Analysis of typed analyses of authentication protocols *

Michele Bugliesi Riccardo Focardi Matteo Maffei
Dipartimento di Informatica
Università Ca'Foscari di Venezia
Via Torino 155, I-30172 Venezia, Italy
{michele,focardi,maffei}@dsi.unive.it

Abstract

This paper contrasts two existing type-based techniques for the analysis of authentication protocols. The former, proposed by Gordon and Jeffrey, uses dependent types for nonces and cryptographic keys to statically regulate the way that nonces are created and checked in the authentication exchange. The latter, proposed by the authors, relies on a combination of static and dynamic typing to achieve similar goals. Specifically, the type system employs dependent ciphertext types to statically define certain tags that determine the typed structure of the messages circulated in the authentication exchange. The type tags are then checked dynamically to verify that each message has the format expected at the corresponding step of the authentication exchange.

This paper compares the two approaches, drawing on a translation of tagged protocols, validated by our system, into protocols that type check with Gordon and Jeffrey's system. This translation gives new insight into the trade-offs between the two techniques, and on their relative expressiveness and precision. In addition, it allows us to port verification techniques from one setting to the other.

1. Introduction

The importance of language-based security in the formal analysis of security protocols dates back to Abadi's seminal work [1] on *secrecy by typing*. Since then, a number of language-based techniques have been applied in the analysis of an increasingly large class of security protocols [2, 3, 4, 6, 7, 8, 12, 13, 16]. These approaches have the advantage of reasoning about security at the language level, thus clarifying *why* a message component is there and *how* security is achieved. This is particularly important for authentication protocols, where flaws are often originated by a certain degree of ambiguity in the encrypted messages circulated in the handshakes. Language-based reasoning on authentication is therefore particularly valuable, as it forces to clarify protocol specifications by making explicit the underlying security mechanisms.

This paper contrasts two of the existing type-based techniques for the analysis of authentication protocols, the *Cryptyc* system proposed in [10, 12, 13] by Gordon and Jeffrey, and the ρ-*spi* system we proposed in [6, 7, 9]. Both the systems use a combination of types and effects for verifying authentication properties formalized as correspondence assertions.

Cryptyc uses dependent types for nonces and cryptographic keys to statically regulate the way that nonces are created and checked in the authentication exchange. The types of nonces express the dependency between each nonce and the message authenticated in the handshake. The types of keys, in turn, make it possible to pass the information on this dependency between the participants in the handshake.

ρ-spi relies on a combination of static and dynamic types and effects to achieve similar goals. Nonce types regulate secrecy and integrity of nonces. As a matter of fact, nonce types (as well as key types) depend just on principal identities: the information about messages to be authenticated is expressed by an additional structure for ciphertexts. This is realized by specific tags attached to the components of the ciphertexts, that also inform on the role that each such component plays in the authentication task. The tags are then checked dynamically to verify that each message has the format expected at the corresponding step of the authentication exchange.

This paper compares the two approaches, drawing on a translation of well-typed (hence safe) ρ-spi protocols into valid Cryptyc protocols. Technically, the crux of the problem is to define the correspondence between ρ-spi tagged

* Work partially supported by EU Contract FET-IST-2001-32617 'Models and Types for Security in Mobile Distributed Systems' (MyThS) and by MIUR Project 'Abstract Interpretation: Design and Applications' (AIDA) .

Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)
1063-6900/05 $20.00 © 2005 IEEE

messages and Cryptyc types. Finding this correspondence is instructive, as it gives new insight into the trade-offs between the two techniques, and on their relative expressiveness and precision.

Besides providing the basis for a formal comparison between the two systems, the translation also allows us to port verification techniques from one setting to the other. Specifically, based on the translation we develop here, we employ the tag-inference procedure developed in [9] for ρ-spi to obtain a fully automated type-inference procedure in Cryptyc, for the class of translated protocols.

At the time of writing, the translation is limited to protocols that do not use sessions keys. We are confident that the results smoothly scale up to this wider class of protocols and, to this purpose, we have suitably extended the ρ-spi type system in order to handle session key authentication. We discuss this issue in detail, illustrating some interesting differences in the way session keys are handled and validated in the two settings. In particular, we notice that Cryptyc is quite liberal, allowing entities to accept non-fresh session keys. Key freshness is then checked by running a nonce handshake based on the session key itself. As noticed by Gordon and Jeffrey [10], this is safe if we assume that session keys cannot be broken. Since this assumption is quite unrealistic, we have preferred to reject this kind of protocols, by always requiring that session key freshness is checked during the key exchange. We discuss how this allows us to tailor the results to a Dolev-Yao attacker model extended with session key corruption.

We remark that the purpose of our analysis is not to show that ρ-spi is more powerful or expressive than Cryptyc (in fact, it is fair to observe that a number of features of ρ-spi are directly inspired by Cryptyc). Rather, our interest and goal here is to explore the extent at which the ρ-spi mechanisms of tagging can be a viable alternative to the corresponding Cryptyc typing. We also remark that the ρ-spi setting considered here is a slight extension of [7]. The main improvement, in terms of expressivity, is the more accurate analysis of protocols in which both the challenge and the response are encrypted. In particular, ρ-spi can now validate (i) protocols authenticating both sent and received messages; this allows us to state properties like "Alice is convinced that Bob has received message M and sent message M'"; (ii) protocols achieving mutual authentication by incorporating both a challenge and a response into the same ciphertext. The complete description of this new ρ-spi version is given in [5].

The rest of the paper is organized as follows: in Section 2, we briefly review the basics of authentication protocols, and the two systems Cryptyc and ρ-spi; Section 3 presents the translation from ρ-spi protocols to Cryptyc typed ones; in Section 4 we discuss the class of protocols that cannot be translated; Section 5 illustrates how the result could be extended to include protocols based on session keys; finally, Section 6 draws some conclusion and discusses future work.

2. Cryptyc and ρ-spi basics

We give a brief review of the Cryptyc and ρ-spi type systems. For lack of space, the presentation is intentionally non-exhaustive: we focus on the main notions and ideas in the two systems and refer the reader to [12] and [7] for details. We start with the basics of authentication protocols.

2.1. Nonce-based Authentication

Authentication protocols based on challenge-response require time-variant parameters like time-stamps or nonces to guarantee message freshness, thus avoiding the so-called replay attacks. Here we focus on nonce-based protocols: a nonce is a value (generally implemented as a random number) used in just one authentication exchange [14].

Suppose A (the *claimant*) wants to authenticate with B (the *verifier*). Nonce-based protocols may be classified into three categories depending on what is encrypted and what is sent in clear. Here, and throughout, we write $\mathsf{Priv}(k_A)$ and $\mathsf{Pub}(k_A)$ to denote the private and public keys of entity A.

Plain-Cipher (PC) B sends out the nonce in clear and receives it back encrypted together with a message which is authenticated. A proves her identity to B by showing the knowledge of the encryption key. For example:

$$A \qquad\qquad B$$
$$\leftarrow\ n\ \text{—}$$
$$\text{—}\ \{B,m,n\}_{\mathsf{Priv}(k_A)}\ \rightarrow$$

This protocol authenticates A sending message m to B, since only A may have generated the ciphertext. The same effect can be achieved in a symmetric cryptosystem using a shared key k_{AB} in place of $\mathsf{Priv}(k_A)$.

Cipher-Plain (CP) B sends out the nonce encrypted and receives it back in clear. A proves her identity to B by showing the knowledge of the decryption key. For example:

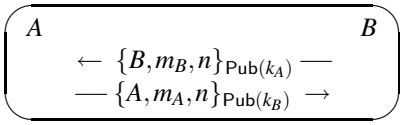
$$A \qquad\qquad B$$
$$\leftarrow\ \{B,m,n\}_{k_{AB}}\ \text{—}$$
$$\text{—}\ n\ \rightarrow$$

where k_{AB} is a symmetric key shared between A and B. This protocol authenticates A receiving message m from B, since only A may have decrypted the ciphertext. (The identifier B is used to "break" the symmetry of the key and avoid the so called *reflection* attacks: when decrypting the message, A knows that it is not a challenge she generated for B in a parallel protocol session). A similar effect is achieved with

Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)
1063-6900/05 $20.00 © 2005 **IEEE**

asymmetric keys, using $\mathsf{Pub}(k_A)$ in place of the symmetric key k_{AB}.

Cipher-Cipher (CC) The nonce is sent out and received back encrypted. This is useful if both entities want to exchange messages. A proves her identity to B by showing the knowledge of either the encryption key (as in PC) or the decryption key (as in CP). For example:



This protocol authenticates A sending message m_A to B and receiving message m_B from B, as only A may have decrypted the first ciphertext. Again, the same effect may be achieved using either the private keys $\mathsf{Priv}(k_B)$ and $\mathsf{Priv}(k_A)$, or a symmetric key k_{AB} in place of the two public keys used in the displayed narration.

Notice that the above mentioned categories are a generalization of POSH (Public Out Secret Home), SOPH (Secret Out Public Home) and SOSH (Secret Out Secret Home), introduced in [12]. We actually need to relax "Secret" into "Cipher" in order to include cases of signed responses, guaranteeing integrity rather than secrecy. For example, as already noticed, PC includes protocols with a cleartext challenge and a signed response, which would not "fit well" into the POSH category.

A well-established technique to formalize properties of the nonce handshakes is based on correspondence assertions [17]. The idea is best illustrated by an example. For instance, in the PC handshake, every time B concludes the protocol convinced to have received a message m from A, written $\mathsf{end}(A,B,m)$, then A has indeed initiated the protocol with B and sent the message m, written $\mathsf{begin}(A,B,m)$. This can be checked by requiring that in every execution sequence, each $\mathsf{end}(A,B,m)$ is preceded by a corresponding $\mathsf{begin}(A,B,m)$.

Both the systems of interest for our analysis, Cryptyc and ρ-spi, rely on correspondence assertions, based on formalizations of protocol narrations into dialects of the spi-calculus.

2.2. Protocol narrations in the spi-calculus

To ease the presentation, below we isolate a syntactic fragment which we take as the common core of the two dialects. The core is given in Table 1: it is not exactly as in the original presentations, but the differences are marginal and harmless.

Messages includes names and variables, for which we introduce two distinct syntactic categories, constructs for asymmetric keys, and tuples. Threads are sequential, and equipped with primitives for input/output, asymmetric and symmetric decryption and dynamic generation of names.

$L,M,N ::=$	*messages*	
	$a,b,\ldots,k,m,n$	names
	x,y,z	variables
	$\mathsf{Priv}(k)$	private key component
	$\mathsf{Pub}(k)$	public key component
	$(M_1,\ldots,M_n)$	tuple
$S ::=$	*threads*	
	$\mathbf{0}$	nil thread
	$\mathsf{in}(M).S$	input
	$\mathsf{out}(M).S$	output
	$\mathsf{decrypt}\ \{\lvert M\rvert\}_N\ \mathsf{as}\ x.S$	asymmetric decryption
	$\mathsf{decrypt}\ \{M\}_N\ \mathsf{as}\ x.S$	symmetric decryption
	$\mathsf{new}(n:T).S$	name generation
	$\mathsf{cast}\ M\ \mathsf{is}\ (\tilde{x}:\tilde{T}).S$	cast
$T ::=$	*types*	
	Un	public (untrusted)
	Top	top type
$f ::=$	*atomic effects*	
	$\mathsf{check}(n)$	fresh nonce

Table 1. Core of spi calculus and ρ-spi calculus

The cast primitive is needed by both the type systems, and has no computational import. All input/output traffic circulates on a unique, anonymous channel. All messages circulating in clear on this channel have type Un, while the type Top acts as the super-type of all messages (encrypted or in clear). Processes are typed according to effects: an effect e is a multiset of atomic effects, noted $[f_1,\ldots,f_n]$. The two type-systems share the atomic effect $\mathsf{check}(n)$, tracking the freshness of the nonce n.

2.3. Types and effects in Cryptyc

The class of Cryptyc terms (in Table 2) includes standard constructs for symmetric and asymmetric encryption. In addition, terms may be formed as elements of sum types (or tagged unions) of the form $(\ell_1(M_1) \mid \cdots \mid \ell_n(M_n))$ where the ℓ_i's are the injections associated with the tagged-union type. The class of processes includes the parallel composition between processes, allowing the interleaving between process executions, the correspondence assertions begin and end, and the primitives match and check having the same semantic behaviour: they proceed only if the checked values are the same.

The type system is built around types and effects. At an abstract level, the rationale of the system may be explained as follows. Within a process, we identify actions that *justify/enable* other actions and, dually, actions that *require* other actions. Typically, actions by one participant justify

Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)
1063-6900/05 $20.00 © 2005 **IEEE**

$L,M,N ::=$	*messages*	
	$\ldots$	as in Table 1
	$\{\|M\|\}_N$	asymmetric encryption
	$\{M\}_N$	symmetric encryption
	$\ell(M)$	tagged union
$P,Q,R,S ::=$	*threads and processes*	
	$\ldots$	as in Table 1
	$P \mid Q$	composition
	$\mathsf{begin}\, L.P$	begin assertion
	$\mathsf{end}\, L.P$	end assertion
	$\mathsf{check}\ M\ \mathsf{is}\ N.P$	nonce check
	$\mathsf{match}\ M\ \mathsf{is}\ N.P$	match
$f ::=$	*atomic effects*	
	$\ldots$	as in Table 1
	$\mathsf{end}\ L$	end event on L
$T,U ::=$	*types*	
	$\ldots$	as in Table 1
	$(x_1 : T_1, \ldots, x_n : T_n)$	dependent record
	$(\ell_1(T_1) \mid \cdots \mid \ell_n(T_n))$	tagged union
	$\ell\,\mathsf{Challes}$, $\ell\,\mathsf{Respes}$	nonce ($\ell = Priv, Pub$)
	$\mathsf{Key}(T)$	asymmetric key
	$\mathsf{SharedKey}(T)$	symmetric key

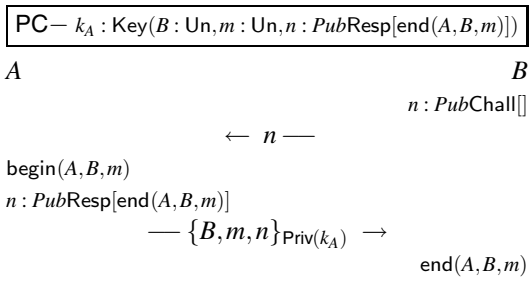
Table 2. Cryptyc syntax and types

and require actions by other participants in the protocol. The exchange of nonces between the participants makes it possible to match the justifying actions of one participant with the requiring actions of another participant.

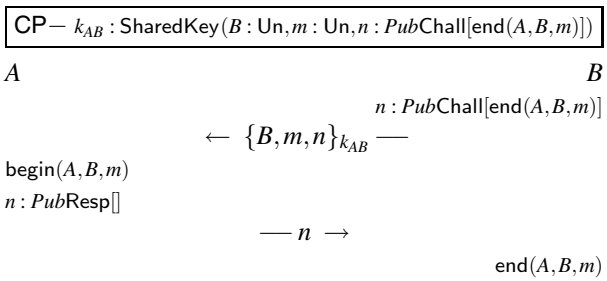
The effects characterize processes, by providing an upper bound on the events that processes may engage in. Specifically, they over-estimate the unmatched (i.e. unjustified) end events: a process (or a pool of processes forming a protocol) is safe if its effect is empty, signalling that every end event is matched by (at least) a corresponding begin event.

The types characterize messages, and in particular the nonces exchanged in the handshakes. Types are related to effects, in that manipulating a nonce may justify or require an action by a process. More precisely, the types of nonces carry, as effects, the enabling actions performed by one participant: upon receiving a nonce, the effect associated with the nonce informs the partner on the enabled actions, which the partner is therefore entitled to carry out locally. The type of the nonce changes during the exchange, reflecting the different "states" of the protocol. The nonce is first sent out at a Challenge type, and then sent back at a Response type: only at this stage, a final check (that the nonce received is the same as the nonce sent) concludes the handshake.

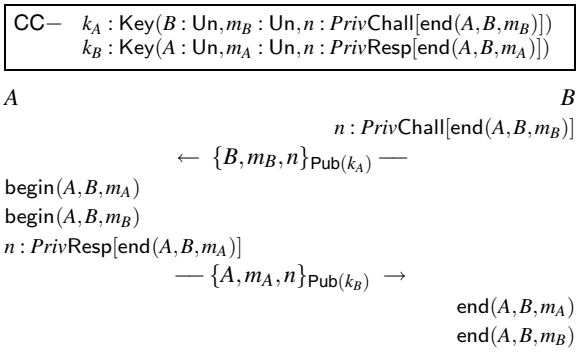
We illustrate these mechanisms in more detail below, by revisiting the handshakes presented in Section 2.1.



The $\mathsf{begin}(A,B,m)$ assertion by A enables an $\mathsf{end}(A,B,m)$ assertion for B, by means of the atomic effect $\mathsf{end}(A,B,m)$ charged on the nonce type by A. While this is the abstract view of the protocol, the mechanism that transfers the effect from A to B is more elaborate and requires auxiliary cast and check actions. The effect is first charged by A when the nonce type is changed (i.e. cast) by A from $Pub\mathsf{Chall}[]$ to $Pub\mathsf{Resp}[\mathsf{end}(A,B,m)]$: it is the cast that requires a $\mathsf{begin}(A,B,m)$ assertion by A. B, in turn, receives the nonce and checks it, i.e. it verifies that it matches the nonce it sent: this check is (implicitly) justified by A's cast and justifies the subsequent $\mathsf{end}(A,B,m)$ assertion by B.



Symmetrically to the previous example, B charges the effect $\mathsf{end}(A,B,m)$ on the nonce's challenge type. Removing the effect when casting the type to $Pub\mathsf{Resp}[]$ requires A to assert $\mathsf{begin}(A,B,m)$. The nonce check justifies an $\mathsf{end}(A,B,m)$ assertion by B.



Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)
1063-6900/05 $20.00 © 2005 **IEEE**

This protocol is the result of the combination of the two previous hand-shakes. In particular, B charges the effect $\mathsf{end}(A,B,m_B)$ on the nonce's challenge type. Casting the type of the nonce to $\mathit{Priv}\mathsf{Resp}[\mathsf{end}(A,B,m_A)]$ requires A to assert $\mathsf{begin}(A,B,m_A)$ and $\mathsf{begin}(A,B,m_B)$. The check of the nonce justifies the $\mathsf{end}(A,B,m_A)$ and $\mathsf{end}(A,B,m_B)$ assertions by B.

As illustrated by the examples, the types of the keys are built around the structure of the encrypted messages and the types of the nonces occurring therein. To allow the re-use of the same key on different ciphertexts with different structure and nonce types, Cryptyc provides tagged union types of the form $\mathsf{Key}(\ell_1(T_1) \mid \cdots \mid \ell_n(T_n))$. Keys with this type may circulate ciphertexts of types $\ell_i(T_i)$: clearly, this requires the use of the injections ℓ_i to form well-typed ciphertexts. To illustrate, the symmetric-key version of the CC handshake discussed above may be expressed relying on union types as follows:

$$\mathsf{CC}{-}k_{AB} : \mathsf{SharedKey}(\ell_1(B:\mathsf{Un}, m_B:\mathsf{Un}, n:\mathit{Priv}\mathsf{Chall}[\mathsf{end}(A,B,m_B)]) \mid \ell_2(A:\mathsf{Un}, m_A:\mathsf{Un}, n:\mathit{Priv}\mathsf{Resp}[\mathsf{end}(A,B,m_A)]))$$

$$A \qquad\qquad B$$

$$\leftarrow \{\ell_1(B,m_B,n)\}_{k_{AB}} \text{ —}$$

$$\text{— } \{\ell_2(A,m_A,n)\}_{k_{AB}} \rightarrow$$

2.4. Tags and Types in ρ-spi

The syntax and the types of ρ-spi are reported in Table 3. The class of terms includes *tagged* terms of the form $C(M)$. Tags require a thorough discussion, as they are the core mechanism of the analysis. They specify the role of each message component. Specifically: all identifiers relevant to authentication are tagged by Id; messages that should be authenticated are tagged by Auth; finally, nonces are tagged by R_H, where $H \in \{\mathsf{PC}, \mathsf{CP}, \mathsf{CC!}, \mathsf{CC?}\}$ denotes the kind of hand-shake and $R \in \{\mathsf{Claim}, \mathsf{Verif}\}$ states whether the entity tagged by Id plays the claimant or verifier role in the handshake. Notice that in CC nonce handshakes we distinguish challenge from response ciphertexts, denoted by CC? and CC!, respectively.

Principal identities are handled explicitly: processes have suitable primitives to declare new symmetric keys shared between two principals I and J and new asymmetric keys for a principal I. The code of a principal I is then specified as $I \triangleright\, !S$, where $!S$ is a replicated sequential thread. The parallel composition is the same as in Cryptyc.

Threads include primitives for symmetric and asymmetric encryption. This makes it easier to check the correct use of tags inside ciphertexts. The begin and end assertions have the same purpose as those in Cryptyc, but more specific format: for example, in $\mathsf{end}(A,B,M_1;M_2)$, B authenticates, at the same time, A receiving M_1 from B and sending M_2 to B. This distinction of messages sent and received is not in the original presentation of ρ-spi, but it is useful here to translate begin and end assertions into the corresponding Cryptyc assertions. As we will see, one event $\mathsf{end}(A,B,M_1;M_2)$ in ρ-spi is mapped into the two events $\mathsf{end}(A,B,M_1)$ and $\mathsf{end}(A,B,M_2)$, in Cryptyc.

$C \in \{\mathsf{Id}, \mathsf{Auth}, R_{\mathsf{PC}}, R_{\mathsf{CP}}, R_{\mathsf{CC?}}, R_{\mathsf{CC!}}\}$ where $R = \mathsf{Verif}, \mathsf{Claim}$
$\ell \in \{\mathit{Pub}, \mathit{Priv}, \mathit{Int}\}$

$M,N ::=$	*messages*	
	$\ldots$	as in Table 1
	$C(M)$	tagged terms
$P,Q ::=$	*processes*	
	$\mathsf{let}\ k = \mathsf{SharedKey}(I,J).P$	symmetric key
	$\mathsf{let}\ k = \mathsf{Key}(I).P$	asymmetric key
	$I \triangleright\, !S$	(replicated) principal
	$P \mid Q$	composition
$S ::=$	*threads*	
	$\ldots$	as in Table 1
	$\mathsf{encrypt}\ \{\lvert M \rvert\}_N\ \mathsf{as}\ x.S$	asymmetric encryption
	$\mathsf{encrypt}\ \{M\}_N\ \mathsf{as}\ x.S$	symmetric encryption
	$\mathsf{begin}(I,J,M_1;M_2).S$	begin assertion
	$\mathsf{end}(I,J,M_1;M_2).S$	end assertion
$T,U ::=$	*types*	
	$\ldots$	as in Table 1
	$\mathsf{Nonce}^{\ell}(I,J)$	nonce type
	$\mathsf{Key}(I)$	asymmetric key
	$\mathsf{SharedKey}(I,J)$	symmetric key
	$\mathsf{Enc}(e_C;e_R)$	ciphertext type
$f ::=$	*atomic effects*	
	$\ldots$	as in Table 1
	$\mathsf{end}(I,J,M_1;M_2)$	end event
	$[?\mid!]\mathsf{Chall}^{\ell}_N(I,J,M)$	challenge effect
	$[?\mid!]\mathsf{Resp}^{\ell}_N(I,J,M)$	response effect

Table 3. ρ-spi syntax and types

Types include nonce, key and ciphertext types. Nonce types regulate the way nonces are used in the authentication task, depending on the kind of hand-shake: the nonces used in PC hand-shakes have type Un, as they are sent in clear on the network. The nonces created by B for being used in CP and CC hand-shakes with A have type $\mathsf{Nonce}^{\ell}(A,B)$. The label $\ell \in \{\mathit{Pub}, \mathit{Priv}, \mathit{Int}\}$ specifies secrecy and integrity properties of the nonce: it is *Pub* in CP hand-shakes, since the environment eventually learns the nonce; it is *Priv* in CC handshakes based on public or symmetric keys, since the nonce

Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)
1063-6900/05 $20.00 © 2005 **IEEE**

remains secret; finally, it is *Int* in CC hand-shakes based on private keys, since only nonce integrity is guaranteed. Notice that nonce types depend just on identity labels as, unlike Cryptyc, they do not convey any information regarding messages exchanged during the hand-shake: they just regulate the scope and the secrecy/integrity of the nonce. Consequently, nonce types do not change during the protocol execution, with the exception of the CP handshakes, where the secrecy of the nonce is lost when it is sent back in clear. This is the only case in which the type of the nonce is cast (from $\mathsf{Nonce}^{Pub}(A,B)$ to Un).

Similarly, key types do not depend on messages but only on the owners of the keys. These types ensure that keys are only used by the authorized principals and are not leaked out to the environment or to other principals. Such simpler types suffice thanks to the rich structure of our ciphertexts: such structure conveys enough information to form the ciphertext types $\mathsf{Enc}(e_C;e_R)$ with the embedded effects defining the ciphertext semantics. In fact, if the ciphertext is a challenge sent by I to J in a hand-shake based on the nonce N and the message M, then $e_C = [\mathsf{Chall}^{\ell}_N(I,J,M)]$, otherwise $e_C = []$. Similarly, if the ciphertext is a response sent by I to J in a hand-shake based on the nonce N and the message M, then $e_R = [\mathsf{Resp}^{\ell}_N(I,J,M)]$, otherwise $e_R = []$. The label ℓ has the same semantics as described above.

Receiving a ciphertext representing a challenge (or a response) sent from I to J containing the nonce N and the message M is tracked by the atomic effect $?\mathsf{Chall}^{\ell}_N(I,J,M)$ (or $?\mathsf{Resp}^{\ell}_N(I,J,M)$), where the label ℓ is used as described above. When ciphertexts are sent rather than received, effects are the same with ! in place of ?. To illustrate, let us consider the following protocol:

PC− $k_A : \mathsf{Key}(A)$
$\{\mathsf{Id}(B),\mathsf{Auth}(m),\mathsf{Verif}_{\mathsf{PC}}(n)\}_{\mathsf{Priv}(k_A)} : \mathsf{Enc}([];[\mathsf{Resp}^{Pub}_n(A,B,m)])$

A B

$n : \mathsf{Un}$

$\leftarrow n —$

$\mathsf{begin}(A,B;m)$

$— \{\ \mathsf{Id}(B),\mathsf{Auth}(m),\mathsf{Verif}_{\mathsf{PC}}(n)\ \}_{\mathsf{Priv}(k_A)} \rightarrow$

$\mathsf{end}(A,B;m)$

Intuitively, the protocol validation proceeds as follows: driven by the tagged structure, the type of the ciphertext is $\mathsf{Enc}([];[\mathsf{Resp}^{Pub}_n(A,B,m)])$. The encryption of such a response requires the atomic effect $!\mathsf{Resp}^{Pub}_n(A,B,m)$, which is justified by $\mathsf{begin}(A,B;m)$. Decrypting this packet justifies $?\mathsf{Resp}^{Pub}_n(A,B,m)$ which is required, together with $\mathsf{check}(n)$, for typing $\mathsf{end}(A,B;m)$. Notice that this assertion states that B authenticates A *sending* message m, since m is after semicolon. The reasoning for the other nonce hand-shakes is similar. Notice that, in CP handshake, B authenticates A *receiving* m, thus m is placed before semicolon in the end assertion.

CP− $k_{AB} : \mathsf{SharedKey}(A,B)$
$\{\mathsf{Id}(B),\mathsf{Auth}(m),\mathsf{Verif}_{\mathsf{CP}}(n)\}_{k_{AB}} : \mathsf{Enc}([\mathsf{Chall}^{Pub}_n(B,A,m)];[])$

A B

$n : \mathsf{Nonce}^{Pub}(A,B)$

$\leftarrow \{\ \mathsf{Id}(B),\mathsf{Auth}(m),\mathsf{Verif}_{\mathsf{CP}}(n)\ \}_{k_{AB}} —$

$\mathsf{begin}(A,B,m;)$

$n : \mathsf{Un}$

$— n \rightarrow$

$\mathsf{end}(A,B,m;)$

CC− $k_A : \mathsf{Key}(A)$, $k_B : \mathsf{Key}(B)$
$\{\mathsf{Id}(B),\mathsf{Auth}(m_B),\mathsf{Verif}_{\mathsf{CC?}}(n)\}_{\mathsf{Pub}(k_A)} : \mathsf{Enc}([\mathsf{Chall}^{Priv}_n(B,A,m_B)];[])$
$\{\mathsf{Id}(A),\mathsf{Auth}(m_A),\mathsf{Claim}_{\mathsf{CC!}}(n)\}_{\mathsf{Pub}(k_B)} : \mathsf{Enc}([];[\mathsf{Resp}^{Priv}_n(A,B,m_A)])$

A B

$n_A : \mathsf{Nonce}^{Priv}(A,B)$

$\leftarrow \{\ \mathsf{Id}(B),\mathsf{Auth}(m_B),\mathsf{Verif}_{\mathsf{CC?}}(n)\ \}_{\mathsf{Pub}(k_A)} —$

$\mathsf{begin}(A,B,m_B;m_A)$

$— \{\ \mathsf{Id}(A),\mathsf{Auth}(m_A),\mathsf{Claim}_{\mathsf{CC!}}(n)\ \}_{\mathsf{Pub}(k_B)} \rightarrow$

$\mathsf{end}(A,B,m_B;m_A)$

3. From ρ-spi to Cryptyc

Conceptually, the relationship between the two systems is simple since the structure of the ρ-spi ciphertexts can be easily mapped to corresponding Cryptyc key types. In the translation, however, there are a number of technical problems that need to be addressed.

A first problem arises from the mechanisms of nonce-checking. In ρ-spi, nonces are checked directly by means of pattern-matching upon input or decryption. This kind of primitive is not directly available in Cryptyc: here, the nonce is received inside a ciphertext, and checked later, only after the successful decryption of the ciphertext. Typically, this is accomplished by a check of the form check x is $n.P$, where n is the nonce and x is the variable that stores the value received in the ciphertext. In order to provide the desired guarantees, the typing rule of the check construct requires that the type at which n was sent be different from the type of the variable x, at which the nonce is received. To realize this sequence of steps in the translation, we need a mechanism to generate a fresh variable (x in the example) to be associated with the nonce checked upon decryption, and then reuse the *same* variable in the check.

A further problem is in the reconstruction of the Cryptyc types of keys. As we noticed in Section 2, the type of a key depends on the structure of the ciphertexts encrypted by that key. Thus, to construct the Cryptyc type of a key used in multiple ciphertexts, we need to identify *all* the ciphertexts encrypted (or decrypted) with that key in the protocol.

Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)
1063-6900/05 $20.00 © 2005 **IEEE**

A third problem is related to the translation of nonce types: in ρ-spi , nonce types depend just on identity labels, while in Cryptyc they may even depend on the messages to authenticate (i.e., messages tagged by Auth) included, together with nonces, inside encrypted challenges and responses. Thus, for translating the type of the nonce, we need to locate such messages.

3.1. The translation

We solve these problems by making the translation dependent on the type derivations in ρ-spi . Furthermore, to facilitate the housekeeping needed in the translation of nonce checking and in the construction of the key types, we give the translation of an *enhanced* type derivation, where we annotate each primitive occurrence (and the messages and names occurring therein) with indexes that make it possible to locate those occurrences. Based on that, we have cheap ways to (i) generate the variables required by the Cryptyc process, and recover them when needed, (ii) collect and identify the ciphertexts needed to generate the correct key types and (iii) inspect ρ-spi effects in order to find out messages, to be authenticated, that are included in challenge or response ciphertexts.

Given a type derivation, the enhanced derivation is constructed by annotating each primitive with a unique *positional* index (the primitive's path in the derivation tree). The occurrences of the ciphertexts are annotated with a further *ciphertext* index: in particular, we assign the same index to all the occurrences of the ciphertexts that are encrypted or decrypted by the same key, and have matching structure. We let Ξ range over enhanced derivation, τ and σ range over positional indexes, and i over ciphertext indexes. We then introduce specific notation to express process and term occurrences within enhanced derivations. This notation is best illustrated by way of examples: $\Xi \vdash \mathsf{in}^{\tau}(M).S$ indicates a thread occurrence in Ξ whose prefix has index τ, while $\Xi \vdash i : \mathsf{decrypt}\ x\ \mathsf{as}\ \{M\}_k.S$ shows the ciphertext index i annotating a decryption. We remark the ciphertext and positional indexes are orthogonal, and may cumulate: for instance in $\Xi \vdash i : \mathsf{decrypt}^{\tau}\ x\ \mathsf{as}\ \{M\}_k.S$, i is the ciphertext index, while τ is the positional index. We make such indexes explicit only when needed in the translation clauses, and omit them otherwise.

The translation of nonce and key types is given in Table 4. The function *CrypticNonceType* rules the translation of nonce types: if the ρ-spi nonce type is Un, then it is not touched by the translation. The only tricky case is when a message M is encrypted in a challenge together with the nonce N, having type $\mathsf{Nonce}^{\ell}(I,J)$, and then authenticated ($\exists M$ s.t. $!\mathsf{Chall}^{\ell}_{N}(I,J,M) \in \Xi$): in this case the translation makes the Cryptyc type depending on M. The translation is deterministic as the linearity of nonce checks implies that no more than one challenge effect labeled by the same nonce can appear in a ρ-spi type derivation for a well-typed process. If the nonce type is $\mathsf{Nonce}^{\ell}(I,J)$ but the nonce is never used for authenticating , then the Cryptyc type is $\ell\ \mathsf{Chall}[\mathsf{end}(I,J)]$. Notice that the type $\mathsf{Nonce}^{Int}(I,J)$ is not translated because it is used in CC hand-shakes based on signatures that cannot be translated into Cryptyc. This is discussed below.

The translation of key types is more elaborate. We let $\mathit{CipherTexts}(k,\Xi)$ be the set of the ciphertexts encrypted and decrypted by the key k in Ξ, together with their ciphertext indexes. Then we construct the type of the key k as the union of Cryptyc tuple types associated with the messages in $\mathit{CipherTexts}(k,\Xi)$. The tuple types, in turn, are defined by cases, depending on the handshake. For instance, a ciphertext such as $\{\mathsf{Id}(J), \mathsf{Verif}_{\mathsf{PC}}(n), \mathsf{Auth}(m)\}_{k_{IJ}}$ is a PC response conveying I's intention to authenticate m with J: the type of n inside the tuple type is therefore $\mathit{Pub}\,\mathsf{Resp}[\mathsf{end}(I,J,m)]$. The reasoning for the other handshakes is similar, with some exceptions that we discuss below:

- **PC** The nonce is sent in clear by J to I, and is returned either encrypted with a symmetric key or signed. The corresponding Cryptyc type is $\mathit{Pub}\mathsf{Resp}[\mathsf{end}(I,J,\overline{y})]$;
- **CP** The nonce is sent encrypted with a symmetric key by J to I and is returned in clear. The corresponding Cryptyc type is $\mathit{Pub}\mathsf{Chall}[\mathsf{end}(I,J,\overline{y})]$. Notice that we do not translate the case in which the nonce is encrypted with I's public key. This case is correctly validated by ρ-spi but is not handled by Cryptyc (cf. Section 4);
- **CC** The nonce is sent and returned encrypted with either a symmetric key or a public key. The corresponding Cryptyc types are, respectively, $\mathit{Priv}\,\mathsf{Chall}[\mathsf{end}(I,J,\overline{y})]$ and $\mathit{Priv}\,\mathsf{Resp}[\mathsf{end}(J,I,\overline{y})]$. Notice that we do not translate the case in which the messages are signed. Also this case is correctly validated by ρ-spi but is not handled by Cryptyc (cf. Section 4).

For ease of presentation, we have omitted the cases of ciphertexts containing two nonces: one for a challenge and one for a response. This typically happens in mutual authentication protocols. The tuple type for these cases can be easily obtained by considering the two types for the two combined nonce hand-shakes. For instance:

$$\begin{aligned}\mathit{TupleType}(k:T,\{\mathsf{Id}(I),\mathsf{Auth}(\overline{M}),\mathsf{Verif}_{\mathsf{CC?}}(n_1),\mathsf{Claim}_{\mathsf{CC!}}(n_2),\dots\}_K)\\ = (x:\mathsf{Un},\overline{y}:\mathsf{Un},\ w:\mathit{Priv}\,\mathsf{Chall}[\mathsf{end}(J,I,\overline{y})],\\ z:\mathit{Priv}\,\mathsf{Resp}[\mathsf{end}(I,J,\overline{y})],\cdots:\mathsf{Un})\end{aligned}$$

is the combination of CC challenge and response. The other possible combinations are: CC! and PC (responses) with CC? and CP (challenges), since these are the only cases in which key types are compatible.

The translation of processes is given in Table 5: as we remarked earlier, it is given inductively on the structure

Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)
1063-6900/05 $20.00 © 2005 **IEEE**

Nonce types

$$\mathit{CrypticNonceType}(n:\mathsf{Un},\Xi) \;=\; \mathsf{Un}$$

$$(\ell \in \{\mathit{Pub},\mathit{Priv}\}) \quad \mathit{CrypticNonceType}(n:\mathsf{Nonce}^{\ell}(I,J),\Xi) \;=\; \begin{cases} \ell\ \mathsf{Chall}\ [\mathsf{end}(I,J,M)] & \text{if } \exists M \text{ s.t. } !\mathsf{Chall}^{\ell}_{n}(I,J,M) \in \Xi \\ \ell\ \mathsf{Chall}\ [\mathsf{end}(I,J)] & \text{otherwise} \end{cases}$$

Key Types

Let $T(\cdot)$, with $T \in \{\mathsf{Key}/\mathsf{SharedKey}\}$, be the type of k in Ξ

$$\begin{aligned} \mathit{CryptycKeyType}(k,\Xi) \;&=\; T(\mathit{msg}_1(T_1) \mid \cdots \mid \mathit{msg}_n(T_n)) \\ &\quad \text{if } \mathit{CipherTexts}(k,\Xi) = \{i_1 : M_1, \ldots, i_k : M_k\} \\ &\quad \text{and } T_i = \mathit{TupleType}(k : T(\cdot), M_i) \\ \mathit{CryptycKeyType}(k,\Xi) \;&=\; T(\mathsf{Top}\ ) \text{ if } \mathit{CipherTexts}(k,\Xi) = \emptyset \end{aligned}$$

The mapping $\mathit{TupleType}(\cdot,\cdot)$ is defined depending on the handshake, as follows

PC:

$$\begin{aligned} &\mathit{TupleType}(k : T, \{|\mathsf{Id}(J), \mathsf{Auth}(\overline{M}), \mathsf{Verif}_{\mathsf{PC}}(n), \ldots|\}_K \\ &\quad = (x:\mathsf{Un}, \overline{y}:\mathsf{Un}, z : \mathit{Pub}\,\mathsf{Resp}[\mathsf{end}(I,J,\overline{y})], \cdots : \mathsf{Un}) \end{aligned}$$

T	K
$\mathsf{SharedKey}(I,J)$	k
$\mathsf{Key}(I)$	$\mathsf{Priv}(k)$

CP:

$$\begin{aligned} &\mathit{TupleType}(k : \mathsf{SharedKey}(I,J), \{|\mathsf{Id}(J), \mathsf{Auth}(\overline{M}), \mathsf{Verif}_{\mathsf{CP}}(n), \ldots|\}_k) \\ &\quad = (x:\mathsf{Un}, \overline{y}:\mathsf{Un}, z : \mathit{Pub}\,\mathsf{Chall}[\mathsf{end}(I,J,\overline{y})], \cdots : \mathsf{Un}) \end{aligned}$$

CC:

$$\begin{aligned} &\mathit{TupleType}(k : T, \{\mathsf{Id}(J), \mathsf{Auth}(\overline{M}), \mathsf{Verif}_{\mathsf{CC?}}(n), \ldots\}_K) \\ &\quad = (x:\mathsf{Un}, \overline{y}:\mathsf{Un}, z : \mathit{Priv}\,\mathsf{Chall}[\mathsf{end}(I,J,\overline{y})], \cdots : \mathsf{Un}) \\ &\mathit{TupleType}(k : T, \{\mathsf{Id}(J), \mathsf{Auth}(\overline{M}), \mathsf{Claim}_{\mathsf{CC!}}(n), \ldots\}_K) \\ &\quad = (x:\mathsf{Un}, \overline{y}:\mathsf{Un}, z : \mathit{Priv}\,\mathsf{Resp}[\mathsf{end}(J,I,\overline{y})], \cdots : \mathsf{Un}) \end{aligned}$$

T	K
$\mathsf{SharedKey}(I,J)$	k
$\mathsf{Key}(I)$	$\mathsf{Pub}(k)$

Table 4. Encoding: Nonce and Key Types

of processes, but relative to a derivation. We comment on the interesting cases below. In the input form, if M_i's are (tagged) names, then the x_i^{τ}'s are (tagged) fresh variables indexed after the positional index associated with the input prefix, and their position in the input tuple; if, instead, M_i's are (tagged) variables, then the x_i^{τ}'s are the same as the M_i's. $|M_i|$, in turn, indicates the *tag-erasure* of the corresponding ρ-spi terms M_i . The reasoning is similar for the decryption form: here, in addition, we use the ciphertext index to choose the injection msg_i used in the ciphertext. In the encryption forms, the variable resulting from the ρ-spi encryption is substituted with the ciphertext in the remaining part of the Cryptyc process. This is needed, as ciphertexts are syntactic terms in spi calculus . The first and third encryption forms correspond to PC and CC! responses, respectively: as explained in Section 2.3, the type of the nonce has to be cast from Chall to Resp. The second encryption form corresponds to CP and CC? challenges (so no cast is needed), while the fourth one translates encryptions with no tagged message component.

In the ρ-spi system, a type cast occurs before nonces used in CP hand-shakes are sent back in clear on the network. Indeed, the possible secrecy of the nonce is lost and the type is cast from $\mathsf{Nonce}^{\mathit{Pub}}(I,J)$ to Un. The ρ-spi cast is immediately translated in the *Cryptyc* cast giving the nonce the type *Pub* Resp[] (which is equal to Un). The clauses for the begin/end assertions complete the definition of the translation. First notice that $\mathsf{begin}(I,J,M_1;M_2)$ events are translated into two separate Cryptyc begin (unless either M_1 or M_2 is the empty tuple, in which case the corresponding begin is dropped from the translation). As we already explained this is necessary since in Cryptyc the authentication of messages sent and received by one principal, is reflected in two separate begin.

The same reasoning applies to the end assertions. In ρ-spi, these assertions are justified by the reception of a nonce either in clear, thus untagged (in CP hand-shakes), or encrypted and tagged (PC and CC hand-shakes). In both cases, the nonce is pattern-matched. The effect system checks that each end is justified by a different (fresh) nonce, thus guaranteeing that a nonce is used at most once. Differently from the ρ-spi system, in Cryptyc nonces are not pattern-matched; rather, they are checked with a special primitive check. The check primitive is used to match the nonce with a variable containing the received value. Such a variable is univocally determined in the translation by the proof-tree

Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)
1063-6900/05 $20.00 © 2005 **IEEE**

$$
\begin{array}{rcl}
[\![\Xi \vdash \mathbf{0}]\!] & = & \mathsf{stop} \\
[\![\Xi \vdash \mathsf{let}\ k = \mathsf{SharedKey}(I,J).P]\!] & = & \mathsf{new}(k : \mathit{CryptycKeyType}(k,\Xi)).[\![\Xi \vdash P]\!] \\
[\![\Xi \vdash \mathsf{let}\ k = \mathsf{Key}(I).P]\!] & = & \mathsf{new}(k : \mathit{CryptycKeyType}(k,\Xi)).[\![\Xi \vdash P]\!] \\
[\![\Xi \vdash I \triangleright !S]\!] & = & [\![\Xi \vdash !S]\!] \\
[\![\Xi \vdash P \mid Q]\!] & = & [\![\Xi \vdash P]\!] \mid [\![\Xi \vdash Q]\!] \\
[\![\Xi \vdash \mathsf{new}(n : T).S]\!]\ (\dagger) & = & \mathsf{new}(n : \mathit{CrypticNonceType}(n : T,\Xi)).[\![\Xi \vdash S]\!] \\
\\
[\![\Xi \vdash \mathsf{in}^{\tau}(M_1,\ldots,M_n).S]\!] & = & \mathsf{in}(x_1^{\tau} : \mathsf{Un},\ldots,x_n^{\tau} : \mathsf{Un}). \\
 & & \mathsf{match}\ (x_1^{\tau},\ldots,x_n^{\tau})\ \mathsf{is}\ (|M_1|,\ldots,|M_n|). \\
 & & [\![\Xi \vdash S]\!] \\
[\![\Xi \vdash \mathsf{out}(M).S]\!] & = & \mathsf{out}(M).[\![\Xi \vdash S]\!] \\
[\![\Xi \vdash i : \mathsf{decrypt}^{\tau}\ x\ \mathsf{as}\ \{M_1,\ldots,M_n\}_K.S]\!] & & \\
 & = & \mathsf{decrypt}\ x\ \mathsf{as}\ \{msg_i(x_1^{\tau} : T_1,\ldots,x_n^{\tau} : T_n)\}_K. \\
 & & \mathsf{match}\ (x_1^{\tau},\ldots,x_n^{\tau})\ \mathsf{is}\ (|M_1|,\ldots,|M_n|). \\
 & & [\![\Xi \vdash S]\!] \\
 & & \text{where} \\
 & & \mathit{CryptycKeyType}(K,\Xi) = \\
 & & \quad \mathsf{Key}(\cdots \mid msg_i(y_1 : T_1,\ldots,y_n : T_n) \mid \ldots) \\
[\![\Xi \vdash i : \mathsf{encrypt}\ \{\mathsf{Id}(J),\mathsf{Auth}(M),\mathsf{Verif}_{\mathsf{PC}}(n),\ldots\}_K\ \mathsf{as}\ z.S]\!] & & \\
 & = & \mathsf{cast}\ n\ \mathsf{is}\ (x : \mathit{Pub}\,\mathsf{Resp}[\mathsf{end}(I,J,M)]). \\
 & & [\![\Xi \vdash S]\!]\,[\{msg_i(J,M,x,\ldots)\}_K/z] \\
[\![\Xi \vdash i : \mathsf{encrypt}\ \{\mathsf{Id}(J),\mathsf{Auth}(M),\mathsf{Verif}_{\mathsf{CP/CC?}}(n),\ldots\}_K\ \mathsf{as}\ z.S]\!] & & \\
 & (*) = & [\![\Xi \vdash S]\!]\,[\{msg_i(J,M,n,\ldots)\}_K/z] \\
[\![\Xi \vdash i : \mathsf{encrypt}\ \{\mathsf{Id}(J),\mathsf{Auth}(M),\mathsf{Verif}_{\mathsf{CC!}}(n),\ldots\}_K\ \mathsf{as}\ z.S]\!] & & \\
 & (*) = & \mathsf{cast}\ n\ \mathsf{is}\ (x : \mathit{Priv}\,\mathsf{Resp}[\mathsf{end}(I,J,M)]). \\
 & & [\![\Xi \vdash S]\!]\,[\{msg_i(J,M,x,\ldots)\}_K/z] \\
[\![\Xi \vdash i : \mathsf{encrypt}\ \{M\}_K\ \mathsf{as}\ z.S]\!] & = & [\![\Xi \vdash S]\!]\,[\{msg_i(M)\}_K/z] \\
[\![\Xi \vdash \mathsf{cast}\ N\ \mathsf{is}\ (x : \mathsf{Un})]\!] & = & \mathsf{cast}\ N\ \mathsf{is}\ (x : \mathit{Pub}\,\mathsf{Resp}[])[\![\Xi \vdash S]\!] \\
[\![\Xi \vdash \mathsf{begin}^{\tau}(I,J,M_1;M_2).S]\!] & = & \mathsf{begin}(I,J,M_1).\mathsf{begin}(I,J,M_2).[\![\Xi \vdash S]\!] \\
[\![\Xi \vdash \mathsf{end}^{\sigma}(I,J,M_1;M_2).S]\!] & = & \mathsf{check}\ x_i^{\tau}\ \mathsf{is}\ n.\mathsf{end}(J,I,M_1).\mathsf{end}(J,I,M_2).[\![\Xi \vdash S]\!] \\
 & & \text{if } \Xi \vdash \mathsf{end}^{\sigma}(I,J,M_1;M_2).S \text{ depends on} \\
 & & \Xi \vdash \mathsf{decrypt}^{\tau}\ z\ \mathsf{as}\ \{M_1,\ldots,M_n\}_N.S' \\
 & & \text{with } M_i = \mathsf{Verif}_{\mathsf{PC}}(n),\mathsf{Verif}_{\mathsf{CC!}}(n),n \\
 & & \text{or on } \Xi \vdash \mathsf{in}^{\tau}(M_1,\ldots,M_n).S' \text{ with } M_i = n
\end{array}
$$

($\dagger$) If $\mathit{CrypticNonceType}(n : T,\Xi) = \ell\ \mathsf{Chall}\ [\mathsf{end}(I,J,M)]$, then all the names and variables in M are free in S.
(*) If the nonce tag is CP, then K is symmetric; if it is CC! or CC? then K is not a private key.

Table 5. Encoding of processes

and the positional indexes.

We exemplify the translation on the symmetric-key version of the first protocol of Section 1. Table 6 reports the ρ-spi code for the narration, and the corresponding Cryptyc code, where we include only the relevant type annotations and disregard the vacuously successful matches. The ρ-spi code is indexed as it would be in the typing derivation, which we leave implicit, and the Cryptyc specification is the result of the translation.

3.2. Properties of the translation

We conclude with the main properties of the translation. We first give a result of computational correspondence, showing that the execution steps of any well-typed ρ-spi process are preserved by its corresponding Cryptyc process. We formalize the statement in terms of labelled transitions, whose definition is standard, and omitted.

Theorem 1 (Preservation of execution steps) *Let P be a*

Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)
1063-6900/05 $20.00 © 2005 **IEEE**

ρ-spi specification	Cryptyc specification
$Protocol \triangleq \mathsf{let}\ k_{AB} = \mathsf{SharedKey}(A,B).$ $(A \triangleright\ !Responder \mid B \triangleright Initiator)$	$Protocol \triangleq \mathsf{new}(k_{AB} : T_{AB})$ $(!Responder \mid !Initiator)$
$Initiator \triangleq$ $\mathsf{new}(n : \mathsf{Un}).$ $\mathsf{out}(n).$ $\mathsf{in}^{\sigma}(z).$ $1 : \mathsf{decrypt}^{\tau}\ z\ \mathsf{as}\ \{\mathsf{Id}(B), \mathsf{Auth}(x), \mathsf{Verif}_{\mathsf{PC}}(n)\}_{k_{AB}}.$ $\mathsf{end}(A,B;x)$	$Initiator \triangleq$ $\mathsf{new}(n : \mathsf{Un}).$ $\mathsf{out}(n).$ $\mathsf{in}(z^{\sigma}).$ $\mathsf{decrypt}\ z\ \mathsf{as}\ \{msg_1(x_1^{\tau}, x_2^{\tau}, x_3^{\tau})\}_{k_{AB}}$ $\mathsf{match}\ (x_1^{\tau}, x_2^{\tau}, x_3^{\tau})\ \mathsf{is}\ (B, x_2^{\tau}, n).$ $\mathsf{check}\ x_3^{\tau}\ \mathsf{is}\ n.$ $\mathsf{end}(A,B,x)$
$Responder \triangleq$ $\mathsf{in}(x).$ $\mathsf{new}(m : \mathsf{Un}).$ $\mathsf{begin}(A,B;m).$ $1 : \mathsf{encrypt}\ \{\mathsf{Id}(B), \mathsf{Auth}(m), \mathsf{Verif}_{\mathsf{PC}}(x)\}_{k_{AB}}\ \mathsf{as}\ z.$ $\mathsf{out}(z)$	$Responder \triangleq$ $\mathsf{in}(x).$ $\mathsf{new}(m : \mathsf{Un}).$ $\mathsf{begin}(A,B,m).$ $\mathsf{cast}\ x\ \mathsf{is}\ (y : Pub\,\mathsf{Resp}[\mathsf{end}(A,B,m)]).$ $\mathsf{out}(\{msg_1(B,m,y)\}_{k_{AB}})$

$$T_{AB} = \mathsf{SharedKey}(msg_1(x : \mathsf{Un}, y : \mathsf{Un}, z : Pub\,\mathsf{Resp}[\mathsf{end}(A,B,y)]))$$

Table 6. An illustration of the translation

ρ-spi process and Ξ an enhanced ρ-spi type derivation for the judgment $\vdash P : []$. *Then*

$$\text{if } P \xrightarrow{\mathsf{in}(M)} P' \text{ then } [\![\Xi \vdash P]\!] \xRightarrow{\mathsf{in}(|M|)} [\![\Xi \vdash P']\!]$$

$$\text{if } P \xrightarrow{\mathsf{out}(M)} P' \text{ then } [\![\Xi \vdash P]\!] \xrightarrow{\mathsf{out}(|M|)} [\![\Xi \vdash P']\!]$$

$$\text{if } P \xrightarrow{\mathsf{begin}(I,J,M_1;M_2)} P' \text{ then } [\![\Xi \vdash P]\!] \xrightarrow{\mathsf{begin}(I,J,M_1)} \bullet \xrightarrow{\mathsf{begin}(I,J,M_2)} [\![\Xi \vdash P']\!]$$

$$\text{if } P \xrightarrow{\mathsf{end}(I,J,\overline{M}_1;\overline{M}_2)} P' \text{ then } [\![\Xi \vdash P]\!] \xrightarrow{\mathsf{end}(I,J,M_1)} \bullet \xrightarrow{\mathsf{end}(I,J,M_2)} [\![\Xi \vdash P']\!]$$

The theorem tells us that the translation preserves all the exchanges and all the assertion events of the ρ-spi protocol. While this notion of operational correspondence is somewhat weak (in fact, the reverse direction of this result, i.e. reflection of execution steps, can also be proved), it is all we need to show that the translation of a safe ρ-spi protocol is a safe Cryptyc protocol.

Definition 1 (Process Safety) *A process P is safe iff, for every execution trace s generated by P, every* $\mathsf{end}(A,B,M)$ *(or* $\mathsf{end}\ (A,B,M_1;M_2)$*) in s is preceded by a distinct* $\mathsf{begin}(A,B,M)$ *(*$\mathsf{begin}(A,B,M_1;M_2)$*).*

In both the type and effect systems, safe processes may be typed with empty effects. The translation preserves type safety, in the following sense:

Theorem 2 (Preservation of Safety) *Let P be a ρ-spi process, and let Ξ an enhanced derivation for the ρ-spi judgment* $I_1 : \mathsf{Un}, \ldots, I_n : \mathsf{Un} \vdash P : []$*, where* $I_1, \ldots, I_n$ *are the identity labels in P. Then* $I_1 : \mathsf{Un}, \ldots, I_n : \mathsf{Un} \vdash [\![\Xi \vdash P]\!] : []$ *is a derivable Cryptyc judgment.*

4. Further kinds of handshakes

We continue our analysis discussing further classes of handshakes which we disregarded in the previous section.

The first kind of protocols we examine includes protocols in which both the challenge and the response are signed, and B wants to authenticate with A the reception of message m_A. For example, consider:

$$\begin{array}{l} A \qquad\qquad\qquad\qquad B \\ \quad \text{—} \{B, n_A, m_A\}_{\mathsf{Priv}(k_A)} \rightarrow \\ \quad\quad \leftarrow \{A, n_A\}_{\mathsf{Priv}(k_B)} \text{—} \end{array}$$

When A receives n_A signed by B, she is guaranteed that B has indeed received m_A. This protocol can be validated in ρ-spi relying on the following tags and types:

$$\begin{array}{l} A \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad B \\ n_A : \mathsf{Nonce}^{Int}(A,B) \\ \text{—} \{\mathsf{Id}(B), \mathsf{Claim}_{\mathsf{CC?}}(n_A), \mathsf{Auth}(m_A)\}_{\mathsf{Priv}(k_A)} \rightarrow \\ \quad\quad \leftarrow \{\mathsf{Id}(A), \mathsf{Verif}_{\mathsf{CC!}}(n_A)\}_{\mathsf{Priv}(k_B)} \text{—} \end{array}$$

This kind of protocols is not validated in Cryptyc. The reason is that Cryptyc does not provide for challenges which

Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)
1063-6900/05 $20.00 © 2005 **IEEE**

are public and untainted, as signature are. The only way to type check the above protocol in the Cryptyc type system is to consider the first challenge as a plaintext, assigning to n_A the type '*Pub* Chall' which cannot convey any effect related to message m_A.

The second kind of protocols we examine are those based on CP handshakes and asymmetric cryptography. For example, consider the following protocol:

$$\begin{array}{rr} A & B \\ \multicolumn{2}{c}{\longrightarrow \{A,n,m\}_{\mathsf{Pub}(k_B)} \rightarrow} \\ \multicolumn{2}{c}{\leftarrow n \longrightarrow} \end{array}$$

This protocol can be validated in ρ-spi using the type n : $\mathsf{Nonce}^{Pub}(A,B)$ and the tagging

$$\{\mathsf{Id}(A), \mathsf{Verif}_{\mathsf{CP}}(n), \mathsf{Auth}(m)\}_{\mathsf{Pub}(k_B)}$$

for the ciphertext. Cryptyc does not handle this kind of protocols [11], as it does not include types for nonces like n above, which is tainted (as it might come from the enemy) and secret.

5. Session Keys

We conclude our analysis by discussing the extensions needed in ρ-spi to validate protocols involving sessions keys, and the relationships of the resulting mechanisms with their corresponding mechanisms in Cryptyc. We discuss the two cases of asymmetric and symmetric cryptography separately, as they have different treatment in Cryptyc.

5.1. Session Keys exchanged through asymmetric cryptography

Let us consider an extended version of the Needham-Schroeder public-key authentication protocol in which A and B exchange a fresh session key k_s:

$$\begin{array}{rr} A & B \\ \multicolumn{2}{c}{\longrightarrow \{A,n_A,k_s\}_{\mathsf{Pub}(k_B)} \rightarrow} \\ \multicolumn{2}{c}{\leftarrow \{B,n_A,n_B,k_s\}_{\mathsf{Pub}(k_A)} \longrightarrow} \\ \multicolumn{2}{c}{\longrightarrow \{n_B\}_{\mathsf{Pub}(k_B)} \rightarrow} \end{array}$$

Notice that the first message does not give any authentication guarantee on the session key k_s: even the enemy might have originated that public key encryption. Thus, B asks for confirmation of the session key k_s in the second message, using the nonce n_B. Only after receiving the third message B is guaranteed that k_s has been sent by A.

The protocol in Cryptyc To check this protocol, Cryptyc exploits a "trust-witness" mechanism that allows B to ask for confirmation about the type of k_s and A to give such a confirmation. This is achieved by appending to the type of n_B the effect '$\mathsf{trust}\ k_s : \mathsf{SharedKey}(T_{ses})$'. ($T_{ses}$ is the type of the messages that will be exchanged through the session key.) Casting the type of n_B to $\mathit{Priv}\mathsf{Resp}[]$ requires A to confirm that the type k_s is $\mathsf{SharedKey}(T_{ses})$ through the special primitive $\mathsf{witness}\ k_s : \mathsf{SharedKey}(T_{ses})$. When B receives back the nonce, he can finally "trust" the type of k_s. The actual cast to the trusted type is done through a special primitive $\mathsf{trust}\ k\ \mathsf{is}\ (x : \mathsf{SharedKey}(T_{ses}))$ which assigns to x the type $\mathsf{SharedKey}(T_{ses})$. The protocol, decorated with types and effects is as follows:

$$\begin{array}{rr} A & B \\ n_A : \mathit{Priv}\mathsf{Chall}[\mathsf{end}(B,A,k_s)] & \\ \multicolumn{2}{c}{\longrightarrow \{A,n_A,k_s\}_{\mathsf{Pub}(k_B)} \rightarrow} \\ & n_A : \mathit{Priv}\mathsf{Resp}[] \\ & n_B : \mathit{Priv}\mathsf{Chall}\ e \\ & e = [\mathsf{end}(A,B,k_s), \mathsf{trust}\ k_s : \mathsf{SharedKey}(T_{ses})] \\ \multicolumn{2}{c}{\leftarrow \{B,n_A,n_B,k_s\}_{\mathsf{Pub}(k_A)} \longrightarrow} \\ n_B : \mathit{Priv}\mathsf{Resp}[] & \\ \multicolumn{2}{c}{\longrightarrow \{n_B\}_{\mathsf{Pub}(k_B)} \rightarrow} \end{array}$$

The protocol in ρ*-spi* In order to handle session keys, the ρ-spi type system is extended with two new tags Key and Key?: the former is used for *communicating* a fresh session key, the latter Key? to *ask for confirmation* about a received session key. Intuitively, Key? is the analogous of the trust-witness mechanism found in Cryptyc. In the example protocol, k_s is tagged by Key in the first ciphertext, while it is tagged by Key? in the second one. All the other tags are used as before. (Notice that the last challenge is correctly tagged as CP, since the last encryption is unnecessary for authentication purposes.)

$$\begin{array}{rr} A & B \\ \multicolumn{2}{c}{\longrightarrow \{\mathsf{Id}(A), \mathsf{Verif}_{\mathsf{CC?}}(n_A), \mathsf{Key}(k_s)\}_{\mathsf{Pub}(k_B)} \rightarrow} \\ \multicolumn{2}{c}{\leftarrow \{\mathsf{Id}(B), \mathsf{Claim}_{\mathsf{CC!}}(n_A), \mathsf{Verif}_{\mathsf{CP}}(n_B), \mathsf{Key?}(k_s)\}_{\mathsf{Pub}(k_A)} \longrightarrow} \\ \multicolumn{2}{c}{\longrightarrow \{n_B\}_{\mathsf{Pub}(k_B)} \rightarrow} \end{array}$$

On A's side, k_s is generated with type $\mathsf{SessionKey}(A,B)$. Let us now illustrate how B gains trust of such a session key type. Initially, k_s is given the type $\mathsf{TaintedKey}(A,B)$, reflecting that such a key is supposed to be shared between A and B, but it might come from the enemy. Tag Key? is used to check that the key comes from A: before A sends back n_B, she is required to check that k_s has indeed type $\mathsf{SessionKey}(A,B)$. Thus, when B receives and checks n_B, he can safely cast the type of k to $\mathsf{SessionKey}(A,B)$ finally asserting $\mathsf{end}(A,B,k_s)$.

Table 7 (in Appendix) reports the full (type-checkable) specifications of the example protocol, with types and tags.

Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)
1063-6900/05 $20.00 © 2005 **IEEE**

5.2. Session Keys exchanged through symmetric cryptography

Since symmetric cryptography provides both secrecy and integrity guarantees on the exchanged key, it is not necessary to check the origin of the session key as done for public key cryptography. However, the freshness of the session key is an independent property that needs to be eventually checked, through some nonce handshake.

Session Keys in Cryptyc The Cryptyc type system is quite liberal, allowing entities to accept non-fresh session keys. Key freshness is then checked by running a nonce handshake based on the session key itself. As an example consider the following protocol:

$$\begin{array}{cc} A & B \\ \multicolumn{2}{c}{\text{---}\{A,k_s\}_K \rightarrow} \\ \multicolumn{2}{c}{\leftarrow n \text{---}} \\ \multicolumn{2}{c}{\text{---}\{A,n,m\}_{k_s} \rightarrow} \end{array}$$

where K is a long-term key shared between A and B. The protocol works as follows: A sends to B the session key k_s encrypted through key K. The session key is then used for authenticating m by a PC nonce handshake. The idea is that the nonce handshake should even check the freshness of k_s. Unfortunately, the protocol is flawed, as shown by the following (standard) *known-key* attack:

$$\begin{array}{cc} E(A) & B \\ \text{The enemy intercepts and stores } \{A,k_s\}_K & \\ \text{together with all the session messages} & \\ \text{encrypted through } k_s & \\ \ldots & \\ \text{The enemy breaks the session key } k_s & \\ \multicolumn{2}{c}{\text{---}\{A,k_s\}_K \rightarrow} \\ \multicolumn{2}{c}{\leftarrow n' \text{---}} \\ \multicolumn{2}{c}{\text{---}\{A,n',m'\}_{k_s} \rightarrow} \end{array}$$

The enemy breaks an old session key and replays the first message to B which begins an authentication session using the broken session key. The enemy can successfully impersonate A as it knows the (broken) key used in the nonce handshake. Cryptyc validates this protocol by assigning type $\mathsf{SharedKey}(x:\mathsf{Un}, y:\mathsf{Un}, z:\mathit{Pub}\mathsf{Resp}[\mathsf{end}(A,B,y)])$ to the session key. As noticed by Gordon and Jeffrey in [10], this liberal use of session keys is safe only if we assume that they can never be broken. This is what is actually done in the standard Dolev-Yao enemy model, and consequently, in Cryptyc.

Session Keys in ρ-spi Since the above assumption is quite unrealistic, we find it safer to reject this kind of protocols, by requiring that session key freshness is always checked during the key exchange step, i.e., that the session key is actually authenticated before being used to exchange or authenticate other messages. This can be achieved by the same tags (i.e., Key and Key?) and types used for asymmetric cryptography and discussed above.

Interestingly, under this stronger assumption, it is possible to show that the safety theorem holds in a Dolev-Yao model extended with session key corruption. We briefly discuss how this model works. Let us assume that each sequential process performs a special action *init* every time it starts the computation and a special action *stop* when it terminates. (This can be easily achieved by forcing the presence of these actions in the process syntax.) Now we can model session key corruption by letting the enemy guess such keys only after the sessions where the keys are used have been closed, i.e., intuitively, only when keys' lifetime is expired. This is formalized in ρ-spi calculus as follows: let s, s' be two execution traces, $|\alpha|_s$ denote the number of occurrences of action α in trace s, and let $\models_E$ represent the deduction system for the standard Dolev Yao enemy model: the judgment $s \models_E k$ means that the enemy can learn k by observing trace s, without breaking cryptography. This standard attacker model can be now enriched by the following deduction rule:

$$\textsc{Broken Session Keys}\quad \frac{|\mathit{init}|_s = |\mathit{stop}|_s \qquad \mathit{fresh}(k) \in s \qquad s \models_E \{M_1,\ldots,M_n\}_k}{s.s' \models_E k}$$

Intuitively, given a trace $s.s'$, in which (i) s has no open sessions (i.e., $|\mathit{init}|_s = |\mathit{stop}|_s$), (ii) k has been freshly generated in s and (iii) k has been used at least once for encrypting a message, then k can be guessed by the enemy.

Requiring the number of *init* to be equal to the number of *stop* in s, guarantees that the enemy cannot break session keys which are still in use, as all the sessions that were active in s have already been terminated. Requiring k to be freshly generated in s is necessary to avoid long-term key corruption.

6. Conclusion

We have analyzed the relationships between the Gordon and Jeffrey's Cryptyc system and our ρ-spi system for the analysis of authentication protocols. Based on a translation of well-typed (and safe) ρ-spi protocols to corresponding well-typed (and safe) Cryptyc protocols, we have drawn a precise relationship between the two systems on a wide class of handshakes:

- PC handshakes in which nonces are sent out in clear, and returned either encrypted with a symmetric key or signed.
- CP handshakes in which nonces are sent out encrypted with a symmetric key and are returned in clear.

Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)
1063-6900/05 $20.00 © 2005 **IEEE**

- CC handshakes in which nonces are sent and returned encrypted with either a symmetric key or a public key.

The translation also reveals cases in which the ρ-spi tags give additional flexibility over Cryptyc (cf. Session 4). This flexibility is paid in ρ-spi in terms of the dynamic checks required on the structure of messages to validate the tags used in the different handshakes. Indeed, we do not regard the dynamic checks as limiting or inconvenient: instead, our contention is that the ρ-spi tagging of ciphertexts is a good specification practice which helps document the intended semantics of the ciphertexts and yields strong compositionality to the handshakes. In fact, as we note in [15], tagged messages may be interpreted without ambiguity, a property that makes it possible to safely compose sessions of arbitrary protocols, as long as the participants agree on the tagging. In addition, the translation given in this paper shows that our tags may be implemented directly into more conventional tags, like the injections employed in Cryptyc: indeed, using a different tag for every different protocol message is one of the possible solution to avoid interferences in real protocols.

We have also discussed the extensions needed to handle session keys in ρ-spi, and contrasted the resulting system with Cryptyc. The main difference is in the security properties that a session key must satisfy before being used for authenticating other messages. In Cryptyc it must be untainted, while in ρ-spi system it must be both untainted and fresh, since the last property is crucial when enemies can break old-session keys.

Plans for the future include work on the reverse translation, from Cryptyc to ρ-spi. This would characterize which Cryptyc idioms can be directly translated into ρ-spi and which cannot, better clarifying the relative merits of the two approaches. Moreover, we would obtain a type-reconstruction algorithm for (a subset of) Cryptyc, based on the tag reconstruction algorithm for ρ-spi [9]. Finally, as we mentioned earlier, our interest is more directly targeted at further investigating the use of tags as a mechanism for protocol specification, design (and consequently, analysis).

Acknowledgements We would like to thank the anonymous referees for their very helpful comments and suggestions.

References

[1] M. Abadi. Secrecy by typing in security protocols. *JACM*, 46(5):749–786, 1999.

[2] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theor. Comput. Sci.*, 298(3):387–415, 2003.

[3] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Automatic validation of protocol narration. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop (CSFW'03)*, pages 126–140. IEEE Computer Society Press, June 2003.

[4] C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. R. Nielson. Control flow analysis can find new flaws too. In *Proceedings of the Workshop on Issues on the Theory of Security (WITS'04)*, ENTCS. Elsevier, 2004.

[5] M. Bugliesi, R. Focardi, and M. Maffei. Authenticity by tagging and typing (full and revised version). Forthcoming.

[6] M. Bugliesi, R. Focardi, and M. Maffei. Compositional analysis of authentication protocols. In *Proceedings of European Symposium on Programming (ESOP 2004)*, volume 2986 of *Lecture Notes in Computer Science*, pages 140–154. Springer-Verlag, 2004.

[7] M. Bugliesi, R.Focardi, and M.Maffei. Authenticity by tagging and typing. In *2nd ACM Workshop on Formal Methods in Security Engineering: From Specifications to Code (FMSE 2004)*. ACM press, October 2004. Affiliated to CCS 2004.

[8] N. Durgin, J. Mitchell, and D. Pavlovic. A compositional logic for proving security properties of protocols. *Journal of Computer Security*, 11(4):677–721, 2003.

[9] R. Focardi, M. Maffei, and F. Placella. Inferring authentication tags. January 2005. In ACM Digital Library Proceedings of IFIP WG 1.7, ACM SIGPLAN and GI FoMSESS Workshop on Issues on the Theory of Security (WITS 2005).

[10] A. Gordon and A. Jeffrey. Cryptyc (cryptographic protocol type checker). http://cryptyc.cs.depaul.edu/.

[11] A. Gordon and A. Jeffrey. Personal communication. January 2005.

[12] A. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. *Journal of Computer Security*, 12(3/4):435–484, 2004.

[13] A. D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. *J. Comput. Secur.*, 11(4):451–519, 2004.

[14] A. Menezes, P. van Oorschot, and S. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[15] M.Maffei. Tags for multi-protocol authentication. In *Proceedings of the 2nd International Workshop on Security Issues in Coordination Models, Languages, and Systems (SECCO '04). To appear*. ENTCS, August 2004.

[16] J. Thayer, J. Herzog, and J. Guttman. Strand spaces: Proving security protocols correct. Journal of Computer Security, 1999. 7(2/3).

[17] T. Woo and S. Lam. "A Semantic Model for Authentication Protocols". In *Proceedings of 1993 IEEE Symposium on Security and Privacy*, pages 178–194, 1993.

Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)
1063-6900/05 $20.00 © 2005 **IEEE**

$Protocol_{\rho-spi} \triangleq$
let $k_A = \mathsf{asym\text{-}key}(A)$.
let $k_B = \mathsf{asym\text{-}key}(B)$.
$(A \triangleright !Initiator_{\rho-spi} \quad | \; B \triangleright !Responder_{\rho-spi})$

$Initiator_{\rho-spi} \triangleq$
$\mathsf{new}(k_s : \mathsf{SessionKey}(A,B))$.
$\mathsf{new}(n_A : \mathsf{Nonce}^{Priv}(A,B))$.
$1 : \mathsf{encrypt}\ \{|\mathsf{Id}(A), \mathsf{Key}(k_s),$
$\quad \mathsf{Verif}_{\mathsf{CC?}}(n_A)|\}_{\mathsf{Pub}(k_B)}\ \mathsf{as}\ x$
$\mathsf{out}(x)$.
$\mathsf{in}(y)$.
$2 : \mathsf{decrypt}\ y\ \mathsf{as}\ \{|\mathsf{Id}(B), \mathsf{Key?}(k_s),$
$\quad \mathsf{Claim}_{\mathsf{CC!}}(n_A), \mathsf{Verif}_{\mathsf{CP}}(x_B)|\}_{\mathsf{Priv}(k_A)}$.

$\mathsf{end}(B,A,k_s)$.

$\mathsf{begin}(A,B,k_s)$.
$3 : \mathsf{encrypt}\ \{x_B\}_{\mathsf{Pub}(k_B)}\ \mathsf{as}\ z$.
$\mathsf{out}(z)$
<use session key k_s>

$Responder_{\rho-spi} \triangleq$
$\mathsf{in}(x)$.
$1 : \mathsf{decrypt}\ x\ \mathsf{as}\ \{|\mathsf{Id}(A), \mathsf{Key}(x_k),$
$\quad \mathsf{Verif}_{\mathsf{CC?}}(x_n)|\}_{\mathsf{Priv}(k_B)}$.
$\mathsf{begin}(B,A,x_k)$.
$\mathsf{new}(n_B : \mathsf{Nonce}^{Pub}(B,A))$.
$2 : \mathsf{encrypt}\ \{|\mathsf{Id}(B), \mathsf{Key?}(x_k), \mathsf{Claim}_{\mathsf{CC!}}(x_n),$
$\quad \mathsf{Verif}_{\mathsf{CP}}(n_B)|\}_{\mathsf{Pub}(k_A)}\ \mathsf{as}\ z$.
$\mathsf{out}(z)$.
$\mathsf{in}(z')$.
$3 : \mathsf{decrypt}\ z'\ \mathsf{as}\ \{n_B\}_{\mathsf{Priv}(k_B)}$

$\mathsf{cast}\ x_k\ \mathsf{is}\ (y_k : \mathsf{SessionKey}(A,B))$
$\mathsf{end}(A,B,x_k)$.
<use session key x_k>

$Protocol_{Cryptyc} \triangleq$
$\mathsf{new}(k_A : \mathsf{Key}(T_A))$.
$\mathsf{new}(k_B : \mathsf{Key}(T_B))$.
$(!Initiator_{Cryptyc} \; | \; !Responder_{Cryptyc})$

$Initiator_{Cryptyc} \triangleq$
$\mathsf{new}(k_s : \mathsf{SharedKey}(T_{ses}))$.
$\mathsf{new}(n_A : Priv\mathsf{Chall}[\mathsf{end}(B,A,k_s)])$.

$\mathsf{out}(\{|msg_1(A,k_s,n_A)|\}_{\mathsf{Pub}(k_B)})$.
$\mathsf{in}(y)$.

$\mathsf{decrypt}\ y\ \mathsf{as}\ \{|msg_2(B,k_s,x_{n_A},x_B)|\}_{\mathsf{Priv}(k_A)}$.
$\mathsf{check}\ x_{n_A}\ \mathsf{is}\ n_A$.
$\mathsf{end}(B,A,k_s)$.
$\mathsf{witness}(k_s : \mathsf{SharedKey}(T_{ses}))$.
$\mathsf{begin}(A,B,k_s)$.
$\mathsf{cast}\ x_B\ \mathsf{is}\ (x'_B : Priv\mathsf{Resp}[])$

$\mathsf{out}(\{msg_3(x'_B)\}_{\mathsf{Pub}(k_B)})$
<use session key k_s>

$Responder_{Cryptyc} \triangleq$
$\mathsf{in}(x)$.

$\mathsf{decrypt}\ x\ \mathsf{as}\ \{|msg_1(A,x_k,x_n)|\}_{\mathsf{Priv}(k_B)}$.
$\mathsf{begin}(B,A,x_k)$.
$\mathsf{new}(n_B : Priv\mathsf{Chall}\left[\begin{array}{l}\mathsf{end}(A,B,x_k),\\ \mathsf{trust}\ x_k : \mathsf{SharedKey}(T)\end{array}\right])$

$\mathsf{cast}\ x_n\ \mathsf{is}\ (x'_n : Priv\mathsf{Resp}[])$.
$\mathsf{out}(\{|msg_2(B,x_k,x'_n,n_B)|\}_{\mathsf{Pub}(k_A)})$.
$\mathsf{in}(z')$.
$\mathsf{decrypt}\ z'\ \mathsf{as}\ \{msg_3(x_{n_B})\}_{\mathsf{Priv}(k_B)}$
$\mathsf{check}\ x_{n_B}\ \mathsf{is}\ n_B$.
$\mathsf{trust}\ x_k\ \mathsf{is}\ (y_k : \mathsf{SharedKey}(T_{ses}))$.
$\mathsf{end}(A,B,x_k)$.
<use session key y_k>

$$
\begin{array}{rcl}
T_B & = & msg_1(z_A : \mathsf{Un}, z_k : \mathsf{Top}\ , z_{n_A} : Priv\mathsf{Chall}[\mathsf{end}(B,A,z_k)]) \\
 & & \quad | \; msg_3(z_{n_B} : Priv\mathsf{Resp}[]) \\
T_A & = & msg_2(z_B : \mathsf{Un}, z_k : \mathsf{Top}\ , z_{n_A} : Private\ \mathsf{Response}\ [], \\
 & & \quad z_{n_B} : Priv\mathsf{Chall}[\mathsf{end}(A,B,z_k), \mathsf{trust}\ z_k : \mathsf{SharedKey}(T_{ses})])
\end{array}
$$

Table 7. Variant of the Needham Schroeder Public-Key with ρ-spi and Cryptyc types.

Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)
1063-6900/05 $20.00 © 2005 **IEEE**