline{\sigma}_2 \) of \( \sigma_1, \sigma_2 \) are defined by:

\[
\overline{\sigma}_1 = \{ \sigma_2(t) \leftarrow \sigma_2(t') \mid t \leftrightarrow t' \in \sigma_1 \} \quad \text{and} \quad \overline{\sigma}_2 = \{ \sigma_1(t) \leftarrow \sigma_1(t') \mid t \leftrightarrow t' \in \sigma_2 \}
\]

**Lemma 17** Let \( \sigma_1 \) and \( \sigma_2 \) be two information enhancing message substitutions wrt. \( M_1 \) and \( M_2 \), respectively, such that \( \text{Dom}(\sigma_1) \cap \text{Dom}(\sigma_2) = \emptyset \). Then, the co-substitutions \( \overline{\sigma}_1 \) and \( \overline{\sigma}_2 \) of \( \sigma_1 \) and \( \sigma_2 \) are information enhancing wrt. \( \text{Dom}(\sigma_1(M_1)) \) and \( \text{Dom}(\sigma_2(M_2)) \), respectively.

**Proof** We prove that \( \overline{\sigma}_2 \) is information enhancing wrt. \( \sigma_1(M_2) \); the proof for \( \overline{\sigma}_1 \) is analogous. Let \( \overline{\sigma}_2 = \{ \{s_1\}^{k_1}_k \leftarrow \{s_1'\}^{k_1}_k, \ldots, \{s_n\}^{k_n}_k \leftarrow \{s_n'\}^{k_n}_k \} \) then \( \overline{\sigma}_2 = \{ \sigma_1(\{s_1\}^{k_1}_k), \ldots, \sigma_1(\{s_n\}^{k_n}_k) \} \leftarrow \sigma_1(\{s_1'\}^{k_1}_k), \ldots, \sigma_1(\{s_n'\}^{k_n}_k) \}. Since \( \sigma_2 \) is information enhancing wrt. \( M_2 \), we know that \( \text{fl}(s_i') = \text{fl}(s_i) \cup M_2 \) and therefore also \( \sigma_1(\text{fl}(s_i')) = \sigma_1(\text{fl}(s_i)) \cup M_1 \). Because \( \sigma_1 \) commutes over flattening, we obtain: \( \text{fl}(\sigma_1(\{s_i\})) = \text{fl}(\sigma_1(\{s_i\})) \cup M_1 \). \( \square \)

**Lemma 18** Let \( \sigma_1, \sigma_2 \) be two message enhancing message substitutions wrt. \( M_1 \) and \( M_2 \), respectively, such that \( \text{Dom}(\sigma_1) \cap \text{Dom}(\sigma_2) = \emptyset \). Let \( M \subseteq A \) be a set of messages, such that \( \sigma_1 \) and \( \sigma_2 \) are injective on \( M \). Then there are information enhancing message substitutions \( \overline{\sigma}_1 \) and \( \overline{\sigma}_2 \) wrt. \( \text{Dom}(\sigma_1(M_1)) \) and \( \text{Dom}(\sigma_2(M_2)) \) such that

\[
\forall t \in M. \overline{\sigma}_1(\sigma_2(t)) = \sigma_2(\overline{\sigma}_1(t))
\]

**Proof** Let \( \overline{\sigma}_1 \) and \( \overline{\sigma}_2 \) be the corresponding co-substitutions of \( \sigma_1 \) and \( \sigma_2 \). By Lemma 17 we know that \( \overline{\sigma}_1 \) and \( \overline{\sigma}_2 \) are information enhancing message substitutions wrt. \( \text{Dom}(\sigma_1(M_1)) \) and \( \text{Dom}(\sigma_2(M_2)) \), respectively.

We prove the push-out by structural induction on the term \( t \): Since \( \sigma_1 \) and \( \sigma_2 \) are injective on \( M \), we know that \( t \notin \text{Dom}(\sigma_1) \) implies \( \sigma_2(t) \notin \text{Dom}(\overline{\sigma}_1) \), and \( t \notin \text{Dom}(\sigma_2) \) implies \( \sigma_1(t) \notin \text{Dom}(\overline{\sigma}_2) \) for all \( t \in M \).

- Let \( t \in \text{TK} \) then \( \overline{\sigma}_1(\sigma_2(t)) = t = \sigma_2(\sigma_1(t)) \).
- Let \( t = t_1; t_2 \) then \( \overline{\sigma}_1(\sigma_2(t_1; t_2)) = \overline{\sigma}_1(\sigma_2(t_1)) \cdot \overline{\sigma}_1(\sigma_2(t_2)) = \overline{\sigma}_2(\sigma_1(t_1); \sigma_1(t_2)). \)
- Let \( t \in \text{Dom}(\sigma_1) \). Hence, \( t = \|t_1\|_{k_1} \) and \( \sigma_1(t) = \|t_1'\|_{k_1} \). Furthermore, \( t \notin \text{Dom}(\sigma_2) \) and therefore \( \sigma_1(t) \notin \text{Dom}(\overline{\sigma}_2) \). Then \( \overline{\sigma}_2(\sigma_1(t)) = \overline{\sigma}_2(\|t_1'\|_{k_1}) = \|\overline{\sigma}_2(\sigma_1(t))\|_{k_1} = \overline{\sigma}_1(\|\overline{\sigma}_2(\sigma_1(t))\|_{k_1}) = \sigma_1(\sigma_2(t_1)) \leftarrow \sigma_1(\sigma_2(t_1)) \cdot \sigma_1(\sigma_2(t_1)). \)
- Let \( t \in \text{Dom}(\sigma_2) \). Analogous to the previous case.
• Let \( t = \{t_i\}_k \notin \text{Dom}(\sigma_1) \cup \text{Dom}(\sigma_2) \). Then \( \sigma_2(\sigma_1(t)) = \sigma_2(\{\sigma_1(t)\}_k) = \{\sigma_2(\sigma_1(t))\}_k = \{\sigma_1(\sigma_2(t))\}_k = \sigma_1(\{\sigma_2(\sigma_1(t))\}_k) = \sigma_1(\{\sigma_1(\sigma_2(t))\}_k) \). □

**Definition 19 (Message Enhancement)** Let \( m, m' \) be two messages and \( M \) be a set of atomic messages. \( m' \) is an enhancement of \( m \) wrt. \( M \), written \( m' \geq_M m \) for short, iff there is an information enhancing message substitution \( \sigma \) wrt. \( M \), such that \( \sigma(m) = m' \) holds.

We write \( m' \geq_{M, \sigma} m \) to identify the corresponding information enhancing message substitution \( \sigma \) with \( \sigma(m) = m' \).

We say that \( m \) is equivalent to \( m' \), in symbols \( m \equiv m' \), iff \( m \geq_B m' \). Furthermore, \( m >_{M, \sigma} m' \) iff \( m \geq_{M, \sigma} m' \) and \( m \neq m' \).

**Example 22** \( \{a; b\}_k \equiv \{m; a; b\}_k \).

We shall also demand that the changed message be dissimilar from an entire set of messages, e.g. from those used in one or more protocols.

**Definition 20 (Collision Freeness)** Let \( m \) be a message and \( M \subseteq A \) be a set of messages. Then, \( m \) is collision free with respect to \( M \) iff for all \( m' \in M \) it is the case that \( m \neq m' \).

**5.3 Protocol Repair**

We have seen that we find bugs in protocols by analyzing a bundle containing a penetrator strand. Often, our analysis suggests a change in the structure of a message, and, in that case, it 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 are then propagated 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 caused by a change in a particular node.

**Definition 21 (Adaptation)** Let \( B \) be a bundle, \( v_0 \) be a positive node in \( B \) and \( \sigma \) be a message substitution. The adaptation \( Ad(B, v_0, \sigma) \) of \( B \) wrt. \( v_0 \) and \( \sigma \) is a graph \( B' \) that is isomorphic to \( B \); we shall use \( \zeta \) to denote the corresponding isomorphism that maps the nodes of \( B \) to nodes of \( B' \). The message of each node in \( B' \) is defined by \( \text{msg}(\zeta(v)) = \text{msg}(v) \) if \( v_0 \not\leq_B v \) and \( \text{msg}(\zeta(v)) = \sigma(\text{msg}(v)) \) otherwise.

Suppose \( B \) is an \( R \)-bundle, then the adaptation \( Ad(B, v_0, \sigma) \) can be considered as a canonical \( R' \)-bundle where \( R' \) is the set of renamed strands embedded in \( Ad(B, v_0, \sigma) \) (c.f. Section 5.1.1).

**Definition 22** An adaptation \( Ad(B, v_0, \sigma) \) of a canonical bundle is safe iff \( \text{Dom}(\sigma) \) originates in \( v_0 \) and \( \sigma \) is information enhancing.

**Lemma 23 (Confluence of Adaptations)** Let \( B \) be a bundle, \( v_1, v_2 \) nodes of \( B \) and \( \sigma_1, \sigma_2 \) information enhancing message substitutions such that \( \text{Dom}(\sigma_1) \cap \text{Dom}(\sigma_2) = \emptyset \) and \( Ad(B, v_i, \sigma_i) \) (\( i = 1, 2 \)) are safe adaptations. Let \( \zeta_i \) be their corresponding node isomorphisms then

\[
Ad(Ad(B, v_1, \sigma_1), \zeta_1(v_2), \sigma_2) = Ad(Ad(B, v_2, \sigma_2), \zeta_2(v_1), \sigma_1)
\]

holds where \( \sigma_1, \sigma_2 \) are the corresponding co-substitutions of \( \sigma_1, \sigma_2 \).

**Proof** Let \( \zeta_1 \) (and \( \zeta_2 \)) be the corresponding node isomorphisms for \( Ad(Ad(B, v_1, \sigma_1), \zeta_1(v_2), \sigma_2) \) (and \( Ad(Ad(B, v_2, \sigma_2), \zeta_2(v_1), \sigma_1) \), respectively). We show that \( \text{msg}(\zeta_1(\zeta_2(v))) = \text{msg}(\zeta_2(\zeta_1(v))) \) holds for each node \( v \) of \( B \). Obviously \( v_k \not\leq_B v \) implies \( \zeta_i(v_k) \not\leq_B \zeta_i(v) \) and also \( \zeta_j(\zeta_i(v_k)) \not\leq_B \zeta_j(\zeta_i(v)) \) for \( i, j \) and \( k = 1, 2 \) and \( i \neq j \).

Let \( v \) an arbitrary node of \( B \). We do a case analysis on the position of \( v \) relative to \( v_1 \) and \( v_2 \):

• Let \( v_1 \not\leq_B v \) and \( v_2 \not\leq_B v \). Then, \( \text{msg}(\zeta_1(\zeta_2(v))) = \text{msg}(\zeta_2(\zeta_1(v))) \).
6 Patch Methods for Protocols

In their seminar paper, Abadi and Needham proposed general guidelines for a protocol design, which cope with the problem of disambiguating messages in a protocol. In particular, the principle 3, agent naming, prescribes that all agent names relevant for a message should be derivable either from the format of the message or from its content. This prevents the reuse of a message in the corresponding step of a different run of the same protocol. Principle 10, recognizing messages and encodings, deals with the second source of misuse and demands that a principal should be able to associate which step a message corresponds to. These principles will result in different ways to fix a flawed protocol.

We shall now discuss various rules to fix a faulty protocol. As already illustrated in Section 2, we use a standard protocol analyzer, like AVISPA, to generate an example of an interleaving of protocol runs in which a honest principal has been spoofed. This example is translated into the strand space notation forming an attack bundle $B$. Additionally, we translate the intended protocol run into a canonical bundle $B^c$. Comparing both bundles provides the necessary information where and how we have to change the protocol. The first step is to construct a coverage for $B$, representing the different views of the honest principals on the protocol runs of the attack. The ultimate goal is to remove all confusions occurring in $B$ by small changes of individual protocol steps. Therefore, we will analyze the various sources for confusions and how they can be avoided by patching the protocol. The type of patch rule used depends on the type of confusion (c.f. Definition 12).

A rule (to patch a protocol) is a patch method, a tuple comprising four elements: i) a name, ii) an input, iii) preconditions, and iv) a patch. The first element is the name of the method, a string, meaningful to the flaw repair performed by the method. The second element is the input: a bundle $B$ describing the attack, a canonical bundle $B^c$ describing the intended run of the protocol, and a distinguished node in $B$ causing a confusion. The third element 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. Finally, the fourth element is the patch, a procedure specifying how to mend the input protocol.

6.1 Patching Protocols with a Message Confusion Flaw

Suppose that a given attack bundle $B$ has a message confusion in some node $v \in B$. This means that a honest principal accepted a message in some step of the protocol that was faked by the penetrator. Since the confusion is fatal, the penetrator reused some encrypted message from another message (possibly from a different protocol step) that he could not construct himself. Hence, the problem is that the encrypted messages within the reused message can be confused with the encrypted message that is part of the expected message. To fix this vulnerability we have to change one of these messages to break the similarity. The simplest way of doing this is to rearrange the information stored in the message. For instance, we may reverse messages of a concatenation occurring in $\text{msg}(v)$. In general, we search for a message $m$ such that $\text{msg}(v) \equiv m$, but $\text{msg}(v) \neq m$. In some cases, and in particular if $\text{msg}(v)$ simply is an encrypted atomic message, there are no messages $m$ satisfying these conditions, and we have to extend $\text{msg}(v)$ with an additional information bit, typically a tag, to resolve the confusion.
Definition 24 (Message-Encoding Rule) The message-encoding rule is the following transformation rule:

Input: \( B^c, B \) with a node \( v \in B \) causing a message confusion with a message \( t = \text{msg}(v) \).

Preconditions:
1. \( \text{msg}(\theta(v)) \) originates at \( v' \) in \( B^c \) at position \( \pi' \).
2. \( v \) belongs to a protocol section with renaming \( \beta \).

Patch:
1. Let \( t' \) be a message and \( B^c \) a graph such that
   
2. \( \sigma = \{ \text{msg}(v') | \pi' \rightarrow t' \} \) is an information enhancing message substitution wrt. \( \beta \).
3. \( t' \) is collision free wrt. \( \text{msg}(v') \) for all nodes \( v' \) in \( B^c \), and
4. \( \text{msg}(v) \) is not accepted for \( \beta(\sigma(\text{msg}(\theta(v)))) \) wrt. \( \beta \) and the known keys \( K_v \) in \( v \).

Then, return \( \text{Ad}(B^c, v', \sigma) \) as a result of the patch.

Example 23 Consider again WMF, which violates Abadi and Needham’s design principle 10. As illustrated in the attack of Example 11, Alice receives the encrypted message \( t \) responding message \( v \in \{ | \} \) only if the messages are equal. We obtain a suitable message \( t \) with \( t \) tk such that

\[ A_B(t) \]

The resulting bundle \( A_B(t) \) looks as follows: (cf. the original definition, given in Example 11):

\[ +a, \{ b; t_a; k \} \leftarrow \rightarrow -a, \{ b; t_a; k \} \]

\[ \text{Init} \quad \text{Serv} \quad \text{Resp} \]

Example 24 In case of the Woo Lam \( \pi_1 \) protocol (see Example 2), there is a message confusion in \( v = \{ r_{\text{resp}}[5, \beta], 5 \} \) with \( t = \{ a; b; n_{b} \} \) and \( t \) originates in \( \{ r_{\text{resp}}[5, \beta], 4 \} \). while in the canonical bundle the corresponding message \( \{ a; b; n_{b} \} \) originates in \( v' = \{ r_{\text{Serv}}, 2 \} \).

Again, we obtain a suitable message \( t' = b; a; n_{b} \) by permuting the messages of \( t \) such that \( \sigma = \{ t \rightarrow t' \} \) is an information enhancing message substitution wrt. \( \beta \). \( t' \) is obviously collision free with all nodes of the canonical bundle \( B^c \) and \( \{ b; t_a; k \} \) is not accepted for \( \{ t_a; b; k \} \) wrt. \( \{ t_a, b \} \) and \( \{ k_{a,b} \} \).

Thus, the resulting bundle \( \text{Ad}(B^c, v', \sigma) \) looks as follows:

\[ +a \quad \rightarrow \quad -a \]

\[ \text{Init} \quad \text{Resp} \quad \text{Serv} \]
Notice that the intruder can elaborate a type flaw attack on a protocol whenever he is able to make a protocol principal accept a message of one type, \( m \), as a message of another, \( m' \). In that case, \( m \) and \( m' \) share the same implementation, \( m \approx m' \). Thus, to remove the protocol type flaw, it suffices to break \( m \approx m' \). So message-encoding will attempt this by rearranging \( m \)'s structure while keeping its meaning intact. If this operation does not suffice to break the confusion between \( m \) and \( m' \), it will then insert into \( m \) vacuous terms, tags actually, as in [20].

Finally, when incurring on these changes, our methods ensure that the new protocol message does not clash with some other.

### 6.2 Patching Protocols with a Cross-Protocol Confusion Flaw

Protocol guarantees are usually implemented via cyphertexts (c.f. the rationale behind an authentication test [18]). To realize authentication, the name of the agents that are relevant for the intended consumption of a cyphertext, namely the originator and the intended recipients, should be all derivable from the cyphertext itself (c.f. principle 3 for protocol design of Abadi and Needham [11]). When this is not the case, the associated message, and the protocol itself, has agent naming problems.

When an agent naming problem occurs, a penetrator can reuse the message corresponding to the \( i \)-th step of the protocol in some run, to camouflage that of the \( i \)-th step but of another run. Thus, the penetrator reuses one or more associated message cyphertexts so as to impersonate the corresponding originator or as to redirect them to an unintended recipient. From a technical point of view, such a message lacks sufficient information about the particular protocol run it was generated for.

We introduce a method designed to fix a faulty protocol with a cross-protocol confusion flaw. We fix this flaw inserting the names of the correspondents that are not explicitly mentioned in the message that has been reused to carry out the attack.

**Definition 25 (Agent-Naming Rule)** The agent-naming rule is the following transformation rule:

**Input:** \( B^c, B \) with a node \( v \in B \) causing a cross-protocol but not a message confusion with a message \( t \).

**Preconditions:**
- \( t \) originates in \( v' \in B \) at position \( \pi \),
- \( v \) belongs to a protocol section with renaming \( \beta \), and
- \( v' \) belongs to a protocol section with renaming \( \beta' \), with \( \beta \neq \beta' \).

**Patch:**
- Let \( t' \) be a message such that
  - \( \sigma = \{ \text{msg}(\theta(v')) | \pi \leftarrow t' \} \) is an information enhancing message substitution wrt. \( \{ m \in TK_{\theta(v')} | \beta'(m) \neq \beta(m) \} \),
  - \( t' \) is collision free wrt. \( \text{msg}(v'') \) for all nodes \( v'' \in B^c \).

Then, return \( Ad(B^c, \theta(v'), \sigma) \) as a result of the patch.

**Example 25** Consider the NSPK protocol in Example [10]. There is a cross-protocol confusion (but not a message confusion) in \( r_{\text{inst}}[3, \beta], 2 \) with \( t = \{ n; n' \}_{k_a} \). The different protocol sections have the following renamings: \( \beta = \{ a \leftarrow a, b \leftarrow c, n \leftarrow n, n' \leftarrow n', k_a \leftarrow k_a, k_b \leftarrow k_c \} \) and \( \beta' = \{ a \leftarrow a, b \leftarrow b, n \leftarrow n, n' \leftarrow n', k_a \leftarrow k_a, k_b \leftarrow k_b \} \). Thus, \( \pi \) is empty and \( \text{msg}(\theta(v')) | \pi = \{ n; n' \}_{k_a} \).

Obviously, \( \beta \) and \( \beta' \) differ only in their renaming of the atomic message \( b \), which results in a patched message \( t' = \{ n; n' \}_{k_a} \) that is to be expected in \( r_{\text{inst}}[3, \beta], 2 \). Therefore, we adjust the protocol using \( \sigma = \{ \{ n; n' \}_{k_a} \leftarrow \{ n; n' \}_{k_a} \} \) and obtain the fixed protocol (cf. the original protocol).
definition, given in Example 6 by $Ad(B^c, \theta(v'), \sigma)$:

\[
\begin{align*}
+i &|a; n|_k & \quad \Rightarrow \quad & -i |a; n|_k \\
\downarrow & & \downarrow & \\
-|n; n'; b|_k & \quad \Leftrightarrow \quad & +|n; n'; b|_k \\
\downarrow & & \downarrow & \\
+|n'|_k & \quad \Rightarrow \quad & -|n'|_k \\
Init & & Resp \\
\end{align*}
\]

6.3 Patching Protocols with a Replay Protection Flaw

The agent-naming rule fails to patch a protocol if the set $\{m \in TK_{v} \mid \beta(m) \neq \beta'(m)\}$ is empty, which means that there are two identical copies of an honest strand in the penetrator bundle contributing to different protocol runs. In other words, the behavior of this principal is deterministic with respect to the messages received from its environment. This gives rise to a multiplicity attack in which the penetrator simply replays a communication from a (pre-)recorded protocol run, and causes an agent to consider that somebody is trying to set up a simultaneous session, when he is not [26].

The session-binding rule deals with faulty protocols that contain this kind of flaw, called replay protection. Two example faulty protocols of this kind are WMF (c.f. Example 7) and DSSK (c.f. Example 9) protocol. None of these protocols satisfies strong authentication protection [26].

In both cases, the responder $b$ participates in the protocol only in a passive way, by receiving some message. Unless the responder stores details about each protocol run, it cannot distinguish copies of such a message replayed by an attacker, from genuine messages in independent protocol runs. Technically speaking, there are no atoms that originate on the responder strand allowing it to actively differentiate individual protocol runs. SHRIMP is equipped with a repair method that introduces a nonce-flow requirement to fix this flaw [1, 35, 26, 18] (c.f. principle 7 for protocol design of Abadi and Needham.) The idea is to use some handshake or challenge-response approach to involve the responder actively in the protocol.

**Definition 26 (Session-Binding Rule)** The session-binding rule is the following transformation rule:

**Input:**

$B^c, B$ with a node $v \in B$ causing a cross-protocol but not a message confusion with a message $|m|_k$.

**Preconditions:**

i) $v$ belongs to a protocol section $T$ with renaming $\beta$,

ii) $v'$ belongs to a protocol section $T'$ with renaming $\beta$ and $T' \neq T$, and

iii) $|m|_k$ originates in $v' \in B$ at position $\pi$.

**Patch:** Introduce a challenge-response between the strand $s$ of $\theta(v)$ and the strand $s'$ of $v''$, where $v''$ is a minimal element wrt. $\preceq_{B'}$ in $\{s \mid s \preceq_{B'} \theta(v)\}$.

Return a bundle $B' = \langle V', (\neg \Rightarrow \cup \Rightarrow) \rangle$ with:

i) $V' = V_{B'} \cup \{v_1, v_2, v_3, v_4\}$,

ii) $\Rightarrow = \Rightarrow_{B'} \cup \{(v_1, v_3), (v_2, v_4)\}$,

iii) $\Rightarrow = \Rightarrow_{B'} \cup \{(s, \#s), (v_1, v_2), ((s', \#s'), (v_3, v_4))\}$,

iv) $msg(v_1) = msg(v_3) = \langle Agent(v_1); Agent(v_1); n \rangle_k$, and

v) $msg(v_2) = msg(v_4) = \langle f(n); Agent(v_1); Agent(v_1) \rangle_{k-1}$,

where $n$ is a nonce originating in $v_1$, $f$ is an arbitrary injective function on nonces, and either $k = k^{-1} \in K_{(s, \#s)} \cap K_{(s', \#s')} \setminus K_P$, or $k$ and $k^{-1}$ are the public and private key of $Agent(v_1)$. 

26
Consider the patched Wide-Mouth-Frog protocol in Example 23. After eliminating the message confusion there is still an attack possible as illustrated in the following penetrator bundle. As Lowe [26] has described the attack, the penetrator impersonates the server and replays the message confusion there is still an attack possible as illustrated in the following penetrator bundle. Since the newly introduced nonce originates in the server’s response making the responder to believe that a second session has been established. As another example consider the attack on the DSSK protocol in Example 12. Again protocol confusion in the first node of the second responder strand with the message confusion there is still an attack possible as illustrated in the following penetrator bundle. Obviously, there is a cross-protocol confusion in the first node of the second responder strand with the message confusion there is still an attack possible as illustrated in the following penetrator bundle. This prevents the application of the agent-naming rule and enables the application of the session-binding rule.

```
Init     Serv     Spy     Resp
```

Obviously, this patch solves the problem of having identical copies of the same honest strand in a penetrator bundle. Since the newly introduced nonce originates in s, it also uniquely originates in a faithful realization of the protocol; i.e. the honest principal will generate individual (i.e. different) nonces for each individual protocol run.

Example 26 Consider the patched Wide-Mouth-Frog protocol in Example 23. After eliminating the message confusion there is still an attack possible as illustrated in the following penetrator bundle. As Lowe [26] has described the attack, the penetrator impersonates the server and replays the message confusion there is still an attack possible as illustrated in the following penetrator bundle. Since the newly introduced nonce originates in the server’s response making the responder to believe that a second session has been established.

```
+ a, \{ a; t_a + d; k \}_k_a \rightarrow - a, \{ a; t_a; b; k \}_k_a \\
\downarrow \\
+ \{ a; t_a + d; k \}_k_a \rightarrow - \{ a; t_a + d; k \}_k_a \\
\downarrow \\
- \{ a; n_b; a \}_k \\
\downarrow \\
+ \{ n_b + 1; b; a \}_k \\
```

Example 27 As another example consider the attack on the DSSK protocol in Example 12. Again we have two identical copies of the responder strand. Similar to the previous example, there is cross-protocol confusion in the first node of the second responder strand with the message confusion there is still an attack possible as illustrated in the following penetrator bundle.

```
Resp     Init     Serv
```

 Again, (Init, 1) is the only \( \preceq_{B_k} \)-minimal element of \( \{ \tau | \tau \preceq_{B_k} \langle \text{Resp}, 1 \rangle \} \) and we choose to use the increment by one as \( f \). With \( K_{\langle \text{Init}, 1 \rangle} \cap K_{\langle \text{Resp}, 1 \rangle} \setminus K_P = \{ k, k_{ab} \} \cap \{ k, k_{bs} \} = \{ k \} \). Hence, we select \( k \) as the encryption key for the challenge response and we obtain the following patched canonical bundle:

```
Init     Serv     Resp
```

```
+ a, \{ a; t_a; b; k \}_k_a \rightarrow - a, \{ a; t_a; b; k \}_k_a \\
\downarrow \\
+ \{ a; t_a + d; k \}_k_a \rightarrow - \{ a; t_a + d; k \}_k_a \\
\downarrow \\
- \{ a; t_a + d; k \}_k_a \\
\downarrow \\
- \{ a; n_b; a \}_k \\
\downarrow \\
+ \{ a; n_b + 1; b; a \}_k \\
```
6.4 Combining Rules

As already mentioned in Section 5.3, our approach runs a protocol verifier and uses its output, in particular the faulty protocol run, to compute the attack bundle and to select an appropriate patch rule to fix the protocol. We apply one rule at a time and resubmit the patched protocol to the protocol verifier. This process iterates until either no more flaws are detected by the verifier, or there are no more applicable rules. Since this approach constitutes some sort of rewrite system on security protocols, it is tempting to analyze formal properties of rewrite systems, like confluence and termination in this setting.

Suppose there is a faulty security protocol containing various flaws. In the first place, the order in which SHRIMP examines the flaws depends on the order in which attacks on the security protocol are discovered by the protocol verifier. Given a particular attack bundle, we may have different messages that allow the penetrator to mount the attack. Concerning a particular message, at most one of the rules message-confusion, agent-naming and session binding would be applicable, because the preconditions of the three rules exclude one another. As far as an attack makes use of a faulty implementation and the attack bundle involves I-traces, there is a choice point between patching the implementation, such that the type confusion is no longer applicable or patching the protocol itself so that any type confusion does not cause any harm. Both patches are independent of each other, since they operate on different levels. If the attacker bundle reveals several flaws, i.e. there are various messages that can be misused to mount an attack, then, lemma 18 guarantees that rules based on adaptations like the message-confusion and the agent-naming rules commute.

There is no strong argument concerning the termination of SHRIMP, although in all our examples (see Section 7), it was never the case that our approach did not terminate. One reason is that rules working on ambiguous messages will disambiguate them with respect to the entire bundle. Therefore, the number of potentially ambiguous messages in a protocol will decrease with each application of such a rule.

7 Results

| Protocol       | Source | Attack Source |
|----------------|--------|---------------|
| Woo-Lam τ₁     | 41     | 20            |
| Otway-Rees     | 9      | 9             |
| Neuman-Stubblebine | 33     | 12            |
| KP             | 36     | 36            |
| NSLPK          | 20     | 20            |
| BAN Yahalom    | 9      | 9             |
| GDOI           | 31     | 31            |

Table 1: Experimental results: protocols repaired by hand, part II

Tables 2 and 1 summarize our results. We considered 40 experiments, of which 20 involve protocols borrowed from the Clark-Jacob library, 4 are variants of some of these protocols (annotated with *); 5 were borrowed from the literature (these 5 protocols are all known to be susceptible to a type flaw attack); and 11 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. Protocol verification was carried out using AVISPA.

Table 2 portrays information about protocols that were fixed fully automatically. 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 $wa_r < sa_r$).

---

6 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.
Table 2: Experimental results: protocols repaired automatically, part I. Columns before and after are used to convey the properties that the associated protocol satisfies, (T)rue and (F)alse (X) means property was not tested.) s stands for secrecy, wa for weak authentication, and sa stands for strong authentication. The subscripts i and r denote the initiator and the responder, respectively.
While secrecy has a definite meaning, authentication is not; however, Lowe’s hierarchy of authentication [27] has become the standard reference in the literature. Not surprisingly, AVISPA’s levels of authentication, namely: weak authentication, and (strong) authentication, correspond to Lowe’s. We refer the reader to [27] for the precise meaning of these terms.

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 $\phi$ might be either $T$, meaning $P$ satisfies $\phi$, $F$, meaning $P$ does not satisfy $\phi$, 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. 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 †.)

SHRIMP is thus able to automatically identify a flaw and a successful candidate patch in 33 faulty protocols. Of these experiments, it applied 12 times agent-naming, 5 times message-encoding, and 17 times session-binding. Notice that SHRIMP was able to repair 18 protocols of the 20 that were borrowed from Clark and Jacob. The other two protocols could not be corrected automatically, since we are currently extending SHRIMP so as to incorporate our extension of the strand spaces to fully account for the theory presented in this paper. So, the faulty protocols in Table 1 have been tested by hand only.

Further development work is also concerned with automatically translating a protocol (respectively an attack) from a strand space notation to AVISPA HLPSL, and back. While most of these translation can be automated, we anticipate human intervention will be required for the formulation of security goals, using AVISPA’s special predicates, namely: witness, (weak) authentication, and authentication on.

We have made SHRIMP try to patch the IKEv2-DS protocol, which is part of AVISPA’s library and an abstraction of IKEv2. We found that, upon an abstraction of 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 session binding suggesting a protocol similar to IKEv2-DSx, which also is part of AVISPA’s library and attack-free.

8 Related Work

We now proceed to compare our method against techniques that are rival in the sense outlined in Section 1.

8.1 On Protocol Repair

R. Choo [13] has also looked at the problem of automated protocol repair. His development framework applies the SHVT model checker [34] to perform a state-space analysis on a (indistinguishableness-based) model of the protocol (encoded using asynchronous product automata) under analysis. If the protocol is faulty, Choo’s framework will first check if the associated attack is captured in the database of attack classes, and then will apply the repair. Unfortunately, the attack classes are not formalized, and it is not clear whether for each class there is an associated repair method. [13] introduces only one repair method; the method indiscriminately inserts the names of all the participants involved in every cyphertext of the protocol. By way of comparison, for each attack class we provide a formal characterization, amenable to mechanization, and an specific repair strategy.
What is more, if applied, our naming repair mechanism will selectively introduce only the agent names required to rule out the attack. Also, the class of replay attacks, fully captured by our methods, comprehends the attack classes involved in [13].

8.2 On Protocol Compiling

The compiling approach to protocol development takes a protocol that is weak, in some security sense, and returns others that is stronger, and made out of modifications of the input one. Example methods in this vein include [22, 16, 2]. They are all based on the seminal work of Canetti on universal composition, which we describe first as it will be the starting point of our comparison.

Canetti’s Universal composition theorem [11] is used to deduce the security of a complex system, from a proof of the security of its constituents. For security protocol development, composition theorems are very attractive, since they assert that if a protocol is secure, when considered in isolation, then it will remain secure, even if simultaneously run an unbounded number of times. In particular, [11] has shown that for universal composition of a protocol to hold, it suffices that each of its runs is independent, and that run independency can be guaranteed prefixing a pre-established Session ID (SID) into every plaintext message, which is to be subject of a cryptoprimitive (encryption or signing) by the protocol. We call this transformation SID session tagging. Although Canetti provides various ways of forming and pre-establishing a SID, it is common for a SID to be set the concatenation of the participants’ names, together with the fresh nonces such participants have all previously generated, and exchanged one another (thus the name pre-established SID).

Canetti’s proof hinges on the assumption of state disjointness across all instances of a protocol, thus, ruling out the possibility of using, for example, long-term keys. Accordingly, Canetti and Rabin [10], later on, showed that Canetti’s results still holds for an, albeit limited, shared randomness, including long-term keys. Put differently, SID session tagging is enough to provide state disjointness.

Yet, Küsters and Tuengerthal [24] have recently shown that SID session tagging is quite a strong transformation for separating runs one another. In particular, they established that universal composition can be achieved on two provisos: first, that the protocol is secure in the single-session setting, and, second, that it satisfies a property they call implicit disjointness. To put it another way, real-world security protocols do not typically use a pre-established SID; therefore, it is necessary to find a further ordinary condition, namely: implicit disjointness, whereby state disjointness can be guaranteed. For a protocol $P$ to satisfy implicit disjointness, two conditions are required. First, participants are not compromised. Second, conversations in a session, especially decryption preceded by encryption, must match. Session matching is, in turn, guaranteed by an appropriate partnering function. Together, implicit disjointness and single-session security, resemble Gutmann’s notion of skeleton [19]. Implicit disjointness can be met simply by selectively inserting randomness in cryptographic material.

In the remaining of this section, we survey protocol compiling methods, stemmed from composition theorems. We shall assume that, after $k$ previous rounds, participants $a_1, \ldots, a_k$ have exchanged fresh nonces, $n_1, \ldots, n_k$, to come out with $SID = a_1; \ldots; a_k; n_1; \ldots; n_k$.

Katz and Yung [23] first showed how to turn a Key Exchange (KE) protocol, known to be secure against a passive adversary, into one that is secure against an active adversary; i.e., into an authenticated KE protocol. Their method transforms each protocol step, $j$. $a_i \to a_{i+1} : j; a_i; m$, into:

$$j : a_i \to a_{i+1} \mid j; a_i; m; \|j; m; SID\|_{K_{a_i}}$$

Later on, Cortier et al. [16] suggested a transformation that guarantees beyond authentication, at the cost of assuming a weaker input protocol that is only functional. Each protocol step, $j$. $a_i \to a_{i+1} : m$, they change into:

$$j : a_i \to a_{i+1} \mid \{m; \|j; m; SID\|_{K_{a_i}}\} \|_{K_{a_{i+1}}}$$
which makes heavy use of cryptoprimitives. Notice that, in particular, "16"’s methods depend on the creation of new signing/encryption key pairs, apart from those that already appear in the original protocol. Finally, Arapinis et al. "2" suggested a straightforward application of Canetti’s universal composition theorem, using SID session tagging, although only cyphertexts are modified. Notice that the plus of this method is that it does not require the creation of new key pairs.

Thus, following the result of "24", all these protocol refinement techniques have several weaknesses. Firstly, most assume the existence and availability of a public key infrastructure. Secondly, they increase both the round complexity, and the message complexity: the extra message exchange implements actually a form of contributary key-agreement protocol; yet, minimizing the number of communications usually is a design goal. Thirdly, they involve additional, indiscriminate computational effort, associated to the extra encrypted material, thus increasing the communication complexity. Fourthly, SID session tagging is as typing tagging (see review [20] below), albeit dynamic, and thus poses the same additional vulnerabilities (the adversary knows SID, and, therefore, can use it to drive the breaking of long-term keys). And fifthly, in all these mechanisms, it is assumed that the participants are all honest: this is restrictive, since it is necessary to consider attacks where the penetrator is part of the group, or even where several participants collude to exclude one or more participants, as in [32]. These properties are all in contrast with our approach, since: we do not require extra key pairs; we selectively include extra rounds, by means of key confirmation; we selectively insert extra bits of information on a specific message; and we minimize the use of tags.

Concluding, even the less heavily cryptographic dependant mechanism of protocol refinement [2] would ‘repair’, for example, NSPK (see Example [6]) as follows:

\[
\begin{align*}
a \rightarrow b & : n \\
b \rightarrow a & : n' \\
a \rightarrow b & : \{a; b; n; n'; a; n_a\}_{K_b} \\
b \rightarrow a & : \{a; b; n; n'; b; n_b\}_{K_a} \\
a \rightarrow b & : \{a; b; n; n'; b; n_b\}_{K_b}
\end{align*}
\]

This increases the number of rounds, and triples the size of the messages to be ciphered. Also, notice that Choo’s approach would not include the extra first two steps, but would tag each message to be ciphered using \(SID = a;b\). By way of comparison, our repair mechanism would yield NSLPK (see Example [14]).

### 8.3 On Further Strategies for Protocol Repair, Based on Old Principles

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 [12] 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, Malladi et al. [29] suggested one should add a session-id contributed to by every participant to any cyphertext in the protocol, as in [11], and Aura [4] suggested one should also use several crypto-algorithms 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 fulfill a stronger security property. What is more, it follows, from Canetti’s universal composition theorem, that these all principles are subsumed by SID session tagging, which inserts even less information.

\footnote{This is indeed supported by, for example, [24], who state that while the use of a pre-established SID is a good design principle, real-world security protocol do not adhere to it, at least not in the explicit way portrayed by this principle.}
8.4 On Protocol Synthesis

Complementary to ours is the work of Perrig and Song [35], 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).

8.5 On Type Flaw Attacks

In the past, several approaches have been developed to deal with potential type flaw attacks. This work can be classified into two different areas: The first area is concerned with changing the representation of messages to prevent type flaw attacks in the first place. Heather et al. [20] use a tagging of messages to identify the type of a message in a unique way. Considering original messages combined with their tagging as new atomic entities, such messages constitute a generated algebraic datatype with non-overlapping ranges of the constructors satisfying the most important properties of the abstract message theory. Since tags themselves will reveal information about a message to a penetrator there are several refinements of this tagging approach to minimize the set of subterms of a message that have to be tagged (e.g. [28, 29]).

The second area is concerned with the verification of protocols that may contain type-flaw attacks. A prominent approach is to replace the standard representation of messages as a freely generated datatype by a more involved datatype dealing with equations between constructor terms. As a consequence, terms representing messages have to be unified modulo a theory modeling the equality relation on messages. While this approach assumes that only entire messages can be interpreted in different ways, Meadows [30, 31] investigated the problem that the implementation of a message could be cut into pieces and one of such pieces might be used to mock another message, e.g. some part of a bit-string representing an encrypted message is reused as a nonce. In her model, messages are represented as bit-streams. Based on an information flow analysis of the protocol and the knowledge how abstract messages are represented as bit-streams, probabilities are computed as to how likely a message could be guessed (or constructed) by a penetrator. This is in contrast to our possibilistic approach, in which we abstract from unlikely events, e.g. that independently guessing a key and a nonce will result in messages that share the same implementation. This is reflected in our notion of originating messages occurring on $I$-traces. The strand space approach excludes protocol runs in which messages are not uniquely originating resulting in a possibilistic approach in a somewhat idealized world. In our approach, for instance, a nonce can be only camouflaged by a message, which itself is camouflaged at some point with the help of the same nonce (c.f. definition 4).

We separate the protocol level operating on abstract messages from its implementation level. This is in contrast to many other approaches that encode implementation details in an equality theory on (abstract) messages. The benefit is that we can use (arbitrary) algebraic specifications to formalize properties (in particular equality and inequality) of the message implementation. This knowledge about the implementation is used to verify side conditions of $I$-traces. In order to apply $I$-traces in the penetrator bundle we have to make sure that both messages of the trace share the same implementation. This is a task that can be given to an automated theorem prover or to specialized deduction system incorporating domain specific knowledge (e.g. SMT-provers).

9 Conclusions

We presented a framework for patching security protocols. The framework is formalized based on the notion of strand spaces, which we have extended to deal also with type flaw attacks. Given a specification for the implementation of messages, an additional $I$-trace rule allows the penetrator
to interchange messages that share the same implementation. While the original purpose of
this extension was to provide a uniform representation language for protocol runs (potentially
containing type flaw attacks), it is also interesting to investigate how the verification methodology
of strand spaces can be lifted to our extended approach.

Based on this formalization, we classify the situations in which a penetrator can reuse messages
communicated by honest principals to mount attacks. This gives rise to various patch rules, which
cope with different types of flaws. It is interesting to see that there are typically two alternative
ways to overcome type flaw attacks. On the one hand, we can change the implementation in
order to avoid the particular equalities on implemented messages; and, on the other hand, we can
change the protocol on the abstract level, in order avoid situations in which one can exploit these
properties in the implementation. The framework has been implemented in the SHRIMP system
and successfully evaluated in a large set of faulty security protocols.

Acknowledgements

This work has been supported by the Deutsche Forschungsgesellschaft DFG under contract Hu737/4-
1 and the Consejo Nacional de Ciencia y Tecnología CONACyT in Mexico under contract 121596.

References

[1] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. IEEE
Transactions on Software Engineering, 22(1):6–15, 1996.

[2] M. Arapinis, S. Delaune, and S. Kremer. From one session to many: Dynamic tags for
security protocols. In I. Cervesato, H. Veith, and A. Voronkov, editors, Proceedings of the
15th Conference on Logic for Programming, Artificial Intelligence, and Reasoning, LPAR
2008, volume 5330 of Lecture Notes in Computer Science, pages 128–142. Springer, 2008.

[3] A. Armando, D. A. Basin, Y. Boichut, Y. Chevalier, L. Compagna, J. Cuéllar, P. H. Drielsma,
P.-C. Héam, O. Kouchnarenko, J. Mantovani, S. Mödersheim, D. von Oheimb, M. Rusinowitch,
J. Santiago, M. Turunen, L. Viganò, and L. Vigneron. The avispa tool for the automated
validation of internet security protocols and applications. In K. Etessami and S. K. Rajamani,
editors, Proceedings of the 17th International Conference in Computer Aided Verification,
CAV’2005, volume 3576 of Lecture Notes in Computer Science, pages 281–285. Springer,
2005.

[4] Tuomas Aura. Strategies against replay attacks. In Computer Security Foundations Workshop
[15], pages 59–69.

[5] D. Basin, S. Mödersheim, and Viganò L. Algebraic intruder deductions. In G. Sutcliffe
and A. Voronkov, editors, Proceedings of the 12th International Conference on Logic for
Programming, Artificial Intelligence, and Reasoning, LPAR 2005, volume 3835 of Lecture
notes in artificial Intelligence, pages 549–564. Springer Verlag, 2005.

[6] D. A. Basin, S. Mödersheim, and L. Viganò. OFMC: A symbolic model checker for security
protocols. International Journal of Information Security, 4(3):181–208, 2005.

[7] J. Biskup and J. Lopez, editors. Proceedings of the 12th European Symposium On Research
In Computer Security, ESORICS 2007, volume 4734 of Lecture Notes in Computer Science.
Springer, 2007.

[8] Bruno Blanchet. An efficient cryptographic protocol verifier based on prolog rules. In Computer
Security Foundations Workshop, pages 82–96. IEEE Computer Science Press, 2001.

[9] M. Burrows, M. Abadi, and R. M. Needham. A logic of authentication. Proceedings of the
Royal Society of London, 426(1):233–71, 1989.
[10] R. Canetti and T. Rabin. Universal composition with joint state. *IACR Cryptology ePrint Archive*, 2002:47–, 2002.

[11] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings of the 42nd Symposium on Foundations of Computer Science, FOCS’01*, pages 136–145. IEEE Computer Society, 2001.

[12] Ulf Carlsen. Cryptographic protocols flaws. In Computer Security Foundations Workshop [13], pages 192–200.

[13] K.-K. Raymond Choo. An integrative framework to protocol analysis and repair: Indistinguishability-based model + planning + model checker. In *Proceedings of the Five-Minute Talks at CSFW’2006*. Computer Security Foundations Workshop, 2006.

[14] *Proceedings of the 7th IEEE Computer Security Foundations Workshop, CSFW’94*. IEEE Computer Society, 1994.

[15] *Proceedings of the 10th IEEE Computer Security Foundations Workshop, CSFW’97*. IEEE Computer Society, 1997.

[16] V. Cortier, B. Warinschi, and E. Zalinescu. Synthesizing secure protocols. In Biskup and Lopez [7], pages 406–421.

[17] D. Dolev and A. C. Yao. On the security of public key protocols. Technical Report 2, Stanford University, Stanford, CA, USA, 1983.

[18] J.-D. Gutman and F.-J. Thayer. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283(2):333–380, 2002.

[19] Joshua D. Gutman. Transformations between cryptographic protocols. In Biskup and Lopez [7], pages 107–123.

[20] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. In *Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW’00*, pages 255–268. IEEE Computer Society, 2000.

[21] D. Hutter and R. Monroy. On the automated correction of protocols with improper message encoding. In *Proceedings of the Joint Workshop on Automated Reasoning for Security Protocol Analysis and Issues in the Theory of Security, ARSPA-WITS’09*, volume 5511 of *Lecture Notes in Computer Science*, pages 138–154. Springer, 2009.

[22] J. Katz and M. Yung. Scalable protocols for authenticated group key exchange. In Dan Boneh, editor, *Advances in Cryptology - CRYPTO 2003, 23rd Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 2003, Proceedings*, volume 2729 of *Lecture Notes in Computer Science*, pages 110–125. Springer, 2003.

[23] J. Katz and M. Yung. Scalable protocols for authenticated group key exchange. *J. Cryptology*, 20(1):85–113, 2007.

[24] R. Küsters and M. Tuengerthal. Composition theorems without pre-established session identifiers. In Y. Chen, G. Danezis, and V. Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011*, pages 41–50. ACM, 2011.

[25] J. C. López-Pimentel, R. Monroy, and D. Hutter. On the automated correction of security protocols susceptible to a replay attack. In Biskup and Lopez [7], pages 594–609.

[26] Gavin Lowe. A family of attacks upon authentication protocols. Technical Report 1997/5, Department of Mathematics and Computer Science, University of Leicester, 1997.
[27] Gavin Lowe. A hierarchy of authentication specifications. In Computer Security Foundations Workshop [15], pages 31–44.

[28] S. Malladi and J. Alves-Foss. How to prevent type-flaw guessing attacks on password protocols. In Proceedings of the 2003 Workshop on Foundations of Computer Security (FCS03), pages 1–12. Technical Report of University of Ottawa, 2003.

[29] S. Malladi, J. Alves-Foss, and R. Heckendorn. On preventing replay attacks on security protocols. In Proceedings of the International Conference on Security and Management, ICSM’02, pages 77–83, 2002. Retrieved February 1, 2007 from http://citeseer.ist.psu.edu/malladi02preventing.html.

[30] Catherine Meadows. Identifying potential type confusion in authenticated messages. In Foundations of Computer Security 02, 2002.

[31] Catherine Meadows. A procedure for verifying security against type confusion attacks. In Proceedings of the 16th IEEE Computer Security Foundations Workshop, CSFW’03, pages 62–74. IEEE Computer Society, 2003.

[32] A. Mukhamedov, S. Kremer, and E. Ritter. Analysis of a multi-party fair exchange protocol and formal proof of correctness in the strand space model. In A.-S. Patrick and M. Yung, editors, Proceedings of the 9th International Conference on Financial Cryptography and Data Security, FC’05, volume 3570 of Lecture Notes in Computer Science, pages 255–269. Springer, 2005.

[33] B.-C. Neuman and S.-G. Stubblebine. A note on the use of timestamps as nonces. Operating Systems Review, 27(2):10–14, 1993.

[34] P. Ochsenschläger, J. Repp, R. Rieke, and U. Nitsche. The SH-verification tool — abstraction-based verification of co-operating systems. Formal Aspects of Computing, 10(4):381–404, 1998.

[35] A. Perrig and D. Song. Looking for diamonds in the desert — extending automatic protocol generation to three-party authentication and key agreement protocols. In Proceedings of the 13th IEEE Computer Security Foundations Workshop, CSFW’00, pages 64–76. IEEE Computer Society, 2000.

[36] Einar Snekkenes. Roles in cryptographic protocols. In Proceedings of the 1992 IEEE Computer Society Symposium on Research in Security and Privacy, pages 105–119. IEEE Computer Society Press, 1992.

[37] Paul Syverson. A taxonomy of replay attacks. In Computer Security Foundations Workshop [14], pages 187–191.

[38] F.-J. Thayer, J.-C. Herzog, and J.-D. Gutman. Strand spaces: Why is a security protocol correct? In Proceedings Symposium on Security and Privacy., pages 160–171. IEEE computer Society, 1998.

[39] F.-J. Thayer, J.-C. Herzog, and J.-D. Gutman. Strand spaces: Proving security protocols correct. Journal of Computer Security, 7(2-3):191–230, 1999.

[40] Mathieu Turuani. The cl-atse protocol analyser. In Term Rewriting and Applications - Proc. of RTA, pages 277–286. Springer, LNCS 4098, 2006.

[41] T. Y. C. Woo and S. S. Lam. A lesson on authentication protocol design. Operating Systems Review, 28(3):24–37, 1994.