Types and Effects for Asymmetric Cryptographic Protocols

Andrew D. Gordon
Microsoft Research

Alan Jeffrey[*]
DePaul University

Submission to *Journal of Computer Security* of August 13, 2002
Revision of June 9, 2003

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

In recent work [GJ03, GJ01], we propose a type-based methodology for checking authenticity properties of security protocols. 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 typechecker. 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 verification to the familiar edit/typecheck/debug cycle.

Still, our previous work 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 our 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.

[*]This material is based upon work supported by the National Science Foundation under Grant No. 0208549.

(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). Our previous system [GJ01] 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 the type of data may change over time.

(3) Our previous system supports a single 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.

In this model trusted agents have to communicate via an untrusted medium, where a malicious opponent can eavesdrop messages, forge messages, establish sessions with trusted agents, and hijack existing sessions. The trusted agents rely on a suite of cryptographic algorithms, which are assumed to provide perfect integrity and confidentiality properties.

1.1 Background

Many methodologies exist for verifying authenticity properties against the opponent model of Dolev and Yao [DY83]. Verification via typechecking 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 [SBP01, THG99] 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 [Low97]. 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	$A \to B:$	N
Event 1	B begins	"B sends A message M"
Message 2	$B \to A:$	$\{M,N\}_K$
Event $1'$	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.

Our earlier work can typecheck the robust safety of protocols based on secure channels [GJ03], and on insecure channels protected by symmetric cryptography [GJ01]. 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 [AB03, 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 [GJ01, AB03]. For example, to facilitate typechecking 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. Another work on types for asymmetric cryptography, though not for authenticity, is Cervesato's typed multiset rewriting [Cer01].

Abadi and Blanchet establish a logic programming formulation of one of their type systems for secrecy properties, and hence obtain an automatic technique for establishing secrecy [AB02]. In recent work, Blanchet extends this logic programming technique to also prove authenticity properties expressed as correspondence assertions [Bla02].

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 secret and untainted, or public and tainted. For example, consider the message:

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

(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, GJ01] 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 come 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, the present 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:

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

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}:

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

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.

On the whole, symmetric key cryptographic protocols 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. Cryptographic 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 they are 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 Thayer [GT02] propose authentication tests for analysing nonce usage. Their incoming tests apply to POSH and SOSH nonces, and their outgoing tests apply to SOPH and SOSH nonces. Our previous work [GJ01] deals only with POSH nonces.

1.3 Remainder of this Paper

Section 2 reviews our 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. Appendix A contains example protocols. Appendix B defines the operational semantics of our calculus. Appendix C includes proofs of correctness for our type system; a technical report [GJ02a] includes omitted details.

An abridged version of this paper appears in a conference proceedings [GJ02b].

Acknowledgements Thanks to Martín Abadi, Adriana Compagnoni, and Dusko Pavlovic for discussions related to this paper. The anonymous referees for *Journal of Computer Security* made invaluable suggestions, especially as regards the introduction of Section 3.

2 Authenticity Properties in Spi (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 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
$\{\!\!\mid M \mid\!\!\}_N$	asymmetric encryption
$k\ (M)$	key-pair component (where k either $\mathsf{Encrypt}$ or $\mathsf{Decrypt}$)

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.)

We write $fn(M)$ for the set of free names of the message M. We write $M\{x{\leftarrow}N\}$ for the outcome of a capture-avoiding substitution of the message N for each free occurrence of the name x in the message M. Next, we give the syntax of processes. Each bound name has a type annotation, written T or U. We postpone the syntax of types to Section 3.

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 decrypt (x bound in P)
$\mathsf{decrypt}\ M\ \mathsf{is}\ \{\!|x{:}T|\!\}_{N^{-1}};P$	asymmetric decrypt (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 typechecking 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.

We write $fn(P)$ for the set of free names of the process P. 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 $\mathsf{out}\ M\ N$ and $\mathsf{inp}\ M\ (x{:}T);P$ are output and input, respectively, along an asynchronous, unordered channel M. If an output $\mathsf{out}\ x\ N$ runs in parallel with an input $\mathsf{inp}\ x\ (y);P$, the two can interact to leave the residual process $P\{y \leftarrow N\}$.
- Process $\mathsf{repeat\ inp}\ M\ (x{:}T);P$ 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 $\mathsf{repeat\ inp}\ M\ (x{:}T);P$.
- A process $\mathsf{split}\ M\ \mathsf{is}\ (x{:}T, y{:}U);P$ 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 $\mathsf{match}\ M\ \mathsf{is}\ (N, y{:}U);P$ 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.

  The match construct is used in cases when a protocol expects a field to have a particular value (for example, A may expect to see her own name in a message) whereas the split construct is used in cases where the protocol does not know the value of a field (for example, B may be prepared to communicate with any principal with appropriate credentials, not just a particular principal A).
- A process $\mathsf{case}\ M\ \mathsf{is\ inl}\ (x{:}T)\ P\ \mathsf{is\ inr}\ (y{:}U)\ Q$ checks the tagged union M. If M is $\mathsf{inl}\ (L)$, the process behaves as $P\{x \leftarrow L\}$. If M is $\mathsf{inr}\ (N)$ it behaves as $Q\{y \leftarrow N\}$. Otherwise, it deadlocks.
- A process $\mathsf{decrypt}\ M\ \mathsf{is}\ \{x{:}T\}_N;P$ 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 $\mathsf{decrypt}\ M\ \mathsf{is}\ \{\!|x{:}T|\!\}_{N^{-1}};P$ decrypts M using asymmetric key N. If M is $\{\!|L|\!\}_{\mathsf{Encrypt}\ (K)}$ and N is $\mathsf{Decrypt}\ (K)$, then the process behaves as $P\{x \leftarrow L\}$. Otherwise, it deadlocks.
- A process $\mathsf{check}\ M\ \mathsf{is}\ N;P$ checks the messages M and N are the same name before executing P. If the equality test fails, the process deadlocks.
- A process $\mathsf{begin}\ L;P$ autonomously asserts an begin-event labelled L, and then behaves as P.

- An process $\mathsf{end}\ L;P$ autonomously asserts an end-event labelled L, and then behaves as P.
- A process $\mathsf{new}\ (x{:}T);P$ 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 $\mathsf{end}\ x;$.
- 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.

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.

```
$Sender(net, private_A, public_B) \triangleq$
  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 $net$ $(challenge_{B1}, \{msg, challenge_{B2}\}_{key_{AB}})$;

$Receiver(net, public_A, private_B) \triangleq$
  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 $B$";

$System(net) \triangleq$
  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 $(pair_B))$
  )
```

Figure 1: An example protocol with correspondence assertions

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 $2'$	A ends	“B received K_{AB} from A”
Event 3	A begins	“A sends M to B”
Message 3	$A \rightarrow B:$	$N_{B1}, \{M, N_{B2}\}_{K_{AB}}$
Event $1'$	B ends	“A generates K_{AB} for B”
Event $3'$	B ends	“A sends M to B”

The process $\mathit{Sender}(\mathit{net}, \mathit{private}_A, \mathit{public}_B)$ defined in Figure 1 is the sender A, parameterized on *net* (the name of the public channel), $\mathit{private}_A$ (A’s private key) and public_B (B’s public key). It generates a fresh session key key_{AB} and a nonce challenge $\mathit{challenge}_A$, and then sends the ciphertext $\{\!|A, \mathit{key}_{AB}, \mathit{challenge}_A|\!\}_{\mathit{public}_B}$ on the public *net* channel. It waits for the acknowledgement message, decrypts it to get the contents $(B, \mathit{key}_{AB}, \mathit{response}_A, \mathit{challenge}_{B1}, \mathit{challenge}_{B2})$ and then checks that the outgoing nonce $\mathit{challenge}_A$ was the same as the incoming nonce $\mathit{response}_A$. If it is, then it responds by sending the response to the nonce challenge $\mathit{challenge}_{B1}$, together with the ciphertext $\{\mathit{msg}, \mathit{challenge}_{B2}\}_{\mathit{key}_{AB}}$.

The process $\mathit{Receiver}(\mathit{net}, \mathit{public}_A, \mathit{private}_B)$ defined in Figure 1 is the receiver B, parameterized on *net* and matching keys public_A and $\mathit{private}_B$. It repeatedly receives a message on the public *net* channel and decrypts it with $\mathit{private}_B$ to get the plaintext of the form $A, \mathit{key}_{AB}, \mathit{challenge}_A$. It responds by generating two nonce challenges $\mathit{challenge}_{B1}$ (which is used to validate key_{AB}) and $\mathit{challenge}_{B2}$ (which is used to validate the message sent encrypted with the key_{AB}) and sending the ciphertext $\{\!|B, \mathit{key}_{AB}, \mathit{challenge}_A, \mathit{challenge}_{B1}|\!\}_{\mathit{public}_A}$ together with the nonce $\mathit{challenge}_{B2}$. It receives back the response $\mathit{response}_{B1}$ together with a ciphertext, and checks that nonce $\mathit{challenge}_{B1}$ is the same as $\mathit{response}_{B1}$ (and so the session key key_{AB} can be trusted). It then decrypts the ciphertext to get the plaintext *msg* together with the response $\mathit{response}_{B2}$, and checks that $\mathit{challenge}_{B2}$ is the same as $\mathit{response}_{B2}$ (and so the message *msg* can be trusted).

Figure 1 is a spi-calculus version of the protocol, making use of syntax sugar defined in Appendix A.4. The top-level process, $\mathit{System}(\mathit{net})$ 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 $\mathit{System}(\mathit{net})$ is robustly safe.

3 Typing Asymmetric Cryptographic Protocols

We now introduce the type-and-effect system used in statically verifying cryptographic communication protocols. The key ideas are:

(1) In general, an effect provides an upper bound on the events of interest that a process may engage in. Here, the events of interest are unmatched end-events, and trust-events tracking the dynamic trust placed in messages.

(2) Just as effects characterize processes, types characterize messages. The types of messages are related to the effects of processes because the type of a message may carry a latent effect that justifies the recipient in performing an unmatched event because it has already been matched in the sender.

(3) Principals that receive a nonce and retransmit it in transformed form are regarded as performing a cast on the type of the nonce. Thus, one latent effect may be received with the nonce and absorbed by the recipient, while another is associated with the nonce and returned to its originator.

(4) The subtyping relation allows the type system to recognize properties of types, especially to distinguish tainted and untainted messages, and public and private messages.

Section 3.1 introduces the type and effect system. Section 3.2 describes how we type messages. Section 3.3 explains the subtyping relation. Section 3.4 explains how we ascribe effects to processes. In Section 3.5 we explain how to type the assertions in the example of the previous section.

3.1 Environments and Judgments

The type and effect system is given as a series of judgments $E \vdash \mathcal{J}$, for example the judgment $E \vdash T$ can be read as 'in environment E we have that T is a type'.

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*

Judgments are given in an *environment* that assigns types to the variables in scope. An environment, E, takes the form $x_1{:}T_1, \ldots, x_n{:}T_n$, and we write $dom(E)$ for $\{x_1, \ldots, x_n\}$.

Environments:

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

An environment, E, is well-formed environment, written $E \vdash \diamond$, when the defined variables are distinct, and for each entry $x{:}T$, the type T is well-formed in E, as explained below. As in our previous type system [GJ01], the type of a variable may contain occurrences of the variable itself.

Rule for Environments:

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

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

3.2 Types for Messages

We give the syntax of types and explain when a message M has type T, written informally $M : T$. The challenge and response types for nonces are deferred to Section 3.4, where they are discussed in the context of effects for processes.

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
$k\ \mathsf{Key}(T)$	encryption or decryption part (where k either $\mathsf{Encrypt}$ or $\mathsf{Decrypt}$)

The free names $fn(T)$ of a type T are defined in the usual way, where the only binder is x being bound in U in the type $(x{:}T,U)$. (The free names in types are introduced by challenge and response types, defined in Section 3.4.) We write $T\{x{\leftarrow}M\}$ for the outcome of a capture-avoiding substitution of the message M for each free occurrence of the name x in the type T.

Next, we informally explain the values described by each type. We say informally that a type is *public* if messages of the type may flow to the opponent. Dually, we say a type is *tainted* if messages from the opponent may flow into the type. Later, we formalize these notions using the subtype relation.

- The type $(x{:}T,U)$ describes a pair (M,N) where $M : T$ and $N : U$. The scope of the variable x consists of the type U. Type $(x{:}T,U)$ is public just if T and U are public, and tainted just if T and U are tainted.
- The type $T+U$ describes a tagged message $\mathsf{inl}\ (M)$ where $M : T$ or $\mathsf{inr}\ (N)$ where $N : U$. Type $T+U$ is public just if T and U are public, and tainted just if T and U are tainted.

- The type Un describes messages that may flow to or from the opponent, which we model as an arbitrary process of the calculus. The type Un is both public and tainted.
- The type Top describes all well-typed messages; it is tainted but not public.
- The type $\mathsf{SharedKey}(T)$ describes symmetric keys for encrypting messages of type T; it is public or tainted just if T is both public and tainted.
- The type $\mathsf{KeyPair}(T)$ describes asymmetric key-pairs for encrypting or signing messages of type T; it is public or tainted just if T is both public and tainted. The key-pair can be used for public-key cryptography just if T is tainted, and for digital signatures just if T is public.
- The type $\mathsf{Encrypt\ Key}(T)$ describes an encryption or signing key for messages of type T; it is public just if T is tainted, and it is tainted just if T is public.
- The type $\mathsf{Decrypt\ Key}(T)$ describes a decryption or verification key for messages of type T; it is public just if T is public, and it is tainted just if T it tainted.

In an environment E, a type T is well-formed, written $E \vdash T$, if $dom(E)$ includes all the variables free in T. It may be a little surprising that this rule does not require each term that occurs in a type to be well-formed, but just that all its free variables be in scope. In fact, the only terms that can occur within a type are the labels of begin- or end-events in an effect occurring within the type. We do not need to regulate or check the types of these labels, as they are simply identifiers for events.

Rule for Types:

(Type)
$$\frac{fn(T) \subseteq dom(E)}{E \vdash T}$$

The formal message typing judgment takes the form $E \vdash M : T$, read 'in environment E, message M has type T'.

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.

Typing Rules for Messages:

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

(Msg Subsum)
$$\frac{E \vdash M : T \quad E \vdash T <: T'}{E \vdash M : T'}$$

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

(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)

$$\frac{\begin{array}{l} E \vdash M : T \\ E \vdash N : \mathsf{Encrypt\ Key}(T) \end{array}}{E \vdash \{\!|M|\!\}_N : \mathsf{Un}}$$

The type-rules are all syntax-directed, and so it is not hard to implement a top-down typechecker for this type system.

3.3 The Subtyping Relation

The *subtyping* relation $E \vdash T <: T'$ means that messages of type T can be used in place of a message of type T'. The environment E tracks the names in scope, and sometimes is omitted in informal discussion.

The interaction of subtyping and dependent types can be quite subtle; our treatment is based on that of Aspinall and Compagnoni [AC01], although our setting is much simpler, due to the absence of higher-order types.

Earlier we gave an informal definition of public and tainted types. Formally, a type's relationship to the type Un of data known to the opponent determines whether it can flow to or from the opponent. We define a type T to be *public* if and only if $T <: \mathsf{Un}$. We define a type T to be *tainted* if and only if $\mathsf{Un} <: T$.

The following tables of rules define the subtyping relation. Note that subtyping is a preorder on types, not a partial order, since all of the types which are both tainted and public are collapsed together, for example $\mathsf{Un} + \mathsf{Un} <: \mathsf{Un} <: \mathsf{Un} + \mathsf{Un}$. 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; symmetric keys $\mathsf{SharedKey}(T)$ and key pairs $\mathsf{KeyPair}(T)$ are invariant. These variances correspond to the usual subtyping rules for references or channels: encryption is a writing operation, and so is contravariant, while decryption is a reading operation, and so is covariant.

Congruence Rules for Subtyping:

(Sub Pair)(where $x \notin dom(E)$)

$$\frac{E \vdash T <: T' \quad E, x{:}T \vdash U <: U' \quad E, x{:}T' \vdash U'}{E \vdash (x{:}T, U) <: (x{:}T', U')}$$

(Sub Sum)

$$\frac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash T + U <: T' + U'}$$

(Sub Key Invar)

$$\frac{E \vdash T <: T' \quad E \vdash T' <: T}{E \vdash \mathsf{SharedKey}(T) <: \mathsf{SharedKey}(T')}$$

(Sub Key Pair Invar)

$$\frac{E \vdash T <: T' \quad E \vdash T' <: T}{E \vdash \mathsf{KeyPair}(T) <: \mathsf{KeyPair}(T')}$$

(Sub Enc Contra)

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

(Sub Dec Co)

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

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)
$E \vdash \mathsf{KeyPair}(\mathsf{Un}) <: \mathsf{Un}$	(Public Keypair)
$E \vdash k\ \mathsf{Key}(\mathsf{Un}) <: \mathsf{Un}$	(Public Key)

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)
$E \vdash \mathsf{Un} <: \mathsf{KeyPair}(\mathsf{Un})$	(Tainted Keypair)
$E \vdash \mathsf{Un} <: k\ \mathsf{Key}(\mathsf{Un})$	(Tainted Key)

We end this section by discussing the two dual applications of key-pairs. We have the following equivalences:

Proposition 1 *Suppose that $E \vdash T$ and $E \vdash \diamond$. Then:*

(1) *T is tainted if and only if $\mathsf{Encrypt\ Key}(T)$ is public if and only if $\mathsf{Decrypt\ Key}(T)$ is tainted.*

(2) *T is public if and only if $\mathsf{Encrypt\ Key}(T)$ is tainted if and only if $\mathsf{Decrypt\ Key}(T)$ is public.*

Proof See Appendix C.4. □

The first case represents public-key applications, where the payload type T is tainted, and the encryption key is public, so that anyone, including the opponent, can encrypt messages. The second case represents digital signature applications, where the payload type T is public, and the decryption key is public, so that anyone, including the opponent, can check signatures.

If we attempt to use the same keypair of type $\mathsf{KeyPair}(T)$ for both applications, T is both public and tainted, and hence equivalent to Un. This matches the common engineering practice that keys used for both public-key and digital signature applications are not to be trusted.

3.4 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 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; and
- $\mathsf{trust}\, M{:}T$, a trust effect used to gain the trust information that data M really has type T.

Overall, the goal when typechecking a protocol is to assign it the empty effect, for then it has no unmatched end-events, and therefore is safe. This section explains the intuitions behind the rules for assigning effects to processes.

The effect determined by our system is only a static approximation to the actual unmatched end-events performed by a process. As with most properties addressed by type systems, a completely accurate determination is undecidable in general. For example, the following protocol (which generates a session key then just discards it) is robustly safe, but is rejected by our system:

$$\mathsf{new}\ (k);\mathsf{inp}\ \mathit{net}\ (c);\mathsf{decrypt}\ c\ \mathsf{is}\ \{x\}_k;\mathsf{end}\ y;$$

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 ($\mathsf{Private}$ nonces). We let ℓ be either Public or $\mathsf{Private}$. We typecheck the above four steps as follows:

(1) A creates the nonce N as having type ℓ $\mathsf{Challenge}$ es, where es is an effect, and sends it to B.

(2) B casts the nonce to a new type ℓ $\mathsf{Response}$ 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 $\mathsf{check}\ N\ \mathsf{is}\ N';$ to check 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 $\mathsf{check}\ \ell\ N$, which is a pre-condition on each process $\mathsf{check}\ N\ \mathsf{is}\ N';$. This pre-condition can only be achieved 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 [GJ01], and is sufficient to typecheck many symmetric key protocols. Asymmetric key protocols, however, often 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 introduce 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 $\mathsf{witness}\, M{:}T;$ which justifies a *trust effect* $\mathsf{trust}\, M{:}T$. A can then use the nonce mechanism described above to communicate this trust effect to B.

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

In this fashion, type information can be exchanged between honest agents, using the same mechanism as authenticity information. This is done without transmitting any extra type information at runtime, and so the types are only used in the static analysis of the protocol, and play no role at runtime.

Effects:

$e, f ::=$	atomic effect
$\mathsf{end}\ L$	end-event labelled with message L
$\mathsf{check}\ \ell\ N$	name-check for a nonce N
$\mathsf{trust}\, M{:}T$	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 $fn(es)$ of an effect es 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 an environment E, an effect es is well-formed, written $E \vdash es$, if $dom(E)$ includes all the names free in es. As we already mentioned in our discussion of the rule (Type), our type system does not regulate the types of the terms serving as event levels within effects. These terms are simply event identifiers. Their types do not matter.

Rule for Effects:

(Effect)

$$\frac{fn(es) \subseteq dom(E)}{E \vdash es}$$

We extend the grammar of types to include nonce types. These come in two varieties: Public nonces (for SOPH and POSH nonce handshakes, which are public at some points in their lifetime) and $\mathsf{Private}$ nonces (for SOSH nonce handshakes, which are never public).

- POSH nonces are sent out with tainted public type $\mathsf{Public\ Challenge}\ [\,]$, and return with untainted public type $\mathsf{Public\ Response}\ es$.
- SOPH nonces are sent out with untainted secret type $\mathsf{Public\ Challenge}\ es$ (with $es \neq [\,]$), and return with tainted public type $\mathsf{Public\ Response}\ [\,]$.
- SOSH nonces are send out with tainted secret type $\mathsf{Private\ Challenge}\ es$, and return with tainted secret type $\mathsf{Private\ Response}\ fs$.

In addition, we introduce challenge-response types $\ell\ \mathsf{CR}\ es\ fs$, which can act as both challenges and responses. These are only required for technical reasons in the proof of correctness, and are not intended for use in user code. The $\ell\ \mathsf{CR}\ es\ fs$ types are used because nonces change type over time—a nonce of type $\ell\ \mathsf{Challenge}\ es$ may be cast to the type $\ell\ \mathsf{Response}\ fs$; in fact, we downcast to the new type $\ell\ \mathsf{CR}\ es\ fs$, which is a subtype of both the original type $\ell\ \mathsf{Challenge}\ es$ and the new type $\ell\ \mathsf{Response}\ fs$.

Nonce Types:

$T,U ::=$	type
$\quad\ldots$	as in Section 3.2
$\quad\ell\ \mathsf{Challenge}\ es$	nonce challenge type
$\quad\ell\ \mathsf{Response}\ es$	nonce response type
$\quad\ell\ \mathsf{CR}\ es\ fs$	challenge-response type
$\ell ::=$	privacy
$\quad\mathsf{Public}$	public
$\quad\mathsf{Private}$	private

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)
$E \vdash es' + fs', es \leq es', fs \leq fs'$ $\Longrightarrow E \vdash \ell\ \mathsf{CR}\ es'\ fs' <: \ell\ \mathsf{CR}\ es\ fs$	(Sub CR)
$E \vdash \ell\ \mathsf{CR}\ es\ fs \Longrightarrow E \vdash \ell\ \mathsf{CR}\ es\ fs <: \ell\ \mathsf{Challenge}\ es$	(Sub CR C)
$E \vdash \ell\ \mathsf{CR}\ es\ fs \Longrightarrow E \vdash \ell\ \mathsf{CR}\ es\ fs <: \ell\ \mathsf{Response}\ fs$	(Sub CR R)

We extend the grammar of processes to include nonce manipulation:

Processes Manipulating Nonces:

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

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

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

These additional constructs add no expressive power at runtime; they are simply annotations to help the typechecker. Therefore, it is convenient and harmless to forbid the opponent from using these additional constructs. Similarly, the check construct adds no expressive power as it may be mimicked by the match construct. We revise our definition of an opponent as follows.

Revised Formulation of Opponent:

A process is *unprivileged* if and only if it contains no checks, casts, witnesses or trusts. An *opponent* O is an assertion-free unprivileged untyped process O.

We can now give rules which calculate the effect of a process. Most of the rules are the same as [GJ01], so we only discuss the rules for asymmetric cryptography, nonce challenges, and dynamic trust here.

The rule for asymmetric decryption is similar to the one for symmetric decryption in [GJ01]: if M is a plaintext of type T and N is a decrypt key of type $\mathsf{Decrypt\ Key}(T)$ then we can decrypt a ciphertext of type Un to reveal the plaintext of type T:

Rule for Asymmetric Cryptography:

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

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

The rules for nonce types are similar to the rules from [GJ01], except that they support SOPH and POSH nonces as well as POSH nonces:

Rules for Challenges and Responses:

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

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

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

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

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 $x \notin dom(E) \cup fn(es)$)

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

The remaining rules are the same as in [GJ01], so we repeat them without comment.

Basic Rules for Processes:

(Proc Subsum)

$$\frac{E \vdash P : es \quad E \vdash fs}{E \vdash P : es + fs}$$

(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 $y \notin dom(E) \cup fn(es)$)

$$\frac{E \vdash M : \mathsf{Un} \quad E, y{:}\mathsf{Un} \vdash P : es}{E \vdash \mathsf{inp}\ M\ (y{:}\mathsf{Un}); P : es}$$

(Proc Repeat Input Un) (where $y \notin dom(E)$)

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

(Proc Par)

$$\frac{E \vdash P : es \quad E \vdash Q : fs}{E \vdash P \mid Q : es + fs}$$

(Proc Stop)

$$\frac{}{E \vdash \mathsf{stop} : [\,]}$$

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

$$\frac{\begin{array}{l} E, x{:}T \vdash P : es \quad E \vdash T \\ T \text{ is } \mathsf{Un} \text{ or } \mathsf{KeyPair}(U) \text{ or } \mathsf{SharedKey}(U) \end{array}}{E \vdash \mathsf{new}\ (x{:}T); P : es}$$

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 $x \notin dom(E) \cup fn(es)$)

$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{SharedKey}(T) \quad E, x{:}T \vdash P : es}{E \vdash \mathsf{decrypt}\ M\ \mathsf{is}\ \{x{:}T\}_N; P : es}$$

Rules for Begins and Ends:

(Proc Begin)

$$\frac{E \vdash L : \mathsf{Top} \quad E \vdash P : es}{E \vdash \mathsf{begin}\ L; P : es - [\mathsf{end}\ L]}$$

(Proc End)

$$\frac{E \vdash L : \mathsf{Top} \quad E \vdash P : es}{E \vdash \mathsf{end}\ L; P : es + [\mathsf{end}\ L]}$$

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 $x \notin dom(E) \cup fn(es)$)

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

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 $xs' \leq xs''$

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.*

Proof Given in Appendix C.8. □

3.5 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.

The protocol uses two nonce handshakes to agree on a session key between A and B, and then an additional nonce handshake to communicate the message M from A to

```
$Sender(net : \mathsf{Un}, private_A : \mathsf{Decrypt}\ \mathsf{K}_A(A), public_B : \mathsf{Encrypt}\ \mathsf{K}_B(B)) \triangleq$
    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 $net$ $(response_{B1}, \{msg, response_{B2}\}_{key_{AB}})$;
```

Figure 2: Proof that the sender is robustly safe

```
$\mathit{Receiver}(\mathit{net} : \mathsf{Un}, \mathit{public}_A : \mathsf{Encrypt}\ \mathsf{K}_A(A), \mathit{private}_B : \mathsf{Decrypt}\ \mathsf{K}_B(B)) \triangleq$
  repeat
    inp $\mathit{net}\ (\mathit{ctext}_1 : \mathsf{Un})$;
    decrypt $\mathit{ctext}_1$ is $\{\!|A, \mathit{untrusted} : \mathsf{Top}, \mathit{challenge}_A : \mathsf{C}_A(A, B, \mathit{key}_{AB})|\!\}_{\mathit{private}_B^{-1}}$;
    // Effect: []
    new $(\mathit{challenge}_{B1} : \mathsf{C}_{B1}(A, B, \mathit{key}_{AB}))$;
    // Effect: [check Public $\mathit{challenge}_{B1}$]
    new $(\mathit{challenge}_{B2} : \mathsf{C}_{B2})$;
    // Effect: [check Public $\mathit{challenge}_{B1}$, check Public $\mathit{challenge}_{B2}$]
    begin “$B$ received $\mathit{untrusted}$ from $A$”;
    // Effect: [end “$B$ received $\mathit{untrusted}$ from $A$”,
                check Public $\mathit{challenge}_{B1}$, check Public $\mathit{challenge}_{B2}$]
    cast $\mathit{challenge}_A$ is $(\mathit{response}_A : \mathsf{R}_A)$;
    out $\mathit{net}\ \{\!|B, \mathit{untrusted}, \mathit{challenge}_A, \mathit{challenge}_{B1}|\!\}_{\mathit{public}_A}, \mathit{challenge}_{B2}$;
    inp $\mathit{net}\ (\mathit{response}_{B1} : \mathsf{R}_{B1}, \mathit{ctext}_3 : \mathsf{Un})$;
    // Effect: [check Public $\mathit{challenge}_{B1}$, check Public $\mathit{challenge}_{B2}$]
    check $\mathit{challenge}_{B1}$ is $\mathit{response}_{B1}$;
    // Effect: [end “$A$ generates $\mathit{untrusted}$ for $B$”,
                trust $\mathit{untrusted}$:$\mathsf{K}_{AB}(A, B)$, check Public $\mathit{challenge}_{B2}$]
    end “$A$ generates $\mathit{untrusted}$ for $B$”;
    // Effect: [trust $\mathit{untrusted}$:$\mathsf{K}_{AB}(A, B)$, check Public $\mathit{challenge}_{B2}$]
    trust $\mathit{untrusted}$ is $(\mathit{key}_{AB} : \mathsf{K}_{AB}(A, B))$;
    decrypt $\mathit{ctext}_3$ is $\{\mathit{msg} : \mathsf{Payload}, \mathit{response}_{B2} : \mathsf{R}_{B2}(A, B, \mathit{msg})\}_{\mathit{key}_{AB}}$;
    // Effect: [check Public $\mathit{challenge}_{B2}$]
    check $\mathit{challenge}_{B2}$ is $\mathit{response}_{B2}$;
    // Effect: [end “$A$ sends $\mathit{msg}$ to $B$”]
    end “$A$ sends $\mathit{msg}$ to $B$”;
```

Figure 3: Proof that the receiver is robustly safe

B:

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 2′	A ends	"B received K_{AB} from A"
Event 3	A begins	"A sends M to B"
Message 3	$A \rightarrow B$:	$N_{B1}, \{M, N_{B2}\}_{K_{AB}}$
Event 1′	B ends	"A generates K_{AB} for B"
Event 3′	B ends	"A sends M to B"

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:

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

The type for K_A is:

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

The type for K_B is:

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

In these messages, the session key k is communicated at an untrusted type Top; only after Message 3 can the participants can agree on the trusted type for the key.

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 Figures 2 and 3, 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.

Gordon and Pucella [GP02] apply the formalism of this paper to the design of some application-level security abstractions for XML web services. They design and implement mechanisms for authenticity and confidentiality for a web service, and formalize the abstract guarantees within a typed object calculus. By giving a semantics for this calculus within the typed spi-calculus of this paper, they reduce the correctness of the design to establishing robust safety for certain processes. By showing that well-typed object calculus programs map to well-typed spi-calculus processes, robust safety follows by Theorem 1 (Robust Safety) of this paper.

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 typecheck, 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 $\mathsf{Principal}$ for principal names.

We show in Section A.4 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	$B \rightarrow A:$	N_B
Event 1	A begins	"A sending M to B"
Message 2	$A \rightarrow B:$	$\{\!\vert A, K_A \vert\!\}_{K_{CA}^{-1}}, \{\!\vert M, B, N_B \vert\!\}_{K_A^{-1}}$
Event $1'$	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 $\mathsf{Payload}$):

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

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 $\mathsf{Decrypt}\ \mathsf{K}_A(a)$ is public (this depends on the $\mathsf{Payload}$ type being public).
- The plaintext of type K_{CA} is public, so $\mathsf{Decrypt}\ \mathsf{K}_{CA}$ 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	$S \to A:$	$\{\!\vert B, K_B \vert\!\}_{K_S^{-1}}$
Event 1	A begins	"A contacting B"
Message 3	$A \to B:$	$\{\!\vert msg_3(A, N_A) \vert\!\}_{K_B}$
Event 2	B begins	"B contacted by A"
Message 4	$B \to S:$	B, A
Message 5	$S \to B:$	$\{\!\vert A, K_A \vert\!\}_{K_S^{-1}}$
Message 6	$B \to A:$	$\{\!\vert msg_6(B, N_A, N_B) \vert\!\}_{K_A}$
Event 2′	A ends	"B contacted by A"
Message 7	$A \to B:$	$\{\!\vert msg_7(N_B) \vert\!\}_{K_B}$
Event 1′	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:

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

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

We omit the details, but it is routine to verify that the protocol can be typed-checked with empty effect, 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.

A.4 Abbreviations Used in Examples

We shall now show that the abbreviations we used in our examples can be defined in our type system. We made use of types for dependent records and tagged unions.

Syntax Sugar for Use in Types:

$T,U ::=$	type
$\ldots$	as in Sections 3.2 and 3.4
$(x_1{:}T_1,\ldots,x_n{:}T_n)$	dependent record
$(\ell_1(T_1) \mid \cdots \mid \ell_n(T_n))$	tagged union

We allowed the construction of messages of record or tagged union type:

Syntax Sugar for Use in Messages:

$L,M,N ::=$	message
...	as in Section 2.1
$(M_1,\ldots,M_n)$	record
$\ell_i(M)$	tagged union
"$a_i \ldots a_n$"	string

In processes, we can make use of pattern-matching:

Syntax Sugar for Use in Processes:

$O,P,Q,R ::=$	process
...	as in Sections 2.1 and 3.4
$\mathsf{match}\ M\ \mathsf{is}\ X;P$	pattern match
$\mathsf{out}\ M\ N;P$	output with residual
$\mathsf{inp}\ M\ (X);P$	pattern matching input
$\mathsf{decrypt}\ M\ \mathsf{is}\ \{X\}_N;P$	pattern matching symmetric decrypt
$\mathsf{decrypt}\ M\ \mathsf{is}\ \{\!\vert X \vert\!\}_{N^{-1}};P$	pattern matching asymmetric decrypt

where X ranges over a grammar of patterns:

Patterns:

$X,Y,Z ::=$	patterns
$x{:}T$	variable
M	constant
$(Y_1,\ldots,Y_{n-1},X_n)$	tuple
$\ell_i(X)$	tagged union
$\{X\}_M$	symmetric ciphertext
$\{\!\vert X \vert\!\}_{M^{-1}}$	asymmetric ciphertext

We will now give definitions for each of these extensions, beginning with types.

Abbreviations for Types:

$$(x_1{:}T_1,\ldots,x_n{:}T_n) \triangleq (x_1{:}T_1,(x_2{:}T_2,(\ldots(x_{n-1}{:}T_{n-1},T_n)\ldots)))$$
$$(\ell_1(T_1) \mid \cdots \mid \ell_n(T_n)) \triangleq (T_1+(T_2+(\cdots(T_{n-1}+T_n)\ldots)))$$

The translations of messages are similarly straightforward.

Abbreviations for Messages:

$(M_1,\ldots,M_n) \triangleq (M_1,(M_2,(\ldots(M_{n-1},M_n)\ldots)))$
$\ell_i(M) \triangleq \mathsf{in}_{i,n}\,(M)$
$\mathsf{in}_{1,1}\,(M) \triangleq M$
$\mathsf{in}_{1,n+1}\,(M) \triangleq \mathsf{inl}\ (M)$
$\mathsf{in}_{i+1,n+1}\,(M) \triangleq \mathsf{inr}\ (\mathsf{in}_{i,n}\,(M))$
“$a_1\ldots a_n$” $\triangleq (a_1,\ldots,a_n)$

We write $\mathsf{out}\ x\ (M);P$ as a simple shorthand for $\mathsf{out}\ x\ M \mid P$:

Abbreviation $\mathsf{out}\ M\ N;P$**:**

$\mathsf{out}\ M\ N;P \triangleq (\mathsf{out}\ M\ N) \mid P$

We define pattern-matching as:

Abbreviations for Pattern Matching:

$\mathsf{inp}\ M\ (X);P \triangleq \mathsf{inp}\ M\ (x);\mathsf{match}\ x\ \mathsf{is}\ X;P$
$\mathsf{decrypt}\ M\ \mathsf{is}\ \{X\}_N;P \triangleq \mathsf{decrypt}\ M\ \mathsf{is}\ \{x\}_N;\mathsf{match}\ x\ \mathsf{is}\ X;P$
$\mathsf{decrypt}\ M\ \mathsf{is}\ \{\!|X|\!\}_{N^{-1}};P \triangleq \mathsf{decrypt}\ M\ \mathsf{is}\ \{\!|x|\!\}_{N^{-1}};\mathsf{match}\ x\ \mathsf{is}\ X;P$
$\mathsf{match}\ M\ \mathsf{is}\ x{:}T;P \triangleq P\{x\leftarrow M\}$
$\mathsf{match}\ M\ \mathsf{is}\ (X);P \triangleq \mathsf{match}\ M\ \mathsf{is}\ X;P$
$\mathsf{match}\ M\ \mathsf{is}\ (N,X_1,\ldots,X_n);P \triangleq \mathsf{match}\ M\ \mathsf{is}\ (N,y);\mathsf{match}\ y\ \mathsf{is}\ (X_1,\ldots,X_n);P$
$\mathsf{match}\ M\ \mathsf{is}\ (X_0,X_1,\ldots,X_n);P \triangleq$
$\quad\mathsf{split}\ M\ \mathsf{is}\ (x,y);\mathsf{match}\ x\ \mathsf{is}\ X_0;\mathsf{match}\ y\ \mathsf{is}\ (X_1,\ldots,X_n);P$
$\mathsf{match}\ M\ \mathsf{is}\ \mathsf{in}_{1,1}\,(X);P \triangleq \mathsf{match}\ M\ \mathsf{is}\ X;P$
$\mathsf{match}\ M\ \mathsf{is}\ \mathsf{in}_{1,n+1}\,(X);P \triangleq \mathsf{case}\ M\ \mathsf{is}\ \mathsf{inl}\ (x)\ \mathsf{match}\ x\ \mathsf{is}\ X;P\ \mathsf{is}\ \mathsf{inr}\ (x)\ \mathsf{stop}$
$\mathsf{match}\ M\ \mathsf{is}\ \mathsf{in}_{i+1,n+1}\,(X);P \triangleq \mathsf{case}\ M\ \mathsf{is}\ \mathsf{inl}\ (x)\ \mathsf{stop}\ \mathsf{is}\ \mathsf{inr}\ (x)\ \mathsf{match}\ x\ \mathsf{is}\ \mathsf{in}_{i,n}\,(X);P$
$\mathsf{match}\ M\ \mathsf{is}\ \{X\}_N P; \triangleq \mathsf{decrypt}\ M\ \mathsf{is}\ \{x\}_N;\mathsf{match}\ x\ \mathsf{is}\ X;P$
$\mathsf{match}\ M\ \mathsf{is}\ \{\!|X|\!\}_{N^{-1}} P; \triangleq \mathsf{decrypt}\ M\ \mathsf{is}\ \{\!|x|\!\}_{N^{-1}};\mathsf{match}\ x\ \mathsf{is}\ X;P$
$\mathsf{match}\ M\ \mathsf{is}\ N;P \triangleq \mathsf{match}\ (M,M)\ \mathsf{is}\ (N,x);P$

Thus we have demonstrated that our core language is powerful enough to describe the examples in this section.

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 to 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).

States:

$A, B, C ::=$	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

We define the operational semantics of a state by giving a reduction relation $As \to Bs$ meaning ‘in state As the program can perform one step of computation and become state Bs’.

State Transitions:

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

The rule (Trans New) makes use of α-conversion to ensure that x is always fresh.

Reachability $As \Rightarrow As'$**:**

(Reach Refl)

$$\frac{}{As \Rightarrow As}$$

(Reach Trans)

$$\frac{As \rightarrow As' \quad As' \Rightarrow As''}{As \Rightarrow As''}$$

An error state is one where an `end` L event is encountered without a matching tuple L in the tuple space.

Error States and Safety:

A state is an *error* if and only if it has the form $[\mathtt{end}\ L;P] + As$ where $L \notin As$.
A process P is *safe* if and only if there is no error state As such that $[P] \Rightarrow As$.

C Properties of the Type System

Some details omitted from this appendix are in a technical report [GJ02a].

C.1 Basics

Proposition 2 (Free Names)

(1) *If* $E \vdash T <: U$ *then* $fn(T) \cup fn(U) \subseteq dom(E)$.

(2) *If* $E \vdash M : T$ *then* $fn(M) \cup fn(T) \subseteq dom(E)$.

(3) *If* $E \vdash P : es$ *then* $fn(P) \cup fn(es) \subseteq dom(E)$.

Lemma 3 *If* $E \vdash \diamond$ *and* $x \notin dom(E)$ *and* $E \vdash T$ *then* $E, x{:}T \vdash \diamond$.

Lemma 4 *If* $E \vdash es$ *and* $es' \leq es$ *then* $E \vdash es'$.

Lemma 5 *If* $E \vdash es$ *and* $E \vdash es'$ *then* $E \vdash es + es'$.

Lemma 6 *If* $E \vdash T <: U$ *then* $E \vdash T$ *and* $E \vdash U$.

Lemma 7 *If* $E \vdash \diamond$ *and* $E \vdash M : T$ *then* $E \vdash T$.

We give a single proof of the following substitutivity and weakening properties.

Lemma 8 (Substitutivity) *If* $E, x{:}T \vdash \mathcal{J}$ *and* $E, x{:}T \vdash \diamond$ *and* $E\{x{\leftarrow}M\} \vdash M : T$ *then* $E\{x{\leftarrow}M\} \vdash \mathcal{J}\{x{\leftarrow}M\}$.

Lemma 9 (Weakening) *If* $E \vdash \mathcal{J}$ *and* $E, E' \vdash \diamond$ *then* $E, E' \vdash \mathcal{J}$.

Proof Lemmas 8 (Substitutivity) and 9 (Weakening) following by proving the following statements in order:

(1) Weakening for any $\mathcal{J}$ not of the form $P : es$.

By induction on the derivation of the judgment $E \vdash \mathcal{J}$. There is no appeal to substitutivity.

(2) Substitutivity for any $\mathcal{J}$ not of the form $P : es$.

By induction on the derivation of the judgment $E, x{:}T \vdash \mathcal{J}$. We appeal to statement (1) in the cases involving bound variables, that is, the rules (Msg Pair), (Sub Pair).

(3) Substitutivity for any $\mathcal{J}$ of the form $P : es$.

By induction on the derivation of the judgment $E, x{:}T \vdash P : es$. We appeal to statement (1) in the cases involving bound variables, such as (Proc Match). Moreover, we appeal to statement (2) in case (Proc Match).

Notice that we rely on (Proc Subsum) in the cases using multiset subtraction, since the inequality $es\{x \leftarrow M\} - fs\{x \leftarrow M\} \leq (es - fs)\{x \leftarrow M\}$ may be a strict inclusion.

(4) Weakening for any $\mathcal{J}$ of the form $P : es$.

By induction on the derivation of the judgment $E \vdash P : es$. We appeal to statement (2) in the case of (Proc Match). □

Lemma 10 (Strengthening) *If $E \vdash \diamond$ and $E, x{:}T \vdash \mathcal{J}$ and $x \notin dom(E) \cup fn(\mathcal{J})$ then $E \vdash \mathcal{J}$.*

Proof An induction on the proof of $E, x{:}T \vdash \mathcal{J}$, making use of Lemma 6 for the case of rule (Sub Pair); Lemma 7 for the cases of rules (Msg Pair), (Proc Split), (Proc Case), (Proc Symm) and (Proc Asymm); and Lemmas 7 and 8 (Substitutivity) for the case of rule (Proc Match). Any case which introduces a bound variable uses Lemma 3. □

Lemma 11 (Bound Weakening) *If $E, x{:}T \vdash \mathcal{J}$ and $E, x{:}T \vdash T' <: T$ then $E, x{:}T' \vdash \mathcal{J}$.*

Proof We prove the following cases in order: when $\mathcal{J}$ is of the form U, of the form es, of the form $U <: U'$, of the form $M : U$, and then finally of the form $P : es$. We prove each case by induction on the derivation of the judgment $E, x{:}T \vdash \mathcal{J}$. □

C.2 Opponent Typability

In this section, we show that any opponent process can be typed in an environment assigning the Un type to each of its free variables.

Derived Rules for Messages of Type Un:

(Msg Pair Un)
$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{Un}}{E \vdash (M, N) : \mathsf{Un}}$$

(Msg Inl Un)
$$\frac{E \vdash M : \mathsf{Un}}{E \vdash \mathsf{inl}\ (M) : \mathsf{Un}}$$

(Msg Inr Un)
$$\frac{E \vdash N : \mathsf{Un}}{E \vdash \mathsf{inr}\ (N) : \mathsf{Un}}$$

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

(Msg Part Un)
$$\frac{E \vdash M : \mathsf{Un}}{E \vdash k\ (M) : \mathsf{Un}}$$

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

Lemma 12 *The rules in the table above are derivable.*

Lemma 13 *If message M and environment E satisfy $E \vdash x : \mathsf{Un}$ for each $x \in fn(M)$, then $E \vdash M : \mathsf{Un}$.*

Proof By structural induction on the message M, and appeal to the rules (Env Good), and all of the derived rules in the table above: (Msg Pair Un), (Msg Inl Un), (Msg Inr Un), (Msg Asymm Un), (Msg Part Un). □

Derived Rules for Processes Manipulating Un:

(Proc Split Un)

$$\frac{E \vdash M : \mathsf{Un} \quad E, x{:}\mathsf{Un}, y{:}\mathsf{Un} \vdash P : []}{E \vdash \mathsf{split}\ M\ \mathsf{is}\ (x{:}\mathsf{Un}, y{:}\mathsf{Un}); P : []}$$

(Proc Match Un)(where $y \notin dom(E)$)

$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{Un} \quad E, y{:}\mathsf{Un} \vdash P : []}{E \vdash \mathsf{match}\ M\ \mathsf{is}\ (N, y{:}\mathsf{Un}); P : []}$$

(Proc Case Un) (where $x \notin dom(E)$ and $y \notin dom(E)$)

$$\frac{E \vdash M : \mathsf{Un} \quad E, x{:}\mathsf{Un} \vdash P : [] \quad E, y{:}\mathsf{Un} \vdash Q : []}{E \vdash \mathsf{case}\ M\ \mathsf{is}\ \mathsf{inl}\ (x{:}\mathsf{Un})\ P\ \mathsf{is}\ \mathsf{inr}\ (y{:}\mathsf{Un})\ Q : []}$$

(Proc Symm Un)(where $x \notin dom(E)$)

$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{Un} \quad E, x{:}\mathsf{Un} \vdash P : []}{E \vdash \mathsf{decrypt}\ M\ \mathsf{is}\ \{x{:}\mathsf{Un}\}_N; P : []}$$

(Proc Asymm Un)(where $x \notin dom(E)$)

$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{Un} \quad E, x{:}\mathsf{Un} \vdash P : []}{E \vdash \mathsf{decrypt}\ M\ \mathsf{is}\ \{\!|x{:}\mathsf{Un}|\!\}_{N^{-1}}; P : []}$$

Lemma 14 *The rules in the table above are derivable.*

Lemma 15 (Opponent Typability) *If process O is an opponent, that is, an assertion-free untyped unprivileged process, and the environment E satisfies $E \vdash x : \mathsf{Un}$ for each $x \in fn(O)$, then $E \vdash O : []$.*

Proof By structural induction on the process O, with appeal to the primitive rules (Proc Output Un), (Proc Input Un), (Proc Repeat Input Un), (Proc Res), (Proc Par), (Proc Stop), and all of the derived rules in the table above: (Proc Split Un), (Proc Match Un), (Proc Case Un), and (Proc Asymm Un). □

C.3 Algorithmic Formulation of Subtyping

We now present an alternative view of subtyping, which is designed to be easier to implement and to reason about. The problem with the existing definition is that it includes the rule (Sub Trans), which makes inductive proofs about subtyping difficult.

We present an algorithmic variant of subtyping, which does not require a transitivity rule, and then show it equivalent to the existing definition.

The algorithmic definition is based around two additional *kinds* of types: public types and tainted types. These have associated judgments $E \vdash \mathit{Tainted}(T)$ and $E \vdash \mathit{Public}(T)$.

Public and Tainted Types:

(*Tainted* Top)
$$\frac{}{E \vdash \mathit{Tainted}(\mathsf{Top})}$$

(*Public* Un)
$$\frac{}{E \vdash \mathit{Public}(\mathsf{Un})}$$

(*Tainted* Un)
$$\frac{}{E \vdash \mathit{Tainted}(\mathsf{Un})}$$

(*Public* Pair)
$$\frac{E \vdash \mathit{Public}(T) \quad E, x{:}T \vdash \mathit{Public}(U)}{E \vdash \mathit{Public}((x{:}T, U))}$$

(*Tainted* Pair)(where $E, x{:}T \vdash U$)
$$\frac{E \vdash \mathit{Tainted}(T) \quad E, x{:}\mathsf{Un} \vdash \mathit{Tainted}(U)}{E \vdash \mathit{Tainted}((x{:}T, U))}$$

(*Public* Sum)
$$\frac{E \vdash \mathit{Public}(T) \quad E \vdash \mathit{Public}(U)}{E \vdash \mathit{Public}(T + U)}$$

(*Tainted* Sum)
$$\frac{E \vdash \mathit{Tainted}(T) \quad E \vdash \mathit{Tainted}(U)}{E \vdash \mathit{Tainted}(T + U)}$$

Cryptographic Keys:

(*Public* Shared)
$$\frac{E \vdash \mathit{Public}(T) \quad E \vdash \mathit{Tainted}(T)}{E \vdash \mathit{Public}(\mathsf{SharedKey}(T))}$$

(*Tainted* Shared)
$$\frac{E \vdash \mathit{Public}(T) \quad E \vdash \mathit{Tainted}(T)}{E \vdash \mathit{Tainted}(\mathsf{SharedKey}(T))}$$

(*Public* Keypair)
$$\frac{E \vdash \mathit{Public}(T) \quad E \vdash \mathit{Tainted}(T)}{E \vdash \mathit{Public}(\mathsf{KeyPair}(T))}$$

(*Tainted* Keypair)
$$\frac{E \vdash \mathit{Public}(T) \quad E \vdash \mathit{Tainted}(T)}{E \vdash \mathit{Tainted}(\mathsf{KeyPair}(T))}$$

(*Public* Enc)
$$\frac{E \vdash \mathit{Tainted}(T)}{E \vdash \mathit{Public}(\mathsf{Encrypt\ Key}(T))}$$

(*Tainted* Enc)
$$\frac{E \vdash \mathit{Public}(T)}{E \vdash \mathit{Tainted}(\mathsf{Encrypt\ Key}(T))}$$

(*Public* Dec)
$$\frac{E \vdash \mathit{Public}(T)}{E \vdash \mathit{Public}(\mathsf{Decrypt\ Key}(T))}$$

(*Tainted* Dec)
$$\frac{E \vdash \mathit{Tainted}(T)}{E \vdash \mathit{Tainted}(\mathsf{Decrypt\ Key}(T))}$$

Nonce Challenge and Responses:

(*Public* Challenge [])
$$\frac{}{E \vdash \mathit{Public}(\mathsf{Public\ Challenge}\ [])}$$

(*Public* Response)
$$\frac{E \vdash es}{E \vdash \mathit{Public}(\mathsf{Public\ Response}\ es)}$$

(*Tainted* Public Challenge [])
$$\frac{}{E \vdash \mathit{Tainted}(\mathsf{Public\ Challenge}\ [])}$$

(*Tainted* Public Response [])
$$\frac{}{E \vdash \mathit{Tainted}(\mathsf{Public\ Response}\ [])}$$

(*Tainted* Private Challenge)

$$\frac{E \vdash es}{E \vdash \mathit{Tainted}(\mathsf{Private\ Challenge}\ es)}$$

(*Tainted* Private Response)

$$\frac{E \vdash es}{E \vdash \mathit{Tainted}(\mathsf{Private\ Response}\ es)}$$

(*Public* CR)

$$\frac{E \vdash es \quad E \vdash fs}{E \vdash \mathit{Public}(\mathsf{Public\ CR}\ es\ \ fs)}$$

Algorithmic Formulation of Subtyping:

(Sub Public Tainted)

$$\frac{E \vdash \mathit{Public}(T) \quad E \vdash \mathit{Tainted}(T')}{E \vdash T <: T'}$$

(Sub Top)

$$\frac{E \vdash T}{E \vdash T <: \mathsf{Top}}$$

(Sub CR C Algo)

$$\frac{E \vdash es + fs \quad es \leq es'}{E \vdash \ell\ \mathsf{CR}\ es\ \ fs <: \ell\ \mathsf{Challenge}\ es'}$$

(Sub CR R Algo)

$$\frac{E \vdash es + fs \quad fs \leq fs'}{E \vdash \ell\ \mathsf{CR}\ es\ \ fs <: \ell\ \mathsf{Response}\ \ fs'}$$

Congruence Rules:

(Sub Pair)(where $x \notin dom(E)$)

$$\frac{E \vdash T <: T' \quad E, x{:}T \vdash U <: U' \quad E, x{:}T' \vdash U'}{E \vdash (x{:}T, U) <: (x{:}T', U')}$$

(Sub Sum)

$$\frac{E \vdash T <: T' \quad E \vdash U <: U'}{E \vdash T + U <: T' + U'}$$

(Sub Key Invar)

$$\frac{E \vdash T <: T' \quad E \vdash T' <: T}{E \vdash \mathsf{SharedKey}(T) <: \mathsf{SharedKey}(T')}$$

(Sub Key Pair Invar)

$$\frac{E \vdash T <: T' \quad E \vdash T' <: T}{E \vdash \mathsf{KeyPair}(T) <: \mathsf{KeyPair}(T')}$$

(Sub Enc Contra)

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

(Sub Dec Co)

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

(Sub Challenge)

$$\frac{E \vdash es}{E \vdash \ell\ \mathsf{Challenge}\ es <: \ell\ \mathsf{Challenge}\ es}$$

(Sub Response)

$$\frac{E \vdash fs}{E \vdash \ell\ \mathsf{Response}\ fs <: \ell\ \mathsf{Response}\ fs}$$

(Sub CR)

$$\frac{E \vdash es' + fs' \quad es \leq es' \quad fs \leq fs'}{E \vdash \ell\ \mathsf{CR}\ es'\ fs' <: \ell\ \mathsf{CR}\ es\ fs}$$

Lemma 16 (Environmental Freedom) *In the algorithmic formulation of subtyping:*

(1) *If $E \vdash Public(T)$ and $dom(E) \subseteq dom(E')$ then $E' \vdash Public(T)$.*

(2) *If $E \vdash Tainted(T)$ and $dom(E) \subseteq dom(E')$ then $E' \vdash Tainted(T)$.*

(3) *If $E \vdash T <: U$ and $dom(E) \subseteq dom(E')$ then $E' \vdash T <: U$.*

Proof By a simultaneous induction on the derivation of the first judgment in each case. □

Lemma 17 (Public Down/Tainted Up) *In the algorithmic formulation of subtyping:*

(1) *If $E \vdash Public(T)$ and $E \vdash T' <: T$ then $E \vdash Public(T')$.*

(2) *If $E \vdash Tainted(T)$ and $E \vdash T <: T'$ then $E \vdash Tainted(T')$.*

Proof By a simultaneous induction on the derivation of the first judgment in each case. □

Lemma 18 (Public Tainted) *In the algorithmic formulation of subtyping:*

(1) *$E \vdash Public(T)$ if and only if $E \vdash T <: \mathsf{Un}$.*

(2) *$E \vdash Tainted(T)$ if and only if $E \vdash \mathsf{Un} <: T$.*

Proof We give the details for part (1); part (2) follows by a symmetric argument. Assume $E \vdash Public(T)$. Since $E \vdash Tainted(\mathsf{Un})$, by (*Tainted* Un), we get $E \vdash T <: \mathsf{Un}$ by (Sub Public Tainted). For the reverse direction, assume $E \vdash T <: \mathsf{Un}$. We have $E \vdash Public(\mathsf{Un})$, by (*Public* Un), so $E \vdash Public(T)$ by Lemma 17 (Public Down/Tainted Up)(1). □

Lemma 19 (Algo Trans) *In the algorithmic formulation of subtyping, the judgments $E \vdash T <: T'$ and $E \vdash T' <: T''$ imply $E \vdash T <: T''$.*

Proof By induction on the derivation of $E \vdash T <: T'$. □

Proposition 20 *The two formulations of $E \vdash T <: T'$ are equivalent.*

Proof Let *ALGO* and *ORIG* be the sets of sentences of the form $E \vdash T <: T'$ generated by the algorithmic and original formulations, respectively.

We can derive each of the original rules using the algorithmic rules. Since *ORIG* is the least set to satisfy the original rules, *ORIG* $\subseteq$ *ALGO*.

Next, we establish the following intermediate results:

(1) If $E \vdash \mathit{Tainted}(T)$ then $E \vdash \mathsf{Un} <: T$ derivable in the original system.

(2) If $E \vdash \mathit{Public}(T)$ then $E \vdash T <: \mathsf{Un}$ derivable in the original system.

The proofs are by induction on the derivations of $E \vdash \mathit{Tainted}(T)$ and $E \vdash \mathit{Public}(T)$. Now, we can derive each of the algorithmic rules using the original rules.

- To derive (Sub Public Tainted), we need to show that $E \vdash T <: T'$ is derivable in the original system if $E \vdash \mathit{Public}(T)$ and $E \vdash \mathit{Tainted}(T')$. By the results (1) and (2) proved above, we have $E \vdash T <: \mathsf{Un}$ and $E \vdash \mathsf{Un} <: T'$, in the original system. By (Sub Trans), we get $E \vdash T <: T'$ in the original system.
- Rule (Sub CR C Algo) follows from (Sub CR) and (Sub CR C). Rule (Sub CR R Algo) follows from (Sub CR) and (Sub CR R). Rules (Sub Challenge) and (Sub Response) follow from (Sub Refl).
- Rules (Sub Top), (Sub Sum), (Sub Key Invar), (Sub Key Pair Invar), (Sub Enc Contra), (Sub Dec Co), and (Sub CR) are shared between both definitions.

Since *ALGO* is the least set to satisfy the algorithmic rules, $\mathit{ALGO} \subseteq \mathit{ORIG}$. □

C.4 Properties of Subtyping

This section collects some properties of the subtype relation needed in our proof of type preservation. The proofs rely on the algorithmic formulation of subtyping presented in the previous section. Proposition 1 from Section 3.3 is equivalent to the following two propositions.

Proposition 21 (Public Key) *Suppose that $E \vdash T$ and $E \vdash \diamond$. Then the following are equivalent:*

(1) $E \vdash \mathsf{Un} <: T$,

(2) $E \vdash \mathsf{Encrypt\ Key}(T) <: \mathsf{Un}$, *and*

(3) $E \vdash \mathsf{Un} <: \mathsf{Decrypt\ Key}(T)$.

Proof

(1) ⇒ (2) By (Sub Enc Contra) and (Public Key), we get:

$$E \vdash \mathsf{Encrypt\ Key}(T) <: \mathsf{Encrypt\ Key}(\mathsf{Un}) <: \mathsf{Un}$$

(2) ⇒ (1) By Lemma 18 (Public Tainted), $E \vdash \mathit{Public}(\mathsf{Encrypt\ Key}(T))$, which itself can only be derived by (*Public* Enc) from $E \vdash \mathit{Tainted}(T)$. Hence (1) follows by Lemma 18 (Public Tainted).

(1) ⇒ (3) By (Public Key) and (Sub Dec Co), we get:

$$E \vdash \mathsf{Decrypt\ Key}(T) <: \mathsf{Decrypt\ Key}(\mathsf{Un}) <: \mathsf{Un}$$

(3) ⇒ (1) By Lemma 18 (Public Tainted), $E \vdash \mathit{Tainted}(\mathsf{Decrypt\ Key}(T))$, which itself can only be derived by (*Tainted* Dec) from $E \vdash \mathit{Tainted}(T)$. Hence (1) follows by Lemma 18 (Public Tainted). □

Proposition 22 (Digital Signature) *Suppose that $E \vdash T$ and $E \vdash \diamond$. Then the following are equivalent:*

(1) $E \vdash T <: \mathsf{Un}$,

(2) $E \vdash \mathsf{Un} <: \mathsf{Encrypt\ Key}(T)$, *and*

(3) $E \vdash \mathsf{Decrypt\ Key}(T) <: \mathsf{Un}$.

Proof Similar to the proof of Proposition 21 (Public Key). □

Lemma 23 (Pair Inversion) *If $E \vdash (x{:}T', U') <: (x{:}T, U)$ then $E \vdash T' <: T$ and $E, x{:}T' \vdash U' <: U$.*

Lemma 24 (Sum Inversion) *If $E \vdash T' + U' <: T + U$ then both $E \vdash T' <: T$ and $E \vdash U' <: U$.*

Lemma 25 (Key Inversion)

(1) *If $E \vdash \mathsf{Encrypt\ Key}(T) <: \mathsf{Encrypt\ Key}(T')$ then $E \vdash T' <: T$.*

(2) *If $E \vdash \mathsf{Decrypt\ Key}(T) <: \mathsf{Decrypt\ Key}(T')$ then $E \vdash T <: T'$.*

C.5 Properties of Message Typing

This section collects some properties of the message typing relation needed in our proof of type preservation.

Lemma 26 (Symm Key Match) *If we have $E \vdash \diamond$ and $E \vdash M : \mathsf{SharedKey}(T_1)$ and $E \vdash M : \mathsf{SharedKey}(T_2)$ then both $E \vdash T_1 <: T_2$ and $E \vdash T_2 <: T_1$.*

Proof By inspection of the type rules for messages, $E \vdash M : \mathsf{SharedKey}(T)$ implies that M is a variable, with $E = E', M{:}U, E''$, and $E \vdash U <: \mathsf{SharedKey}(T_1)$. Since this may be derived via (Sub Public Tainted) or (Sub Key Invar), there are two possibilities.

(A) $E \vdash \mathit{Public}(U)$, $E \vdash \mathit{Public}(T_1)$, and $E \vdash \mathit{Tainted}(T_1)$.

(B) $U = \mathsf{SharedKey}(U_1)$, $E \vdash U_1 <: T_1$, and $E \vdash T_1 <: U_1$.

Now, given $E \vdash \diamond$ the variables listed in E are distinct, so $E \vdash M : \mathsf{SharedKey}(T_2)$ further implies that $E \vdash U <: \mathsf{SharedKey}(T_2)$. We have two further possibilities:

(C) $E \vdash \mathit{Public}(U)$, $E \vdash \mathit{Public}(T_2)$, and $E \vdash \mathit{Tainted}(T_2)$.

(D) $U = \mathsf{SharedKey}(U_2)$, $E \vdash U_2 <: T_2$, and $E \vdash T_2 <: U_2$.

Overall, there are four combinations to consider. In combinations (AC), (AD), and (BC), we can show that both $E \vdash \mathit{Public}(T_2)$ and $E \vdash \mathit{Tainted}(T_1)$, and hence that $E \vdash T_2 <: T_1$ via (Sub Public Tainted). In combination (BD), we have that $U_1 = U_2$, and $E \vdash T_2 <: T_1$ follows by transitivity of subtyping. The converse, $E \vdash T_1 <: T_2$, follows by symmetric considerations. □

Lemma 27 (Keypair Match) *If $E \vdash \diamond$, $E \vdash M : \mathsf{KeyPair}(T_1)$, and $E \vdash M : \mathsf{KeyPair}(T_2)$ then both $E \vdash T_1 <: T_2$ and $E \vdash T_2 <: T_1$.*

Lemma 28 (Asymm Key Match) *If we have $E \vdash \mathsf{Encrypt}\ (N) : \mathsf{Encrypt\ Key}(T)$ and $E \vdash \mathsf{Decrypt}\ (N) : \mathsf{Decrypt\ Key}(U)$ then $E \vdash T <: U$.*

C.6 End-events, Trust-events, Check-events

We are now almost ready to prove the type preservation result which is the core of the robust safety result. Before we do so, however, we need to analyse the notion of effect. Since our type system contains a notion of *latent effect* in the nonce types, we must consider all of the effects an effect multiset might have. For example, in the environment $x{:}\ell\ \mathsf{CR}\ [\mathsf{end}\ L]\ []$, the effect $\mathsf{check}\ \ell\ x$ allows not only the side-effect $\mathsf{check}\ \ell\ x$ but also the latent effect $\mathsf{end}\ L$. For this reason, we define the *closure* of an effect given an environment to be the multiset of possible effects given by including all of the latent effects of any checked nonces. For example:

$$\mathit{closure}(x{:}\ell\ \mathsf{CR}\ [\mathsf{end}\ L]\ [])[\mathsf{check}\ \ell\ x] = [\mathsf{check}\ \ell\ x, \mathsf{end}\ L]$$

The function $\mathit{closure}(E, es)$ is partial, since some multisets contain ‘nonce cycles’ such as:

$$\begin{aligned}
&\mathit{closure}(x{:}\ell\ \mathsf{CR}\ [\mathsf{end}\ L, \mathsf{check}\ \ell\ x]\ [], [\mathsf{check}\ \ell\ x])\\
&\quad= \quad [\mathsf{end}\ L, \mathsf{check}\ \ell\ x] + \mathit{closure}(x{:}\ell\ \mathsf{CR}\ [\mathsf{end}\ L, \mathsf{check}\ \ell\ x]\ [], [\mathsf{check}\ \ell\ x])\\
&\quad= \quad [\mathsf{end}\ L, \mathsf{check}\ \ell\ x, \mathsf{end}\ L, \mathsf{check}\ \ell\ x] +\\
&\quad\qquad\qquad \mathit{closure}(x{:}\ell\ \mathsf{CR}\ [\mathsf{end}\ L, \mathsf{check}\ \ell\ x]\ [], [\mathsf{check}\ \ell\ x])\\
&\quad= \quad \cdots
\end{aligned}$$

For nonce acyclic multisets, the closure function is well-defined.

The Closure of an Effect Given an Environment $\mathit{closure}(E, es)$**:**

$$\mathit{closure}(E, [\mathsf{end}\ L]) \triangleq [\mathsf{end}\ L]$$

$$\mathit{closure}(E, [\mathsf{check}\ \ell\ x]) \triangleq \begin{cases} [\mathsf{check}\ \ell\ x] + \mathit{closure}(E, es + fs) & \text{if } E(x) = \ell\ \mathsf{CR}\ es\ fs \\ \varnothing & \text{otherwise} \end{cases}$$

$$\mathit{closure}(E, [\mathsf{trust}\, M{:}T]) \triangleq [\mathsf{trust}\, M{:}T]$$

$$\mathit{closure}(E, es + fs) \triangleq \mathit{closure}(E, es) + \mathit{closure}(E, fs)$$

$$\mathit{closure}(E, []) \triangleq []$$

Next, we define three properties of an effect es paired with an environment E. First, the pair (E, es) is trust-proper if every trust effect $\mathsf{trust}\, M{:}T$ in its closure is legitimate with respect to the environment E. Second, the pair (E, es) is check-proper if its closure is well-defined, is nonce-linear (that is, has no duplicate name-check effects), and is check-typed (that is, the names being checked have suitable types). Third, the pair (E, es) is end-proper for a message multiset Ls if Ls dominates the multiset of labels of end-events in the closure of (E, es). These three properties are part of the invariant guaranteed by the type system.

Trust-Proper Environment/Effect Pairs:

Let (E, es) be *trust-proper* if and only if
for every $(\mathsf{trust}\, M{:}T) \in \mathit{closure}(E, es)$ we have $E \vdash M{:}T$.

Check-Proper Environment/Effect Pairs:

Let (E, es) be *check-proper* if and only if
(1) $closure(E, es)$ is well-defined,
(2) $closure(E, es)$ is nonce-linear, that is,
there is no x such that $[\mathsf{check}\ \ell\ x, \mathsf{check}\ \ell\ x] \leq closure(E, es)$,
(3) (E, es) is *check-typed*, that is, $\mathsf{check}\ \ell\ x \in closure(E, es)$ implies either
$E(x) = \ell\ \mathsf{Challenge}\ es_C$ for some es_C, or $E(x) = \ell\ \mathsf{CR}\ es_C\ es_R$ for some es_C, es_R.

End-Proper Environment/Effect Pairs:

Let (E, es) be *end-proper* for Ls if and only if $[L \mid \mathsf{end}\ L \in closure(E, es)] \leq Ls$.

Lemma 29 *If:*

(1) $T = \ell\ \mathsf{CR}\ (es'_C)\ (es'_R)$

(2) $T' = \ell\ \mathsf{CR}\ (es_C + es'_C)\ (es_R + es'_R)$

(3) $es = es' + es_C + es_R$

(4) $(E, x{:}T, es)$ *is check-proper*

then $closure(E, x{:}T', es') \leq closure(E, x{:}T, es)$.

Proof First we use a routine induction to show that if $\mathsf{check}\ \ell\ x \notin closure(E, x{:}T, fs)$ then $closure(E, x{:}T', fs) = closure(E, x{:}T, fs)$.

Next, we show the main result by induction on the definition of $closure(E, x{:}T', es')$. The only interesting case is when $es' = [\mathsf{check}\ \ell\ x]$. We have:

$$\begin{array}{rcl} closure(E, x{:}T, es) & = & closure(E, x{:}T, [\mathsf{check}\ \ell\ x] + es_C + es_R) \\ & = & closure(E, x{:}T, es_C + es_R + es'_C + es'_R) + [\mathsf{check}\ \ell\ x] \end{array}$$

Since $(E, x{:}T, es)$ is check-proper, this means that:

$$\mathsf{check}\ \ell\ x \notin closure(E, x{:}T, es_C + es_R + es'_C + es'_R)$$

Thus, we can use the previous induction to show that:

$$\begin{array}{rcl} closure(E, x{:}T', es') & = & closure(E, x{:}T', [\mathsf{check}\ \ell\ x]) \\ & = & closure(E, x{:}T', es_C + es'_C + es_R + es'_R) + [\mathsf{check}\ \ell\ x] \\ & = & closure(E, x{:}T, es_C + es'_C + es_R + es'_R) + [\mathsf{check}\ \ell\ x] \\ & = & closure(E, x{:}T, [\mathsf{check}\ \ell\ x] + es_C + es_R) \\ & = & closure(E, x{:}T, es) \end{array}$$

The other cases are routine. □

C.7 Type Preservation

In this section, we define the invariant on computation states induced by the type system. We prove it is preserved by state transitions. Let a *nominal* type be one that is either Un, $\ell\ \mathsf{Challenge}\ es$, $\mathsf{KeyPair}(T)$, $\mathsf{SharedKey}(T)$, or a challenge-response type $\ell\ \mathsf{CR}\ es'\ fs'$. Let a *nominal environment* E be one where $E(x)$ is nominal for every $x \in dom(E)$.

Good State:

(State)(where $es = es_1 + \cdots + es_n$)

$$\frac{\begin{array}{c} E \vdash \diamond \\ E \vdash es \\ (E, es) \text{ is trust-proper} \\ (E, es) \text{ is check-proper} \\ (E, es) \text{ is end-proper for } Ls \\ E \text{ is nominal} \\ E \vdash P_1 : es_1 \quad \cdots \quad E \vdash P_n : es_n \end{array}}{E \vdash [P_1] + \cdots + [P_n] + Ls : es}$$

Theorem 2 (Type Preservation) *If $E \vdash As : es$ and $As \to As'$ then we can find E' and es' such that $E' \vdash As' : es'$.*

Proof For any $x \in fn(As') - fn(As)$, if $E = E', x{:}T, E''$ then we can use Lemmas 9 (Weakening) and 8 (Substitutivity) to get that $(E', y{:}T, E')\{y \leftarrow x\} \vdash As$ and so without loss of generality, we can assume that $x \notin dom(E)$.

The proof proceeds by a case analysis of the derivation of the state transition $As \to As'$. We only show the most interesting case of the proof.

(Trans Cast)

$$[\mathsf{cast}\ x\ \mathsf{is}\ (y{:}U); P] + Ps + Ls \to [P\{y \leftarrow x\}] + Ps + Ls$$

We have $E \vdash As : es$ so $es = es_1 + es_2$ with $E \vdash \mathsf{cast}\ x\ \mathsf{is}\ (y{:}U); P : es_1$ and $E \vdash Ps : es_2$. Only (Proc Cast) can derive $E \vdash \mathsf{cast}\ x\ \mathsf{is}\ (y{:}U); P : es_1$ so $es_1 = es_C + es_R + es_1'$ and $U = \ell\ \mathsf{Response}\ es_R$ with $E \vdash x : \ell\ \mathsf{Challenge}\ es_C$ and E, $y{:}\ell\ \mathsf{Response}\ es_R \vdash P : es_1'$ and $y \notin dom(E) \cup fn(es_1')$.

The judgment $E \vdash x : \ell\ \mathsf{Challenge}\ es_C$ must have come from an application $E_0, x{:}T \vdash x{:}T$ of (Msg x), with $E = E_0, x{:}T$, followed by a number of subsumption steps implying that $E \vdash T <: \ell\ \mathsf{Challenge}\ es_C$ by transitivity.

Assume that we can find a nominal type T' such that $E' \vdash T' <: T$ and $E' \vdash T' <: \ell\ \mathsf{Response}\ es_R$ and $closure(E', es') \leq closure(E, es)$, where we let $E' = E_0, x{:}T'$ and $es' = es_1' + es_2$.

Then:

- Since $E \vdash \diamond$ and $E \vdash T' <: T$, it follows by Lemma 11 (Bound Weakening) that $E' \vdash \diamond$.

- Since (E,es) is trust-proper and $closure(E',es') \leq closure(E,es)$, it follows that (E',es') is trust-proper.
- Since (E,es) is check-proper and $closure(E',es') \leq closure(E,es)$, it follows that (E',es') is check-proper.
- Since (E,es) is end-proper for Ls and $closure(E',es') \leq closure(E,es)$ it follows that (E',es') is end-proper for Ls.
- Since E is nominal and T' is nominal, it follows that E' is nominal.
- Since $E,y{:}\ell\ \mathsf{Response}\ es_R \vdash P : es'_1$ we have by Lemmas 8 (Substitutivity) and 11 (Bound Weakening) that $E' \vdash P\{y \leftarrow x\} : es'_1$. Since $E \vdash Ps : es_2$ we have by Lemma 11 (Bound Weakening) that $E' \vdash Ps : es_2$.

So we have found E' and es' such that $E' \vdash As' : es'$, as required.

All that remains is to find an appropriate T'. We proceed by case analysis of the rule used to derive $E \vdash T <: \ell\ \mathsf{Challenge}\ es_C$:

(1) (Sub Challenge): $T = \ell\ \mathsf{Challenge}\ es_C$. Let $T' \triangleq \ell\ \mathsf{CR}\ es_C\ es_R$, which is nominal, and satisfies both $E \vdash T' <: \ell\ \mathsf{Challenge}\ es_C$ and also $E \vdash T' <: \ell\ \mathsf{Response}\ es_R$. We can calculate, using Lemma 29:

$$\begin{aligned}
closure(E',es') &= closure(E_0, x{:}\ell\ \mathsf{CR}\ es_C\ es_R, es'_1 + es_2)\\
&\leq closure(E_0, x{:}\ell\ \mathsf{CR}\ []\ [], es_C + es_R + es'_1 + es_2)\\
&= closure(E_0, x{:}\ell\ \mathsf{CR}\ []\ [], es_1 + es_2)\\
&= closure(E_0, x{:}\ell\ \mathsf{Challenge}\ es_C, es_1 + es_2)\\
&= closure(E,es)
\end{aligned}$$

(2) (Sub CR C Algo): $T = \ell\ \mathsf{CR}\ es'_C\ es'_R$. Let $T' \triangleq \ell\ \mathsf{CR}\ (es_C + es'_C)\ (es_R + es'_R)$, which is nominal, and satisfies both $E \vdash T' <: \ell\ \mathsf{Challenge}\ es_C$ and also $E \vdash T' <: \ell\ \mathsf{Response}\ es_R$. We can calculate, using Lemma 29:

$$\begin{aligned}
&closure(E',es')\\
&\quad= closure(E_0, x{:}\ell\ \mathsf{CR}\ (es_C + es'_C)\ (es_R + es'_R), es'_1 + es_2)\\
&\quad\leq closure(E_0, x{:}\ell\ \mathsf{CR}\ es'_C\ es'_R, es_C + es_R + es'_1 + es_2)\\
&\quad= closure(E_0, x{:}\ell\ \mathsf{CR}\ es'_C\ es'_R, es_1 + es_2)\\
&\quad= closure(E,es)
\end{aligned}$$

(3) (Sub Public Tainted), where $\ell = \mathsf{Private}$. This means that we have both $E \vdash Public(T)$ and $E \vdash Tainted(\ell\ \mathsf{Challenge}\ es_C)$. Let $T' \triangleq T$ and $E' = E$, then use (Sub Public Tainted) again to derive $E' \vdash T' <: \ell\ \mathsf{Response}\ es_R$.

(4) (Sub Public Tainted), where $\ell = \mathsf{Public}$. This means we have $E \vdash Public(T)$ and $E \vdash Tainted(\ell\ \mathsf{Challenge}\ es_C)$, and so $es_C = []$. From the definition of a public type, and the requirement that T is nominal, there are these cases to consider:

(a) $T = \mathsf{Public\ CR}\ es'_C\ es'_R$. Let $T' \triangleq \mathsf{Public\ CR}\ (es_C + es'_C)\ (es_R + es'_R)$, which is nominal, and satisfies both $E \vdash T' <: \mathsf{Public\ CR}\ es'_C\ es'_R$ and also $E \vdash T' <: \mathsf{Public\ Response}\ es_R$, and we can calculate, using Lemma 29:

$$\begin{array}{ll} \multicolumn{2}{l}{closure(E', es')} \\ = & closure(E_0, x{:}\ell\ \mathsf{CR}\ (es_C + es'_C)\ (es_R + es'_R), es'_1 + es_2) \\ \leq & closure(E_0, x{:}\ell\ \mathsf{CR}\ es'_C\ es'_R, es_C + es_R + es'_1 + es_2) \\ = & closure(E_0, x{:}\ell\ \mathsf{CR}\ es'_C\ es'_R, es_1 + es_2) \\ = & closure(E, es) \end{array}$$

(b) $T = \mathsf{Un}$. Let $T' \triangleq \mathsf{Public\ CR}\ []\ es_R$, which is nominal, and satisfies both $E \vdash T' <: \mathsf{Un}$ and also $E \vdash T' <: \mathsf{Public\ Response}\ es_R$, and we can calculate, using Lemma 29:

$$\begin{array}{rcl} closure(E', es') & = & closure(E_0, x{:}\ell\ \mathsf{CR}\ es_C\ es_R, es'_1 + es_2) \\ & \leq & closure(E_0, x{:}\ell\ \mathsf{CR}\ []\ [], es_C + es_R + es'_1 + es_2) \\ & = & closure(E_0, x{:}\ell\ \mathsf{CR}\ []\ [], es_1 + es_2) \\ & = & closure(E_0, x{:}\mathsf{Un}, es_1 + es_2) \\ & = & closure(E, es) \end{array}$$

(c) $T = \mathsf{Public\ Challenge}\ []$, which uses a similar argument.

(d) $T = \mathsf{SharedKey}(\mathsf{Un})$, which uses a similar argument.

(e) $T = \mathsf{KeyPair}(\mathsf{Un})$, which uses a similar argument. □

The theorem justifies the intended meaning of an effect of a process: it is an upper bound on the end-events that may be performed by the process.

C.8 Safety and Robust Safety

Our main application of Theorem 2 (Type Preservation) is to establish safety and robust safety properties of well-typed processes.

Lemma 30 *If $E \vdash As : es$ and $As \Rightarrow As'$ then there are E' and es' such that $E' \vdash Ps' : es'$.*

Proof An induction on the derivation of $As \Rightarrow As'$, making use of Theorem 2 (Type Preservation). □

Theorem 3 (Safety) *If $x_1{:}\mathsf{Un}, \ldots, x_n{:}\mathsf{Un} \vdash P : []$ then P is safe.*

Proof Consider an arbitrary state As such that $[P] \Rightarrow As$. Let $As = Ps + Ls$, and let $E = x_1{:}\mathsf{Un}, \ldots, x_n{:}\mathsf{Un}$. We have by (State), $E \vdash [P] : []$ and so by Lemma 30, we can find E' and es' such that $E' \vdash As : es'$. Then if $Ps = [\mathsf{end}\ L; P'] + Ps'$, we must have $\mathsf{end}\ L \in es'$ and so $L \in Ls$. Thus, As is not an error state, and so P is safe. □

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

Proof Consider any opponent O. Suppose $fn(O) - \{x_1,\ldots,x_n\} = \{y_1,\ldots,y_m\}$. Let $E = x_1{:}\mathsf{Un},\ldots,x_n{:}\mathsf{Un},\ y_1{:}\mathsf{Un},\ldots,y_m{:}\mathsf{Un}$. By Lemma 15 (Opponent Typability), we have $E \vdash O : []$. By Lemma 9 (Weakening), $E \vdash P : []$. By (Proc Par), $E \vdash P \mid O : []$. By Theorem 3 (Safety), $P \mid O$ is safe. □

References

[AB02] M. Abadi and B. Blanchet. Analyzing security protocols with secrecy types and logic programs. In *29th ACM Symposium on Principles of Programming Languages*, pages 33–44, 2002.

[AB03] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. *Theoretical Computer Science*, 298(3):387–415, 2003.

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

[AC01] D. Aspinall and A. Compagnoni. Subtyping dependent types. *Theoretical Computer Science*, 266(1–2):273–309, 2001.

[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.

[Bla02] B. Blanchet. From secrecy to authenticity in security protocols. In *9th International Static Analysis Symposium (SAS'02)*, volume 2477 of *Lecture Notes in Computer Science*, pages 242–259. Springer, 2002.

[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.

[Cer01] 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.

[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 report. University of York, 1997.

[DMP01] N. Durgin, J.C. Mitchell, and D. Pavlovic. A compositional logic for protocol correctness. In *14th IEEE Computer Security Foundations Workshop*, pages 241–255. 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.

[GJ01] 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. Extended version to appear in *J. Computer Security*.

[GJ02a] A.D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. Technical Report MSR–TR–2002–31, Microsoft Research, August 2002.

[GJ02b] 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.

[GJ03] A.D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. *Theoretical Computer Science*, 300:379–409, 2003.

[GP02] A.D. Gordon and R. Pucella. Validating a web service security abstraction by typing. In *2002 ACM Workshop on XML Security*, pages 62–73. ACM Press, 2002. An extended version appears as Technical Report MSR–TR–2002–108, Microsoft Research, December 2002.

[GT02] J.D. Guttman and F.J. Thayer. Authentication tests and the structure of bundles. *Theoretical Computer Science*, 283(2):333–380, 2002.

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

[HS00] 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.

[Low96] 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.

[Low97] G. Lowe. A hierarchy of authentication specifications. In *10th IEEE Computer Security Foundations Workshop*, pages 31–43. IEEE Computer Society Press, 1997.

[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.

[SBP01] D. Song, S. Berezin, and A. Perrig. Athena, a novel approach to efficient automatic security protocol analysis. *Journal of Computer Security*, 9(1,2):47–74, 2001.

[Sch98] S.A. Schneider. Verifying authentication protocols in CSP. *IEEE Transactions on Software Engineering*, 24(9):741–758, 1998.

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

[THG99] F.J. Thayer Fábrega, J.C. Herzog, and J.D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7:191–230, 1999.

[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 Computer Society Symposium on Research in Security and Privacy*, pages 178–194, 1993.