Types and Effects for Asymmetric Cryptographic Protocols

Submitted to 2002 IEEE Computer Security Foundations Workshop

DRAFT of Jan 28 2002

Abstract

We present the first type and effect system for proving authenticity properties of security protocols based on asymmetric cryptography. The most significant new features of our type system are: (1) a separation of *public types* (for data possibly sent to the opponent) from *tainted types* (for data possibly received from the opponent) via a subtype relation; (2) *trust effects*, to guarantee that tainted data does not, in fact, originate from the opponent; and (3) *challenge/response types* to support a variety of idioms used to guarantee message freshness. We illustrate the applicability of our system via protocol examples.

1 Motivation

Gordon and Jeffrey recently proposed a type-based methodology for checking authenticity properties of security protocols [GJ01c, GJ01a]. First, specify properties by annotating an executable description of a protocol with correspondence assertions [WL93]. Second, annotate the protocol with suitable types. Third, verify the assertions by running a type-checker. A type-correct protocol is secure against a malicious opponent conforming to the Dolev and Yao assumptions [DY83]; the opponent may eavesdrop, generate, and replay messages, but can only encrypt or decrypt messages if it knows the appropriate key. This methodology is promising because it requires no state-space exploration, requires little interactive effort per protocol, and reduces the verification problem to the familiar edit/type-check/debug cycle.

Still, their system applies only to symmetric-key cryptography and only to one style of nonce handshake, a significant limitation. The goal of this paper is to enrich their type and effect system so as to apply the methodology to a wider class of protocols based on both symmetric and asymmetric cryptography. To do so, we need to solve the following three problems.

(1) Let us say data is *tainted* if it may have been generated by the opponent, otherwise *untainted*, and *public* if it may be revealed to the opponent, otherwise *secret*. Now, in symmetric protocols, data is either secret and untainted (because it is sent encrypted, and the opponent is ignorant of the key) or it is both public and tainted (because it is sent in the clear). In asymmetric protocols, the situation is subtler because of public keys: data may be both secret and tainted (if sent encrypted with an honest agent's public key) or public and untainted (if sent encrypted with an honest agent's private key). Gordon and Jeffrey's system [GJ01a] has one type, Un, for public, tainted data, and every other type is both secret and untainted. Here, we need to be more flexible; we use a subtype relation to represent whether a type is tainted and whether it is public.

(2) Types can represent the degree of trust we place in data. In symmetric protocols, the degree of trust, and hence the types of data, is fixed. On the other hand, in asymmetric protocols, the degree of trust may increase over time as new information arises, for example, from nonce challenges. We introduce *trust effects* to model how new information may change the type of existing data.

(3) Gordon and Jeffrey's system [GJ01a] supports one format for proving freshness via nonce handshakes: the challenge in the clear, the response encrypted. Asymmetric protocols may use other styles: both challenge and response encrypted; or the challenge encrypted, the response in the clear. To accommodate these other styles, we introduce new *challenge/response* types.

1.1 Background

Many methodologies exist for verifying authenticity properties against the opponent model of Dolev and Yao [DY83]. Verification via type-checking is one of only a few, recent techniques that requires little interactive effort per protocol, while not bounding protocol or opponent size. Other such techniques include automatic tools for strand spaces [Son99, THG98] and rank functions [HS00, Sch98]. Other effective approaches include model-checking [Low96, MCJ97], which typically puts bounds on the protocol and opponent, and techniques relying on theorem-proving [Bol96, Pau98] or epistemic logics [BAN89, DMP01], which typically require lengthy expert interaction.

Woo and Lam's correspondence assertions [WL93] are safety properties, specifying what is known as injective agreement [Low95]. Given a description of the sequence of messages exchanged by principals in a protocol, we annotate it with labelled events marking the progress of each principal through the protocol. We divide these events into two kinds, begin-events and end-events. Event labels typically indicate the names of the principals involved and their roles in the protocol. For example, to specify an authenticity property of a simple nonce handshake we decorate it with begin-events and end-events as follows.

Message 1	$A \rightarrow B:$	N
Event 1	B begins	"B sends A message M"
Message 2	$B \rightarrow A:$	$\{M,N\}_K$
Event 2	A ends	"B sends A message M"

A protocol is *safe* if in all protocol runs, every assertion of an end-event corresponds to a distinct, earlier assertion of a begin-event with the same label. A protocol is *robustly safe* if it is safe in the presence of any hostile opponent who can capture, modify, and replay messages, but cannot forge assertions.

Previous work can type-check the robust safety of protocols based on secure channels [GJ01c], and on insecure channels protected by symmetric cryptography [GJ01a]. These two papers are the only prior work on authenticity by typing. They build on Abadi's pioneering work [Aba99] on secrecy by typing for symmetric-key cryptographic protocols. Abadi and Blanchet [AB01, AB02] extend Abadi's original system to establish secrecy properties for asymmetric protocols. The present paper is a parallel development for authenticity properties. Technically, it is not simply a routine combination of previous papers [GJ01a, AB01]. For example, to facilitate type-checking our formalism, each bound variable is annotated with a single type. A feature of Abadi and Blanchet's treatment of tainted data is that a bound variable may assume an arbitrary number of types, depending on its context, and therefore they suppress type annotations.

Like earlier work on types for cryptographic protocols, we take a binary view of the world as consisting of a system of honest protocol participants plus a dishonest opponent. We leave a finer-grained analysis as future work.

1.2 Our Three Main Contributions

Separation of trust and secrecy. In a cryptographic protocol based on symmetric cryptography, data is typically either both secret and untainted or both public and tainted. For example, consider the message:

$$A \rightarrow B: \quad A, \{M\}_{K_{AB}}$$

(We write $\{M\}_{K_{AB}}$ for the outcome of encrypting M using a symmetric algorithm with key K_{AB}.) The principal name A is public and tainted (since it is sent in plaintext) but the payload M and the shared key K_{AB} are secret and untainted (since they are never sent in plaintext, and are known only to honest principals).

On the other hand, in a cryptographic protocol based on asymmetric cryptography, secrecy and taintedness are independent. Data may be secret and tainted, or public and untainted. For example, if K_B is B's public key and K_A^{-1} is A's private key, consider the message:

$$A \rightarrow B: \quad \{\!|M|\!\}_{K_A^{-1}}, \{\!|N|\!\}_{K_B}$$

(We write $\{\!|M|\!\}_{K_A^{-1}}$ for the outcome of encrypting M using an asymmetric algorithm with private key K_A^{-1}, and $\{\!|N|\!\}_{K_B}$ for the outcome of encrypting N with public key K_B.) Now, B considers:

- M is public (since the opponent knows K_A and so can decrypt the ciphertext $\{\!|M|\!\}_{K_A^{-1}}$) but untainted (since it is encrypted with A's private key, and so must have originated from the honest agent A).
- N is secret (since the opponent does not know K_B^{-1} so cannot decrypt the ciphertext $\{\!|B|\!\}_{K_B}$) but tainted (since it is encrypted with B's public key, and so could have originated from a dishonest intruder).

Previous type systems [Aba99, GJ01a] feature a type, here called Un, for all messages known to the opponent. Here, to support asymmetric cryptography, we admit some types that are public without being tainted, and others that are tainted without being public. We relate these types to Un via a subtype relation. As usual, we say T is a subtype of U, written $T <: U$, to mean that data of type T may be used in situations expecting data of type U. A type T is *public* if $T <: \mathsf{Un}$, that is, it may be sent to the opponent. A type T is *tainted* if $\mathsf{Un} <: T$, that is, it may be sent from the opponent.

Our recognition of tainted types—as distinct from public types—has many parallels in analyses of non-cryptographic aspects of security. The Perl programming language [WCS96] can track at runtime whether or not scalar data is tainted, to catch bugs in code dealing with untrusted inputs. An extension of the simply-typed λ-calculus [ØP97] uses annotations on each type constructor to track whether or not data can be trusted, either because it originates from or has been endorsed by an honest participant. Similarly, an experimental extension [STFW01] of C qualifies types as tainted or untainted to allow the static detection of issues with format strings. The Secure Lambda Calculus [HR98] uses subtyping to track security levels. To the best of our knowledge, this paper is the first to use types to track both public and tainted data in the presence of cryptography.

Dynamic trust. In asymmetric protocols, the degree of trust we place in tainted data may increase as we receive new information. For example, consider the following variant of the Needham–Schroeder–Lowe [NS78, Low96] public-key protocol, extended to include a key exchange initiated by A:

$$\begin{array}{lll} \text{Message 1} & A \to B: & \{\!|A, K_{AB}, N_A|\!\}_{K_B} \\ \text{Message 2} & B \to A: & \{\!|B, K_{AB}, N_A, N_B|\!\}_{K_A} \\ \text{Message 3} & A \to B: & \{\!|N_B|\!\}_{K_B} \end{array}$$

After receiving Message 1, B regards the session key K_{AB} as tainted; it may come from A, but it may also come from the opponent, since the key K_B is public. In Message 2, B sends A a nonce N_B, encrypted together with the tainted key K_{AB} under K_A, and hence hidden from the opponent. Now, A only replies with Message 3 if the session key it receives in Message 2 matches the key it issued in Message 1. Therefore, on successful receipt of the secret N_B in Message 3, B trusts that K_{AB} did not in fact come from the opponent. So it is safe for B to send a secret message to A encrypted with the key K_{AB}:

$$\text{Message 4} \quad B \to A: \quad \{M\}_{K_{AB}}$$

In this protocol, B's trust in the session key K_{AB} is *dynamic* in that it changes over time: initially K_{AB} is tainted, but after Message 3 it is known to be untainted.

We model dynamic trust by introducing *trust effects*, that allow the type of a nonce to make assertions about the type of other data. In the typed form of our example, the type of N_B asserts that K_{AB} has the type of keys known only to honest participants.

Symmetric key cryptographic protocols typically do not require dynamic trust: data is either trusted or untrusted for the whole run of the protocol, and its trust status does not change during a particular run. Over time, symmetric key cryptographic protocols may downgrade their trust in data due to key-compromise or other long-term attacks on the cryptosystem. Still, such attacks are outside our model, and are left for future work.

Nonce handshake styles. Protocols use nonce handshakes to establish message freshness, and hence to thwart replay attacks. The type and effect system of this paper supports three handshake idioms:

- *Public Out Secret Home (POSH)*: the nonce goes out in the clear and returns encrypted.
- *Secret Out Public Home (SOPH)*: the nonce goes out encrypted and returns in the clear.
- *Secret Out Secret Home (SOSH)*: the nonce goes out encrypted and returns encrypted.

SOSH nonces are useful in asymmetric protocols, such as the protocol described above, where if either N_A or N_B is learned by the opponent, the protocol can be compromised. The novel feature of SOSH nonces in our type system is that they can be relied upon for authenticity even when they are tainted (for example, when they are encrypted with a public key) because we have two cases:

- If the nonce was generated by the opponent, then only the opponent can perform the equality check at the end of the nonce handshake, so no honest agent ever relies on the authenticity information carried by the nonce.
- If the nonce was generated by an honest agent, then the opponent never learns of it (since the nonce is secret) and so it is safe for honest agents to rely on the authenticity information carried by it.

In contrast, POSH and SOPH nonces cannot be relied upon when tainted. The Needham–Schroeder–Lowe protocol relies on N_A and N_B being SOSH nonces, since they are encrypted with public keys and hence tainted.

Guttman and Fábrega [GF00] call POSH and SOPH nonces incoming and outgoing tests, respectively; they do not discuss SOSH nonces. Gordon and Jeffrey [GJ01a] deal only with POSH nonces.

1.3 Remainder of this Paper

Section 2 reviews Gordon and Jeffrey's methodology for specifying authenticity properties of protocols. Section 3 describes our new type and effect system, and describes its application to some examples. Section 4 concludes.

2 Specifying Authenticity Properties in the Spi-Calculus (Review)

We formalise our type and effect system in a version of the spi-calculus [AG99], a concurrent language based on the π-calculus [Mil99] augmented with the Dolev–Yao model of cryptography. Section 2.1 reviews the syntax and informal semantics of a spi-calculus extended with correspondence assertions [WL93]. Section 2.2 shows how to specify an example protocol. Later, we show it is robustly safe by typing.

2.1 A Spi-Calculus with Correspondence Assertions

First, here is the syntax of messages.

Names, Messages

m, n, x, y, z	name: variable, channel, nonce, key, key-pair
$L, M, N ::=$	message
x	name
(M, N)	pair formation
$\mathsf{inl}\ (M)$	left injection
$\mathsf{inr}\ (M)$	right injection
$\{M\}_N$	symmetric encryption

$k\ (M)$	(where k either Encrypt or Decrypt)	key-pair component
$\{\!\|M\|\!\}_N$		asymmetric encryption

These messages are:

- A message x is a name, representing a channel, nonce, symmetric key, or asymmetric key-pair.
- A message (M,N) is a pair. From this primitive we can describe any finite record.
- Messages $\mathsf{inl}\ (M)$ and $\mathsf{inr}\ (M)$ are tagged unions, differentiated by the distinct tags inl and inr. With these primitives we can encode any finite tagged union.
- A message $\{M\}_N$ is the ciphertext obtained by encrypting the plaintext M with the symmetric key N.
- A message $\{\!|M|\!\}_N$ is the ciphertext obtained by encrypting the plaintext M with the asymmetric encryption key N.
- A message $\mathsf{Decrypt}\ (M)$ extracts the decryption key component from the key pair M, and $\mathsf{Encrypt}\ (M)$ extracts the encryption key component from the key pair M.

An asymmetric key-pair p has two dual applications: public-key encryption and digital signature. In the first, $\mathsf{Encrypt}\ (p)$ is public and $\mathsf{Decrypt}\ (p)$ is secret. In the second, $\mathsf{Encrypt}\ (p)$ is secret and $\mathsf{Decrypt}\ (p)$ is public. For each key-pair, our type system tracks whether the encryption or decryption key is public, but it makes no difference to our syntax or operational semantics. (Hence, a single key-pair cannot be used both for public-key encryption and digital signature; this is often regarded as an imprudent practice, but nonetheless is beyond our formalism.)

Next, we give the syntax of processes. Each bound name has a type annotation, written T or U. We postpone describing the syntax of types to the next section.

Processes:

$O,P,Q,R ::=$	process
$\mathsf{out}\ M\ N$	output
$\mathsf{inp}\ M\ (x{:}T);P$	input (x bound in P)
$\mathsf{repeat\ inp}\ M\ (x{:}T);P$	replicated input (x bound in P)
$\mathsf{split}\ M\ \mathsf{is}\ (x{:}T,y{:}U);P$	pair splitting (x bound in U and P; y bound in P)
$\mathsf{match}\ M\ \mathsf{is}\ (N,y{:}T);P$	pair matching (y bound in P)
$\mathsf{case}\ M\ \mathsf{is\ inl}\ (x{:}T)\ P\ \mathsf{is\ inr}\ (y{:}U)\ Q$	union case (x bound in P; y bound in Q)
$\mathsf{decrypt}\ M\ \mathsf{is}\ \{x{:}T\}_N;P$	symmetric-key decryption (x bound in P)
$\mathsf{decrypt}\ M\ \mathsf{is}\ \{\!\|x{:}T\|\!\}_{N^{-1}};P$	asymmetric-key decryption (x bound in P)
$\mathsf{check}\ M\ \mathsf{is}\ N;P$	nonce-checking
$\mathsf{begin}\ L;P$	begin-assertion
$\mathsf{end}\ L;P$	end-assertion
$\mathsf{new}\ (x{:}T);P$	name generation (x bound in P)
$P \mid Q$	composition
stop	inactivity

The type annotations on bound names are used for type-checking but play no role at runtime; they do not affect the operational behaviour of processes. In examples, for the sake of brevity, we sometimes omit type annotations.

The free and bound names of a process are defined as usual. We write $P\{x{\leftarrow}N\}$ for the outcome of a capture-avoiding substitution of the message N for each free occurrence of the name x in the process P. We identify processes up to the consistent renaming of bound names, for example when $y \notin fn(P)$, we equate $\mathsf{new}\ (x{:}T);P$ with $\mathsf{new}\ (y{:}T);(P\{x{\leftarrow}y\})$.

Next, we give informal semantics for process behaviour and process safety; formal definitions appear in Appendix B. These processes are:

- Processes out M N and inp M $(x{:}T);P$ are output and input, respectively, along an asynchronous, unordered channel M. If an output out x N runs in parallel with an input inp x $(y);P$, the two can interact to leave the residual process $P\{y{\leftarrow}N\}$.
- Process repeat inp M $(x{:}T);P$ is replicated input, which behaves like input, except that each time an input of N is performed, the residual process $P\{y{\leftarrow}N\}$ is spawned off to run concurrently with the original process repeat inp M $(x{:}T);P$.
- A process split M is $(x{:}T,y{:}U);P$ splits the pair M into its two components. If M is (N,L), the process behaves as $P\{x{\leftarrow}N\}\{y{\leftarrow}L\}$. Otherwise, it deadlocks, that is, does nothing.
- A process match M is $(N,y{:}U);P$ splits the pair M into its two components, and checks that the first one is N. If M is (N,L), the process behaves as $P\{y{\leftarrow}L\}$. Otherwise, it deadlocks.
- A process case M is inl $(x{:}T)$ P is inr $(y{:}U)$ Q checks the tagged union M. If M is inl (L), the process behaves as $P\{x{\leftarrow}L\}$. If M is inr (N) it behaves as $Q\{y{\leftarrow}N\}$. Otherwise, it deadlocks.
- A process decrypt M is $\{x{:}T\}_N;P$ decrypts M using symmetric key N. If M is $\{L\}_N$, the process behaves as $P\{x{\leftarrow}L\}$. Otherwise, it deadlocks. We assume there is enough redundancy in the representation of ciphertexts to detect decryption failures.
- A process decrypt M is $\{\!|x{:}T|\!\}_{N^{-1}};P$ decrypts M using asymmetric key N. If M is $\{\!|L|\!\}_{\mathsf{Encrypt}\ (K)}$ and N is Decrypt (K), then the process behaves as $P\{x{\leftarrow}L\}$. Otherwise, it deadlocks.
- A process check M is $N;P$ checks the messages M and N are the same name before executing P. If the equality test fails, the process deadlocks.
- A process begin $L;P$ autonomously asserts an begin-event labelled L, and then behaves as P.
- An process end $L;P$ autonomously asserts an end-event labelled L, and then behaves as P.
- A process new $(x{:}T);P$ generates a new name x, whose scope is P, and then runs P. (This abstractly represents nonce or key generation.)
- A process $P \mid Q$ runs processes P and Q in parallel.
- The process stop is deadlocked.

Safety:

A process P is *safe* if and only if for every run of the process and for every L,
there is a distinct begin-event labelled L preceding every end-event labelled L.

We are mainly concerned not just with safety, but with robust safety, that is, safety in the presence of an arbitrary hostile opponent. In the untyped spi-calculus [AG99], the opponent is modelled by an arbitrary process. In our typed spi-calculus, we do not consider completely arbitrary attacker processes, but restrict ourselves to *opponent* processes that satisfy two mild conditions:

- Opponents cannot assert events: otherwise, no process would be robustly safe, because of the opponent end x;.
- Opponents do not have access to trusted data, so any type occurring in the process must be Un.

Opponents and Robust Safety:

A process P is *assertion-free* if and only if it contains no begin- or end-assertions.
A process P is *untyped* if and only if the only type occurring in P is Un.
An *opponent* O is an assertion-free untyped process O.
A process P is *robustly safe* if and only if $P \mid O$ is safe for every opponent O.

$Sender(net, private_A, public_B) \triangleq$
  new (key_{AB});
  new $(challenge_A)$;
  begin "A generates key_{AB} for B";
  out net $\{\!|A, key_{AB}, challenge_A|\!\}_{public_B}$;
  inp net $(ctext_2, challenge_{B2})$;
  decrypt $ctext_2$
    is $\{\!|B, key_{AB}, response_A, challenge_{B1}|\!\}_{private_A^{-1}}$;
  check $challenge_A$ is $response_A$;
  end "B received key_{AB} from A";
  new (msg);
  begin "A sends msg to B";
  out net $(challenge_{B1}, \{msg, challenge_{B2}\}_{key_{AB}})$;

$System(net) \triangleq$
  new $(pair_A)$; new $(pair_B)$; (
    $Sender(net,$ Decrypt $(pair_A),$ Encrypt $(pair_B))$ |
    $Receiver(net,$ Encrypt $(pair_A),$ Decrypt $(pair_B))$ |
    out net (Encrypt $(pair_A)$, Encrypt $(pair_B)$)
)

$Receiver(net, public_A, private_B) \triangleq$
  repeat
    inp net $(ctext_1)$;
    decrypt $ctext_1$
     is $\{\!|A, key_{AB}, challenge_A|\!\}_{private_B^{-1}}$;
    new $(challenge_{B1})$;
    new $(challenge_{B2})$;
    begin "B received key_{AB} from A";
    out net
      $(\{\!|B, key_{AB}, challenge_A, challenge_{B1}|\!\}_{public_A},$
       $challenge_{B2})$;
    inp net $(response_{B1}, ctext_3)$;
    check $challenge_{B1}$ is $response_{B1}$;
    end "A generates key_{AB} for B";
    decrypt $ctext_3$ is $\{msg, response_{B2}\}_{key_{AB}}$;
    check $challenge_{B2}$ is $response_{B2}$;
    end "A sends msg to B";

Figure 1: An example protocol with correspondence assertions

2.2 Specifying an Example

We show how to program a simple cryptographic protocol in our formalism. This protocol is a version of Needham-Schroeder-Lowe [NS78, Low96] modified to illustrate the various features of our type system. (The protocol is different from the version discussed in Section 1.) The protocol shares a session key K_{AB} between participants A and B, and uses this key to send a message M from A to B. The protocol should guarantee the authenticity properties:

(1) A believes she shares the key K_{AB} with B.
(2) B believes he shares the key K_{AB} with A.
(3) B believes message M was sent by A.

We specify the protocol informally as follows:

Event 1	A begins	"A generates K_{AB} for B"
Message 1	$A \rightarrow B$:	$\{\!|A, K_{AB}, N_A|\!\}_{K_B}$
Event 2	B begins	"B received K_{AB} from A"
Message 2	$B \rightarrow A$:	$\{\!|B, K_{AB}, N_A, N_{B1}|\!\}_{K_A}, N_{B2}$
Event 3	A ends	"B received K_{AB} from A"
Event 4	A begins	"A sends M to B"
Message 3	$A \rightarrow B$:	$N_{B1}, \{M, N_{B2}\}_{K_{AB}}$
Event 5	B ends	"A generates K_{AB} for B"
Event 6	B ends	"A sends M to B"

Figure 1 is a spi-calculus version of the protocol. The top-level process, $System(net)$ generates two fresh key pairs $pair_A$ and $pair_B$, and places a single sender and a single receiver in parallel. We publish the public encryption keys of A and B, to allow the attacker access to them. The parameter net is a communications channel, on which the attacker may send or receive, representing the untrusted network. For simplicity,

Figure 1 includes just one sender and one receiver; it is easy to extend the program to run multiple senders and receivers in parallel.

Given the assertions embedded in the program, our formal specification is simply the following:

Authenticity: The process $System(net)$ is robustly safe.

3 Authenticity by Typing for Asymmetric Cryptographic Protocols

Section 3.1 describes informally how we type messages. Section 3.2 explains the subtyping relation. Section 3.3 explains how we ascribe effects to processes, respectively. The full details of the type system are left to Appendix C. In Section 3.4 we explain how to type the assertions in the example of the previous section.

3.1 Types for Messages

We give the syntax of types and explain when a message M has type T, written informally $M : T$.

Apart from challenge/response types, deferred to the next section, here is the syntax of our types.

Types:

$S, T, U ::=$		type
$(x{:}T, U)$		dependent pair type (x bound in U)
$T + U$		sum type
Un		data known to the opponent
Top		top
$\mathsf{SharedKey}(T)$		shared-key type
$\mathsf{KeyPair}(T)$		asymmetric key-pair
$k\ \mathsf{Key}(T)$	(where k either $\mathsf{Encrypt}$ or $\mathsf{Decrypt}$)	encryption or decryption part

Many of these types are standard or appear in earlier work on spi [GJ01a]. Messages of type $(x{:}T, U)$ are dependent records (M, N), where $M : T$, and $N : U\{x{\leftarrow}T\}$. Messages of type $T + U$ are tagged unions, either $\mathsf{inl}\ (M)$ with $M : T$ or $\mathsf{inr}\ (N)$ with $N : U$. Messages of type Un are arbitrary, untrusted data known to the opponent. Any typeable message is also of type Top. Messages of type $\mathsf{SharedKey}(T)$ are names representing symmetric keys for encypting data of type T to yield a ciphertext of type Un.

We need some new types for asymmetric cryptography. A message of type $\mathsf{KeyPair}(T)$ is a name representing an asymmetric key-pair for encrypting data of type T. Messages of types $\mathsf{Encrypt\ Key}(T)$ or $\mathsf{Decrypt\ Key}(T)$ take the form $\mathsf{Encrypt}\ p$ or $\mathsf{Decrypt}\ p$, respectively, where $p : \mathsf{KeyPair}(T)$.

The formal message typing judgment takes the form $E \vdash M : T$, where E is an *environment*, that assigns types to the names in scope. An environment takes the form $x_1{:}T_1, \ldots, x_n{:}T_n$.

Our typing rules rely on a subtyping relation on types, written $E \vdash T <: U$. Intuitively, this means that any message of type T also is of type U. We explain subtyping in detail in the next section.

The formal typing rules defining $E \vdash M : T$ are mostly standard [GJ01a]. Full details are in Appendix C. Here are some samples, the rules for applying subtyping and for typing asymmetric operations

Type Rules for Messages:

$$\text{(Msg Subsum)}\quad \frac{E \vdash M : T \quad E \vdash T <: U}{E \vdash M : U} \qquad \text{(Msg Part)}\quad \frac{E \vdash M : \mathsf{KeyPair}(T)}{E \vdash k\ (M) : k\ \mathsf{Key}(T)} \qquad \text{(Msg Asymm)}\quad \frac{E \vdash M : T \quad E \vdash N : \mathsf{Encrypt\ Key}(T)}{E \vdash \{\!|M|\!\}_N : \mathsf{Un}}$$

The type-rules in Appendix C are all syntax-directed, and so it is routine to implement a top-down type-checker for this type system.

3.2 The Subtyping Relation

The *subtyping* relation $E \vdash T <: U$ means that messages of type T can be used in place of a message of type U. The environment E tracks the names in scope, and sometimes is omitted in informal discussion.

A type's relationship to the type Un of data known to the opponent determines whether it can be sent to or received from the opponent. Let a type T be *public* if and only if $T <: \mathsf{Un}$. Let a type T be *tainted* if and only if $\mathsf{Un} <: T$.

The following tables of rules define the subtyping relation. Subtyping is reflexive and transitive, and has a top element Top:

Basic rules for subtyping:

$E \vdash T \Longrightarrow E \vdash T <: T$	(Sub Refl)
$E \vdash S <: T, E \vdash T <: U \Longrightarrow E \vdash S <: U$	(Sub Trans)
$E \vdash T \Longrightarrow E \vdash T <: \mathsf{Top}$	(Sub Top)

Pair types $(x : T, U)$, sum types $T + U$ and decryption key types $\mathsf{Decrypt\ Key}(T)$ are covariant; encryption key types $\mathsf{Encrypt\ Key}(T)$ are contravariant:

Congruence Rules for Subtyping:

(Sub Pair)(where $x \notin dom(E)$)
$$\frac{E \vdash T <: T' \quad E, x{:}T \vdash U <: U'}{E \vdash (x{:}T, U) <: (x{:}T', U')}$$

(Sub Sum)
$$\frac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash T + U <: T' + U'}$$

(Sub Enc Key)
$$\frac{E \vdash T' <: T}{E \vdash \mathsf{Encrypt\ Key}(T) <: \mathsf{Encrypt\ Key}(T')}$$

(Sub Dec Key)
$$\frac{E \vdash T <: T'}{E \vdash \mathsf{Decrypt\ Key}(T) <: \mathsf{Decrypt\ Key}(T')}$$

A pair type $(x : \mathsf{Un}, \mathsf{Un})$ contains only public data, so is itself public. Similarly, the sum type $\mathsf{Un} + \mathsf{Un}$, the symmetric key type $\mathsf{SharedKey}(\mathsf{Un})$, the asymmetric key type $k\ \mathsf{Key}(\mathsf{Un})$, and the key pair type $\mathsf{KeyPair}(\mathsf{Un})$ are all public types:

Subtyping Rules for Public Types:

$E \vdash (x{:}\mathsf{Un}, \mathsf{Un}) <: \mathsf{Un}$	(Public Pair)
$E \vdash \mathsf{Un} + \mathsf{Un} <: \mathsf{Un}$	(Public Sum)
$E \vdash \mathsf{SharedKey}(\mathsf{Un}) <: \mathsf{Un}$	(Public Shared Key)
$E \vdash k\ \mathsf{Key}(\mathsf{Un}) <: \mathsf{Un}$	(Public Key)
$E \vdash \mathsf{KeyPair}(\mathsf{Un}) <: \mathsf{Un}$	(Public Keypair)

A pair type $(x : \mathsf{Un}, \mathsf{Un})$ contains only tainted data, so is itself tainted. Similarly, the sum type $\mathsf{Un} + \mathsf{Un}$, the symmetric key type $\mathsf{SharedKey}(\mathsf{Un})$, the asymmetric key type $k\ \mathsf{Key}(\mathsf{Un})$, and the key pair type $\mathsf{KeyPair}(\mathsf{Un})$ are all tainted types:

Subtyping Rules for Tainted Types:

$E \vdash \mathsf{Un} <: (x{:}\mathsf{Un}, \mathsf{Un})$	(Tainted Pair)
$E \vdash \mathsf{Un} <: \mathsf{Un} + \mathsf{Un}$	(Tainted Sum)
$E \vdash \mathsf{Un} <: \mathsf{SharedKey}(\mathsf{Un})$	(Tainted Shared Key)
$E \vdash \mathsf{Un} <: k\ \mathsf{Key}(\mathsf{Un})$	(Tainted Key)
$E \vdash \mathsf{Un} <: \mathsf{KeyPair}(\mathsf{Un})$	(Tainted Keypair)

We end this section by discussing the two dual applications of key-pairs of type $\mathsf{KeyPair}(T)$.

- In public-key applications, the payload type T should be tainted, since anyone, including the opponent, can encrypt messages. If T is tainted, then $\mathsf{Un} <: T$. The type constructor for encryption keys is contravariant, so $\mathsf{Encrypt\ Key}(T) <: \mathsf{Encrypt\ Key}(\mathsf{Un}) <: \mathsf{Un}$. Hence, $\mathsf{Encrypt\ Key}(T)$ is public.
- In digital signature applications, the payload type T should be public, since anyone, including the opponent, can check signatures. If T is public, then $T <: \mathsf{Un}$. The type constructor for decryption keys is covariant, so $\mathsf{Decrypt\ Key}(T) <: \mathsf{Decrypt\ Key}(\mathsf{Un}) <: \mathsf{Un}$. Hence, $\mathsf{Decrypt\ Key}(T)$ is public.

Note in particular that if a nïave programmer attempts to use a key of type $\mathsf{Key}(T)$ for both public-key and digital signature, then they will discover that $T <: \mathsf{Un} <: T$, and so $\mathsf{Key}(T) = \mathsf{Un}$. This enforces the common engineering practice that keys which are used for both public-key and digital signature applications are not to be trusted.

3.3 Effects for Processes

We write $E \vdash P : es$ to mean that the process P is well-typed in environment E, and that the effect es is an upper bound on the certain aspects of the behaviour P. An effect is a multiset (that is, an unordered list) of *atomic effects*. These can take three forms:

- $\mathsf{end}\ L$, used to track the unmatched end-events of a process.
- $\mathsf{check\ Public}\ N$ and $\mathsf{check\ Private}\ N$, used to track how often a nonce has been used.
- $\mathsf{trust}\ M{:}T$, a trust effect used to gain the trust information that data M really has type T.

Overall, the goal when type-checking a protocol is to assign it the empty effect, for then it has no unbalanced end-events, and therefore is safe. This section explains the intuitions behind the rules for assigning effects to processes, which in part rely on challenge/response types for nonces.

Let e stand for an atomic effect, and let es stand for an *effect*, that is, a multiset $[e_1, \ldots, e_n]$ of atomic effects. We write $es + es'$ for the multiset union of the two multisets es and es', that is, their concatenation. We write $es - es'$ for the multiset subtraction of es' from es, that is, the outcome of deleting an occurrence of each atomic effect in es' from es. If an atomic effect does not occur in an effect, then deleting the atomic effect leaves the effect unchanged.

The interesting part of the effect system for processes is how it handles nonce handshakes. Each nonce handshake breaks down into several steps:

(1) Participant A creates a fresh nonce and sends it to B inside a message M.
(2) Participant B returns the nonce to A inside message N.
(3) Participant A checks that she received the same nonce as she sent. From this (and some trust in the cryptography used to encrypt secret messages) she knows that B must have been involved in the dialogue.
(4) To avoid vulnerability to replay of messages containing the nonce, A subsequently discards the nonce and refuses to accept it again.

Our type system requires us to distinguish nonces which may be published to the untrusted agents (Public nonces) from ones which may not (Private nonces). We let ℓ be either Public or Private. We type-check the above four steps as follows:

(1) A creates the nonce N as having type $\ell\ \mathsf{Challenge}\ es$, where es is an effect, and sends it to B.
(2) B casts the nonce to a new type $\ell\ \mathsf{Response}\ fs$, where fs is also an effect, and returns it to A. In order to do this, B must ensure that the effect $es + fs$ is justified.

(3) After receiving the newly cast nonce, A uses a name-check check N is N'; to check nonce equality of the original nonce challenge with the new nonce response. If this check succeeds, A can assume that the effect $es + fs$ is justified.

(4) To guarantee that each nonce N is only checked once, we introduce a new atomic effect check ℓ N, which is introduced each time a check N is N'; is used. This can only be justified by freshly generating the nonce N, which ensures that each nonce is only ever checked once.

This four-phase process extends the treatment of POSH nonces in earlier work [GJ01a], and is sufficient to type check symmetric key protocols. Asymmetric key protocols, however, have dynamic trust, where the trust in a piece of data may increase over time. In our system, trust is given by knowing the type of data, so dynamic trust is modelled by allowing the type of some data to change over time. We model this by introducing two new statements, which allow A to communicate to B that a piece of data M has type T:

(1) A knows that M has type T, and executes witness M:T; which justifies a *trust effect* trust M:T. A can then use the nonce mechanism described above to communicate this trust effect to B.

(2) B executes trust M is $(x$:$T)$; which gives M type T by binding M to variable x of type T. This requires a trust effect trust M:T.

In this fashion, type information can be exchanged between honest agents, using the same mechanism as authenticity information.

Effects:

e, f ::=	atomic effect
end L	end-event labelled with message L
check ℓ N	name-check for a nonce N
trust M:T	trust that a message M has type T
es, fs ::=	effect
$[e_1, \ldots, e_n]$	multiset of atomic effects

Effects contain no name binders, so the free names of an effect are the free names of the message and types they contain. We write $es\{x \leftarrow M\}$ for the outcome of a capture-avoiding substitution of the message M for each free occurrence of the name x in the effect es.

In Appendix C we define $E \vdash es$ meaning 'in environment E, the effect es is well-formed'.

We extend the grammar of types to include nonce types. These come in two varieties: Public nonces (for SOPH and POSH nonce handshakes) and Private nonces (for SOSH nonce handshakes). Note that:

- POSH nonces are sent out with tainted public type Public Challenge [], and return with untainted secret type Public Response es.
- SOPH nonces are sent out with untainted secret type Public Challenge es, and return with tainted public type Public Response [].
- SOSH nonces are send out with tainted secret type Private Challenge es, and return with tainted secret type Private Response fs.

These properties are captured by the subtyping rules for nonce types.

Nonce Types:

T, U ::=	type
...	as in Section 3.1
ℓ Challenge es	nonce challenge type

ℓ Response es	nonce response type
$\ell ::=$	privacy
Public	public
Private	private

Subtyping Rules for Nonce Types:

$E \vdash$ Public Challenge $[] <:$ Un	(Public Challenge $[]$)
$E \vdash fs \Longrightarrow E \vdash$ Public Response $fs <:$ Un	(Public Response)
$E \vdash$ Un $<:$ Public Challenge $[]$	(Tainted Public Challenge $[]$)
$E \vdash$ Un $<:$ Public Response $[]$	(Tainted Public Response $[]$)
$E \vdash es \Longrightarrow E \vdash$ Un $<:$ Private Challenge es	(Tainted Private Challenge)
$E \vdash es \Longrightarrow E \vdash$ Un $<:$ Private Response es	(Tainted Private Response)

We extend the grammar of processes to include nonce manipulation:

Processes Manipulating Nonces:

$O,P,Q,R ::=$	process
...	as in Section 2.1
cast M is $(x{:}T);P$	nonce-casting
witness $M{:}T;P$	witness testimony
trust M is $(x{:}T);P$	trusted-casting

In a process cast M is $(x{:}T);P$ or trust M is $(x{:}T);P$, the name x is bound; its scope is the process P.

- The process cast M is $(x{:}T);P$ casts the message M to the type T, by binding the variable x to M, and then running P. (This process can only be typed by our type system if M has type ℓ Challenge es and T is of the form ℓ Response es.)
- The process witness $M{:}T;P$ requires that M has type T. It justifies any number of effects of the form trust $M{:}T$.
- The process trust M is $(x{:}T);P$ casts the message M to the type T, by binding the variable x to M, and then running P. (This process requires an effect trust $M{:}T$ to be justified: this allows type information to be communicated amongst honest agents.)

We can now give rules which calculate the effect of a process. Most of the rules are the same as [GJ01a], so are given in Appendix C. We only provide the rules for asymmetric cryptography, nonce challenges, and dynamic trust here.

The rule for asymmetric decryption is similar to the one for symmetric decryption: if M is a plaintext of type T and K is a decrypt key of type Decrypt Key(T) then we can decrypt a ciphertext of type Un to reveal the plaintext of type T:

Rule for Asymmetric Cryptography:

(Proc Asymm) (where $x \notin dom(E) \cup fn(es)$)

$$\frac{E \vdash M : \text{Un} \quad E \vdash N : \text{Decrypt Key}(T) \quad E, x{:}T \vdash P : es}{E \vdash \text{decrypt } M \text{ is } \{\!|x{:}T|\!\}_{N^{-1}}; P : es}$$

The rules for nonce types are similar to the rules from [GJ01a], except that they support SOPH and POSH nonces as well as POSH nonces:

Rules for Challenges and Responses:

(Proc Cast) (where $x \notin dom(E) \cup fn(fs)$)

$$\frac{E \vdash M : \ell\ \mathsf{Challenge}\ es_C \quad E, x{:}\ell\ \mathsf{Response}\ es_R \vdash P : fs}{E \vdash \mathsf{cast}\ M\ \mathsf{is}\ (x{:}\ell\ \mathsf{Response}\ es_R); P : es_C + es_R + fs}$$

(Proc Check)

$$\frac{E \vdash M : \ell\ \mathsf{Challenge}\ es_C \quad E \vdash N : \ell\ \mathsf{Response}\ es_R \quad E \vdash P : fs}{E \vdash \mathsf{check}\ M\ \mathsf{is}\ N; P : (fs - (es_C + es_R)) + [\mathsf{check}\ \ell\ M]}$$

(Proc Challenge) (where $x \notin dom(E) \cup fn(es - [\mathsf{check}\ \ell\ x])$)

$$\frac{E, x{:}\ell\ \mathsf{Challenge}\ fs \vdash P : es}{E \vdash \mathsf{new}\ (x{:}\ell\ \mathsf{Challenge}\ fs); P : es - [\mathsf{check}\ \ell\ x]}$$

The rules for trust effects are new in this paper. A process $\mathsf{witness}\, M{:}T; P$ requires that message M has type T, and allows the process P to use the trust effect $\mathsf{trust}\, M{:}T$ many times; A process $\mathsf{trust}\, M\, \mathsf{is}\, (x{:}T); P$ makes use of the trust effect $\mathsf{trust}\, M{:}T$ to use M with type T:

Rules for Witness Testimony and Trusted-Casting:

(Proc Witness)

$$\frac{E \vdash M : T \quad E \vdash P : es + [\mathsf{trust}\, M{:}T, \ldots, \mathsf{trust}\, M{:}T]}{E \vdash \mathsf{witness}\, M{:}T; P : es}$$

(Proc Trust) (where $x \notin dom(E) \cup fn(es)$)

$$\frac{E \vdash M : \mathsf{Top} \quad E, x{:}T \vdash P : es}{E \vdash \mathsf{trust}\, M\, \mathsf{is}\, (x{:}T); P : es}$$

Finally, we state the safety theorem for this type system. The proof depends on identifying a suitable runtime invariant and showing it is preserved by the operational semantics.

Theorem 1 (Robust Safety) *If* $x_1{:}\mathsf{Un}, \ldots, x_n{:}\mathsf{Un} \vdash P : []$ *then* P *is robustly safe.*

3.4 Typing the Example

We now show that the process $System(net)$ has empty effect, and so by Theorem 1 (Robust Safety) is robustly safe. We give other examples in Appendix A, including an example using signed certificates. Each nonce has two types: one type when it is used as a nonce challenge, and one for when it is used as a response. The types for N_A are:

$$\begin{aligned} \mathsf{C}_A(a,b,k) &= \mathsf{Private\ Challenge}\ [\mathsf{end}\ (\text{“}a \text{ generates } k \text{ for } b\text{”})] \\ \mathsf{R}_A &= \mathsf{Private\ Response}\ [] \end{aligned}$$

The types for N_{B1} are:

$$\begin{aligned} \mathsf{C}_{B1}(a,b,k) &= \mathsf{Public\ Challenge}\ [\mathsf{end}\ (\text{“}b \text{ received } k \text{ from } a\text{”}), \mathsf{trust}\, k{:}\mathsf{K}_{AB}(a,b)] \\ \mathsf{R}_{B1} &= \mathsf{Public\ Response}\ [] \end{aligned}$$

The types for N_{B2} are:

$$\begin{aligned} \mathsf{C}_{B2} &= \mathsf{Public\ Challenge}\ [] \\ \mathsf{R}_{B2}(a,b,m) &= \mathsf{Public\ Response}\ [\mathsf{end}\ (\text{“}a \text{ sends } m \text{ to } b\text{”})] \end{aligned}$$

Keys have only one type, giving the type of the plaintext encrypted with the key. The type for K_{AB} is:

$$\mathsf{K}_{AB}(a,b) = \mathsf{SharedKey}(m{:}\mathsf{Payload}, r{:}\mathsf{R}_{B2}(a,b,m))$$

$Sender(net : \mathsf{Un}, private_A : \mathsf{Decrypt}\ \mathsf{K}_A(A), public_B : \mathsf{Encrypt}\ \mathsf{K}_B(B)) \triangleq$

new $(key_{AB} : \mathsf{K}_{AB}(A,B))$;
// Effect: []
new $(challenge_A : \mathsf{C}_A(A,B,key_{AB}))$;
// Effect: [check Private $challenge_A$]
begin "A generates key_{AB} for B";
out net $\{\!|A, key_{AB}, challenge_A|\!\}_{public_B}$;
inp net $(ctext_2 : \mathsf{Un}, challenge_{B2} : \mathsf{C}_{B2})$;
decrypt $ctext_2$ is $\{\!|B, key_{AB}, response_A : \mathsf{R}_A, challenge_{B1} : \mathsf{C}_{B1}(A,B,key_{AB})|\!\}_{private_A^{-1}}$;
// Effect: [check Private $challenge_A$, end "A generates key_{AB} for B"]
check $challenge_A$ is $response_A$;
// Effect: [end "B received key_{AB} from A", end "A generates key_{AB} for B"]
end "B received key_{AB} from A";
new $(msg : \mathsf{Payload})$;
// Effect: [end "A generates key_{AB} for B"]
begin "A sends msg to B";
// Effect: [end "A generates key_{AB} for B", end "A sends msg to B"]
witness key_{AB}:$\mathsf{K}_{AB}(A,B)$;
// Effect: [end "A generates key_{AB} for B", trust key_{AB}:$\mathsf{K}_{AB}(A,B)$, end "A sends msg to B"]
cast $challenge_{B1}$ is $(response_{B1} : \mathsf{R}_{B1})$;
// Effect: [end "A sends msg to B"]
cast $challenge_{B2}$ is $(response_{B2} : \mathsf{R}_{B2}(A,B,msg))$;
// Effect: []
out net $(response_{B1}, \{msg, response_{B2}\}_{key_{AB}})$;

$Receiver(net : \mathsf{Un}, public_A : \mathsf{Encrypt}\ \mathsf{K}_A(A), private_B : \mathsf{Decrypt}\ \mathsf{K}_B(B)) \triangleq$

repeat
inp net $(ctext_1 : \mathsf{Un})$;
decrypt $ctext_1$ is $\{\!|A, untrusted : \mathsf{Top}, challenge_A : \mathsf{C}_A(A,B,key_{AB})|\!\}_{private_B^{-1}}$;
// Effect: []
new $(challenge_{B1} : \mathsf{C}_{B1}(A,B,key_{AB}))$;
// Effect: [check Public $challenge_{B1}$]
new $(challenge_{B2} : \mathsf{C}_{B2})$;
// Effect: [check Public $challenge_{B1}$, check Public $challenge_{B2}$]
begin "B received $untrusted$ from A";
// Effect: [end "B received $untrusted$ from A", check Public $challenge_{B1}$, check Public $challenge_{B2}$]
cast $challenge_A$ is $(response_A : \mathsf{R}_A)$;
out net $\{\!|B, untrusted, challenge_A, challenge_{B1}|\!\}_{public_A}, challenge_{B2}$;
inp net $(response_{B1} : \mathsf{R}_{B1}, ctext_3 : \mathsf{Un})$;
// Effect: [check Public $challenge_{B1}$, check Public $challenge_{B2}$]
check $challenge_{B1}$ is $response_{B1}$;
// Effect: [end "A generates $untrusted$ for B", trust $untrusted$:$\mathsf{K}_{AB}(A,B)$, check Public $challenge_{B2}$]
end "A generates $untrusted$ for B";
// Effect: [trust $untrusted$:$\mathsf{K}_{AB}(A,B)$, check Public $challenge_{B2}$]
trust $untrusted$ is $(key_{AB} : \mathsf{K}_{AB}(A,B))$;
decrypt $ctext_3$ is $\{msg : \mathsf{Payload}, response_{B2} : \mathsf{R}_{B2}(A,B,msg)\}_{key_{AB}}$;
// Effect: [check Public $challenge_{B2}$]
check $challenge_{B2}$ is $response_{B2}$;
// Effect: [end "A sends msg to B"]
end "A sends msg to B";

Figure 2: Proof that the example is robustly safe

The type for KA is:

$$\mathsf{K}_A(a) \quad = \quad \mathsf{Key}(b{:}\mathsf{Principal}, k{:}\mathsf{Top}, r_A{:}\mathsf{R}_A, c_{B1}{:}\mathsf{C}_{B1}(a,b,k))$$

The type for KB is:

$$\mathsf{K}_B(b) \quad = \quad \mathsf{Key}(a{:}\mathsf{Principal}, k{:}\mathsf{Top}, c_A{:}\mathsf{C}_A(a,b,k))$$

We can then check that the encryption keys for each of the participants is public:

- The types Principal, Top, R_A and $\mathsf{C}_{B1}(a,b,k)$ are all tainted,
  so the record type $(b{:}\mathsf{Principal}, k{:}\mathsf{Top}, r_A{:}\mathsf{R}_A, c_{B1}{:}\mathsf{C}_{B1}(a,b,k))$ is tainted,
  so the encryption key type Encrypt $\mathsf{K}_A(a)$ is public.
- The types Principal, Top and $\mathsf{C}_A(a,b,k)$ are all tainted,
  so the record type $(a{:}\mathsf{Principal}, k{:}\mathsf{Top}, c_A{:}\mathsf{C}_A(a,b,k))$ is tainted,
  so the encryption key type Encrypt $\mathsf{K}_B(b)$ is public.

In Figure 2, we annotate the participants in the protocol with types and appropriate casts, to ensure that the protocol is robustly safe. When we typecheck the receiver, we cannot initially trust the session key, so we have to give it type Top rather than key type. It is only once message 3 has arrived that we know that the key is really from A and not fabricated by an intruder, at which point we can cast it to $key_{AB} : \mathsf{K}_{AB}(A,B)$. This is justified by the trust effect $\mathsf{trust}\, key_{AB} : \mathsf{K}_{AB}(A,B)$ which is communicated as part of nonce challenge $challenge_{B1}$.

4 Conclusions and Further Work

This paper presents a type and effect system for asymmetric cryptographic protocols. The main new ideas are (1) to identify the separate notions of public and tainted types, defined formally via subtyping; (2) to formalize the way nonces increase the degree of trust in data via trust effects; and (3) to support different styles of nonce handshake via challenge/response types. Examples show how to model common features of asymmetric protocols such as key exchange and the use of signed certificates.

The Cryptyc project [GJ01b] includes a tool for type-checking symmetric key protocols, and have used this tool to verify most of the protocols in the Clark–Jacob survey [CJ97]. We expect that this tool could easily be extended to include the type and effect system described here.

The long-term aims of all the work on typing cryptographic protocols are to find secrecy and authenticity types that are as compellingly intuitive as BAN formulas, are easy to type-check, have a precise semantics, and support a wide range of cryptographic transforms and protocol idioms. This paper represents solid progress towards these goals.

Still, several limitations remain to be addressed. Our types for encryption give every ciphertext type Un, so we cannot model some forms of nested cryptography such as “sign-then-encrypt” or “encrypt-then-sign”. Our attacker model assumes that every opponent is completely untrusted: they only have access to data of type Un; this does not model attacks where opponents are partially trusted (for example, M may have a public key K_M which is trusted to give authenticity information about M but not about A or B). Also, the attacker model does not support key-compromise attacks. Our encryption model does not include other encryption technologies such as hashing, Diffie–Hellman key exchange and constructing keys from pass phrases.

A Other Examples

A.1 Abbreviations Used in Examples

In these examples, we make use of the following syntax sugar:

- Dependent record types $(x_1{:}T_1, \ldots, x_n{:}T_n)$, rather than just pairs.
- Tagged union types $(\ell_1(T_1) \mid \cdots \mid \ell_n(T_n))$ rather than just binary choice $T + U$.
- Strings "$a_1 \ldots a_n$" used in correspondence assertions.
- A public, tainted type Principal for principal names.

We show in the full version of this paper that these constructs can be derived from our base language.

A.2 Authentication using certificates

A simple authentication protocol using certificates is the ISO Public Key Two-Pass Unilateral Authentication Protocol described by Clark and Jacob [CJ97]. In this protocol, a principal A sends a certificate for her public key K_A together with a message encrypted with her private key K_A^{-1} to principal B. The certificate is encrypted with the private key K_{CA}^{-1} of a certificate authority CA. The protocol, simplified to remove messages unrelated to authenticity, is:

Message 1	$B \to A$:	N_B
Event 1	A begins	"A sending M to B"
Message 2	$A \to B$:	$\{\!\{A, K_A\}\!\}_{K_{CA}^{-1}}, \{\!\{M, B, N_B\}\!\}_{K_A^{-1}}$
Event 2	B ends	"A sending M to B"

Translating the protocol into the spi-calculus with correspondence assertions is routine, but we have to provide types for the participants. The type of A's key is (for any public type Payload):

$$\mathsf{K}_A(a : \mathsf{Principal}) = \mathsf{Key}(msg : \mathsf{Payload}, b : \mathsf{Principal}, n : \mathsf{Public\ Response}\ [\mathsf{end}\ \text{"}a\text{ sending }msg\text{ to }b\text{"}])$$

The type of the certificate authority CA's key is:

$$\mathsf{K}_{CA} = \mathsf{Key}(a : \mathsf{Principal}, k_A : \mathsf{K}_A(a))$$

We can then check that the participants public keys are public:

- The plaintext of type $\mathsf{K}_A(a)$ is public so Decrypt $\mathsf{K}_A(a)$ is public (this depends on the Payload type being public).
- The plaintext of type K_{CA} is public, so Decrypt K_{CA} is public.

It is then routine to verify that this protocol typechecks and is effect-free, and so is robustly safe.

A.3 Needham–Schroeder–Lowe

The full Needham–Schroeder–Lowe [NS78, Low96] protocol makes use of a certificate authority S which validates the public keys K_A and K_B of principals A and B, by encrypting the public keys with private encryption key K_S^{-1}. A and B use S to find each others public keys, then use two SOSH nonce handshakes to establish contact:

Message 1	$A \to S$:	A, B
Message 2	$S \to A$:	$\{\!\|B, K_B\|\!\}_{K_S^{-1}}$
Event 1	A begins	"A contacting B"
Message 3	$A \to B$:	$\{\!\|msg_3(A, N_A)\|\!\}_{K_B}$
Event 2	B begins	"B contacted by A"
Message 4	$B \to S$:	B, A
Message 5	$S \to B$:	$\{\!\|A, K_A\|\!\}_{K_S^{-1}}$
Message 6	$B \to A$:	$\{\!\|msg_6(B, N_A, N_B)\|\!\}_{K_A}$
Event 3	A ends	"B contacted by A"
Message 7	$A \to B$:	$\{\!\|msg_7(N_B)\|\!\}_{K_B}$
Event 4	B ends	"A contacting B"

Translating the protocol into the spi-calculus with correspondence assertions is routine, but we have to provide types for the participants. The type of A and B's keys is:

$$
\begin{array}{l}
\mathsf{K}_P(p : \mathsf{Principal}) = \mathsf{Key}(\\
\quad msg_3(q : \mathsf{Principal}, n_Q : \mathsf{Private\ Challenge}\ [\text{end "}p\text{ contacted by }q\text{"}]) \\
\quad \mid msg_6(q : \mathsf{Principal}, n_P : \mathsf{Private\ Response}\ [], n_Q : \mathsf{Private\ Challenge}\ [\text{end "}p\text{ contacting }q\text{"}]) \\
\quad \mid msg_7(\mathsf{Private\ Response}\ []) \\
)
\end{array}
$$

The type of S's key is:

$$\mathsf{K}_S = \mathsf{Key}(p : \mathsf{Principal}, k_P : \mathsf{K}_P(p))$$

We can then check that the participants public keys are public:

- The plaintext of type $\mathsf{K}_P(p)$ is tainted, so $\mathsf{Encrypt}\ \mathsf{K}_P(p)$ is public (note that this depends on private nonce types being tainted).
- The plaintext of type K_S is public, so $\mathsf{Decrypt}\ \mathsf{K}_S$ is public.

It is then routine to verify that NSL typechecks is effect-free, and so is robustly safe. In the type for msg_6 we require q's name to be present, otherwise the type for msg_6 is not well-formed; this is the basis of Lowe's attack on the original Needham–Schroeder public key protocol.

B Operational Semantics and Safety

Processes include correspondence assertion events $\mathsf{begin}\ L$ and $\mathsf{end}\ L$ which describe the authenticity properties expected of the protocol. We take a new approach to formalizing correspondence assertions via a tuple space metaphor. Informally, we regard these events as analogous to put and get in a fictitious secure tuple space similar to Linda [CG89]. When a $\mathsf{begin}\ L$ event takes place, we add L to the secure tuple space. When an $\mathsf{end}\ L$ event takes place, we remove L from the tuple space: a violation of the security requirements of the protocol have taken place if L is not present. In reality, this tuple space does not exist, so we need the type system ensure that every $\mathsf{end}\ L$ event is guaranteed to succeed. In an implementation of a typechecked protocol, $\mathsf{begin}\ L$ and $\mathsf{end}\ L$ events can be implemented as no-ops, since the type checker guarantees that the $\mathsf{end}\ L$ will succeed.

We define a *state As* of a protocol to be a tuple space (that is, a multiset of tuples which have been begun but not ended) and a thread pool (that is, a multiset of executing threads).

Activities

$A,B,C ::=$	activity
$\quad L$	$\quad$ tuple labelled L
$\quad P$	$\quad$ process P
$Ls ::= [L_1,\ldots,L_n]$	tuple space: multiset of tuples
$Ps,Qs ::= [P_1,\ldots,P_n]$	thread pool: multiset of processes
$As,Bs,Cs ::= Ls + Ps$	state: tuple space plus thread pool

The free names $fn(As)$ of a state As are defined in the usual way. We define the operational semantics of a state by giving a reduction relation $As \rightarrow Bs$ meaning 'in state As the program can perform one step of computation and become state Bs'.

State Transitions:

$[\mathsf{out}\ x\ M] + [\mathsf{inp}\ x\ (y{:}T);P] + As \rightarrow [P\{y{\leftarrow}M\}] + As$	(Trans I/O)
$[\mathsf{out}\ x\ M] + [\mathsf{repeat\ inp}\ x\ (y{:}T);P] + As \rightarrow [P\{y{\leftarrow}M\}] + [\mathsf{repeat\ inp}\ x\ (y{:}T);P] + As$	(Trans Repl I/O)
$x \notin fn(As) \Rrightarrow [\mathsf{new}\ (x{:}T);P] + As \rightarrow [P] + As$	(Trans New)
$[P \mid Q] + As \rightarrow [P] + [Q] + As$	(Trans Par)
$[\mathsf{stop}] + As \rightarrow As$	(Trans Stop)
$[\mathsf{split}\ (M,N)\ \mathsf{is}\ (x{:}T,y{:}U);P] + As \rightarrow [P\{x{\leftarrow}M\}\{y{\leftarrow}N\}] + As$	(Trans Split)
$[\mathsf{match}\ (M,N)\ \mathsf{is}\ (M,y{:}U);P] + As \rightarrow [P\{y{\leftarrow}N\}] + As$	(Trans Match)
$[\mathsf{case\ inl}\ (M)\ \mathsf{is\ inl}\ (x{:}T)\ P\ \mathsf{is\ inr}\ (y{:}U)\ Q] + As \rightarrow [P\{x{\leftarrow}M\}] + As$	(Trans Inl)
$[\mathsf{case\ inr}\ (N)\ \mathsf{is\ inl}\ (x{:}T)\ P\ \mathsf{is\ inr}\ (y{:}U)\ Q] + As \rightarrow [Q\{y{\leftarrow}N\}] + As$	(Trans Inr)
$[\mathsf{decrypt}\ \{M\}_N\ \mathsf{is}\ \{x{:}T\}_N;P] + As \rightarrow [P\{x{\leftarrow}M\}] + As$	(Trans Symm)
$[\mathsf{decrypt}\ \{\!|M|\!\}_{\mathsf{Encrypt}\ (N)}\ \mathsf{is}\ \{\!|x{:}T|\!\}_{\mathsf{Decrypt}\ (N)^{-1}};P] + As \rightarrow [P\{x{\leftarrow}M\}] + As$	(Trans Asymm)
$[\mathsf{begin}\ L;P] + As \rightarrow [L] + [P] + As$	(Trans Begin)
$[L] + [\mathsf{end}\ L;P] + As \rightarrow [P] + As$	(Trans End)
$[\mathsf{check}\ x\ \mathsf{is}\ x;P] + As \rightarrow [P] + As$	(Trans Check)
$[\mathsf{cast}\ x\ \mathsf{is}\ (y{:}T);P] + As \rightarrow [P\{y{\leftarrow}x\}] + As$	(Trans Cast)
$[\mathsf{witness}\ M{:}T;P] + As \rightarrow [P] + As$	(Trans Witness)
$[\mathsf{trust}\ M\ \mathsf{is}\ (x{:}T);P] + As \rightarrow [P\{x{\leftarrow}M\}] + As$	(Trans Trust)

An error state is one where an $\mathsf{end}\ L$ event is encountered without a matching tuple L in the tuple space.

Error States and Safety:

A state is an *error* iff it has the form $[\mathsf{end}\ L;P] + As$ where $L \notin As$.
A process P is *safe* iff there is no error state As such that $[P] \rightarrow^* As$.

C Full Definition of the Type System

In this section, we give the full definition of the type system for the spi calculus with correspondence assertions. The type system is given as a number of judgements of the form $E \vdash \mathcal{J}$.

Environments:

$D,E ::=$	environment
$\quad x_1{:}T_1,\ldots,x_n{:}T_n$	$\quad$ unordered set of entries

Judgments $E \vdash \mathcal{J}$:

$E \vdash \diamond$	good environment
$E \vdash es$	good effect es
$E \vdash T$	good type T
$E \vdash T <: U$	subtyping
$E \vdash M : T$	good message M of type T
$E \vdash P : es$	good process P with effect es

A well-formed environment E is one where $E \vdash \diamond$.

Rules for Environments:

(Env Good)(where $E = x_1{:}T_1, \ldots, x_n{:}T_n$)

$$\frac{E \vdash T_i \quad \forall i \in 1..n \quad x_1, \ldots, x_n \text{ distinct}}{E \vdash \diamond}$$

In an environment E, a well-formed effect es is one where $E \vdash es$.

Rules for Effects:

(Effect $\varnothing$)
$$\frac{}{E \vdash []}$$

(Effect End)
$$\frac{E \vdash es \quad E \vdash L : \mathsf{Top}}{E \vdash es + [\mathsf{end}\ L]}$$

(Effect Check)
$$\frac{E \vdash es \quad E \vdash N : \ell\ \mathsf{Challenge}\ fs}{E \vdash es + [\mathsf{check}\ \ell\ N]}$$

(Effect Trust)
$$\frac{E \vdash es \quad E \vdash M : \mathsf{Top} \quad E \vdash T}{E \vdash es + [\mathsf{trust}\ M{:}T]}$$

In an environment E, a well-formed type T is one where $E \vdash T$.

Rules for Types:

(Type Pair)(where $x \notin dom(E)$)
$$\frac{E \vdash T \quad E, x{:}T \vdash U}{E \vdash (x{:}T, U)}$$

(Type Sum)
$$\frac{E \vdash T \quad E \vdash U}{E \vdash T + U}$$

(Type Top)
$$\frac{}{E \vdash \mathsf{Top}}$$

(Type Un)
$$\frac{}{E \vdash \mathsf{Un}}$$

(Type Key)
$$\frac{E \vdash T}{E \vdash \mathsf{SharedKey}(T)}$$

(Type Key-Pair)
$$\frac{E \vdash T}{E \vdash \mathsf{KeyPair}(T)}$$

(Type Part)
$$\frac{E \vdash T}{E \vdash k\ \mathsf{Key}(T)}$$

(Type Challenge)
$$\frac{E \vdash es}{E \vdash \ell\ \mathsf{Challenge}\ es}$$

(Type Response)
$$\frac{E \vdash es}{E \vdash \ell\ \mathsf{Response}\ es}$$

In an environment E, a well-formed type T is a subtype of a well-formed type U whenever $E \vdash T <: U$, as defined in Section 3.1 and 3.3. We repeat the rules here for completeness:

Basic rules for subtyping:

$E \vdash T \Longrightarrow E \vdash T <: T$	(Sub Refl)
$E \vdash S <: T, E \vdash T <: U \Longrightarrow E \vdash S <: U$	(Sub Trans)
$E \vdash T \Longrightarrow E \vdash T <: \mathsf{Top}$	(Sub Top)

Congruence Rules for Subtyping:

(Sub Pair)(where $x \notin dom(E)$)
$$\frac{E \vdash T <: T' \quad E, x{:}T \vdash U <: U'}{E \vdash (x{:}T, U) <: (x{:}T', U')}$$

(Sub Sum)
$$\frac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash T + U <: T' + U'}$$

(Sub Enc Key)
$$\frac{E \vdash T' <: T}{E \vdash \mathsf{Encrypt\ Key}(T) <: \mathsf{Encrypt\ Key}(T')}$$

(Sub Dec Key)
$$\frac{E \vdash T <: T'}{E \vdash \mathsf{Decrypt\ Key}(T) <: \mathsf{Decrypt\ Key}(T')}$$

Subtyping Rules for Public Types:

$E \vdash (x{:}\mathsf{Un}, \mathsf{Un}) <: \mathsf{Un}$	(Public Pair)
$E \vdash \mathsf{Un} + \mathsf{Un} <: \mathsf{Un}$	(Public Sum)
$E \vdash \mathsf{SharedKey}(\mathsf{Un}) <: \mathsf{Un}$	(Public Shared Key)
$E \vdash k\ \mathsf{Key}(\mathsf{Un}) <: \mathsf{Un}$	(Public Key)
$E \vdash \mathsf{KeyPair}(\mathsf{Un}) <: \mathsf{Un}$	(Public Keypair)

Subtyping Rules for Tainted Types:

$E \vdash \mathsf{Un} <: (x{:}\mathsf{Un}, \mathsf{Un})$	(Tainted Pair)
$E \vdash \mathsf{Un} <: \mathsf{Un} + \mathsf{Un}$	(Tainted Sum)
$E \vdash \mathsf{Un} <: \mathsf{SharedKey}(\mathsf{Un})$	(Tainted Shared Key)
$E \vdash \mathsf{Un} <: k\ \mathsf{Key}(\mathsf{Un})$	(Tainted Key)
$E \vdash \mathsf{Un} <: \mathsf{KeyPair}(\mathsf{Un})$	(Tainted Keypair)

Subtyping Rules for Nonce Types:

$E \vdash \mathsf{Public\ Challenge}\ [] <: \mathsf{Un}$	(Public Challenge [])
$E \vdash fs \Longrightarrow E \vdash \mathsf{Public\ Response}\ fs <: \mathsf{Un}$	(Public Response)
$E \vdash \mathsf{Un} <: \mathsf{Public\ Challenge}\ []$	(Tainted Public Challenge [])
$E \vdash \mathsf{Un} <: \mathsf{Public\ Response}\ []$	(Tainted Public Response [])
$E \vdash es \Longrightarrow E \vdash \mathsf{Un} <: \mathsf{Private\ Challenge}\ es$	(Tainted Private Challenge)
$E \vdash es \Longrightarrow E \vdash \mathsf{Un} <: \mathsf{Private\ Response}\ es$	(Tainted Private Response)

In an environment E, a well-formed message M of type T is one where $E \vdash M : T$.

Type Rules for Messages:

(Msg x)
$$\frac{}{E', x{:}T, E'' \vdash x : T}$$

(Msg Subsum)
$$\frac{E \vdash M : S \quad E \vdash S <: U}{E \vdash M : U}$$

(Msg Pair)
$$\frac{E \vdash M : T \quad E \vdash N : U\{x \leftarrow M\}}{E \vdash (M, N) : (x{:}T, U)}$$

(Msg Inl)
$$\frac{E \vdash M : T \quad E \vdash U}{E \vdash \mathsf{inl}\ (M) : T + U}$$

(Msg Inr)
$$\frac{E \vdash T \quad E \vdash N : U}{E \vdash \mathsf{inr}\ (N) : T + U}$$

(Msg Symm)
$$\frac{E \vdash M : T \quad E \vdash N : \mathsf{SharedKey}(T)}{E \vdash \{M\}_N : \mathsf{Un}}$$

(Msg Part)
$$\frac{E \vdash M : \mathsf{KeyPair}(T)}{E \vdash k\ (M) : k\ \mathsf{Key}(T)}$$

(Msg Asymm)
$$\frac{E \vdash M : T \quad E \vdash N : \mathsf{Encrypt\ Key}(T)}{E \vdash \{\!|M|\!\}_N : \mathsf{Un}}$$

In an environment E, a well-formed process P with effect es is one where $E \vdash P : es$.

Basic Rules for Processes:

(Proc Output Un)
$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{Un}}{E \vdash \mathsf{out}\ M\ N : []}$$

(Proc Input Un) (where $y \notin dom(E) \cup fn(es)$)
$$\frac{E \vdash M : \mathsf{Un} \quad E, y{:}\mathsf{Un} \vdash P : es}{E \vdash \mathsf{inp}\ M\ (y{:}T); P : es}$$

(Proc Par)
$$\frac{E \vdash P : es \quad E \vdash Q : fs}{E \vdash P \mid Q : es + fs}$$

(Proc Repeat Input Un)
$$\frac{E \vdash M : \mathsf{Un} \quad E, y{:}\mathsf{Un} \vdash P : []}{E \vdash \mathsf{repeat\ inp}\ M\ (y{:}T); P : []}$$

(Proc Stop)
$$\frac{}{E \vdash \mathsf{stop} : []}$$

(Proc Res) (where $x \notin dom(E) \cup fn(es)$)
$$\frac{E, x{:}T \vdash P : es \quad T \text{ is Un or KeyPair}(U) \text{ or SharedKey}(T)}{E \vdash \mathsf{new}\ (x{:}T); P : es}$$

Rules for Processes Manipulating Products and Sums:

(Proc Split) (where $x, y \notin dom(E) \cup fn(es)$ and $x \neq y$)
$$\frac{E \vdash M : (x{:}T, U) \quad E, x{:}T, y{:}U \vdash P : es}{E \vdash \mathsf{split}\ M\ \mathsf{is}\ (x{:}T, y{:}U); P : es}$$

(Proc Match) (where $y \notin dom(E) \cup fn(es)$)
$$\frac{E \vdash M : (x{:}T, U) \quad E \vdash N : T \quad E, y{:}U\{x{\leftarrow}N\} \vdash P : es}{E \vdash \mathsf{match}\ M\ \mathsf{is}\ (N, y{:}U\{x{\leftarrow}N\}); P : es}$$

(Proc Case) (where $x \notin dom(E) \cup fn(es)$ and $y \notin dom(E) \cup fn(fs)$)
$$\frac{E \vdash M : T + U \quad E, x{:}T \vdash P : es \quad E, y{:}U \vdash Q : fs}{E \vdash \mathsf{case}\ M\ \mathsf{is\ inl}\ (x{:}T)\ P\ \mathsf{is\ inr}\ (y{:}U)\ Q : es \vee fs}$$

Rules for Cryptography:

(Proc Symm) (where $x \notin dom(E) \cup fn(es)$)
$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{SharedKey}(T) \quad E, x{:}T \vdash P : es}{E \vdash \mathsf{decrypt}\ M\ \mathsf{is}\ \{x{:}T\}_N; P : es}$$

(Proc Asymm) (where $x \notin dom(E) \cup fn(es)$)
$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{Decrypt\ Key}(T) \quad E, x{:}T \vdash P : es}{E \vdash \mathsf{decrypt}\ M\ \mathsf{is}\ \{\!|x{:}T|\!\}_{N^{-1}}; P : es}$$

Rules for Begins and Ends:

(Proc Begin)
$$\frac{E \vdash L : T \quad E \vdash P : es}{E \vdash \mathsf{begin}\ L; P : es - [\mathsf{end}\ L]}$$

(Proc End)
$$\frac{E \vdash L : T \quad E \vdash P : es}{E \vdash \mathsf{end}\ L; P : es + [\mathsf{end}\ L]}$$

Rules for Challenges and Responses:

(Proc Cast) (where $x \notin dom(E) \cup fn(fs)$)
$$\frac{E \vdash M : \ell\ \mathsf{Challenge}\ es_C \quad E, x{:}\ell\ \mathsf{Response}\ es_R \vdash P : fs}{E \vdash \mathsf{cast}\ M\ \mathsf{is}\ (x{:}\ell\ \mathsf{Response}\ es_R); P : es_C + es_R + fs}$$

(Proc Check)

$$\frac{E \vdash M : \ell\ \mathsf{Challenge}\ es_C \quad E \vdash N : \ell\ \mathsf{Response}\ es_R \quad E \vdash P : fs}{E \vdash \mathsf{check}\ M\ \mathsf{is}\ N; P : (fs - (es_C + es_R)) + [\mathsf{check}\ \ell\ M]}$$

(Proc Challenge) (where $x \notin dom(E) \cup fn(es - [\mathsf{check}\ \ell\ x])$)

$$\frac{E, x{:}\ell\ \mathsf{Challenge}\ fs \vdash P : es}{E \vdash \mathsf{new}\ (x{:}\ell\ \mathsf{Challenge}\ fs); P : es - [\mathsf{check}\ \ell\ x]}$$

Rules for Witness Testimony and Trusted-Casting:

(Proc Witness)

$$\frac{E \vdash M : T \quad E \vdash P : es + [\mathsf{trust}\,M{:}T, \ldots, \mathsf{trust}\,M{:}T]}{E \vdash \mathsf{witness}\,M{:}T; P : es}$$

(Proc Trust) (where $x \notin dom(E) \cup fn(es)$)

$$\frac{E \vdash M : \mathsf{Top} \quad E, x{:}T \vdash P : es}{E \vdash \mathsf{trust}\,M\,\mathsf{is}\,(x{:}T); P : es + [\mathsf{trust}\,M{:}T]}$$

The type-and-effect rules for processes $E \vdash P : es$ rely on some multiset algebra, which we define here for unordered sequences $[x_1, \ldots, x_n]$ for some grammar ranged over by x.

Multiset algebra $xs + xs'$**,** $xs \leq xs'$**,** $xs - xs'$**,** $x \in xs$**,** $xs \vee xs'$

$[x_1, \ldots, x_m] + [y_1, \ldots, y_n] \triangleq [x_1, \ldots, x_m, y_1, \ldots, y_n]$
$xs \leq xs'$ if and only if $xs + xs'' = xs'$ for some xs''
$xs - xs' \triangleq$ the smallest xs'' such that $xs \leq xs'' + xs'$
$x \in xs$ if and only if $[x] \leq xs$
$xs \vee xs' \triangleq$ the smallest xs'' such that $xs \leq xs''$ and $xs' \leq xs''$

References

[AB01] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In *Foundations of Software Science and Computation Structures (FoSSaCS 2001)*, volume 2030 of *Lectures Notes in Computer Science*, pages 25–41. Springer, 2001.

[AB02] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. In *29th ACM Symposium on Principles of Programming Languages (POPL'02)*, 2002. To appear.

[Aba99] M. Abadi. Secrecy by typing in security protocols. *Journal of the ACM*, 46(5):749–786, September 1999.

[AG99] M. Abadi and A.D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148:1–70, 1999.

[BAN89] M. Burrows, M. Abadi, and R.M. Needham. A logic of authentication. *Proceedings of the Royal Society of London A*, 426:233–271, 1989.

[Bol96] D. Bolignano. An approach to the formal verification of cryptographic protocols. In *Third ACM Conference on Computer and Communications Security*, pages 106–118, 1996.

[CG89] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[CJ97] J. Clark and J. Jacob. A survey of authentication protocol literature. Unpublished manuscript, 1997.

[DMP01] Nancy Durgin, John C. Mitchell, and Dusko Pavlovic. A compositional logic for protocol correctness. In *14th IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 2001.

[DY83] D. Dolev and A.C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT–29(2):198–208, 1983.

[GF00] J.D. Guttman and F.J. Thayer Fábrega. Authentication tests. In *2000 IEEE Computer Society Symposium on Research in Security and Privacy*, 2000.

[GJ01a] 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. An extended version appears as Microsoft Research Technical Report MSR–TR–2001–49, May 2001.

[GJ01b] A.D. Gordon and A. Jeffrey. The Cryptyc Project. At *http://cryptyc.cs.depaul.edu/*, 2001.

[GJ01c] A.D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. In *Mathematical Foundations of Programming Semantics 17*, Electronic Notes in Theoretical Computer Science. Elsevier, 2001. To appear. Pages 99–120 of the Preliminary Proceedings, BRICS Notes Series NS-01-2, BRICS, University of Aarhus, May 2001. An extended version appears as Microsoft Research Technical Report MSR–TR–2001–48, May 2001.

[HR98] N. Heintze and J.G. Riecke. The SLam calculus: Programming with secrecy and integrity. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 365–377, 1998.

[HS00] J. Heather and S. Schneider. Towards automatic verification of authentication protocols on an unbounded network. In *13th Computer Security Foundations Workshop*, pages 132–143. IEEE Computer Society Press, 2000.

[Low95] G. Lowe. A hierarchy of authentication specifications. In *10th Computer Security Foundations Workshop*, pages 31–43. IEEE Computer Society Press, 1995.

[Low96] G. Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using CSP and FDR. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems (TACAS'96)*, volume 1055 of *Lectures Notes in Computer Science*, pages 147–166. Springer, 1996.

[MCJ97] 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.

[Mil99] R. Milner. *Communicating and Mobile Systems: the π-Calculus*. Cambridge University Press, 1999.

[NS78] R.M. Needham and M.D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[ØP97] P. Ørbæk and J. Palsberg. Trust in the λ-calculus. *Journal of Functional Programming*, 3(2):75–85, 1997.

[Pau98] L.C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6:85–128, 1998.

[Sch98] S.A. Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9), September 1998.

[Son99] D.X. Song. Athena: a new efficient automatic checker for security protocol analysis. In *12th Computer Security Foundations Workshop*. IEEE Computer Society Press, 1999.

[STFW01] U. Shankar, K. Talwar, J.S. Foster, and D. Wagner. Detecting format string vulnerabilities with type qualifiers. In *10th USENIX Security Symposium*, 2001.

[THG98] F.J. Thayer Fábrega, J.C. Herzog, and J.D. Guttman. Strand spaces: Why is a security protocol correct? In *1998 IEEE Computer Society Symposium on Research in Security and Privacy*, 1998.

[WCS96] L. Wall, T. Christiansen, and R.L. Schwartz. *Programming Perl*. O'Reilly Associates, 2nd edition, 1996.

[WL93] T.Y.C. Woo and S.S. Lam. A semantic model for authentication protocols. In *IEEE Symposium on Security and Privacy*, pages 178–194, 1993.