

Pattern-matching Spi-calculus

Christian Haack and Alan Jeffrey

DePaul University

Abstract. Cryptographic protocols often make use of nested cryptographic primitives, for example signed message digests, or encrypted signed messages. Gordon and Jeffrey’s prior work on types for authenticity did not allow for such nested cryptography. In this work, we present the *pattern-matching spi-calculus*, which is an obvious extension of the spi-calculus to include pattern-matching as primitive. The novelty of the language is in the accompanying type system, which uses the same language of patterns to describe complex data dependencies which cannot be described using prior type systems. We show that any appropriately typed process is guaranteed to satisfy a strong robust safety property.

1 Introduction

Background. Cryptographic protocols are prone to subtle errors, in spite of the fact that they are often relatively small, and so are a suitable target for formal and automated verification methods. One line of such research is the development of domain-specific languages and logics, such as BAN logic [6], strand spaces [22], CSP [20, 21] MSR [8] and the spi-calculus [3]. These languages are based on the Dolev–Yao model of cryptography [10], and often use Woo and Lam’s correspondence assertions [23] to model authenticity. Techniques for proving correctness include rank functions [21, 16, 15], theorem provers [5, 19, 9], model checkers [17, 18] and type systems [1, 2, 7, 12, 13, 11].

Towards more complete and realistic cryptographic type systems. Type systems for interesting languages are incomplete, that is they fail to type-check some safe programs. Because of the subtlety of cryptographic protocols and their desired safety properties, this is especially true for cryptographic type systems. Such systems often cover some cryptographic primitives while neglecting others, for instance, [12, 11] does not include asymmetric cryptography, [2] treats public encryption keys but not signing keys and [13] covers full symmetric and asymmetric cryptography but not hashing. Moreover, even if primitives are included, it is often the case that some safe *and common* programs that use these primitives do not type-check, for example [13] treats full asymmetric cryptography, but does not allow for nested cryptography such as sign-then-encrypt. In this paper, we will use the techniques developed in [12, 13, 11] to reason about protocols making use of asymmetric cryptography, symmetric cryptography and hashing.

Small core language. While increasing the completeness of a cryptographic type system, it is also important to keep the system tractable, so that rigorous safety proofs are still feasible. For that reason, we chose to define a very small core language and obtain the full language through derived forms. The core language is extremely parsimonious: its only constructs for messages are tupling, asymmetric encryption and those for asymmetric keys. We show that symmetric encryption, hashing, and message tagging are all derived operators from this small core.

Authorization types. The language of types is small, too. It contains key types for key pairs, encryption and decryption keys. Moreover, it contains parameterized *authorization types* of the forms $\text{Public}(M)$ and $\text{Secret}(M)$. Typically, the parameter M is a list of principal names. For instance, if principal B receives from an untrusted channel a ciphertext $\{M\}_{esA}$ encrypted with A ’s private signing key esA , then the plaintext M is of type $\text{Public}(A, ())$, because M is a public message that has been authorized by A . On the other hand, if B receives a ciphertext $\{M\}_{epB}$ encrypted with B ’s public encryption key epB and it is clear from the context that this ciphertext comes from a trusted source, then the plaintext M has type $\text{Secret}(B)$, because it is a secret message that has been authorized for B .

Patterns and nested cryptography. The process language combines the suite of separate message destructor’s and equality checks from previous systems [12, 13, 11] into one pattern matching construct. Patterns at the process level are convenient, and are similar to the communication techniques used in

other specification languages [22, 8, 4]. Notably, our system uses patterns not only in processes but also in types. This permits types for nested use of cryptographic primitives, which would otherwise not be possible. For example, previous type systems [12, 13, 11] could express data dependencies such as

$$(\exists a : \text{Princ}, \exists m : \text{Msg}, \exists b : \text{Princ}, [! \text{begun}(a, b, m)])$$

where $! \text{begun}(a, b, m)$ is an *effect* ensuring that principals a and b have agreed on message m . In this paper, we extend these systems to deal with more complex data dependencies such as

$$\{ \#(\exists a : \text{Princ}, \exists m : \text{Msg}, \exists b : \text{Princ}) \}_{dk^{-1}} [! \text{begun}(a, b, m)]$$

where the effect $! \text{begun}(a, b, m)$ makes use of variables a , b and m which are doubly nested in the scope of a decryption $\{ \cdot \}_{dk^{-1}}$ and a hash function $\#(\cdot)$: such data dependencies were not previously allowed because the occurrences of a , b and m in $! \text{begun}(a, b, m)$ would be considered out of scope.

Reusable long-term keys. Another form of incompleteness is that previous systems have often been designed for verifying small (yet, subtle) protocol sketches in isolation, but not for verifying larger cryptographic systems where the same key may be used for multiple protocols. For instance, in [13] when a signing key for A is generated, its type specification fixes a finite number of message types that this key may sign. A more realistic approach for larger, possibly extensible, cryptosystems would be to generate a key for encrypting *arbitrary* data authorized by A . We show how the combination of key types, authorization types and message tagging allow keys to be generated independently of the protocols for which they will be used.

Notational conventions. If the meta-variable x ranges over set S , then \vec{x} ranges over finite sequences over S , and \bar{x} ranges over finite subsets of S .

2 An Introductory Example

Before the technical exposition, we want to convey a flavor of the type system by discussing a simple example. Consider the following simple sign-then-encrypt protocol:

$$\begin{array}{l} A \text{ begins! } (M, A, B) \\ A \rightarrow B \quad \{ \{ \text{sec}(M, B) \}_{esA} \}_{epB} \\ B \text{ ends } (M, A, B) \end{array}$$

Alice wants to send Bob a secret message M . To this end, she first tags M together with Bob's name with a message tag sec . Message tags are traditionally used so that receivers can identify which protocol a message belongs to and which role it plays in the protocol. In our system, message tags communicate additional important type information. Next, Alice encrypts the tagged message $sec(M, B)$ with her secret signing key esA to authenticate herself. Because M is to be kept secret, Alice finally encrypts the message with Bob's public encryption key. The begin- and end-statements are Woo-Lam correspondence specifications [23]. They specify that Alice begins a protocol session (M, A, B) , which Bob ends after message reception.

Protocol specification in pattern-matching spi. Here are Alice's and Bob's side of this protocol expressed in pattern-matching spi calculus:

$$\begin{array}{l} P_A \triangleq \text{begin!}(B, M); \text{out } net \{ \{ \text{sec}(B, M) \}_{esA} \}_{epB} \\ P_B \triangleq \text{inp } net \{ \{ \text{sec}(B, \exists x) \}_{dsA^{-1}} \}_{dpB^{-1}}; \text{end}(B, x) \end{array}$$

The variable net represents an untrusted channel and dsA and dpB are the matching decryption keys for esA and epB . An output statement of the form $(\text{out } net \ N)$ sends a message N out on channel net . A statement of the form $(\text{inp } net \ X; P)$ inputs a message from channel net and then attempts to match the message against pattern X . If the pattern match succeeds then P gets executed, otherwise execution gets stuck. Existentials in patterns indicate which variables get bound as part of the pattern match. In the input pattern above, the variable x gets bound, whereas B , dsA and dpB are constants that must be matched exactly.

Type annotations. For a type-checker to verify the protocol's correctness (and also for us to better understand and document it), it is necessary that we annotate the protocol with types. For our example, the types for the free variables are:

M : Secret	M will not be revealed to the opponent
epB : PublicCryptoEK(B)	epB is B 's public encryption key
dpB : PublicCryptoDK(B)	dpB is B 's matching decryption key
esA : SigningEK(A)	esA is A 's private signing key
dsA : SigningDK(A)	dsA is A 's matching signature verification key

No type annotations are necessary in P_A , because P_A does not have input statements. In P_B we add two type annotations. The input variable x is annotated with Secret. Moreover, we add a postcondition to the input statement that indicates that a (x, A, B) -session can safely be ended after a successful pattern match. Here is the annotated version of P_B :

$$P_B \triangleq \text{inp } net \{ \{ \{ sec(B, \exists x : \text{Secret}) \} \}_{dsA^{-1}} \}_{dpB^{-1}} [! \text{begun}(x, A, B)]; \text{end}(B, x)$$

These type annotations, together with our Robust Safety Theorem are enough to ensure the safety of this protocol in the presence of an arbitrary opponent.

3 A Spi Calculus with Pattern Matching

3.1 Messages

As usual in spi calculi, messages are modeled as elements of an algebraic datatype. They may be built from atomic names and variables by pairing and asymmetric-key encryption. Moreover, there are two special symbolic operators Enc and Dec with the following meanings: if message M represents a key pair, then $\text{Enc}(M)$ represents its encryption and $\text{Dec}(M)$ its decryption part. This language of messages is extremely parsimonious; in Section 5 we show how to introduce derived forms for constructs such as symmetric key encryption and message tagging.

In th presentation of messages, we include asymmetric-key encryption $\{M\}_N$ which encrypts plaintext M with encryption key N . We also allow messages $\{M\}_{N^{-1}}$ which represents the encryption of plaintext M with the encryption key which matches decryption key N . This is clearly not an implementable operation: it is used in the next section when we discuss *patterns*.

Messages:

x, y, z	variables
m, n	names
$L, M, N ::=$	message
n	name
x	variable
$()$	empty message
(M, N)	message pair
$\{M\}_N$	M encrypted under encryption key N
$\{M\}_{N^{-1}}$	M encrypted under inverse of decryption key N
$\text{Enc}(M)$	encryption part of key pair M
$\text{Dec}(M)$	decryption part of key pair M

Syntactic restriction: No subterms of the form $\{M\}_{(\text{Dec}(N))^{-1}}$.

Define: A message M is *implementable* if it contains no subterms of the form $\{M\}_{N^{-1}}$.

Because of the restriction that we never build a message of the form $\{M\}_{(\text{Dec}(N))^{-1}}$, we have to be careful with our definition of substitution. This is standard, except for when we substitute into a term of the form $\{M\}_{N^{-1}}$.

Substitution into Messages:

$$(\llbracket M \rrbracket_{N^{-1}})\{\sigma\} \triangleq \begin{cases} \llbracket M\{\sigma\} \rrbracket_{\text{Enc}(L)} & \text{if } N\{\sigma\} = \text{Dec}(L) \\ \llbracket M\{\sigma\} \rrbracket_{(N\{\sigma\})^{-1}} & \text{otherwise} \end{cases}$$

We will often use the following abbreviation: $(M_1, \dots, M_n) \triangleq (M_1, (\dots, (M_n, ()) \dots))$

3.2 Patterns

Patterns are of the form $\{\vec{x}.M \mid \bar{A}\}$, where M is a *pattern body* and \bar{A} an *assertion set*. Assertion sets are only used in type-checking, so we delay their discussion until Section 4.2. The variables \vec{x} act as binders. A message N matches a pattern $\{\vec{x}.M \mid \bar{A}\}$ if it is of the form $N = M\{\vec{x} \leftarrow \vec{L}\}$, in which case variables \vec{x} will be bound to messages \vec{L} . The pattern body M may have multiple occurrences of the same variable and it may contain variables that are not mentioned in \vec{x} : such variables are regarded as constants and must be matched exactly. For instance, the pattern $\{x.(x, \llbracket x \rrbracket_y) \mid \bar{A}\}$ is matched by messages of the form $(M, \llbracket M \rrbracket_y)$, but not by messages $(M, \llbracket M \rrbracket_z)$ or $(M, \llbracket N \rrbracket_y)$.

Patterns:

$X, Y, Z ::=$ pattern
 $\{\vec{x}.M \mid \bar{A}\}$ pattern matching term M binding \vec{x}

Syntactic restrictions: $\vec{x} \subseteq \text{fv}(M)$ and \vec{x} distinct.

Define: A pattern $\{\vec{x}.M \mid \bar{A}\}$ is *implementable* if $(\text{fn}(M), \text{fv}(M) - \vec{x}, M \Vdash \vec{x})$.

Importantly, not all patterns are implementable. For instance, the patterns $\{x.dk.\llbracket x \rrbracket_{dk^{-1}} \mid \bar{A}\}$ and $\{x.\llbracket x \rrbracket_{ek} \mid \bar{A}\}$ are not implementable, because they would allow access to the plaintext without knowing the decryption key. On the other hand, $\{x.\llbracket x \rrbracket_{dk^{-1}} \mid \bar{A}\}$ and $\{x.\llbracket x \rrbracket_{\text{Enc}(k)} \mid \bar{A}\}$ are implementable patterns. A syntactic restriction forbids non-implementable input patterns in processes. We formalize the notion of implementable pattern by making use of the Dolev–Yao ‘derivable message’ judgment $\bar{M} \Vdash \bar{N}$ meaning ‘An agent which knows messages \bar{M} can construct messages \bar{N} .’

Dolev–Yao Derivability, $\bar{M} \Vdash \bar{N}$:

(DY Id)	(DY And)	(DY Nil)	(DY Pair)	(DY Split)
$\frac{}{\bar{M}, N \Vdash N}$	$\frac{\bar{M} \Vdash N_1 \dots \bar{M} \Vdash N_k}{\bar{M} \Vdash N_1, \dots, N_k}$	$\frac{}{\bar{M} \Vdash ()}$	$\frac{\bar{M} \Vdash N, N'}{\bar{M} \Vdash (N, N')}$	$\frac{\bar{M}, N, N' \Vdash L}{\bar{M}, (N, N') \Vdash L}$
(DY Encrypt)	(DY Decrypt)	(DY Unencrypt)	(DY Key)	
$\frac{\bar{M} \Vdash N, N'}{\bar{M} \Vdash \llbracket N' \rrbracket_N}$	$\frac{\bar{M} \Vdash N \quad \bar{M}, N' \Vdash L}{\bar{M}, \llbracket N' \rrbracket_{N^{-1}} \Vdash L}$	$\frac{\bar{M} \Vdash N \quad \bar{M}, N' \Vdash L}{\bar{M}, \llbracket N' \rrbracket_{\text{Enc}(N)} \Vdash L}$	$\frac{\bar{M} \Vdash N \quad k \in \{\text{Enc}, \text{Dec}\}}{\bar{M} \Vdash k(N)}$	

We use some convenient syntactic abbreviations that treat patterns as if they were messages containing binding existentials. These ‘derived forms’ for patterns are defined below. For example:

$$\begin{aligned} & \llbracket \llbracket \text{sec}(B, \exists x : \text{Secret}) \rrbracket_{dsA^{-1}} \rrbracket_{dpB^{-1}} [\text{!begun}(x, A, B)] \\ & \equiv \{x. \llbracket \llbracket \text{sec}(B, x) \rrbracket_{dsA^{-1}} \rrbracket_{dpB^{-1}} \mid x : \text{Secret}, \text{!begun}(x, A, B)\} \end{aligned}$$

Derived Forms for Patterns:

$$\begin{aligned} M & \triangleq \{M \mid \}; \quad T \triangleq \{x.x \mid x : T\} \text{ for fresh } x; \quad \exists x \triangleq \{x.x \mid \}; \quad \dots \triangleq (\exists x) \text{ for fresh } x; \\ \llbracket X \rrbracket_N & \triangleq \{\vec{x}. \llbracket M \rrbracket_N \mid \bar{A}\}, \text{ if } X = \{\vec{x}.M \mid \bar{A}\}; \quad \llbracket X \rrbracket_{N^{-1}} \triangleq \{\vec{x}. \llbracket M \rrbracket_{N^{-1}} \mid \bar{A}\}, \text{ if } X = \{\vec{x}.M \mid \bar{A}\}; \\ X[\bar{B}] & \triangleq \{\vec{x}.M \mid \bar{A}, \bar{B}\}, \text{ if } X = \{\vec{x}.M \mid \bar{A}\}; \\ (X_1, \dots, X_n) & \triangleq \{\vec{x}_1, \dots, \vec{x}_n. (M_1, \dots, M_n) \mid \bar{A}_1, \dots, \bar{A}_n\}, \text{ if } X_i = \{\vec{x}_i.M_i \mid \bar{A}_i\} \end{aligned}$$

3.3 Processes

The spi-calculus with patterns is a variant of the spi-calculus, where we add pattern-matching as a primitive capability (in the spi-calculus it is derived).

Processes:

$O, P, Q, R ::=$	process
$\text{out } N M$	asynchronous output of M on N
$\text{inp } N X; P$	input from N against pattern X
$\text{new } n:T; P$	name generation
$P \mid Q$	parallel composition
$!P$	replication
0	inactivity

Syntactic restrictions:

- In $(\text{out } N M)$, both N and M are implementable messages.
- In $(\text{inp } N X; P)$, N is an implementable message and X is an implementable pattern.

Scope:

- The scope of \vec{x} in $(\text{inp } N \{\vec{x}. M \mid \bar{A}\}; P)$ is M , \bar{A} and P .
- The scope of n in $\text{new } n:T; P$ is P .

The interesting construct is input of the form $(\text{inp } N X; P)$. If a message L is received on channel N , then L is matched against pattern X . If this match succeeds, then it results in a binding and the process continuation P is executed under this binding. This is captured by the main reduction rule for pattern-matching spi:

$$(\text{out } L M \{\vec{x} \leftarrow \vec{N}\} \mid \text{inp } L \{\vec{x}. M \mid \bar{A}\}; P) \rightarrow (P \{\vec{x} \leftarrow \vec{N}\})$$

3.4 Specifying Authenticity by Correspondence Assertions

Following [12, 13, 11], we specify authenticity properties by inserting correspondence assertions into protocol specifications. In this paper, we only consider *non-injective* or *many-to-one* correspondences [11], where each begin-many assertion may match many end assertions. We do not believe that treating one-to-one correspondences should pose any technical difficulties, and postpone it for future work.

Correspondence Assertions:

$O, P, Q, R ::=$	process
\dots	as in Section 3.3
$\text{begin}!(L); P$	begin-many assertion
$\text{end}(L); P$	end assertion

A process is *safe* whenever at run-time each $\text{end}(M)$ is preceded by a $\text{begin}!(M)$ (precise definitions can be found in the appendix). For example, consider process P :

$$P \triangleq P_A \mid P_B, \text{ where } P_A \triangleq (\text{begin}!(M, A, B); \text{out } \text{net } (M, B))$$

$$P_B \triangleq (\text{inp } \text{net } (\exists x, B) [! \text{begun}(x, A, B)]; \text{end}(x, A, B))$$

Process P is safe in isolation, but we are really interested in safety in the presence of an opponent. An *opponent process* is an untyped process that does not contain correspondence assertions. A process P is called *robustly safe* whenever $(O \mid P)$ is safe for all opponent processes O . The example process P is not robustly safe, because $(\text{out } \text{net } N \mid P)$ is not safe, and we ensure robust safety by adding encryption:

$$P \triangleq \text{new } k : \text{SigningKP}(A); (\text{out } \text{net } (\text{Dec}(k)) \mid P_A(\text{Enc}(k)) \mid P_B(\text{Dec}(k)))$$

$$P_A(ek) \triangleq (\text{begin}!(M, A, B); \text{out } \text{net } \{\{M, B\}_{ek}\})$$

$$P_B(dk) \triangleq (\text{inp } \text{net } \{\exists x : \text{Public}, B\}_{dk^{-1}} [! \text{begun}(x, A, B)]; \text{end}(x, A, B))$$

Here, $\text{SigningKP}(A)$ is a type for principal A 's signing key pair and Public is a type for public, untainted data. The crucial property of our system is that processes that only make use of public data are robustly safe (we will return in Section 4.3 to the definition of a public type):

Theorem (Robust Safety) *If \vec{T} are public types and $(\vec{n} : \vec{T} \vdash P)$, then P is robustly safe.*

4 Highlights of the Type System

4.1 Environments

As is usual in most type systems, we give our judgments relative to a *typing environment*. In our case, this typing environment is used to:

- Track the names of bound variables, for example dk and x .
- Give types to messages, for example $dk : \text{SigningDK}(A)$ and $\{x, B\}_{dk^{-1}} : \text{Un}$.
- List the begun assertions which can be ended safely, for example $!\text{begun}(x, A, B)$.

The environment containing these assertions would be:

$$dk, x, dk : \text{SigningDK}(A), \{x, B\}_{dk^{-1}} : \text{Un}, !\text{begun}(x, A, B)$$

A significant difference to previous type systems for the spi-calculus [12, 13, 11] is that we are unifying the notions of *variable environment* and *process effect* into a common language of environments. In these previous papers, the above environment would be split into the variable environment $dk : \text{SigningDK}(A)$, and the process effect trust $(\{x, B\}_{dk^{-1}} : \text{Un}), !\text{begun}(x, A, B)$.

Environments:

$A, B, C, D ::=$	assertions
$M : T$	type assertion
$!\text{begun}(M)$	begun-many assertion
$E, F, G ::=$	environments
$\vec{x}; \vec{A}$	environment
$\text{dom}(\vec{x}; \vec{A}) \triangleq \vec{x}$	environment domain

We now discuss the interesting aspects of this type system; see Appendix B for details.

4.2 Typed Pattern Matching

We can now explain the assertion component of a pattern $\{\vec{x}. M \mid \vec{A}\}$: it gives the precondition \vec{A} which must be satisfied by any process that constructs a term matching the pattern. For example, the pattern $(\exists x : \text{Public}, B)[!\text{begun}(x, A, B)]$ is a derived form for $\{x. (x, B) \mid x : \text{Public}, !\text{begun}(x, A, B)\}$. For M to match this pattern in environment E , it must be the case that M is well-typed (i.e. M is of type Top), M is of the form $M = (N, B)$, message N is of type Public and $!\text{begun}(M, A, B)$ is contained in E .

Typed Pattern Matching (where $X = \{\vec{x}. N \mid \vec{A}\}$):

$E \vdash M \in X \triangleq E \vdash M : \text{Top}, \vec{A}\{\vec{x} \leftarrow \vec{N}\}$	where $M = N\{\vec{x} \leftarrow \vec{N}\}$	Match
$E, M \in X \vdash \mathcal{J} \triangleq E, M : \text{Top}, \vec{A}\{\vec{x} \leftarrow \vec{N}\} \vdash \mathcal{J}$	where $M = N\{\vec{x} \leftarrow \vec{N}\}$	Unmatch

4.3 Kinds and Subkinding

A message is *publishable* if it may be sent to an untrusted target. For instance, ciphertext $\{M\}_{ek}$ is publishable if the decryption key for ek is secret. If ek is a signing key, whose corresponding decryption key is public, then $\{M\}_{ek}$ is only publishable if M is already publishable.

A message is *untainted* if it has been received from a trusted source. For instance, if dk is a signature verification key whose matching signing key is secret, then M is untainted even if $\{M\}_{dk^{-1}}$ is tainted. If dk is a key whose matching encryption key is public, then M is only untainted if $\{M\}_{dk^{-1}}$ is already untainted.

An important part of the type system is a *kinding relation* $(T :: K)$ that assigns kinds K to types T . The type system is designed so that the following statements hold:

- If $(T :: K)$ and $\text{Public} \in K$, then members of type T are publishable.
- If $(T :: K)$ and $\text{Tainted} \notin K$, then members of type T are untainted.

We say that type T is *public* (respectively *tainted*) if $(T :: K \ni \text{Public})$ (respectively $(T :: K \ni \text{Tainted})$) for some kind K .

Kinds and Subkinding:

$$K, H, J \subseteq \{\text{Public}, \text{Tainted}\} \quad \frac{(\text{Public} \in H) \Rightarrow (\text{Public} \in K) \quad (\text{Tainted} \in K) \Rightarrow (\text{Tainted} \in H)}{K \leq H}$$

4.4 Types and Subtyping

We will now give the grammar of types, together with the definition of kinding and subtyping. We discuss each of the types in more detail below.

Types:

$T, U, V ::=$	types
$K \text{ Top}$	top type
$K \text{ Auth}(L)$	authorized type
$(K, H) \text{ KP}(X)$	key type
$\text{KT} ::=$	key type symbols
EK	encryption key
DK	decryption key
KP	key pair

Kinding $T :: K$:

$$K \text{ Top} :: K; \quad K \text{ Auth}(L) :: K; \quad (K, H) \text{ KP}(X) :: K \cap H; \quad (K, H) \text{ EK}(X) :: K; \quad (K, H) \text{ DK}(X) :: H$$

Kinds are used to define subtyping. The rule (Subty Public Tainted) states that any message of public type also has any tainted type, as in [13]. The subtyping rules (Subty Top) and (Subty Auth) are new and have not been part of [13].

Subtyping, $T \leq U$:

(Subty Refl)	(Subty Top)	(Subty Auth)	(Subty Public Tainted)
$\frac{}{T \leq T}$	$\frac{T :: K \quad K \leq H}{T \leq H \text{ Top}}$	$\frac{K \leq H}{K \text{ Auth}(L) \leq H \text{ Auth}(L)}$	$\frac{T :: K \cup \{\text{Public}\} \quad U :: H \cup \{\text{Tainted}\}}{T \leq U}$

4.5 Top Types

Top types have the form $K \text{ Top}$ and are the most general types of kind K , by (Subty Top). Moreover, $\{\text{Tainted}\} \text{ Top}$ is the greatest type of the entire type hierarchy. We define the following derived forms:

Derived Forms for Top Types:

$$\text{Secret} \triangleq \emptyset \text{ Top}; \quad \text{Public} \triangleq \{\text{Public}\} \text{ Top}; \quad \text{Top} \triangleq \text{Tainted} \triangleq \{\text{Tainted}\} \text{ Top}; \\ \text{Un} \triangleq \{\text{Public}, \text{Tainted}\} \text{ Top}$$

4.6 Authorization Types

A novel feature of this type system is *authorization types*. A message $M : K \text{ Auth}(L)$ is a message of kind K which requires authorization by or for L .

Derived Forms for Authorization Types:

$$\text{Secret}(L) \triangleq \emptyset \text{ Auth}(L); \quad \text{Tainted}(L) \triangleq \{\text{Tainted}\} \text{ Auth}(L); \\ \text{Public}(L) \triangleq \{\text{Public}\} \text{ Auth}(L); \quad \text{Un}(L) \triangleq \{\text{Public}, \text{Tainted}\} \text{ Auth}(L)$$

In meaningful authorization types, parameter L is either a list of principal names (possibly terminated by the empty message) or a single principal name. For example, $\text{Public}(A, B, C, ())$ is the type of public messages M that require authorizations *by* principals A , B and C . These authorizations are acquired by A , B and C digitally signing M . On the other hand, $\text{Secret}(A)$ is the type of secret messages M that require authorization *for* A . This authorization is acquired by encrypting M with A 's public encryption key.

4.7 Key Types

In this system, key types are extremely general: in examples, we will often use specialized derived key types for applications such as signing, as discussed in Section 5.2.

The key type $(K, H)KT(X)$ contains a pattern X . These keys will be used to encrypt plaintext messages M to produce ciphertexts which have an authorization type $J\text{Auth}(L)$. In order to form the ciphertext, we require the pair (M, L) to match the pattern X . The key type $(K, H)KT(X)$ also contains a kind K , which is the kind of the encryption key, and a kind H , which is the kind of the decryption key. For example, in Section 5.2 we define principal A 's signing key to be:

$$\text{SigningEK}(A) \triangleq (\emptyset, \{\text{Public}\})\text{EK}(\exists x : \text{Secret}(A, y), \exists y)$$

A key esA of type $\text{SigningEK}(A)$ is a secret encryption key, whose matching decryption key is public. Thus, it is a signing key. It is typically used to encrypt messages M of type $\text{Public}(A, \vec{B})$ to produce ciphertexts $\{\!\{M}\!\}_{esA}$ of type $\text{Public}(\vec{B})$: thus, by signing the message, A removes her name from the list of principals required to authorize it.

4.8 Output and Input

The interesting rules for the process judgment $E \vdash P$ are for input and output. We assume that every channel is untrusted, that is of type Un . Untrusted channels may transmit messages of type Un . Moreover, untrusted channel names are themselves of type Un , and may, thus, be sent and received on untrusted channels:

$$\frac{E \vdash N : \text{Un}, M : \text{Un}}{E \vdash \text{out } N M} \quad \frac{\vec{x} \cap \text{dom}(E) = \emptyset \quad E \vdash N : \text{Un} \quad \vec{x}, E, M : \text{Un} \vdash \bar{A} \quad \vec{x}, E, \bar{A} \vdash P}{E \vdash \text{inp } N \{\vec{x}. M \mid \bar{A}\}; P}$$

In the output rule, message M has to be of type Un in order to be sent out on the untrusted channel N . Note that M may also be sent out if M 's type is any other public type, because each public type is a subtype of Un .

4.9 Encryption

There are two typing rules for encryption, which only differ in the kind attributes of the types. The first rule applies to encryption with a trusted key:

$$\frac{\text{Tainted} \notin K \cup H^{-1} \quad E \vdash N : (K, H)\text{EK}(X), (M, L) \in X}{E \vdash \{\!\{M}\!\}_N : \text{Public}(L)} \quad \begin{array}{l} \text{Public}^{-1} \triangleq \text{Tainted} \\ \text{Tainted}^{-1} \triangleq \text{Public} \end{array}$$

The condition $\text{Tainted} \notin K \cup H^{-1}$ expresses that the ciphertext is only publishable if the encryption key is untainted and the corresponding decryption key is not public. Otherwise, the following rule is used for encryption:

$$\frac{\text{Tainted} \in K \cup H^{-1} \quad J = (J' - \{\text{Tainted}\}) \cup (K - \{\text{Public}\}) \quad E \vdash N : (K, H)\text{EK}(X), (M, L) \in X, M : J' \text{Top}}{E \vdash \{\!\{M}\!\}_N : J\text{Auth}(L)}$$

Note that here the ciphertext type $J\text{Auth}(L)$ is only public if the plaintext type $J' \text{Top}$ is public, and is tainted if the encryption key is tainted.

4.10 Decryption

There are two typing rules for decryption, which only differ in how they treat kinds and authorizations. The first rule applies if both the decryption key and the ciphertext are untainted, and is the inverse of the rule for encryption with a trusted key:

$$\frac{\text{Tainted} \notin H \cup J \quad E \vdash N : (K, H) \text{DK}(X) \quad E, (M, L) \in X \vdash B}{E, \{\!|M|\!\}_{N^{-1}} : J \text{Auth}(L) \vdash B}$$

The second decryption rule applies if we cannot trust the ciphertext; in particular we do not know who has authorized the ciphertext:

$$\frac{\text{Tainted} \in J \quad E \vdash N : (K, H) \text{DK}(X) \quad x, E, (M, x) \in X \vdash B \quad (\text{Tainted} \in H \cup K^{-1}) \Rightarrow (x, E, M : J \text{Top}, x : \text{Top} \vdash (M, x) \in X)}{E, \{\!|M|\!\}_{N^{-1}} : J \text{Top} \vdash B}$$

Note that when we apply this rule, the authorization is unknown, so we replace it by a fresh variable x , which acts as a placeholder for the ‘real’ authorization. If the decryption key is untrusted, then we have an additional requirement: we can only add (M, x) to the assumption list if it is derivable from $(x, E, M : J \text{Top}, x : \text{Top})$; as a result, untrusted keys can only be used when the pattern X is quite ‘weak’.

4.11 Summary of Technical Properties

Here is a brief outline of the crucial technical properties of the type system, as proved in a longer version of this paper [14]. Firstly, the type system satisfies substitutivity in the following sense:

Theorem (Substitutivity) *If $(E \vdash J)$ and $\text{dom}(E) = \text{dom}(\sigma)$, then $(E\{\sigma\} \vdash J\{\sigma\})$*

Note that there is no condition on the substitution σ , except that its domain must coincide with E ’s domain. The second important theorem is (Cut):

Theorem (Cut) *If E is nominal, $(E \vdash \bar{A})$ and $(\bar{A} \vdash \bar{B})$, then $(E \vdash \bar{B})$.*

An environment E is nominal, if $\text{dom}(E) = \emptyset$, each type assertion in E is of the form $(n : T)$, and there are no duplicate type assertions $n : T, n : U$. The nominality condition is needed to ensure that E is ‘key-unique’, that is, E does not assign two unrelated key types to the same name. Using substitutivity and (Cut), it is not so hard to prove that state reductions preserve well-typedness:

Theorem (Type Preservation) *If $\vdash \bar{A} :: P$ and $(\bar{A} :: P) \rightarrow (\bar{B} :: Q)$, then $\vdash \bar{B} :: Q$.*

Safety is a simple corollary of type preservation:

Theorem (Safety) *If $(\bar{n} : \bar{T} \vdash P)$, then P is safe.*

As usual, we model opponent processes as well-typed, too. An *opponent process* is a process with no correspondence assertions, where all type annotations on names are Un and all input patterns are of the form $\{\bar{x}. M \mid M : \text{Un}\}$ for some \bar{x}, M .

Theorem (Dolev-Yao \Rightarrow Typability) *If $(\bar{M} \Vdash \bar{N})$ and $\text{fv}(\bar{M}, \bar{N}) = \bar{x}$, then $(\bar{x}; \bar{M} : \text{Un} \vdash \bar{N} : \text{Un})$.*

Theorem (Opponent Typability) *If P is an opponent process, $\text{fn}(P) = \bar{n}$ and $\text{fv}(P) = \bar{x}$, then $(\bar{x}; \bar{n} : \text{Un}, \bar{x} : \text{Un} \vdash P)$.*

Robust safety is a simple corollary of safety and opponent typability:

Theorem (Robust Safety) *If \bar{T} are public types and $(\bar{n} : \bar{T} \vdash P)$, then P is robustly safe.*

5 Derived Forms and Examples

5.1 Tagging

In previous type systems for cryptographic protocols [12, 13, 11], message tags were introduced using *tagged union types*. These types are sound, and they allow a key to be used in more than one protocol, but unfortunately they require the protocol suite to be known *before* the key is generated, since the plaintext type of the key is given as the tagged union of all the messages in the protocol suite. In this paper, we adopt a variant of *dynamic types* to allow a key to be generated with no knowledge of the protocol suite it will be used for.

In our system, we give message tags a type of the form $\ell : X \rightarrow \text{Auth}(Y)$, which can be used to tag messages M of kind $(J \cup \text{Tainted})$ to get tagged messages $\ell(M) : J \text{Auth}(L)$. For example, our previous protocol becomes:

$$\begin{aligned} P &\triangleq \text{new } k : \text{SigningKP}(A); (\text{out } \text{net} (\text{Dec}(k)) \mid P_A(\text{Enc}(k)) \mid P_B(\text{Dec}(k))) \\ P_A(ek) &\triangleq \text{begin!}(M, A, B); \text{out } \text{net} \{\!\{ \text{snd}(M, B) \}\!\}_{ek} \\ P_B(dk) &\triangleq \text{inp } \text{net} \{\!\{ \text{snd}(\exists x : \text{Public}, B) \}\!\}_{dk^{-1}}[\text{!begun}(x, A, B)]; \text{end}(x, A, B) \\ \text{snd} &: (\exists x : \text{Public}, \exists b : \text{Public}) \rightarrow \text{Auth}(\exists a : \text{Public}, \dots)[\text{!begun}(x, a, b)] \end{aligned}$$

Tags are not primitive in the pattern-matching spi-calculus, instead we can encode tags as public key pairs, and message tagging as encryption. We treat message tags ℓ as names with a globally agreed type.

$$\begin{aligned} \ell(M) &\triangleq \{\!\{ M \}\!\}_{\text{Enc}(\ell)}; & \ell(X) &\triangleq \{\!\{ X \}\!\}_{\text{Enc}(\ell)}; \\ (X \rightarrow \text{Auth}(Y)) &\triangleq (\{\!\{ \text{Public} \}\!\}, \{\!\{ \text{Public} \}\!\}) \text{KP}(X, Y) \end{aligned}$$

We obtain the following derived typing rules for tagging:

$$\begin{array}{c} \frac{E \vdash \ell : X \rightarrow \text{Auth}(Y), M : (J \cup \{\text{Tainted}\}) \text{Top}, (M, L) \in (X, Y)}{E \vdash \ell(M) : J \text{Auth}(L)} \\ \frac{\text{Tainted} \notin J \quad E \vdash \ell : X \rightarrow \text{Auth}(Y) \quad E, (M, L) \in (X, Y) \vdash B}{E, \ell(M) : J \text{Auth}(L) \vdash B} \\ \frac{\text{Tainted} \in J \quad E \vdash \ell : X \rightarrow \text{Auth}(Y) \quad E, M : J \text{Top} \vdash M \in X \quad E, M \in X \vdash B}{E, \ell(M) : J \text{Top} \vdash B} \end{array}$$

5.2 Signing Keys

A goal of this type system is to allow principals to have just one signing key, which can be used for any protocol, rather than requiring different signing key types for different protocols. Message tags are then used to ensure the correctness of each protocol.

The type for a signing key is designed to support nested signatures, for example $\{\!\{ \{\!\{ M \}\!\}_{esA} \}\!\}_{esB}$ is a message M signed by A (using her signing key $esA : \text{SigningEK}(A)$) and B (using his signing key $esB : \text{SigningEK}(B)$). This signed message can be given type $\{\!\{ \{\!\{ M \}\!\}_{esA} \}\!\}_{esB} : \text{Secret}$ as long as $M : \text{Secret}(A, B, y)$ for some y , and can be given type $\{\!\{ \{\!\{ M \}\!\}_{esA} \}\!\}_{esB} : \text{Public}$ as long as $M : \text{Public}(A, B, y)$ for some y . This form of nested signing was not supported by [12, 13, 11].

$$\begin{array}{c} \text{SigningKT}(L) \triangleq (\emptyset, \{\!\{ \text{Public} \}\!\}) \text{KT}(\exists x : \text{Secret}(L, y), \exists y) \\ \frac{\text{Tainted} \notin J \quad E \vdash N : \text{SigningEK}(L), M : J \text{Auth}(L, L')}{E \vdash \{\!\{ M \}\!\}_N : J \text{Auth}(L')} \\ \frac{\text{Tainted} \notin J \quad E \vdash N : \text{SigningDK}(L) \quad E, M : \text{Secret}(L, L') \vdash B}{E, \{\!\{ M \}\!\}_{N^{-1}} : J \text{Auth}(L') \vdash B} \\ \frac{\text{Tainted} \in J \quad E \vdash N : \text{SigningDK}(L) \quad x, E, M : \text{Secret}(L, x) \vdash B}{E, \{\!\{ M \}\!\}_{N^{-1}} : J \text{Top} \vdash B} \end{array}$$

These type rules are enough to verify the protocol in Section 5.1. A long version of this paper [14] contains a proof that this protocol is well-typed.

5.3 Public Encryption Keys

Public encryption is dual to signing: the encryption key is public, and the decryption key is kept secret. One crucial difference is that although our type system supports nested uses of signatures, it does not support similar nested uses of public-key encryption. As a result, although we can support sign-then-encrypt, we cannot support encrypt-then-sign, due to the well-known problems with encrypt-then-sign applications (see, for instance, the analysis of the CCITT X.509 protocol in [6]).

$$\begin{array}{c} \text{PublicCryptoKT}(L) \triangleq (\{\text{Public}\}, \emptyset) \text{KT}(\exists x : \text{Secret}(L), \dots) \\ \frac{E \vdash N : \text{PublicCryptoEK}(L), M : \text{Secret}(L)}{E \vdash \{M\}_N : \text{Un}} \\ \frac{E \vdash N : \text{PublicCryptoDK}(L) \quad E, M : \text{Un} \vdash M : \text{Secret}(L) \quad E, M : \text{Secret}(L) \vdash B}{E, \{M\}_{N^{-1}} : \text{Un} \vdash B} \end{array}$$

The following example was not typable in previous Cryptyc systems:

$$\begin{array}{l} A \text{ begins! } (M, A, B) \\ A \rightarrow B \quad \{\{sec(M, B)\}_{esA}\}_{epB} \\ B \text{ ends } (M, A, B) \end{array}$$

It can now be typechecked using the types:

$$\begin{array}{l} esA : \text{SigningEK}(A) \\ epB : \text{PublicCryptoEK}(B) \\ sec : (\exists x : \text{Secret}(a, b), \exists b : \text{Public}) \rightarrow \text{Auth}(\exists a : \text{Public}, \dots)[!begun(x, a, b)] \end{array}$$

This is very similar to the example in Section 5.1 except that now the message is a secret shared between A and B . A long version of this paper [14] contains a proof that this protocol type-checks.

5.4 Symmetric Keys

Symmetric cryptography is not primitive in pattern-matching spi-calculus, instead we encode it using asymmetric cryptography:

$$\begin{array}{c} \{M\}_N \triangleq \{M\}_{\text{Enc}(N)}; \quad \{X\}_N \triangleq \{X\}_{\text{Enc}(N)}; \\ \text{SymK}(X) \triangleq (\emptyset, \emptyset) \text{KP}(X, \dots) \\ \frac{E \vdash N : \text{SymK}(X), M \in X}{E \vdash \{M\}_N : \text{Un}} \quad \frac{E \vdash N : \text{SymK}(X) \quad E, M \in X \vdash B}{E, \{M\}_N : \text{Un} \vdash B} \end{array}$$

These are the obvious analogues of the rules for encryption given in [12].

5.5 Hashing

We can encode hashing as encryption with a hashing key, where the matching decryption key has been discarded.

$$\begin{array}{c} \#(M) \triangleq \text{hash}(\{M\}_{ekH}) \text{ where } ekH : (\{\text{Public}\}, \emptyset) \text{EK}(\dots) \\ \text{and } \text{hash} : \{\exists x : \text{Secret}(y)\}_{ekH} \rightarrow \text{Auth}(\exists y) \\ \frac{E \vdash M : \text{Secret}(L)}{E \vdash \#(M) : \text{Public}(L)} \quad \frac{E, M : \text{Secret}(L) \vdash B}{E, \#(M) : \text{Secret}(L) \vdash B} \end{array}$$

From this point on, we assume that each environment E is implicitly extended by the above type assertions for the special global names ekH and $hash$. We can then adapt the example from Section 5.1 to allow A to sign the message digest of M rather than signing the entire message:

$$\begin{array}{l} A \text{ begins! } (M, A, B) \\ A \rightarrow B \quad M, \{\#\text{snd}(B, M)\}_{esA} \\ B \text{ ends } (M, A, B) \end{array}$$

This example uses the types which were introduced in previous examples, a full version is given in a long version of this paper [14].

6 Conclusions and Further Work

In this paper, we have shown how pattern-matching types can be used to express complex data dependencies, in particular how they can be used to provide authenticity typings for nested uses of cryptography. This work shows that such types can be used to prove many-to-one correspondences; we expect the extension to one-to-one correspondences to be fairly straightforward, although it will require some notion of *linear* environment. We also hope to extend the attacker model, for example to cope with key compromise.

A Operational Semantics

Structural Process Equivalence, $P \equiv Q$:

$P \equiv P$	(Struct Refl)
$P \equiv Q \Rightarrow Q \equiv P$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow P \mid Q \equiv P \mid R$	(Struct Par)
$P \mid \mathbf{0} \equiv P$	(Struct Par Zero)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$!P \equiv P \mid !P$	(Struct Repl Par)

State Transition, $(\bar{A} :: P) \rightarrow (\bar{B} :: Q)$:

(Redn Equiv)	
$\frac{P \equiv P' \quad (\bar{A} :: P') \rightarrow (\bar{B} :: Q') \quad Q' \equiv Q}{(\bar{A} :: P) \rightarrow (\bar{B} :: Q)}$	
$n \notin \text{fn}(\bar{A}, Q) \Rightarrow (\bar{A} :: (\text{new } n:T; P) \mid Q) \rightarrow (\bar{A}, n:T :: P \mid Q)$	(Redn New)
$(\bar{A} :: (\text{begin}!(M); P) \mid Q) \rightarrow (\bar{A}, !\text{begun}(M) :: P \mid Q)$	(Redn Begin)
$(\bar{A}, !\text{begun}(M) :: (\text{end}(M); P) \mid Q) \rightarrow (\bar{A}, !\text{begun}(M) :: P \mid Q)$	(Redn End)
$(\bar{A} :: (\text{out } L M \{\vec{x} \leftarrow \vec{N}\} \mid \text{inp } L \{\vec{x}. M \mid \bar{A}\}; P) \mid Q) \rightarrow (\bar{A} :: P \{\vec{x} \leftarrow \vec{N}\} \mid Q)$	(Redn IO)

B Type System

Good Environment, $E \vdash \diamond$:

(Good Env)
$\frac{\text{fv}(\bar{A}) \subseteq \bar{x}}{\bar{x}; \bar{A} \vdash \diamond}$

Valid Assertions, $E \vdash \bar{A}$:

(Id)	(And)	(Empty)
$\frac{}{E, A \vdash A}$	$\frac{}{E \vdash A_1 \quad \dots \quad E \vdash A_n \quad n \geq 0}$	$\frac{}{E \vdash \diamond}$
$E, A \vdash A$	$E \vdash A_1, \dots, A_n$	$E \vdash () : \text{Public}$
(Sub)	(Pair)	
$\frac{E \vdash M : T \quad T \leq U \quad \text{fv}(U) \subseteq \text{dom}(E)}{E \vdash M : U}$	$\frac{E \vdash M : K \text{ Top}, N : K \text{ Top}}{E \vdash (M, N) : K \text{ Top}}$	
(Enc Part)	(Dec Part)	
$\frac{E \vdash M : (K, H) \text{ KP}(X)}{E \vdash \text{Enc}(M) : (K, H) \text{ EK}(X)}$	$\frac{E \vdash M : (K, H) \text{ KP}(X)}{E \vdash \text{Dec}(M) : (K, H) \text{ DK}(X)}$	

(Encrypt Trusted)

Tainted $\notin K \cup H^{-1}$

$E \vdash N : (K, H) \text{EK}(X), (M, L) \in X$

$E \vdash \{\!\{M}\!\}_N : \text{Public}(L)$

(Encrypt Untrusted)

Tainted $\in K \cup H^{-1} \quad J = (J' - \{\text{Tainted}\}) \cup (K - \{\text{Public}\})$

$E \vdash N : (K, H) \text{EK}(X), (M, L) \in X, M : J' \text{Top}$

$E \vdash \{\!\{M}\!\}_N : J \text{Auth}(L)$

Left Rules, $E, \bar{A} \vdash B$:

(Unsub)

$\text{fv}(T) \subseteq \text{dom}(E)$

$E, M : U \vdash A \quad T \leq U$

$E, M : T \vdash A$

(Split)

$E, M : K \text{Top}, N : K \text{Top} \vdash A$

$E, (M, N) : K \text{Top} \vdash A$

(Decrypt Trusted)

Tainted $\notin H \cup J$

$E \vdash N : (K, H) \text{DK}(X) \quad E, (M, L) \in X \vdash B$

$E, \text{decrypt}(M, N) : J \text{Auth}(L) \vdash B$

(Decrypt Untrusted)

Tainted $\in J \quad E \vdash N : (K, H) \text{DK}(X) \quad x \notin \text{dom}(E) \quad x, E, (M, x) \in X \vdash B$

$(\text{Tainted} \in H \cup K^{-1}) \Rightarrow (x, E, M : J \text{Top}, x : \text{Top} \vdash (M, x) \in X)$

$E, \text{decrypt}(M, N) : J \text{Top} \vdash B$

where $\text{decrypt}(M, N) \triangleq \begin{cases} \{\!\{M}\!\}_{\text{Enc}(L)} & \text{if } N = \text{Dec}(L) \\ \{\!\{M}\!\}_{N^{-1}} & \text{otherwise} \end{cases}$

Well-typed Processes, $E \vdash P$:

(Proc Out)

$E \vdash N : \text{Un}, M : \text{Un}$

$E \vdash \text{out } N M$

(Proc In)

$\bar{x} \cap \text{dom}(E) = \emptyset$

$E \vdash N : \text{Un} \quad \bar{x}, E, M : \text{Un} \vdash \bar{A} \quad \bar{x}, E, \bar{A} \vdash P$

$E \vdash \text{inp } N \{\bar{x}. M \mid \bar{A}\}; P$

(Proc New)

$E, n : T \vdash P \quad n \notin \text{fn}(E)$

$E \vdash \text{new } n:T; P$

(Proc Par)

$E \vdash P \quad E \vdash Q$

$E \vdash P \mid Q$

(Proc Repl)

$E \vdash P$

$E \vdash !P$

(Proc Stop)

$E \vdash \diamond$

$E \vdash \mathbf{0}$

(Proc Begin Many)

$E, !\text{begun}(M) \vdash P$

$E \vdash \text{begin}!(M); P$

(Proc End)

$E \vdash !\text{begun}(M) \quad E \vdash P$

$E \vdash \text{end}(M); P$

Well-typed Computation States, $\vdash \bar{A} \text{ ::: } P$:

(State)

$\bar{A} \text{ nominal} \quad \bar{A} \vdash \bar{A}' \quad \bar{A}' \vdash P$

$\vdash \bar{A} \text{ ::: } P$

References

1. M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999.
2. M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In *Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 25–41. Springer, 2001.
3. M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999.
4. C. Bodei, M. Buchholtz, P. Degano, F. Nielson, and H. Riis Nielson. Automatic validation of protocol narration. In *Proc. CSFW03*, pages 126–140. IEEE Press, 2003.
5. D. Bolignano. An approach to the formal verification of cryptographic protocols. In *Third ACM Conference on Computer and Communications Security*, pages 106–118, 1996.
6. M. Burrows, M. Abadi, and R.M. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.
7. I. Cervesato. Typed MSR: Syntax and examples. In *First International Workshop on Mathematical Methods, Models and Architectures for Computer Network Security*, volume 2052 of *Lecture Notes in Computer Science*, pages 159–177. Springer, 2001.
8. I. Cervesato, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. A meta-notation for protocol analysis. In *Proc. IEEE Computer Security Foundations Workshop*, pages 55–69, 1999.
9. E. Cohen. TAPS: A first-order verifier for cryptographic protocols. In *13th IEEE Computer Security Foundations Workshop*, pages 144–158. IEEE Computer Society Press, 2000.
10. D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(2):198–208, 1983.
11. A. D. Gordon and A. S. A. Jeffrey. Typing one-to-one and one-to-many correspondences in security protocols. In *Proc. Int. Software Security Symp.*, volume 2609 of *Lecture Notes in Computer Science*, pages 263–282. Springer-Verlag, 2002.
12. A.D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop*, pages 145–159. IEEE Computer Society Press, 2001.
13. A.D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. In *15th IEEE Computer Security Foundations Workshop*, pages 77–91. IEEE Computer Society Press, 2002.
14. C. Haack and A. S. A. Jeffrey. Pattern-matching spi-calculus (longer draft). Available from <http://fpl.cs.depaul.edu/ajeffrey/fast04Long.pdf>, 2004.
15. J. Heather. ‘Oh! . . . Is it really you?’ Using rank functions to verify authentication protocols. PhD thesis, Royal Holloway, University of London, 2000.
16. J. Heather and S. Schneider. Towards automatic verification of authentication protocols on an unbounded network. In *13th IEEE Computer Security Foundations Workshop*, pages 132–143. IEEE Computer Society Press, 2000.
17. G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer, 1996.
18. W. Marrero, E.M. Clarke, and S. Jha. Model checking for security protocols. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*, 1997. Preliminary version appears as Technical Report TR-CMU-CS-97-139, Carnegie Mellon University, May 1997.
19. L.C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.
20. A.W. Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *8th IEEE Computer Security Foundations Workshop*, pages 98–107. IEEE Computer Society Press, 1995.
21. S.A. Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9):741–758, 1998.
22. F.J. Thayer Fábrega, J.C. Herzog, and J.D. Guttman. Strand spaces: Why is a security protocol correct? In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 160–171, 1998.
23. T.Y.C. Woo and S.S. Lam. A semantic model for authentication protocols. In *IEEE Computer Society Symposium on Research in Security and Privacy*, pages 178–194, 1993.