PATTERN-MATCHING SPI-CALCULUS*

Christian Haack
DePaul University

Alan Jeffrey
Bell Labs, Lucent Technologies
and 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. Type systems usually are tailored to a particular idiom, for example [2] treats public encryption keys but not signing keys, and [13]

*This material is based upon work supported by the National Science Foundation under Grant No. 0208459.

covers full symmetric and asymmetric cryptography but not nested uses of cryptography. In this paper, we will use the techniques developed in [12, 13, 11] to reason about protocols making use of nested 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 $\mathsf{Public}(M)$ and $\mathsf{Secret}(M)$. 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 $\mathsf{Public}\langle A\rangle$, because M is a public message that has been authorized *by* A.

Patterns and nested cryptography. The process language combines the suite of separate message destructors and equality checks from previous systems [12, 13, 11] into one patten 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

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

where $!\mathsf{begun}(a, b, m)$ 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

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

where the effect $!\mathsf{begun}(a, b, m)$ 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 $!\mathsf{begun}(a, b, m)$ 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:

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

The begin- and end-statements are Woo-Lam correspondence assertions [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:

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

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 (out *net* N) sends a message N out on channel *net*. A statement of the form (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:

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

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 $\mathsf{Enc}(M)$ represents its encryption and $\mathsf{Dec}(M)$ its decryption part.

In the 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
$\mathsf{Enc}(M)$	encryption part of key pair M

$\mathsf{Dec}(M)$	decryption part of key pair M

Syntactic restriction: No subterms of the form $\{\!|M|\!\}_{(\mathsf{Dec}(N))^{-1}}$.
Define: A message M is *implementable* if it contains no subterms $\{\!|M|\!\}_{N^{-1}}$.

Because of the restriction that we never build messages $\{\!|M|\!\}_{(\mathsf{Dec}(N))^{-1}}$, 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:

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

We will write the list $\langle M_1, \ldots, M_n \rangle$ as shorthand for $(M_1, (\ldots, (M_n, ())\ldots))$

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 $\{x.\, (x, \{\!|x|\!\}_y) \mid \bar{A}\}$ is matched by messages of the form $(M, \{\!|M|\!\}_y)$, but not by messages $(M, \{\!|M|\!\}_z)$ or $(M, \{\!|N|\!\}_y)$.

Patterns:

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

Syntactic restrictions: $\vec{x} \subseteq \mathsf{fv}(M)$ and $\vec{x}$ distinct.
Define: A pattern $\{\vec{x}.\, M \mid \bar{A}\}$ is *implementable* if $(\mathsf{fn}(M), \mathsf{fv}(M) - \vec{x}, \bar{M} \Vdash \vec{x})$.

Importantly, not all patterns are implementable. For instance, the patterns $\{x, dk.\, \{\!|x|\!\}_{dk^{-1}} \mid \bar{A}\}$ and $\{x.\, \{\!|x|\!\}_{ek} \mid \bar{A}\}$ are not implementable, because they would allow access to the plaintext without knowing the decryption key. On the other hand, $\{x.\, \{\!|x|\!\}_{dk^{-1}} \mid \bar{A}\}$ and $\{x.\, \{\!|x|\!\}_{\mathsf{Enc}(k)} \mid \bar{A}\}$ 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 $\bar{M} \Vdash N$ meaning 'An agent which knows messages $\bar{M}$ can construct messages N.'

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

$$\text{(DY Id)}\ \frac{}{\bar{M}, N \Vdash N} \qquad \text{(DY And)}\ \frac{\bar{M} \Vdash N_1 \ \ldots\ \bar{M} \Vdash N_k}{\bar{M} \Vdash N_1, \ldots, N_k} \qquad \text{(DY Nil)}\ \frac{}{\bar{M} \Vdash ()} \qquad \text{(DY Pair)}\ \frac{\bar{M} \Vdash N, N'}{\bar{M} \Vdash (N, N')}$$

$$\text{(DY Split)}\ \frac{\bar{M}, N, N' \Vdash L}{\bar{M}, (N, N') \Vdash L} \qquad \text{(DY Key)}\ \frac{\bar{M} \Vdash N \quad k \in \{\mathsf{Enc}, \mathsf{Dec}\}}{\bar{M} \Vdash k(N)}$$

$$\text{(DY Encrypt)}\ \frac{\bar{M} \Vdash N, N'}{\bar{M} \Vdash \{\!|N'|\!\}_N} \qquad \text{(DY Decrypt)}\ \frac{\bar{M} \Vdash N \quad \bar{M}, N' \Vdash L}{\bar{M}, \{\!|N'|\!\}_{N^{-1}} \Vdash L} \qquad \text{(DY Unencrypt)}\ \frac{\bar{M} \Vdash N \quad \bar{M}, N' \Vdash L}{\bar{M}, \{\!|N'|\!\}_{\mathsf{Enc}(N)} \Vdash L}$$

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:

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

Derived Forms for Patterns:

$M \triangleq \{M \mid \}$; $\quad T \triangleq \{x . x \mid x : T\}$ for fresh x;
$\exists x \triangleq \{x . x \mid \}$; $\quad \ldots \triangleq (\exists x)$ for fresh x;
$\{\!|X|\!\}_N \triangleq \{\vec{x} . \{\!|M|\!\}_N \mid \bar{A}\}$, if $X = \{\vec{x} . M \mid \bar{A}\}$;
$\{\!|X|\!\}_{N^{-1}} \triangleq \{\vec{x} . \{\!|M|\!\}_{N^{-1}} \mid \bar{A}\}$, if $X = \{\vec{x} . M \mid \bar{A}\}$;
$X[\bar{B}] \triangleq \{\vec{x} . M \mid \bar{A}, \bar{B}\}$, if $X = \{\vec{x} . M \mid \bar{A}\}$;
$\langle X_1, \ldots, X_n \rangle \triangleq \{\vec{x}_1, \ldots, \vec{x}_n . \langle M_1, \ldots, M_n \rangle \mid \bar{A}_1, \ldots, \bar{A}_n\}$, if $X_i = \{\vec{x}_i . M_i \mid \bar{A}_i\}$

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

Syntactic restrictions:

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

Scope:

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

3.4 Specifying Authenticity by Correspondence Assertions

Following [12, 13, 11], we specify authenticity properties by inserting correspondence assertions into protocol specifications.

Correspondence Assertions:

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

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

$$P \triangleq P_A \mid P_B, \text{ where } P_A \triangleq (\mathsf{begin!}(M,A,B);\ \mathsf{out}\ \mathit{net}\ (M,B))$$
$$P_B \triangleq (\mathsf{inp}\ \mathit{net}\ (\exists x,B)[!\mathsf{begun}(x,A,B)];\ \mathsf{end}(x,A,B))$$

Process P is safe in isolation, but we are really interested in safety in the presence of an opponent. 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 (out *net* $N \mid P$) is not safe, and we ensure robust safety by adding encryption:

$$P \triangleq \mathsf{new}\ k : \mathsf{SigningKP}(A);\ (\mathsf{out}\ \mathit{net}\ (\mathsf{Dec}(k)) \mid P_A(\mathsf{Enc}(k)) \mid P_B(\mathsf{Dec}(k)))$$
$$P_A(ek) \triangleq \mathsf{begin!}(M,A,B);\ \mathsf{out}\ \mathit{net}\ \{\!|M,B|\!\}_{ek}$$
$$P_B(dk) \triangleq \mathsf{inp}\ \mathit{net}\ \{\!|\exists x : \mathsf{Public}, B|\!\}_{dk^{-1}}[!\mathsf{begun}(x,A,B)];\ \mathsf{end}(x,A,B)$$

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 message types, for example $dk : \mathsf{SigningDK}(A)$ and $\{\!|x,B|\!\}_{dk^{-1}} : \mathsf{Un}$.
- List correspondences that have begun, for example $!\mathsf{begun}(x,A,B)$.

The environment containing these assertions would be:

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

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.

Environments:

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

4.2 Typed Pattern Matching

We can now explain the assertion component of a pattern $\{\vec{x}.M \mid \overline{A}\}$: it gives the precondition $\overline{A}$ which must be satisfied by any process that constructs a term matching the pattern. For example, the pattern $(\exists x : \mathsf{Public}, B)[!\mathsf{begun}(x,A,B)]$ is a derived form for $\{x.(x,B) \mid x : \mathsf{Public}, !\mathsf{begun}(x,A,B)\}$.

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

$E \vdash M \in X \triangleq E \vdash M : \mathsf{Top}, \overline{A}\{\vec{x} \leftarrow \vec{N}\}$, where $M = N\{\vec{x} \leftarrow \vec{N}\}$	Match
$E, M \in X \vdash \mathcal{J} \triangleq E, M : \mathsf{Top}, \overline{A}\{\vec{x} \leftarrow \vec{N}\} \vdash \mathcal{J}$, 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. A message is *untainted* if it has been received from a trusted source. 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 $\mathsf{Public} \in K$, then members of type T are publishable.
- If $(T :: K)$ and $\mathsf{Tainted} \notin K$, then members of type T are untainted.

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

Kinds and Subkinding:

$$K,H,J \subseteq \{\mathsf{Public},\mathsf{Tainted}\}$$

$$\frac{(\mathsf{Public} \in H) \Rightarrow (\mathsf{Public} \in K) \quad (\mathsf{Tainted} \in K) \Rightarrow (\mathsf{Tainted} \in H)}{K \leq H}$$

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

Kinding $T :: K$:

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

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)
$$\frac{}{T \leq T}$$

(Subty Top)
$$\frac{T :: K \quad K \leq H}{T \leq H\,\mathsf{Top}}$$

(Subty Auth)
$$\frac{K \leq H}{K\,\mathsf{Auth}(L) \leq H\,\mathsf{Auth}(L)}$$

(Subty Public Tainted)
$$\frac{T :: K \cup \{\mathsf{Public}\} \quad U :: H \cup \{\mathsf{Tainted}\}}{T \leq U}$$

4.5 Top Types

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

Derived Forms for Top Types:

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

4.6 Authorization Types

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

Derived Forms for Authorization Types:

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

In meaningful authorization types, parameter L is usually a list of principal names. For example, $\mathsf{Public}\langle A, B, C\rangle$ 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.

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 $J\,\mathsf{Auth}(L)$. 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:

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

A key esA of type $\mathsf{SigningEK}(A)$ 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 $\mathsf{Public}(A,\vec{B})$ to produce ciphertexts $\{\!|M|\!\}_{esA}$ of type $\mathsf{Public}(\vec{B})$: 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.

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

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:

$$\frac{\mathsf{Tainted} \notin K \cup H^{-1} \qquad E \vdash N : (K,H)\,\mathsf{EK}(X),\ (M,L) \in X}{E \vdash \{\!|M|\!\}_N : \mathsf{Public}(L)}$$

$$\mathsf{Public}^{-1} \triangleq \mathsf{Tainted} \qquad \mathsf{Tainted}^{-1} \triangleq \mathsf{Public}$$

The condition $\mathsf{Tainted} \notin K \cup H^{-1}$ 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:

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

Note that here the ciphertext type $J\,\mathsf{Auth}(L)$ is only public if the plaintext type $J'\,\mathsf{Top}$ 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:

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

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

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

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 $(x,E,M : J\,\mathsf{Top}, x : \mathsf{Top})$; as a result, untrusted keys can only be used when the pattern X is quite 'weak'.

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 $\ell : X \rightarrow \mathsf{Auth}(Y)$, which can be used to tag messages M of kind $(J \cup \mathsf{Tainted})$ to get tagged messages $\ell(M) : J\mathsf{Auth}(L)$. For example, our previous protocol becomes:

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

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.

$$
\begin{array}{c}
\ell(M) \triangleq \{\!|M|\!\}_{\mathsf{Enc}(\ell)}; \qquad \ell(X) \triangleq \{\!|X|\!\}_{\mathsf{Enc}(\ell)}; \\
(X \rightarrow \mathsf{Auth}(Y)) \triangleq (\{\mathsf{Public}\}, \{\mathsf{Public}\})\,\mathsf{KP}(X,Y)
\end{array}
$$

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 $esA : \mathsf{SigningEK}(A)$) and B (using his signing key $esB : \mathsf{SigningEK}(B)$). This message can be given type $\{\!|\{\!|M|\!\}_{esA}|\!\}_{esB} : \mathsf{Secret}$ as long as $M : \mathsf{Secret}\langle A,B,y\rangle$ for some y, and type $\{\!|\{\!|M|\!\}_{esA}|\!\}_{esB} : \mathsf{Public}$ as long as $M : \mathsf{Public}\langle A,B,y\rangle$ for some y. This form of nested signing was not supported by [12, 13, 11].

$$
\mathsf{Signing}KT(L) \triangleq (\emptyset, \{\mathsf{Public}\})\,KT(\exists x : \mathsf{Secret}(L,y), \exists y)
$$

A long version of this paper [14] contains a proof that the protocol in Section 5.1 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]).

$$\mathsf{PublicCrypto}KT(L) \triangleq (\{\mathsf{Public}\},\emptyset)\,KT(\exists x : \mathsf{Secret}(L),\ldots)$$

5.4 Symmetric Keys

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

$$\{M\}_N \triangleq \{\!|M|\!\}_{\mathsf{Enc}(N)};\ \ \{X\}_N \triangleq \{\!|X|\!\}_{\mathsf{Enc}(N)};\ \ \mathsf{SymK}(X) \triangleq (\emptyset,\emptyset)\,\mathsf{KP}(X,\ldots)$$

5.5 Hashing

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

$$\begin{array}{ll} \#(M) \triangleq hash(\{\!|M|\!\}_{ekH}) & \text{where } ekH : (\{\mathsf{Public}\},\emptyset)\,\mathsf{EK}(\ldots) \\ & \text{and } hash : \{\!|\exists x : \mathsf{Secret}(y)|\!\}_{ekH} \rightarrow \mathsf{Auth}(\exists y) \end{array}$$

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:

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

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

Appendix

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)
$$\frac{P \equiv P' \quad (\bar{A} ::: P') \rightarrow (\bar{B} ::: Q') \quad Q' \equiv Q}{(\bar{A} ::: P) \rightarrow (\bar{B} ::: Q)}$$

$n \notin \mathsf{fn}(\bar{A}, Q) \Rightarrow (\bar{A} ::: (\mathsf{new}\ n{:}T; P) \mid Q) \rightarrow (\bar{A}, n{:}T ::: P \mid Q)$ (Redn New)
$(\bar{A} ::: (\mathsf{begin!}(M); P) \mid Q) \rightarrow (\bar{A}, !\mathsf{begun}(M) ::: P \mid Q)$ (Redn Begin)
$(\bar{A}, !\mathsf{begun}(M) ::: (\mathsf{end}(M); P) \mid Q) \rightarrow (\bar{A}, !\mathsf{begun}(M) ::: P \mid Q)$ (Redn End)
$(\bar{A} ::: (\mathsf{out}\ L\ M\{\vec{x} \leftarrow \vec{N}\} \mid \mathsf{inp}\ L\ \{\vec{x}.\ M \mid \bar{A}\}; P) \mid Q) \rightarrow (\bar{A} ::: P\{\vec{x} \leftarrow \vec{N}\} \mid Q)$ (Redn IO)

Good Environment, $E \vdash \diamond$**:**

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

Right Rules, $E \vdash \bar{A}$**:**

(Id)
$$\frac{E, A \vdash \diamond}{E, A \vdash A}$$

(And)
$$\frac{E \vdash A_1 \quad \cdots \quad E \vdash A_n \quad n \geq 0}{E \vdash A_1, \ldots, A_n}$$

(Empty)
$$\frac{E \vdash \diamond}{E \vdash () : \mathsf{Public}}$$

(Sub)
$$\frac{E \vdash M : T \quad T \leq U \quad \mathsf{fv}(U) \subseteq \mathsf{dom}(E)}{E \vdash M : U}$$

(Pair)
$$\frac{E \vdash M : K\,\mathsf{Top},\ N : K\,\mathsf{Top}}{E \vdash (M,N) : K\,\mathsf{Top}}$$

(Enc Part)
$$\frac{E \vdash M : (K,H)\mathsf{KP}(X)}{E \vdash \mathsf{Enc}(M) : (K,H)\mathsf{EK}(X)}$$

(Dec Part)
$$\frac{E \vdash M : (K,H)\mathsf{KP}(X)}{E \vdash \mathsf{Dec}(M) : (K,H)\mathsf{DK}(X)}$$

(Encrypt Trusted)
$$\frac{\mathsf{Tainted} \notin K \cup H^{-1} \quad E \vdash N : (K,H)\mathsf{EK}(X),\ (M,L) \in X}{E \vdash \{\!|M|\!\}_N : \mathsf{Public}(L)}$$

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

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

(Unsub)
$$\frac{\mathsf{fv}(T) \subseteq \mathsf{dom}(E) \quad E,\ M : U \vdash A \quad T \leq U}{E,\ M : T \vdash A}$$

(Split)
$$\frac{E,\ M : K\,\mathsf{Top},\ N : K\,\mathsf{Top} \vdash A}{E,\ (M,N) : K\,\mathsf{Top} \vdash A}$$

(Decrypt Trusted)

$$\frac{\mathsf{Tainted} \notin H \cup J \quad E \vdash N : (K,H)\,\mathsf{DK}(X) \quad E,(M,L) \in X \vdash B}{E,\, \mathit{decrypt}(M,N) : J\,\mathsf{Auth}(L) \vdash B}$$

(Decrypt Untrusted)

$$\frac{\mathsf{Tainted} \in J \quad E \vdash N : (K,H)\,\mathsf{DK}(X) \quad x \notin \mathsf{dom}(E) \quad x,E,(M,x) \in X \vdash B \quad (\mathsf{Tainted} \in H \cup K^{-1}) \;\Rightarrow\; (x,E,M : J\,\mathsf{Top}, x : \mathsf{Top} \vdash (M,x) \in X)}{E,\, \mathit{decrypt}(M,N) : J\,\mathsf{Top} \vdash B}$$

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

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

(Proc Out)

$$\frac{E \vdash N : \mathsf{Un},\ M : \mathsf{Un}}{E \vdash \mathsf{out}\, N\, M}$$

(Proc In)

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

(Proc New)

$$\frac{E,n : T \vdash P \quad n \notin \mathsf{fn}(E)}{E \vdash \mathsf{new}\, n{:}T;\ P}$$

(Proc Par)

$$\frac{E \vdash P \quad E \vdash Q}{E \vdash P \mid Q}$$

(Proc Repl)

$$\frac{E \vdash P}{E \vdash\, !P}$$

(Proc Stop)

$$\frac{E \vdash \diamond}{E \vdash \mathbf{0}}$$

(Proc Begin Many)

$$\frac{E,\ !\mathsf{begun}(M) \vdash P}{E \vdash \mathsf{begin!}(M);\ P}$$

(Proc End)

$$\frac{E \vdash\, !\mathsf{begun}(M) \quad E \vdash P}{E \vdash \mathsf{end}(M);\ P}$$

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

(State)

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

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.