Authenticity by Typing for Security Protocols

Andrew D. Gordon
Microsoft Research

Alan Jeffrey
DePaul University

May 2001

Technical Report
MSR–TR–2001–49

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

Publication History

A portion of this work will appear in the proceedings of the *14th IEEE Computer Security Foundations Workshop (CSFW 14)*, Cape Breton, June 11–13, 2001. The proceedings will be published by the IEEE Computer Society.

Affiliation

Alan Jeffrey is with DePaul University. The two authors completed part of this work at Microsoft Research in Cambridge and part at DePaul University in Chicago.

Abstract

We propose a new method to check authenticity properties of cryptographic protocols. First, code up the protocol in the spi-calculus of Abadi and Gordon. Second, specify authenticity properties by annotating the code with correspondence assertions in the style of Woo and Lam. Third, figure out types for the keys, nonces, and messages of the protocol. Fourth, check that the spi-calculus code is well-typed according to a novel type and effect system presented in this paper. Our main theorem guarantees that any well-typed protocol is robustly safe, that is, its correspondence assertions are true in the presence of any opponent expressible in spi. It is feasible to apply this method by hand to several well-known cryptographic protocols. It requires little human effort per protocol, puts no bound on the size of the opponent, and requires no state space enumeration. Moreover, the types for protocol data provide some intuitive explanation of how the protocol works. This paper describes our method and gives some simple examples. Our method has led us to the independent rediscovery of flaws in existing protocols and to the design of improved protocols.

Contents

1 Verifying Correspondences by Typing Spi

We propose a new method for analysing authenticity properties of cryptographic protocols. Our proposal builds on and develops two existing ideas: Woo and Lam's idea of correspondence assertions for specifying authentication properties of protocols [41], and Abadi's idea of checking security properties of cryptographic protocols by type-checking [1].

Woo and Lam's idea of correspondence assertions is very simple. Starting from some 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. Moreover, 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, before running a protocol to authenticate its presence to another principal B, an initiator A asserts a begin-event labelled "initiator A authenticating itself to responder B". After satisfactory completion of the protocol, the principal B asserts an end-event with the same label. A protocol satisfies these assertions if in all protocol runs, and in the presence of a hostile opponent, every assertion of an end-event corresponds to a distinct, earlier assertion of a begin-event with the same label. The hostile opponent can capture, modify, and replay messages, but cannot forge assertions.

Woo and Lam's paper [41] describes a formal semantics for correspondence assertions but suggests no verification techniques. Marrero, Clarke, and Jha [30] base a model-checker for security protocols on correspondence assertions. This paper formalises correspondence assertions as new commands in the spi-calculus [3], a concurrent programming language equipped with abstract forms of cryptographic primitives. We expect it would not be difficult to adapt the techniques of this paper to other concurrent languages.

There is a variety of different formulations of authenticity properties of protocols, and even a little controversy [6, 16, 27, 13]. Still, we adopt correspondence assertions because they are simple, precise, and flexible. They are simple annotations of a protocol expressed as a program. They have a precise semantics. They are flexible in the sense that by annotating a protocol in different ways we can express different authenticity intentions and guarantees. Correspondence assertions allow us to express what Lowe [27] calls injective agreement between protocol runs. In a formal comparison of authenticity properties, Focardi, Gorrieri, and Martinelli [14] formulate a property that systematically generalises the equational properties proved in the original work on spi [3], and show that this generalisation is strictly weaker than agreement. Therefore, there is some evidence that the authentication properties proved in this paper are at least as strong as in the original work.

By verifying suitable correspondence assertions, our method can rule out problems such as vulnerability to replay attacks or confusions of identity. Still, like most other formal methods for analysing authenticity protocols, our method deliberately abstracts from the details of the underlying encryption algorithms, and therefore cannot detect protocol weaknesses deriving from inadequacies in these algorithms.

Abadi's idea of type-checking secrecy properties of cryptographic protocols in the spi-calculus is part of a surge of interest in types for security. Other work includes

type systems for checking untrusted mobile code [26, 32, 19], for checking access control [25, 37], and, most recently, other type systems for cryptographic primitives [35, 2]. Abadi's original system establishes secrecy properties, and features some unusual constructs that allow any opponent to be type-checked. This paper develops some of the constructs in Abadi's system, and proposes a new type and effect system [15, 29] for the spi-calculus. For a well-typed program containing correspondence assertions, a type safety theorem guarantees the program satisfies the assertions.

Our new method is the following. First, code up the protocol in the spi-calculus. Second, specify authenticity properties expected of the protocol by annotating the code with correspondence assertions. Third, figure out types for the keys, nonces, and messages of the protocol. Fourth, check that the spi-calculus code is well-typed. The type safety theorem guarantees the soundness of the authenticity properties specified in the second step. The theorem asserts these properties hold in the presence of an opponent represented by an arbitrary spi process. Therefore, a limitation of the theorem is that it does not rule out attacks that cannot be expressed in the spi-calculus. On the other hand, it does not limit the size of the attacker in any way. We have applied this method to several protocols by hand, and have re-discovered some known flaws.

Our method is one of only a few formal analyses that require little human effort per protocol, while putting no bound on the size of the protocol or opponent. Other examples include Song's mechanisation [38] of strand spaces [39], Heather and Schneider's algorithm [24, 22] for computing Schneider's rank functions [36], and Cohen's resolution-based theorem prover TAPS [10]. Non-examples include most approaches based on model-checking [28], which are automatic but require bounds on the size of the opponent or the protocols, and most approaches based on theorem-proving [8, 34], which impose no bound on opponent or protocol size, but require lengthy and expert human intervention.

Our method is also one of only a few where analysing a protocol involves no exploration or enumeration of the possible states or messages of the protocol, and so is decidable even for protocols with no bound on the size of the principals. The only other such methods we know of are those based on proof-checking belief logics [9, 17]. Like constructing a proof in a belief logic, the work of devising types for a protocol in our system amounts to writing down a formal argument explaining the protocol. Failing to find a proof or a typing can suggest possible attacks on the protocol. Unlike most belief logics, our method has a precise computational basis.

In this paper, we only consider type checking, not type synthesis. Type checking (where the computer checks user-defined typings) is easily seen to be decidable, and provides a straightforward top-down algorithm for protocol verification. Type synthesis (where the computer derives the typings itself) would be harder.

In summary, our new method enjoys a rare and attractive combination of strengths:

- It needs little human effort per protocol.
- It puts no bound on the size of the principals.
- It needs no state space enumeration per protocol.
- It has a precise computational foundation.
- It is decidable.

On the other hand, the type system on which our method is based has limitations. Like all type systems, it is incomplete in the sense that perfectly well-behaved code can fail to type-check. For example, we have found that certain uses of nonces cannot be type-checked. Our system is also limited to symmetric-key cryptography. We leave the study of types for other cryptographic primitives as future work.

The new technical contribution of this paper is a type and effect system for proving correspondence assertions that supports the cryptographic primitives of the spi-calculus. A series of examples supports its usefulness. In earlier work [18], we proposed a type system for proving correspondence assertions about non-cryptographic communication protocols in the π-calculus. The system of the present paper copes with untrusted opponents, encryption primitives, and synchronisation via nonce handshakes, additional features essential for cryptographic protocols.

Contents of this Paper

Section 2 presents the spi-calculus, and illustrates programming of security protocols. Section 3 extends the spi-calculus with correspondence assertions, and shows how they can specify authenticity properties. Section 4 describes our type and effect system. Section 5 discusses further examples. Section 6 concludes.

2 Programming Protocols

This section reviews the syntax and informal semantics of the spi-calculus, and explains how to express a simple protocol example as a spi-calculus program.

Milner, Parrow, and Walker's π-calculus [31] is a parsimonious formalism for concurrency. It explains many different kinds of computation by reducing them to exchanges of names on communication channels. An important constituent of the calculus is a name generation operator for generating fresh names, which identify communication channels.

Abadi and Gordon's spi-calculus [3] is an extension of the π-calculus with abstract forms of encryption and decryption, akin to the idealised versions introduced by Dolev and Yao [12]. The atomic names of the spi-calculus represent the random numbers of cryptographic protocols, such as encryption keys and nonces, as well as channels. The name generation operator abstractly represents the fresh generation of unguessable random numbers such as keys and nonces. We can describe cryptographic protocols by programming them in the spi-calculus.

There are several existing spi-calculus techniques, such as notions of bisimulation, for reasoning about authenticity properties. The new contribution of this paper is a type system for reasoning about authenticity. Our preliminary experience is that establishing authenticity properties by typing is much less labour intensive than constructing bisimulations.

2.1 Review of the Spi-Calculus

There are in fact several versions of spi. The main difference between the spi-calculus presented in this section and the original version [3] is that each binding occurrence of a name is annotated with a type, T. (We postpone defining the set of types till Section 4.) Choosing these type annotations is part of our verification method; they are needed for type-checking processes, but do not affect the runtime behaviour of processes.

We assume an infinite set of atomic names or variables, ranged over by m, n, x, y, and z. For the sake of simplicity in presenting our type system, this version of the spi-calculus, unlike the original, does not distinguish names from variables. The set of messages, which includes the set of names, is given by the grammar in the following table.

Names and Messages:

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

- A message (M,N) is a pair, and $()$ is an empty tuple. With these primitives 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.

We regard messages as abstract representations of the bit strings manipulated by cryptographic protocols. We assume there is enough redundancy in the format that we can tell apart the different kinds of messages.

Free names of a message $fn(M)$:

$fn(x) \stackrel{\Delta}{=} \{x\}$
$fn(()) \stackrel{\Delta}{=} \varnothing$
$fn(M,N) \stackrel{\Delta}{=} fn(M) \cup fn(N)$
$fn(\mathsf{inl}\ (M)) \stackrel{\Delta}{=} fn(M)$
$fn(\mathsf{inr}\ (M)) \stackrel{\Delta}{=} fn(M)$
$fn(\{M\}_N) \stackrel{\Delta}{=} fn(M) \cup fn(N)$

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.

The set of processes is defined by the grammar:

Processes:

$O,P,Q,R ::=$	process
$\mathsf{out}\ M\ N$	output
$\mathsf{inp}\ M\ (x{:}T);P$	input
$\mathsf{split}\ M\ \mathsf{is}\ (x{:}T,y{:}U);P$	pair splitting
$\mathsf{case}\ M\ \mathsf{is}\ \mathsf{inl}\ (x{:}T)\ P\ \mathsf{is}\ \mathsf{inr}\ (y{:}U)\ Q$	union case
$\mathsf{decrypt}\ M\ \mathsf{is}\ \{x{:}T\}_N;P$	decryption
$\mathsf{check}\ M\ \mathsf{is}\ N;P$	name-check
$\mathsf{new}\ (x{:}T);P$	name generation
$P \mid Q$	composition
$\mathsf{repeat}\ P$	replication
stop	inactivity

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 $P\{y{\leftarrow}M\}$.
- 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 $\mathsf{case}\ M\ \mathsf{is}\ \mathsf{inl}\ (x{:}T)\ P\ \mathsf{is}\ \mathsf{inr}\ (y{:}U)\ Q$ 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 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 $\mathsf{check}\ M\ \mathsf{is}\ N;P$ checks the messages M and N are the same name before executing P. If the equality test fails, the process deadlocks.
- A process $\mathsf{new}\ (x{:}T);P$ generates a new name x, whose scope is P, and then runs P.
- A process $P \mid Q$ runs processes P and Q in parallel.
- A process $\mathsf{repeat}\ P$ replicates P arbitrarily often. So $\mathsf{repeat}\ P$ behaves like $P \mid \mathsf{repeat}\ P$.
- The process stop is deadlocked.

Each binding occurrence of a name bears a type annotation. These types play a role in type-checking but have 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.

Free names of a process $fn(P)$:

$fn(\mathsf{out}\ M\ N \stackrel{\Delta}{=} fn(M) \cup fn(N)$

$fn(\mathsf{inp}\ M\ (x{:}T);P \stackrel{\Delta}{=} fn(M) \cup fn(T) \cup (fn(P) - \{x\})$

$fn(\mathsf{split}\ M\ \mathsf{is}\ (x{:}T, y{:}U); P) \triangleq fn(M) \cup fn(T) \cup (fn(U) - \{x\}) \cup (fn(P) - \{x, y\})$

$fn(\mathsf{case}\ M\ \mathsf{is}\ \mathsf{inl}\ (x{:}T)\ P\ \mathsf{is}\ \mathsf{inl}\ (y{:}U)\ Q\ ) \triangleq fn(M) \cup fn(T) \cup (fn(P) - \{x\}) \cup fn(U) \cup (fn(Q) - \{y\})$

$fn(\mathsf{decrypt}\ M\ \mathsf{is}\ \{x{:}T\}_N; P) \triangleq fn(M) \cup fn(T) \cup fn(N) \cup (fn(P) - \{x\})$

$fn(\mathsf{check}\ M\ \mathsf{is}\ N; P) \triangleq fn(M) \cup fn(N) \cup fn(P)$

$fn(\mathsf{new}\ (x{:}T); P) \triangleq fn(T) \cup (fn(P) - \{x\})$

$fn(P \mid Q) \triangleq fn(P) \cup fn(Q)$

$fn(\mathsf{repeat}\ P) \triangleq fn(P)$

$fn(\mathsf{stop}) \triangleq \varnothing$

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 $\mathsf{new}\ (y{:}T); (P\{x \leftarrow y\})$. We will often elide stop from the end of processes, and we will write $\mathsf{out}\ x\ M; P$ as shorthand for $\mathsf{out}\ x\ M \mid P$.

2.2 Programming an Example

This section shows how to program a simple cryptographic protocol in spi. The protocol is intended to allow a fixed principal A to send a series of messages to another fixed principal B via a public channel, assuming they both share a secret key K. The idea is simply that A encrypts each message. Of course, for many purposes this protocol is actually far too simple: it is vulnerable to an attacker intercepting and replaying a message, so that B may accept the message twice though A sent it just once. In the next section, we introduce correspondence assertions to specify that B should accept a message M no more times than A sent M, and we discuss a standard guard against replay attacks, based on nonces.

In a common notation, we can summarise this flawed protocol as follows:

$$\text{Message 1} \quad A \rightarrow B : \quad \{M\}_K$$

Although standard, this notation leaves implicit details of both protocol behaviour and security goals. One of the original purposes of the spi-calculus was to make protocol behaviour explicit in an executable format. We can program the protocol in spi as follows.

First, we describe the behaviour of the sender and receiver.

$$\begin{array}{ll} \mathit{FlawedSender}(\mathit{net}, \mathit{key}) \triangleq & \mathit{FlawedReceiver}(\mathit{net}, \mathit{key}) \triangleq \\ \quad \mathsf{repeat} & \quad \mathsf{repeat} \\ \quad\quad \mathsf{new}\ (\mathit{msg}); & \quad\quad \mathsf{inp}\ \mathit{net}\ (\mathit{ctext}); \\ \quad\quad \mathsf{out}\ \mathit{net}\ \{\mathit{msg}\}_{\mathit{key}} & \quad\quad \mathsf{decrypt}\ \mathit{ctext}\ \mathsf{is}\ \{\mathit{msg}\}_{\mathit{key}} \end{array}$$

These are:

- The process $FlawedSender(net, key)$ is the sender A, parameterized on *net* (the name of the public channel) and *key* (the shared secret key). It repeatedly generates a fresh name *msg*, and then sends the ciphertext $\{msg\}_{key}$ on the public *net* channel.

  (In passing from the informal notation to the spi-calculus, we have determined that the plaintexts of the sent messages are freshly generated, rather than say being predetermined. It is easy to adapt this process to take a list of predetermined plaintexts as parameter.)
- The process $FlawedReceiver(net, key)$ is the receiver B, parameterized on *net* and *key* It repeatedly receives a message on the public *net* channel, binds it to variable *ctext*, and attempts to decrypt it with key *key*.

We specify the behaviour of the whole system running in the protocol by generating a fresh name *key*—the shared secret key—and then by placing the sender and receiver in parallel.

$$\begin{array}{l} FlawedSystem(net, done) \triangleq \\ \quad \mathsf{new}\ (key); \\ \quad\quad (FlawedSender(net, key) \mid FlawedReceiver(net, key)) \end{array}$$

Most protocols analysed with the spi-calculus have been programmed in this style.

3 Specifying Protocols

Woo and Lam [41] introduce correspondence assertions, a method for specifying protocol authenticity properties, such as properties that are violated by replay or man-in-the-middle attacks. The method depends on principals asserting labelled begin- and end-events during the course of a protocol. The idea is that each end-event should correspond to a distinct, preceding begin-event with the same label. Otherwise there is an error in the protocol.

To formalize these ideas, in Section 3.1, we enrich our spi-calculus with assertions of begin- and end-events. Then, in Section 3.2, we illustrate how to specify an authenticity property of our example protocol, and show in fact that the protocol is flawed. In Section 3.3 we fix the flaw by adding a standard nonce handshake.

3.1 A Spi-Calculus with Correspondence Assertions

First, we introduce the following notation for events, using messages as labels.

Events:

$\mathsf{begin}\ L$	begin-event labelled with message L
$\mathsf{end}\ L$	end-event labelled with message L

Second, we add processes to assert begin- and end-events.

Processes:

$O, P, Q, R ::=$	process
$\ldots$	as in Section 2.1
$\mathsf{begin}\ L; P$	begin-assertion
$\mathsf{end}\ L; P$	end-assertion

Assertions are autonomous in that they act independently without any synchronisation with other processes.

- The begin-assertion $\mathsf{begin}\ L; P$ autonomously asserts a $\mathsf{begin}\ L$ event, and then behaves as P.
- The end-assertion $\mathsf{end}\ L; P$ autonomously asserts an $\mathsf{end}\ L$ event, and then behaves as P.

Free names of a process $fn(P)$:

$fn(\mathsf{begin}\ M; P) \triangleq fn(M) \cup fn(P)$
$fn(\mathsf{end}\ M) \triangleq fn(M)$

Given this informal semantics, we give an informal definition of process safety. (We formalize these definitions in Appendix B via a trace semantics for the spi-calculus.)

Safety:

A process P is *safe* if and only if
for every run of the process and for every L,
there is a distinct $\mathsf{begin}\ L$ event for every $\mathsf{end}\ L$ event.

For example:

- Process $\mathsf{begin}\ L; \mathsf{end}\ L$ is safe.
- Process $\mathsf{begin}\ L; \mathsf{end}\ L; \mathsf{end}\ L$ is unsafe because of the unmatched $\mathsf{end}\ L$.
- Process $\mathsf{begin}\ L; \mathsf{begin}\ L; \mathsf{end}\ L$ is safe; the unmatched $\mathsf{begin}\ L$ does not affect safety.
- Process $\mathsf{begin}\ L; \mathsf{begin}\ L; \mathsf{end}\ L; \mathsf{end}\ L$ is safe; here there are two correspondences, both named L.
- Process $\mathsf{begin}\ L; \mathsf{end}\ L; \mathsf{begin}\ L'; \mathsf{end}\ L'$ is safe.
- Process $\mathsf{begin}\ L; \mathsf{end}\ L'; \mathsf{begin}\ L'; \mathsf{end}\ L$ is unsafe.

Safety does not require begin- and end-assertions to be properly bracketed:

- Process $\mathsf{begin}\ L; \mathsf{begin}\ L'; \mathsf{end}\ L'; \mathsf{end}\ L$ is safe.
- Process $\mathsf{begin}\ L; \mathsf{begin}\ L'; \mathsf{end}\ L; \mathsf{end}\ L'$ is safe.

Finally, consider the parallel process $\mathsf{begin}\ L \mid \mathsf{end}\ L$. This process either asserts a $\mathsf{begin}\ L$ event followed by an $\mathsf{end}\ L$ event, or it asserts an $\mathsf{end}\ L$ event followed by a $\mathsf{begin}\ L$ event. Because of the latter run, the process is unsafe.

We are mainly concerned not just with safety, but with safety in the presence of an arbitrary hostile opponent, which we call robust safety. (This use of "robust" to describe a property invariant under composition with an arbitrary environment follows Grumberg and Long [20]). In the untyped spi-calculus [3], 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 are not required to be well-typed: we model this using a type Un for untyped, untrusted data. This is discucssed further in Section 4

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.
A process P is *robustly safe* if and only if
  $P \mid O$ is safe for every opponent O.

3.2 Specifying the Example

Recall the protocol example of Section 2.2. Two fixed principals A and B share a key K with which A sends a sequence of messages to B. We introduce begin- and end-events labelled M for each message M. The sender asserts a begin-event labelled M before sending M, and the receiver asserts an end-event labelled M after successfully receiving a message M.

We express this idea informally as follows:

Event 1	A begins	M
Message 1	$A \rightarrow B$:	$\{M\}_K$
Event 2	B ends	M

We express the idea formally by inserting assertion processes into the spi-calculus de-

scriptions of the sender and receiver. We update our definitions as follows.

$$
\begin{array}{ll}
\mathit{CheckedSender}(\mathit{net},\mathit{key}) \triangleq & \mathit{CheckedReceiver}(\mathit{net},\mathit{key}) \triangleq \\
\quad \mathsf{repeat} & \quad \mathsf{repeat} \\
\quad\quad \mathsf{new}\ (\mathit{msg}); & \quad\quad \mathsf{inp}\ \mathit{net}\ (\mathit{ctext}); \\
\quad\quad \mathsf{begin}\ \mathit{msg}; & \quad\quad \mathsf{decrypt}\ \mathit{ctext}\ \mathsf{is}\ \{\mathit{msg}\}_{\mathit{key}}; \\
\quad\quad \mathsf{out}\ \mathit{net}\ \{\mathit{msg}\}_{\mathit{key}} & \quad\quad \mathsf{end}\ \mathit{msg}
\end{array}
$$

$$
\begin{array}{l}
\mathit{CheckedSystem}(\mathit{net}) \triangleq \\
\quad \mathsf{new}\ (\mathit{key}); \\
\quad\quad (\mathit{CheckedSender}(\mathit{net},\mathit{key}) \mid \mathit{CheckedReceiver}(\mathit{net},\mathit{key}))
\end{array}
$$

Next, we precisely state the authenticity property we desire (but that is actually violated by the protocol).

Authenticity: The process $\mathit{CheckedSystem}(\mathit{net})$ is robustly safe. (Breaks.)

If the protocol is safe, each end *msg* has a distinct corresponding begin *msg*, and therefore *B* accepts each message no more times than *A* sent it. Moreover, if the protocol is robustly safe, no attacker can violate this property.

It is easy to prove that this protocol is safe, since the protocol itself never duplicates messages. Still, the protocol is not robustly safe since a suitable attacker can violate this safety property.

$$
\begin{array}{l}
\mathit{Attacker}(\mathit{net}) \triangleq \\
\quad \mathsf{inp}\ \mathit{net}\ (\mathit{ctext}); \mathsf{out}\ \mathit{net}\ (\mathit{ctext}); \mathsf{out}\ \mathit{net}\ (\mathit{ctext})
\end{array}
$$

Here is an unsafe run of the process $\mathit{CheckedSystem}(\mathit{net}) \mid \mathit{Attacker}(\mathit{net})$. The sender $\mathit{CheckedSender}(\mathit{net},\mathit{key})$ generates a name *msg*, performs a single begin *msg*; event, and sends the ciphertext $\{\mathit{msg}\}_{\mathit{key}}$ on *net*. The attacker $\mathit{Attacker}(\mathit{net})$ receives this message, and then sends two copies of on *net*. The receiver then receives one of these copies, successfully decrypts it, and asserts an end *msg* event. So far so good. But now another iteration of the body of $\mathit{CheckedReceiver}(\mathit{net},\mathit{key})$ receives the second copy, successfully decrypts it, and asserts another end *msg* event. Because of the second end-event is unmatched, the run breaks the authenticity property displayed above.

3.3 Fixing the Example

A standard countermeasure against replay attacks is to include a *nonce*, a randomly generated bit-string, in each ciphertext to ensure its uniqueness. The following variant of our protocol is now initiated by the receiver, who sends a new nonce N to the sender, to guard against replays of the encrypted form of the message M.

Event 1	A begins	M
Message 1	$B \to A$:	N
Message 2	$A \to B$:	$\{M,N\}_K$
Event 2	B ends	M

In the spi-calculus, nonces are represented by names, and creation of fresh nonces by name generation. We program the revised protocol as follows:

$$
\begin{array}{ll}
\mathit{FixedSender}(\mathit{net},\mathit{key}) \triangleq & \mathit{FixedReceiver}(\mathit{net},\mathit{key}) \triangleq \\
\quad \mathsf{repeat} & \quad \mathsf{repeat} \\
\quad\quad \mathsf{inp}\ \mathit{net}\ (\mathit{nonce}); & \quad\quad \mathsf{new}\ (\mathit{nonce}); \\
\quad\quad \mathsf{new}\ (\mathit{msg}); & \quad\quad \mathsf{out}\ \mathit{net}\ \mathit{nonce}; \\
\quad\quad \mathsf{begin}\ \mathit{msg}; & \quad\quad \mathsf{inp}\ \mathit{net}\ (\mathit{ctext}); \\
\quad\quad \mathsf{out}\ \mathit{net}\ \{\mathit{msg},\mathit{nonce}\}_{\mathit{key}} & \quad\quad \mathsf{decrypt}\ \mathit{ctext} \\
 & \quad\quad\quad \mathsf{is}\ \{\mathit{msg},\mathit{nonce}'\}_{\mathit{key}}; \\
 & \quad\quad \mathsf{check}\ \mathit{nonce}\ \mathsf{is}\ \mathit{nonce}'; \\
 & \quad\quad \mathsf{end}\ \mathit{msg}
\end{array}
$$

The process $\mathsf{check}\ \mathit{nonce}\ \mathsf{is}\ \mathit{nonce}'; P$ checks that *nonce* and nonce' are the same name before executing P. For the sake of simplicity, in this example and others in the paper we omit error recovery code: upon receiving a ciphertext containing an unexpected nonce, an instance of the receiver just terminates. The whole system and its authenticity property are now:

$$
\begin{array}{l}
\mathit{FixedSystem}(\mathit{net}) \triangleq \\
\quad \mathsf{new}\ (\mathit{key}); \\
\quad\quad (\mathit{FixedSender}(\mathit{net},\mathit{key}) \mid \mathit{FixedReceiver}(\mathit{net},\mathit{key}))
\end{array}
$$

Authenticity: The process $\mathit{FixedSystem}(\mathit{net})$ is robustly safe.

Given our modifications, this property is true. A direct proof is possible, but tricky, since we must quantify over all possible attackers. The original paper on the spi-calculus includes a verification via equational reasoning of a protocol similar to that embodied in $\mathit{FixedSystem}(\mathit{net})$. The point of our type system, presented next, is to provide an efficient way of proving this specification, and others like it.

4 Typing Protocols

This section describes the heart of our method for analysing authenticity properties of protocols: a dependent type and effect system for statically verifying correspondence assertions by type-checking.

Section 4.1 and Section 4.2 explain informally how to type messages and how to ascribe effects to processes, respectively. We present the type and effect system formally in Section 4.3. Finally, in Section 4.4 we explain how to type the assertions in the example of the previous section.

4.1 Types for Messages

There is an objection in principle to a security analysis based on type-checking processes: it may be reasonable to assume that honest principals conform to typing rules, but it is imprudent to assume the same of the opponent. As previously discussed, our

general model of the opponent is any untyped, assertion-free process. The objection to a typed analysis is that we may miss attacks by ruling out processes that happen not to conform to our typing rules. On the internet, famously, nobody knows you're a dog. Likewise, nobody knows your code failed the type-checker.

To answer this objection, Abadi [1] introduces an *untrusted type* (which we call Un) for public messages, those exposed to the opponent. Every message and every opponent is typable if all their free variables are assigned the Un type. The type represents the unconstrained messages that an arbitrary process manipulates. Since any opponent can be typed in this trivial way we have not limited the power of opponents.

To illustrate this, here are some informal typing rules for messages and processes (for brevity, we elide some technical requirements on free names). Messages of the Un type may be output, input, paired, split apart, encrypted, and decrypted, with no constraints.

- If $M : \mathsf{Un}$ and $N : \mathsf{Un}$ then $\mathsf{out}\ M\ N$ is well-typed.
- If $M : \mathsf{Un}$ and P is well-typed then $\mathsf{inp}\ M\ (x{:}\mathsf{Un}); P$ is well-typed.
- If $M : \mathsf{Un}$ and $N : \mathsf{Un}$ then $(M, N) : \mathsf{Un}$.
- If $M : \mathsf{Un}$ and P is well-typed then $\mathsf{split}\ M\ \mathsf{is}\ (x{:}\mathsf{Un}, y{:}\mathsf{Un}); P$ is well-typed.
- If $M : \mathsf{Un}$ and $N : \mathsf{Un}$ then $\{M\}_N : \mathsf{Un}$.
- If $M : \mathsf{Un}$ and $N : \mathsf{Un}$ and P is well-typed then $\mathsf{decrypt}\ M\ \mathsf{is}\ \{x{:}\mathsf{Un}\}_N; P$ is well-typed.

When modelling protocols, we assume that all the names and messages exposed to the opponent—representing public data and channels—are of this type. Names and messages not publicly disclosed may be assigned other types, known as *trusted types*.

Messages of the trusted type $\mathsf{Key}(T)$ are symmetric keys for encrypting messages of type T. When encrypting with a $\mathsf{Key}(T)$, the plaintext must have type T, and the resulting ciphertext is given untrusted type. Using the rules above for Un, we can send and receive ciphertexts on untrusted channels. When decrypting with a $\mathsf{Key}(T)$, if we succeed we know the plaintext must have been encrypted with the same key, and therefore our typing rules assign it type T.

- If $M : T$ and $N : \mathsf{Key}(T)$ then $\{M\}_N : \mathsf{Un}$.
- If $M : \mathsf{Un}$ and $N : \mathsf{Key}(T)$ and P is well-typed then $\mathsf{decrypt}\ M\ \mathsf{is}\ \{x{:}T\}_N; P$ is well-typed.

The remaining trusted types are more standard. Messages of type $\mathsf{Ch}(T)$ are channels communicating data of type T. Messages of type $(x{:}T, U)$ are dependent pairs where the first element has type T and the second element has type U. The variable x is bound, and has scope U. (The need for such dependent types arises later, when we introduce a type for nonces.) The only message of the empty tuple type $()$ is the empty tuple $()$. Messages of type $T + U$ are tagged unions. A union of type $T + U$ is either of the form $\mathsf{inl}\ (M)$ where M has type T, or of the form $\mathsf{inr}\ (N)$ where N has type U. As a technical convenience, to simplify some abbreviations introduced in Appendix A.5, we introduce the empty type, $\mathsf{0}$. There are no messages of this type. Other base types

such as int or boolean could easily be added to this language: we expect they would produce no technical difficulties.

Types:

$T,U ::=$	type
Un	untrusted type
$\mathsf{Key}(T)$	shared-key type
$\mathsf{Ch}(T)$	channel type
$()$	empty tuple type
$(x{:}T,U)$	dependent pair type
$T+U$	variant type
$\mathsf{0}$	empty type

For example:

- $\mathsf{Key}(\mathsf{Un})$: key for encrypting untrusted data
- $\mathsf{Ch}(\mathsf{Un})$: channel for communicating untrusted data
- $\mathsf{Key}(\mathsf{Key}(\mathsf{Un}) + \mathsf{Ch}(\mathsf{Un}))$: key for encrypting either a key for encrypting untrusted data or a channel for communicating untrusted data

4.2 Effects for Processes

Our effect system tracks the unmatched end-assertions of a process. In its most basic form, our main judgment

$$P : [\mathsf{end}\ L_1, \ldots, \mathsf{end}\ L_n]$$

means that the effect $[\mathsf{end}\ L_1, \ldots, \mathsf{end}\ L_n]$, is an upper bound on the multiset (or unordered list) of end-events that P may assert without asserting a matching begin-event. Hence, if $P : []$ then every end-event in P has a matching begin-event, that is, P is safe.

Let e stand for an *atomic effect*. One kind of atomic effect is $\mathsf{end}\ L$. The second kind is $\mathsf{check}\ N$; we explain later its use to track nonce name-checking. 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.

Tracking Correspondences in Sequential Code

Given this notation, the typing rules for $\mathsf{begin}\ L;P$ and $\mathsf{end}\ L;P$ are essentially:

- If $P : es$ then $\mathsf{begin}\ L;P : (es - [\mathsf{end}\ L])$.
- If $P : es$ then $\mathsf{end}\ L;P : (es + [\mathsf{end}\ L])$.

These rules are enough to check correspondences in sequential code, for example:

- $\mathsf{end}\ L : [\mathsf{end}\ L]$

- $\mathsf{begin}\ L; \mathsf{end}\ L : []$
- $\mathsf{end}\ L; \mathsf{end}\ L : [\mathsf{end}\ L, \mathsf{end}\ L]$
- $\mathsf{begin}\ L; \mathsf{end}\ L; \mathsf{end}\ L : [\mathsf{end}\ L]$
- $\mathsf{begin}\ L; \mathsf{begin}\ L; \mathsf{end}\ L; \mathsf{end}\ L : []$

Transferring Effects between Parallel Processes

Our rules for assigning effects to communications and compositions are similar to those in previous work on effect systems for the π-calculus [11, 18].

- If $M : \mathsf{Ch}(T)$ and $N : T$ then $\mathsf{out}\ M\ N : []$.
- If $M : \mathsf{Ch}(T)$ and $P : es$ then $\mathsf{inp}\ M\ (x{:}T); P : es$.
- If $P : es_P$ and $Q : es_Q$ then $P \mid Q : (es_P + es_Q)$.

When computing the effect of the composition $P \mid Q$ of two processes, we simply compute the multiset union of the effects of the processes. This rule in itself does not allow a begin-assertion in P, say, to account for an end-assertion in Q. For example, the parallel composition $\mathsf{begin}\ L \mid \mathsf{end}\ L$ has effect $[\mathsf{end}\ L]$, while in contrast the sequential composition $\mathsf{begin}\ L; \mathsf{end}\ L$ has effect $[]$. In the parallel case, we cannot assume that the begin-event precedes the end-event so we must conservatively assign the effect $[\mathsf{end}\ L]$. In the sequential case, the syntax guarantees that the begin-event precedes the end-event so we can assign the effect $[]$. Somehow we need to be able to show that temporal precedences are established between parallel processes. Recall our *FixedSystem* example: we need to show that a distinct begin *msg* precedes each end *msg*, even though these assertions are running in parallel.

Typing Nonce Handshakes

A nonce handshake guarantees temporal precedence between events in parallel processes. In this paper, we consider a particular idiom for nonce handshakes, referred to by Guttman and Thayer as *incoming tests* [21]. Other idioms are possible, for example Guttman and Thayer's *outgoing tests*, but we leave these for future work. Incoming tests break down into several steps.

(1) The receiver creates a fresh nonce and publishes it.

(2) The sender embeds the nonce in a ciphertext.

(3) The receiver looks for the nonce in a received ciphertext. Finding the nonce encrypted under a shared private key proves the sender recently generated the ciphertext. If this is the first and only time the nonce is found, there is a one-to-one correspondence between finding the nonce and the creation of the ciphertext by the sender.

(4) To avoid vulnerability to replay of messages containing the nonce, the receiver subsequently discards the nonce and no longer looks for it.

We type-check these four steps as follows.

(1) The receiver creates the nonce N in the untrusted type Un. This allows the nonce to be sent on an untrusted channel, and reflects that it can be received and copied by the opponent as well as the sender.

(2) The sender embeds the nonce in a ciphertext as a message of a new trusted type $\mathsf{Nonce}\ es$, where es is an effect. The sender casts the nonce $N : \mathsf{Un}$ to this trusted type using the new process $\mathsf{cast}\ N\ \mathsf{is}\ (x{:}\mathsf{Nonce}\ es); P$. At runtime, this process simply binds the message N to the variable x of type $\mathsf{Nonce}\ es$, and then runs P. The sender uses the variable x to embed the nonce in the ciphertext.

(3) After decrypting a ciphertext containing a nonce $N' : \mathsf{Nonce}\ es$, the receiver uses a name-check $\mathsf{check}\ N\ \mathsf{is}\ N'; Q$ to check for the nonce $N : \mathsf{Un}$ which it made public earlier. Only a cast can populate the type $\mathsf{Nonce}\ es$. So the presence of the message $N' : \mathsf{Nonce}\ es$ proves there was a preceding execution of a cast process. Our type system ensures that at most one name-check process checks for the presence of each nonce $N : \mathsf{Un}$. Therefore, if the check succeeds, we are guaranteed a one-to-one correspondence between the check and the preceding process that cast N into the type $\mathsf{Nonce}\ es$. Note that the safety of this step relies on global agreement between the trusted participants as to the types of each of the messages.

(4) To guarantee that each nonce N is the subject of no more than one name-check, we introduce a new atomic effect, written $\mathsf{check}\ N$. In general, our main judgment takes the form,

$$P : [\mathsf{end}\ L_1, \ldots, \mathsf{end}\ L_m, \mathsf{check}\ N_1, \ldots, \mathsf{check}\ N_n]$$

and means the multiset $[\mathsf{end}\ L_1, \ldots, \mathsf{end}\ L_m]$ is an upper bound on the end-events P asserts without previously asserting a corresponding begin-event, and that the multiset $[\mathsf{check}\ N_1, \ldots, \mathsf{check}\ N_n]$ is an upper bound on the multiset of free nonces name-checked by P. We include $\mathsf{check}\ N$ in the effect of a name-check $\mathsf{check}\ N\ \mathsf{is}\ N'; Q$ on a nonce N. When checking name generation $\mathsf{new}\ (N{:}\mathsf{Un}); P$, we check that $\mathsf{check}\ N$ occurs at most once in the effect of P. This guarantees that each free name is the subject of no more than one name-check.

In summary, our type and effect system provides a solution to the problem of guaranteeing temporal precedences between parallel processes: for every successful execution of a process $\mathsf{check}\ N\ \mathsf{is}\ N'; Q$, where $N' : \mathsf{Nonce}\ es$, there is a distinct preceding execution of a process $\mathsf{cast}\ N\ \mathsf{is}\ (x{:}\mathsf{Nonce}\ es); P$, even if the name-check and the cast are in parallel processes.

The following rules for computing the effect of casts and name-checks exploit this temporal precedence. They allow us to guarantee by typing that those end-events following the name-check and listed in the effect es of the type $\mathsf{Nonce}\ es$ are matched by distinct begin-events that precede the cast. This effect is transferred from the name-check to the cast; the effect es is added to the effect of a cast, and is subtracted from the effect of a name-check.

- If $N : \mathsf{Un}$ and $P : es_P$ then $\mathsf{cast}\ N\ \mathsf{is}\ (x{:}\mathsf{Nonce}\ es); P : (es_P + es)$.

- If $N : \mathsf{Un}$ and $N' : \mathsf{Nonce}\ es$ and $Q : es_Q$
  then $\mathsf{check}\ N\ \mathsf{is}\ N'; Q : ((es_Q - es) + [\mathsf{check}\ N])$.
- If $P : es_P$ then $\mathsf{new}\ (N); P : (es_P - [\mathsf{check}\ N])$.

To illustrate these rules, we compute the effect of a nonce handshake that guarantees the safety of a correspondence between a begin-event labelled m in one process and an end-event with the same label in another. We consider fixed, global names m, n, and c. We assume m:T for some type T. We assume n:Un is the name of a nonce that somehow is already shared between the two processes. We assume c:Ch((Nonce [end m])) is the name of a trusted channel shared by the processes. (To focus on casting and checking nonces, we communicate the nonce n over the trusted channel c; in realistic examples, nonces are sent encrypted on untrusted channels.)

The first process

$$\begin{array}{rcl} P & = & \mathsf{begin}\ m; \\ & & \mathsf{cast}\ n\ \mathsf{is}\ (n'{:}\mathsf{Nonce}\ [\mathsf{end}\ m]); \\ & & \mathsf{out}\ c\ n' \end{array}$$

begins the correspondence, casts n into the type Nonce [end m], and then sends it on c.

We have $\mathsf{cast}\ n\ \mathsf{is}\ (n'{:}\mathsf{Nonce}\ [\mathsf{end}\ m]); \mathsf{out}\ c\ n' : [\mathsf{end}\ m]$ and therefore $P : []$. The second process

$$\begin{array}{rcl} Q & = & \mathsf{inp}\ c\ (x{:}\mathsf{Nonce}\ [\mathsf{end}\ m]); \\ & & \mathsf{check}\ n\ \mathsf{is}\ x; \\ & & \mathsf{end}\ m \end{array}$$

receives a name x off the channel c, checks that n equals x, and if so ends the correspondence.

We have $\mathsf{end}\ m : [\mathsf{end}\ m]$, and $\mathsf{check}\ n\ \mathsf{is}\ x; \mathsf{end}\ m : ([\mathsf{end}\ m] - [\mathsf{end}\ m]) + [\mathsf{check}\ n]$, and therefore $Q : [\mathsf{check}\ n]$. Now, by the rules for name generation and composition, we get that $R = \mathsf{new}\ (n{:}\mathsf{Un}); (P \mid Q) : []$. So R is safe.

On the other hand, consider the process $R' = \mathsf{new}\ (n{:}\mathsf{Un}); (P' \mid Q \mid Q)$ where we have duplicated Q and where the process

$$\begin{array}{rcl} P' & = & \mathsf{begin}\ m; \\ & & \mathsf{cast}\ n\ \mathsf{is}\ (n'{:}\mathsf{Nonce}\ [\mathsf{end}\ m]); \\ & & (\mathsf{out}\ c\ n' \mid \mathsf{out}\ c\ n') \end{array}$$

is a variation of P' that duplicates the nonce. Now, R' is unsafe, because the two copies of Q can each receive one of the duplicate nonces sent by P'. Therefore both can assert an end-event, but only one is accounted for by the begin-assertion by P'. The process R' does not type-check, because it name-checks the nonce n more than once. We can derive $P' : []$, but the whole process R' fails the rule for name generation, because process $P' \mid Q \mid Q$ has effect $[\mathsf{check}\ n, \mathsf{check}\ n]$ so the condition $n \notin fn([\mathsf{check}\ n, \mathsf{check}\ n] - [\mathsf{check}\ n])$ is false.

Effects and Atomic Effects

Given these motivations for and examples of assigning effects to processes, here is the grammar of effects and atomic effects.

Effects:

$e, f ::=$	atomic effect
$\quad \mathsf{end}\ L$	$\quad$ end-event labelled with message L
$\quad \mathsf{check}\ N$	$\quad$ name-check for a nonce N
$es, fs ::=$	effect
$\quad [e_1, \ldots, e_n]$	$\quad$ multiset of atomic effects

Free names, $fn(es)$, of an effect es:

$fn(\mathsf{end}\ L) \triangleq fn(L)$
$fn(\mathsf{check}\ N) \triangleq fn(N)$
$fn([e_1, \ldots, e_n]) \triangleq fn(e_1) \cup \cdots \cup fn(e_n)$

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.

Additional Types and Processes

We end this section by completing the grammars of types and processes with the new type and new processes we need for typing nonce handshakes. We add a type for nonces, and we give rules defining the set $fn(T)$ of any type T.

Types:

$T, U ::=$	type
$\quad \ldots$	$\quad$ as in Section 4.1
$\quad \mathsf{Nonce}\ es$	$\quad$ nonce type

Free names, $fn(T)$, of a type T:

$fn(\mathsf{Ch}(T)) \triangleq fn(T)$
$fn((x{:}T, U)) \triangleq fn(T) \cup (fn(U) - \{x\})$
$fn(()) \triangleq \varnothing$
$fn(T + U) \triangleq fn(T) \cup fn(U)$
$fn(\mathsf{0}) \triangleq \varnothing$
$fn(\mathsf{Un}) \triangleq \varnothing$
$fn(\mathsf{Key}(T)) \triangleq fn(T)$
$fn(\mathsf{Nonce}\ es) \triangleq fn(es)$

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.

As we explained, we add a process to cast untrusted data into nonce type. Moreover, we add a new process for pattern matching pairs.

Processes:

$O,P,Q,R ::=$	process
...	as in Sections 2.1 and 3.1
cast M is $(x{:}T);P$	cast to nonce type
match M is $(N,y{:}U);P$	pair pattern matching

In a process cast M is $(x{:}T);P$, the name x is bound; its scope is the process P. In a process match M is $(N,y{:}U);P$, the name y is bound; its scope of the process P.

- The process cast M is $(x{:}T);P$ casts the message M to the type T, by binding the variable x to M, and then running P. (This process can only be typed by our type system if T is of the form Nonce es.)
- The process match M is $(N,y);P$ is similar to split M is $(x,y);P$ except that it checks that the first component of M is equal to N before extracting the second component (which is bound to y in P). If the equality test fails, then the process deadlocks.

Pair pattern matching is a generalization of π-calculus name equality testing, since $[M=N]P$ can be written match $(M,())$ is $(N,y);P$.

Free names of a process $fn(P)$:

$fn(\text{cast } M \text{ is } (x{:}T);P) \triangleq fn(M) \cup fn(T) \cup (fn(P) - \{x\})$

$fn(\text{match } M \text{ is } (N,y{:}U);P) \triangleq fn(M) \cup fn(N) \cup fn(U) \cup (fn(P) - \{y\})$

Pair pattern matching is used in the protocol examples in Appendix A.

4.3 Typing Rules

In this section, we formally define the judgments of our type and effect system.

These judgments all depend on an *environment*, E, that defines the types of all variables in scope. An environment takes the form $x_1{:}T_1,\ldots,x_n{:}T_n$ and defines the type T_i for each variable x_i. The *domain*, $dom(E)$, of an environment E is the set of variables whose types it defines.

Environments:

$D,E ::=$	environment
$\varnothing$	empty
$E,x{:}T$	entry
$dom(x_1{:}T_1,\ldots,x_n{:}T_n) \triangleq \{x_1,\ldots,x_n\}$	domain of an environment

The following are the five judgments of our type and effect system. They are inductively defined by rules presented in the following tables.

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

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

Rules for Environments:

(Env $\varnothing$) $\dfrac{}{\varnothing \vdash \diamond}$ (Env x) (where $x \notin dom(E)$) $\dfrac{E \vdash T}{E, x{:}T \vdash \diamond}$

These standard rules define an environment $x_1{:}T_1, \ldots, x_n{:}T_n$ to be well-formed just if each of the names $x_1, \ldots, x_n$ are distinct, and each of the types T_i is well-formed.

Rules for Effects:

(Effect $\varnothing$) $\dfrac{E \vdash \diamond}{E \vdash \varnothing}$ (Effect End) $\dfrac{E \vdash es \quad E \vdash L : T}{E \vdash es + [\mathsf{end}\ L]}$ (Effect Check) $\dfrac{E \vdash es \quad E \vdash N : \mathsf{Un}}{E \vdash es + [\mathsf{check}\ N]}$

These rules define an effect $[e_1, \ldots, e_n]$ to be well-formed just if for each atomic effect $e_i = \mathsf{end}\ L$, message L has type T for some type T, and for each atomic effect $e_i = \mathsf{check}\ N$, message N has type Un.

Rules for Types:

(Type Un) $\dfrac{E \vdash \diamond}{E \vdash \mathsf{Un}}$ (Type Chan) $\dfrac{E \vdash T}{E \vdash \mathsf{Ch}(T)}$ (Type Pair) $\dfrac{E, x{:}T \vdash U}{E \vdash (x{:}T, U)}$ (Type Unit) $\dfrac{E \vdash \diamond}{E \vdash ()}$

(Type Variant) $\dfrac{E \vdash T \quad E \vdash U}{E \vdash T + U}$ (Type Empty) $\dfrac{E \vdash \diamond}{E \vdash 0}$ (Type Key) $\dfrac{E \vdash T}{E \vdash \mathsf{Key}(T)}$ (Type Nonce) $\dfrac{E \vdash es}{E \vdash \mathsf{Nonce}\ es}$

According to these rules a type is well-formed just if every effect occurring in the type is itself well-formed.

Next, we present the rules for deriving the judgment $E \vdash M : T$ that assigns a type T to a message M. We split the rules into three tables: first, the rule for variables; second, rules for manipulating data of trusted type; and third, rules for assigning the untrusted type to arbitrary messages.

Rule for Variables:

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

Rules for Messages of Trusted Type:

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

(Msg Unit)
$$\frac{E \vdash \diamond}{E \vdash () : ()}$$

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

Rules for Messages of Untrusted Type:

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

(Msg Unit Un)
$$\frac{E \vdash \diamond}{E \vdash () : \mathsf{Un}}$$

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

Recall from Section 4.1 the principle that any message can be assigned the untrusted type Un, provided its free variables are also untrusted. Using just the rules in the first and third tables of message typing rules, we can prove:

Lemma 1 *If* $fn(M) \subseteq \{x_1, \ldots, x_n\}$ *then* $x_1{:}\mathsf{Un}, \ldots, x_n{:}\mathsf{Un} \vdash M : \mathsf{Un}$.

Proof By structural induction on the message M. □

A message may be assigned both a trusted and an untrusted type. For example:

- $x{:}\mathsf{Un}, y{:}\mathsf{Un} \vdash (x,y){:}(z{:}\mathsf{Un}, \mathsf{Un})$ by (Msg Pair)
- $x{:}\mathsf{Un}, y{:}\mathsf{Un} \vdash (x,y){:}\mathsf{Un}$ by (Msg Pair Un)

Finally, we present the rules for assigning effects to processes. To state the rule for name-generation we introduce the notion of a *generative type*. A type is generative if it is untrusted or if it is a key or channel type. A process $\mathsf{new}\ (x{:}T); P$ is only well-typed if T is generative. This rule prevents the fresh generation of names of, for example, the $\mathsf{Nonce}\ es$ type; it is crucial to our system that the only way of populating this type is via a cast process.

Generative Types:

A type is *generative* if and only if
it takes the form $\mathsf{Ch}(T)$, Un, or $\mathsf{Key}(T)$.

Basic Rules for Processes:

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

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

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

(Proc Repeat)
$$\frac{E \vdash P : []}{E \vdash \mathsf{repeat}\ P : []}$$

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

(Proc Res) (where $x \notin fn(es - [\mathsf{check}\ x])$)
$$\frac{E, x{:}T \vdash P : es \quad T \text{ is generative}}{E \vdash \mathsf{new}\ (x{:}T); P : es - [\mathsf{check}\ x]}$$

(Proc Subsum)
$$\frac{E \vdash P : es \quad E \vdash es'}{E \vdash P : es + es'}$$

We discussed informal versions of the rules (Proc Begin), (Proc End), (Proc Par), and (Proc Res) previously. The rule (Proc Repeat) requires the effect of the replicated process P to be empty. If P had a non-empty effect, then somehow we might assign an infinite effect to $\mathsf{repeat}\ P$ but this would not be useful. Assigning an effect to a whole process is useful because if the effect is empty then the process is safe. Any process enclosing $\mathsf{repeat}\ P$ can only match a finite number of atomic effects arising from $\mathsf{repeat}\ P$, and so must have a non-empty effect. So typing $\mathsf{repeat}\ P$ is only useful if P has an empty effect. The rule (Proc Stop) says the inactive process has empty effect. The effect of a process is an upper bound on the behaviour of a process; the rule (Proc Subsum) allows us to weaken this upper bound by enlarging the effect.

The rule (Proc Case), in the following table, uses an operator $\vee$ defined as follows. Let the multiset ordering $es \leq es'$ mean there is an effect es'' such that $es + es'' = es'$. Then we write $es \vee es'$ for the least effect es'' in this ordering such that both $es \leq es''$ and $es' \leq es''$. Note that $(es \vee es') = ((es - es') + es')$.

Rules for Processes Manipulating Trusted Types:

(Proc Output)
$$\frac{E \vdash x : \mathsf{Ch}(T) \quad E \vdash M : T}{E \vdash \mathsf{out}\ x\ M : []}$$

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

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

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

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

(Proc Decrypt) (where $x \notin fn(es)$)
$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash y : \mathsf{Key}(T) \quad E, x{:}T \vdash P : es}{E \vdash \mathsf{decrypt}\ M\ \mathsf{is}\ \{x{:}T\}_y; P : es}$$

(Proc Cast) (where $x \notin fn(es)$)
$$\frac{E \vdash M : \mathsf{Un} \quad E, x{:}\mathsf{Nonce}\ fs \vdash P : es}{E \vdash \mathsf{cast}\ M\ \mathsf{is}\ (x{:}\mathsf{Nonce}\ fs); P : es + fs}$$

(Proc Check)
$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{Nonce}\ fs \quad E \vdash P : es}{E \vdash \mathsf{check}\ M\ \mathsf{is}\ N; P : (es - fs) + [\mathsf{check}\ M]}$$

We discussed informal versions of the rules (Proc Input), (Proc Output), (Proc Cast), and (Proc Check) previously. Rule (Proc Split) is a standard rule to allow a pair $M : (x{:}T, U)$ to be split into two components named $x{:}T$ and $y{:}U$, where x may occur free in the type U. The conditions $x \notin fn(es)$ and $y \notin fn(es)$ prevent the bound variables x and y from appearing out of scope in the effect es. In the rule (Proc Match), the message $N : T$ is meant to match the first component of the pair $M : (x{:}T, U)$, and the variable $y{:}U$ gets bound to the second component. Again, the condition $y \notin fn(es)$ prevents y from appearing out of scope in es. The rule (Proc Case) is a standard rule for checking inspections of tagged unions. In the rule (Proc Decrypt), the ciphertext M is of untrusted type, Un, the key y is of type $\mathsf{Key}(T)$, and the plaintext, bound to x, has type T. The condition $x \notin fn(es)$ prevents x from appearing out of scope in the effect es.

Rules for Processes Manipulating Untrusted Types:

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

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

(Proc Match Un) (where $y \notin fn(es)$)
$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{Un} \quad E, y{:}\mathsf{Un} \vdash P : es}{E \vdash \mathsf{match}\ M\ \mathsf{is}\ (N, y{:}\mathsf{Un}); P : es}$$

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

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

(Proc Cast Un) (where $x \notin fn(es)$)
$$\frac{E \vdash M : \mathsf{Un} \quad E, x{:}\mathsf{Un} \vdash P : es}{E \vdash \mathsf{cast}\ M\ \mathsf{is}\ (x{:}\mathsf{Un}); P : es}$$

(Proc Check Un)
$$\frac{E \vdash M : \mathsf{Un} \quad E \vdash N : \mathsf{Un} \quad E \vdash P : es}{E \vdash \mathsf{check}\ M\ \mathsf{is}\ N; P : es}$$

These rules are similar to those in the previous table in how they compute effects of processes, but differ in that all messages are of untrusted type. These rules are needed to type-check opponents.

Our rules for processes conform to the principle, stated in Section 4.1, that any opponent can be typed if all its free variables are assigned the type Un.

Lemma 2 (Opponent Typability) *If O is an opponent, that is, an untyped, assertion-free process, and $fn(O) \subseteq \{x_1, \ldots, x_n\}$ then $x_1{:}\mathsf{Un}, \ldots, x_n{:}\mathsf{Un} \vdash O : []$.*

Proof By structural induction on O, with appeal to Lemma 1. □

The following theorem, proved in Appendix B, says a process is safe if it can be assigned the empty effect.

Theorem 1 (Safety) *If $E \vdash P : [\,]$ then P is safe.*

Combined, Lemma 2 (Opponent Typability) and Theorem 1 (Safety) establish our main result, that our type and effect system guarantees robust safety.

Theorem 2 (Robust Safety) *If $x_1{:}\mathsf{Un}, \ldots, x_n{:}\mathsf{Un} \vdash P : [\,]$ then P is robustly safe.*

Proof For any untyped, assertion-free O, find $x_{n+1}, \ldots, x_{n+m}$ such that $fn(O) \subseteq \{x_1, \ldots, x_{n+m}\}$. By Lemma 2 (Opponent Typability), we have $x_1 : \mathsf{Un}, \ldots, x_{n+m} : \mathsf{Un} \vdash O : [\,]$. By a standard weakening lemma, proved in the full version, $x_1{:}\mathsf{Un}, \ldots, x_n{:}\mathsf{Un} \vdash P : [\,]$ implies $x_1{:}\mathsf{Un}, \ldots, x_{n+m}{:}\mathsf{Un} \vdash P : [\,]$. So by rule (Proc Par) we have $x_1 : \mathsf{Un}, \ldots, x_{n+m} : \mathsf{Un} \vdash P \mid O : [\,]$, and so by Theorem 1 (Safety), $P \mid O$ is safe. Thus, P is robustly safe. □

4.4 Typing the Example

Our example $\mathit{FixedSystem}(\mathit{net})$ from Section 3.3 uses a nonce handshake over the public channel *net* to transfer messages from the sender to the receiver. Here we show how to prove the example's correspondence assertions by choosing suitable types and adding a cast process.

The sender receives a nonce *nonce* off the public channel *net*, performs a begin-event to indicate it is sending a message *msg*, embeds the nonce and the message in a ciphertext encrypted with the shared key *key*, and returns the ciphertext to the receiver on *net*. Any public channel should be accessible to the opponent, so we assign *net* the untrusted type Un, and since *nonce* is sent on these channels, they too must have the untrusted type. We fix some arbitrary type *Msg* and assume each *msg* is of this type. To type-check the correspondence between begin- and end-assertions made by the sender and receiver, respectively, we add a cast process to the sender to cast the nonce into the type $\mathsf{Nonce}\ [\mathsf{end}\ \mathit{msg}]$. Therefore, the shared key has type $\mathsf{Key}(\mathit{msg}{:}\mathit{Msg}, \mathit{nonce}{:}\mathsf{Nonce}\ [\mathsf{end}\ \mathit{msg}])$; the first component of the ciphertext is the actual message, and the second component is a nonce proving it is safe to assert an $\mathsf{end}\ \mathit{msg}$ event.

Therefore, we introduce the types

$$\begin{aligned}
&\mathit{Msg} \text{ some arbitrary type}\\
&\mathit{Network} \triangleq \mathsf{Un}\\
&\mathit{MyNonce}\ (\mathit{msg}) \triangleq \mathsf{Nonce}\ [\mathsf{end}\ \mathit{msg}]\\
&\mathit{MyKey} \triangleq \mathsf{Key}(\mathit{msg}{:}\mathit{Msg}, \mathit{nonce}{:}\mathit{MyNonce}\ (\mathit{msg}))
\end{aligned}$$

and we type the sender as follows, where we display the effects of bracketed subprocesses to the right.

$$\begin{array}{l}
\mathit{TypedSender}(\mathit{net}{:}\mathit{Network}, \mathit{key}{:}\mathit{MyKey}) : [\,] \triangleq \\
\quad \mathsf{repeat} \\
\quad\quad \left.\begin{array}{l}
\mathsf{inp}\ \mathit{net}\ (\mathit{nonce}{:}\mathsf{Un}); \\
\mathsf{new}\ (\mathit{msg}{:}\mathit{Msg}); \\
\left.\begin{array}{l}
\mathsf{begin}\ \mathit{msg}; \\
\left.\begin{array}{l}
\mathsf{cast}\ \mathit{nonce} \\
\quad \mathsf{is}\ (\mathit{nonce}'{:}\mathit{MyNonce}\ (\mathit{msg})); \\
\mathsf{out}\ \mathit{net}\ \{\mathit{msg}, \mathit{nonce}'\}_{\mathit{key}}\ [\,]
\end{array}\right\} [\mathsf{end}\ \mathit{msg}]
\end{array}\right\} [\,]
\end{array}\right\} [\,]
\end{array}$$

Next, we type the receiver. Like the sender, it is effect-free, that is, it can be assigned the empty effect.

$$
\begin{array}{l}
\mathit{TypedReceiver}(\mathit{net}{:}\mathit{Network}, \mathit{key}{:}\mathit{MyKey}) : [] \triangleq \\
\quad \mathsf{repeat} \\
\qquad \left.\begin{array}{l}
\mathsf{new}\ (\mathit{nonce}{:}\mathsf{Un}); \\
\mathsf{out}\ \mathit{net}\ \mathit{nonce}; \\
\mathsf{inp}\ \mathit{net}\ (\mathit{ctext}{:}\mathsf{Un}); \\
\mathsf{decrypt}\ \mathit{ctext} \\
\qquad \mathsf{is}\ \{\mathit{msg}{:}\mathit{Msg}, \mathit{nonce}'{:}\mathit{MyNonce}\ (\mathit{msg})\}_{\mathit{key}}; \\
\left.\begin{array}{l}\mathsf{check}\ \mathit{nonce}\ \mathsf{is}\ \mathit{nonce}'; \\ \mathsf{end}\ \mathit{msg}\end{array}\right\}[\mathsf{check}\ \mathit{nonce}]
\end{array}\right\}[]
\end{array}
$$

Since the sender and receiver are both effect-free, the whole system is also effect-free:

$$
\begin{array}{l}
\mathit{TypedSystem}(\mathit{net}{:}\mathit{Network}) : [] \triangleq \\
\quad \mathsf{new}\ (\mathit{key}{:}\mathit{MyKey}); \\
\qquad (\mathit{TypedSender}(\mathit{net}, \mathit{key}) \mid \mathit{TypedReceiver}(\mathit{net}, \mathit{key}))
\end{array}
$$

By Theorem 2 (Robust Safety), it follows that $\mathit{TypedSystem}(\mathit{net}{:}\mathit{Network})$ is robustly safe. This proves the following authenticity property by typing.

Authenticity: The process $\mathit{TypedSystem}(\mathit{net})$ is robustly safe.

5 Further Protocol Examples

We have applied our method to several cryptographic protocols from the literature. We verified some protocols, found flaws in others, but also found at least one incompleteness in our method. Details are in an appendix, but we can summarise our experience as follows.

- Abadi and Gordon [3] propose a nonce-based variation of the Wide Mouth Frog key-exchange protocol [9]. We can verify authenticity properties of Abadi and Gordon's protocol by typing. Abadi and Gordon prove an equationally-specified authenticity property by constructing a bisimulation relation based on an elaborate invariant; our proof of correspondence assertions by typing took considerably less time.
- Woo and Lam [40] propose a nonce-based authentication protocol. Trying to type-check the protocol exposes known flaws in the protocol and suggests a known simplification [4, 5].
- Otway and Rees [33] propose another nonce-based key exchange protocol. The nonces used by the protocol to prove freshness are kept secret; hence the protocol does not fit the idiom that can be checked by our type system. Still, we can type-check a more efficient version of the protocol suggested by Abadi and Needham [4]. The typing suggests a further simplification.

In each case, there is a spi-calculus representation of the protocol in which there are arbitrarily many participant principals and arbitrarily many sessions.

6 Summary and Conclusion

To summarise, we reviewed the spi-calculus, a formalism for precisely describing the behaviour of security protocols based on cryptography. We embedded Woo and Lam's correspondence assertions in spi as a way of specifying authenticity properties. We devised a new type and effect system that proves authenticity properties, simply by type-checking.

To conclude, the examples in this paper, together with others we have investigated, suggest that this is a promising technique for checking protocols, since it requires little human effort to type a protocol, and the types of protocol data document how the protocol works.

Acknowledgements

Thanks to Martín Abadi, Gavin Lowe, Dusko Pavlovic, Simon Peyton Jones, Benjamin Pierce, Corin Pitcher, James Riely, and Andre Scedrov for discussions about this work. The anonymous referees for the *IEEE Computer Security Foundations Workshop* provided invaluable feedback. C.A.R. Hoare suggested several improvements to a draft. Alan Jeffrey was supported in part by Microsoft Research during some of the time we worked on this paper.

A Protocol Examples

In this appendix we describe details of the examples mentioned in Section 5. Section A.1 describes Abadi and Gordon's version of Wide Mouth Frog. Section A.2 discusses Woo and Lam's authentication protocol. Section A.3 discusses Otway and Rees's key-exchange protocol. Finally, we present a new typed protocol for secure message streams in Section A.4.

Abbreviations Used in Examples

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

- Dependent record types $(x_1{:}T_1, \ldots, x_n{:}T_n)$, rather than just pairs. These come with a constructor $(M_1, \ldots, M_n)$ and a destructor $\mathsf{match}\ M\ \mathsf{is}\ (x_1{:}T_1, \ldots, x_n{:}T_n); P$.
- Tagged union types $(\ell_1(T_1) \mid \cdots \mid \ell_n(T_n))$ rather than just binary choice $T + U$. These come with a constructor $\ell_i(M)$ and a destructor $\mathsf{match}\ M\ \mathsf{is}\ \ell_i(x{:}T); P$.
- Dependent function types $(x{:}T) \to U$. These come with an appropriate function declaration and application syntax.

We show in Section A.5 that these constructs can be derived from our base language.

A.1 Abadi and Gordon's Variant of Wide Mouth Frog

The original paper on the spi-calculus [3] includes a lengthy proof of authenticity and secrecy properties for a variation of the Wide Mouth Frog key distribution protocol [9] based on nonce handshakes instead of timestamps. In this section, we show how to type-check this protocol.

To begin with we look at an unsafe version of the protocol, to illustrate how attempting to type-check a protocol may expose flaws. This broken protocol consists of a sender (Alice), a receiver (Bob) and a server (Sam). Alice wishes to contact Bob, and asks Sam to establish her credentials:

Event 1	A begins	"A sending B key K_{AB}"
Message 1	$A \to S$	A
Message 2	$S \to A$	N_S
Message 3	$A \to S$	$A, \{B, K_{AB}, N_S\}_{K_{AS}}$
Message 4	$S \to B$	$()$
Message 5	$B \to S$	N_B
Message 6	$S \to B$	$\{A, K_{AB}, N_B\}_{K_{BS}}$
Event 2	B ends	"A sending B key K_{AB}"

(For the sake of readability, we use "A sending B key K_{AB}" as a shorthand for the message (A, B, K_{AB}).)

This protocol can be compromised by an intruder I impersonating Sam, if Alice

acts both as a sender and a receiver:

Event $\alpha.1$	A begins	"A sending B key K_{AB}"
Message $\alpha.1$	$A \rightarrow I$	A
Message $\beta.4$	$I \rightarrow A$	$()$
Message $\beta.5$	$A \rightarrow I$	N_A
Message $\alpha.2$	$I \rightarrow A$	N_A
Message $\alpha.3$	$A \rightarrow I$	$A, \{B, K_{AB}, N_A\}_{K_{AS}}$
Message $\beta.6$	$I \rightarrow A$	$\{B, K_{AB}, N_A\}_{K_{AS}}$
Event $\beta.2$	A ends	"B sending A key K_{AB}"

At this point, Alice believes that she has been contacted by Bob, when in fact she has been contacted by the intruder.

We can easily express this protocol in the spi-calculus, and use begin M and end M statements to specify the desired correspondence property.

We define $FlawedSender(net, alice, key)$ to be the sender, using *net* as the insecure communications medium, acting on behalf of *alice* using secret key *key* (in order to bootstrap the system, we have the sender receive *bob*'s name from the network, so the attacker can create as many concurrent sessions as they like):

```
FlawedSender(
  net:Network, alice:Princ, key:WMFKey(alice)
) ≜
  repeat
    inp net (bob:Princ);
    new (sKey:SKey);
    begin "alice sending bob key sKey";
    out net (alice);
    inp net (nonce:Un);
    cast nonce is (nonce':WMFNonce(alice, bob, sKey));
    out net (alice, {bob, sKey, nonce'}key);
```

We define $FlawedReceiver(net, bob, key)$ to be the receiver, using *net* as the insecure

communications medium, acting on behalf of *bob*, using secret key *key*:

$$
\begin{array}{l}
\mathit{FlawedReceiver}(\\
\quad \mathit{net}{:}\mathit{Network}, \mathit{bob}{:}\mathit{Princ}, \mathit{key}{:}\mathit{WMFKey}(\mathit{bob}) \\
) \triangleq \\
\quad \mathsf{repeat} \\
\qquad \mathsf{inp}\ \mathit{net}\ (); \\
\qquad \mathsf{new}\ (\mathit{nonce}{:}\mathsf{Un}); \\
\qquad \mathsf{out}\ \mathit{net}\ (\mathit{nonce}); \\
\qquad \mathsf{inp}\ \mathit{net}\ (\mathit{ctext}{:}\mathsf{Un}); \\
\qquad \mathsf{decrypt}\ \mathit{ctext} \\
\qquad\quad \mathsf{is}\ \{\mathit{alice}{:}\mathit{Princ}, \\
\qquad\qquad \mathit{sKey} : \mathit{SKey}, \\
\qquad\qquad \mathit{nonce}' : \mathit{WMFNonce}(\mathit{alice}, \mathit{bob}, \mathit{sKey})\}_{\mathit{key}}; \\
\qquad \mathsf{check}\ \mathit{nonce}\ \mathsf{is}\ \mathit{nonce}'; \\
\qquad \mathsf{end}\ \text{“}\mathit{alice}\ \text{sending}\ \mathit{bob}\ \text{key}\ \mathit{sKey}\text{”}
\end{array}
$$

We define $\mathit{FlawedServer}(\mathit{net}, \mathit{lookup})$ to be the server, using *net* as the insecure communications medium, making use of a trusted database lookup function *lookup* to access the secret keys:

$$
\begin{array}{l}
\mathit{FlawedServer}(\mathit{net}{:}\mathit{Network}, \mathit{lookup}{:}\mathit{WMFLookup}) \triangleq \\
\quad \mathsf{repeat} \\
\qquad \mathsf{inp}\ \mathit{net}\ (\mathit{alice}{:}\mathit{Princ}); \\
\qquad \mathsf{new}\ (\mathit{nonceA}{:}\mathsf{Un}); \\
\qquad \mathsf{out}\ \mathit{net}\ (\mathit{nonceA}); \\
\qquad \mathsf{inp}\ \mathit{net}\ (\mathit{alice}, \mathit{ctext}{:}\mathsf{Un}); \\
\qquad \mathsf{let}\ \mathit{keyA} : \mathit{WMFKey}(\mathit{alice}) = \mathit{lookup}\,(\mathit{alice}); \\
\qquad \mathsf{decrypt}\ \mathit{ctext} \\
\qquad\quad \mathsf{is}\ \{\mathit{bob}{:}\mathit{Princ}, \\
\qquad\qquad \mathit{sKey} : \mathit{SKey}, \\
\qquad\qquad \mathit{nonceA}' : \mathit{WMFNonce}(\mathit{alice}, \mathit{bob}, \mathit{sKey})\}_{\mathit{keyA}}; \\
\qquad \mathsf{check}\ \mathit{nonceA}\ \mathsf{is}\ \mathit{nonceA}'; \\
\qquad \mathsf{out}\ \mathit{net}\ (); \\
\qquad \mathsf{inp}\ \mathit{net}\ (\mathit{nonceB}{:}\mathsf{Un}); \\
\qquad \mathsf{cast}\ \mathit{nonceB} \\
\qquad\quad \mathsf{is}\ (\mathit{nonceB}'{:}\mathit{WMFNonce}(\mathit{alice}, \mathit{bob}, \mathit{sKey})); \\
\qquad \mathsf{let}\ \mathit{keyB} : \mathit{WMFKey}(\mathit{bob}) = \mathit{lookup}\,(\mathit{bob}); \\
\qquad \mathsf{out}\ \mathit{net}\ \{\mathit{alice}, \mathit{sKey}, \mathit{nonceB}'\}_{\mathit{keyB}}
\end{array}
$$

Then we can try to define the types appropriately. For most of the types, it is fairly routine(for the *WMFLookup* type, we need to use an appropriate function type, and for the *SKey* type, we need an appropriate *Msg* type for the payload, but these do not play

an important role in the typing) :

$$
\begin{array}{l}
\mathit{Network} \triangleq \mathsf{Un} \\
\mathit{Princ} \triangleq \mathsf{Un} \\
\mathit{WMFLookup} \triangleq (\mathit{princ}{:}\mathit{Princ}) \to \mathit{WMFKey}(\mathit{princ}) \\
\mathit{SKey} \triangleq \mathsf{Key}(\mathit{Msg}) \\
\mathit{WMFNonce}(\mathit{alice}, \mathit{bob}, \mathit{sKey}) \triangleq \\
\quad \mathsf{Nonce}\ [\mathsf{end}\ \text{“}\mathit{alice}\ \text{sending}\ \mathit{bob}\ \text{key}\ \mathit{sKey}\text{”}] \\
\mathit{WMFKey}(\mathit{princ}) \triangleq \mathsf{Key}(\mathit{WMFMsg}(\mathit{princ}))
\end{array}
$$

The problem comes when we try to give a definition for *WMFMsg*, which is the type of the plaintext of messages used in the WMF protocol. In order to type-check Message 3, we require:

$$
\begin{array}{l}
\mathit{WMFMsg}(\mathit{alice}) = \\
\quad (\mathit{bob}{:}\mathit{Princ}, \mathit{sKey}{:}\mathit{SKey}, \mathit{nonce}{:}\mathit{WMFNonce}(\mathit{alice}, \mathit{bob}, \mathit{sKey}))
\end{array}
$$

and in order to type-check Message 6, we require:

$$
\begin{array}{l}
\mathit{WMFMsg}(\mathit{bob}) = \\
\quad (\mathit{alice}{:}\mathit{Princ}, \mathit{sKey}{:}\mathit{SKey}, \mathit{nonce}{:}\mathit{WMFNonce}(\mathit{alice}, \mathit{bob}, \mathit{sKey}))
\end{array}
$$

Unfortunately, these requirements are inconsistent, since the roles of *alice* and *bob* have been swapped. This is the root of the attack on this broken WMF, which relies on the fact that the key for *alice* is being used in two incompatible ways, depending on whether *alice* is acting as the sender or the receiver.

This is an example of a type-flaw attack [23] and may be solved by the standard solution of adding tag information to messages. This is akin to the use of tagged union types in type-safe languages like ML or Haskell. In this case, we have the type for Message 3 of the protocol:

$$
\begin{array}{l}
\mathit{WMFMsg}_3(\mathit{alice}) \triangleq \\
\quad (\mathit{bob}{:}\mathit{Princ}, \mathit{sKey}{:}\mathit{SKey}, \mathit{nonce}{:}\mathit{WMFNonce}(\mathit{alice}, \mathit{bob}, \mathit{sKey}))
\end{array}
$$

and the type for Message 6:

$$
\begin{array}{l}
\mathit{WMFMsg}_6(\mathit{bob}) \triangleq \\
\quad (\mathit{alice}{:}\mathit{Princ}, \mathit{sKey}{:}\mathit{SKey}, \mathit{nonce}{:}\mathit{WMFNonce}(\mathit{alice}, \mathit{bob}, \mathit{sKey}))
\end{array}
$$

and we can define $\mathit{WMFMsg}(\mathit{princ})$ as the tagged union of these two types:

$$
\begin{array}{l}
\mathit{WMFMsg}(\mathit{princ}) \triangleq \\
\quad (\mathit{msg}_3(\mathit{WMFMsg}_3(\mathit{princ})) \mid \mathit{msg}_6(\mathit{WMFMsg}_6(\mathit{princ})))
\end{array}
$$

We can then check that the safe versions of the principals are effect-free. The sender, receiver, and server are given in Figure 1.

The key database has to implement the *lookup* function, and be effect-free. In practice, an implementation would require access to a secure database, but in this example,

we can just hard-wire in the principal names and keys, and use pattern-matching to define the database:

$$\begin{array}{l} \mathit{KeyDB}(\mathit{lookup}{:}\mathit{WMFLookup}, \mathit{princ}_1{:}\mathit{Princ}, \mathit{key}_1{:}\mathit{WMFKey}(\mathit{princ}_1), \ldots, \\ \qquad \mathit{princ}_n{:}\mathit{Princ}, \mathit{key}_n{:}\mathit{WMFKey}(\mathit{princ}_n)) \stackrel{\Delta}{=} \\ \quad \mathsf{function} \\ \qquad \mathit{lookup}\,(\mathit{princ}_1) : \mathit{WMFKey}(\mathit{princ}_1)\ \mathsf{is\ return}\ \mathit{key}_1 \\ \qquad \vdots \\ \qquad \mathit{lookup}\,(\mathit{princ}_n) : \mathit{WMFKey}(\mathit{princ}_n)\ \mathsf{is\ return}\ \mathit{key}_n \end{array}$$

We define a *Wide Mouth Frog configuration* to be a process of the form:

$$\begin{array}{l} \mathsf{new}\ (\mathit{lookup}{:}\mathit{WMFLookup}); \\ \mathsf{new}\ (\mathit{princ}_1{:}\mathit{Princ}); \ldots \\ \mathsf{new}\ (\mathit{princ}_n{:}\mathit{Princ}); \\ \mathsf{new}\ (\mathit{key}_1{:}\mathit{WMFKey}(\mathit{princ}_1)); \ldots \\ \mathsf{new}\ (\mathit{key}_n{:}\mathit{WMFKey}(\mathit{princ}_n)); \\ \mathit{FixedSender}(\mathit{net}, \mathit{princ}_1, \mathit{key}_1) \mid \cdots \mid \\ \mathit{FixedSender}(\mathit{net}, \mathit{princ}_n, \mathit{key}_n) \mid \\ \mathit{FixedReceiver}(\mathit{net}, \mathit{princ}_1, \mathit{key}_1) \mid \cdots \mid \\ \mathit{FixedReceiver}(\mathit{net}, \mathit{princ}_n, \mathit{key}_n) \mid \\ \mathit{FixedServer}(\mathit{net}, \mathit{lookup}) \mid \\ \mathit{KeyDB}(\mathit{lookup}, \mathit{princ}_1, \mathit{key}_1, \ldots, \mathit{princ}_n, \mathit{key}_n) \end{array}$$

We can then apply the results of this paper to get:

- Any Wide Mouth Frog configuration is effect-free, and hence robustly safe.

Thus, we have shown the Wide Mouth Frog protocol to satisfy this particular safety property for an arbitrary number of principals, sessions, and in the presence of an arbitrary attacker and well-typed database implementation.

The use of tagged unions to represent the different message types which are sent in a protocol is a common technique, and corresponds to the final phrase of Principle 10 of Abadi and Needham [4]:

> If an encoding is used to present the meaning of a message, then it should be possible to tell which encoding is being used. In the common case where the encoding is protocol dependent, it should be possible to deduce that the message belongs to this protocol, and in fact to a particular run of the protocol, and to know its number in the protocol.

Many protocols use ad hoc techniques such as incrementing timestamps, or juggling the order of participant names to encode message numbers implicitly. Our type system makes these ad hoc solutions formal, as an instance of the standard technique of using tagged union types.

A.2 Woo and Lam's Authentication Protocol

Woo and Lam [40] propose a server-based symmetric-key authentication protocol. Alice wishes to authenticate herself to Bob, and does so by responding to a nonce challenge with a message which Bob can ask the trusted server to decrypt:

Event 1	A begins	"A authenticates to B"
Message 1	$A \rightarrow B$:	A
Message 2	$B \rightarrow A$:	N_B
Message 3	$A \rightarrow B$:	$\{msg_3(N_B)\}_{K_{AS}}$
Message 4	$B \rightarrow S$:	$\{msg_4(A, \{msg_3(N_B)\}_{K_{AS}})\}_{K_{BS}}$
Message 5	$S \rightarrow B$:	$\{msg_5(N_B)\}_{K_{BS}}$
Event 2	B ends	"A authenticates to B"

(In the original protocol, the messages were untagged, but we have provided tags for the reasons discussed in the previous section.) Abadi and Needham [4] demonstrate that this protocol is not robustly safe, because message 5 does not mention A.

The possibility of this attack is made clear when we try to type-check the protocol. We have types:

$$
\begin{aligned}
&WLKey(princ) \triangleq \mathsf{Key}((WLMsg(princ))) \\
&WLMsg(princ) \triangleq (msg_3(WLMsg_3(princ)) \mid \\
&\qquad\qquad msg_4(WLMsg_4(princ)) \mid \\
&\qquad\qquad msg_5(WLMsg_5(princ))) \\
&WLMsg_3(alice) \triangleq (nonce{:}WLNonce(alice, bob)) \\
&WLMsg_4(bob) \triangleq (alice{:}Princ, ctext{:}\mathsf{Un}) \\
&WLMsg_5(bob) \triangleq (nonce{:}WLNonce(alice, bob)) \\
&WLNonce(alice, bob) \triangleq \mathsf{Nonce}\,[\mathsf{end}\ \text{"}alice\text{ authenticates to }bob\text{"}] \\
&WLLookup \triangleq (princ{:}Princ) \rightarrow WLKey(princ)
\end{aligned}
$$

At this point it becomes clear that the protocol is not well-typed, since the types are not well-formed: $WLMsg_3(alice)$ contains an unbound occurrence of *bob* and $WLMsg_5(bob)$ contains an unbound occurrence of *alice*. Abadi and Needham observe that Message 5 should be changed to:

$$\text{Message 5'} \quad S \rightarrow B : \quad \{msg_5(A, N_B)\}_{K_{BS}}$$

but did not make any similar observation for Message 3. Their strengthened protocol allows Bob to know that Alice is talking to somebody, but does not allow Bob to know that Alice is talking to Bob. For example, one possible run, where Alice begins a

dialogue with Charlie, but is authenticated to Bob is:

Event $\alpha.1$	A begins	"A authenticates to C"
Message $\alpha.1$	$A \rightarrow I:$	A
Message $\beta.1$	$I \rightarrow B:$	A
Message $\beta.2$	$B \rightarrow I:$	N_B
Message $\alpha.2$	$I \rightarrow A:$	N_B
Message $\alpha.3$	$A \rightarrow I:$	$\{msg_3(N_B)\}_{K_{AS}}$
Message $\beta.3$	$I \rightarrow B:$	$\{msg_3(N_B)\}_{K_{AS}}$
Message $\beta.4$	$B \rightarrow S:$	$\{msg_4(A, \{msg_3(N_B)\}_{K_{AS}})\}_{K_{BS}}$
Message $\beta.5$	$S \rightarrow B:$	$\{msg_5(N_B)\}_{K_{BS}}$
Event $\beta.2$	B ends	"A authenticates to B"

This attack is noted by Anderson and Needham [5], and is stopped by a similar change to the protocol:

$$\text{Message 3'} \quad A \rightarrow B: \quad \{msg_3(B, N_B)\}_{K_{AS}}$$

Finally, our type system makes clear that the encryption of message 4 is unnecessary, since all the data is of type Un, and so can safely be sent in plaintext, as suggested by Abadi and Needham [4]:

$$\text{Message 4'} \quad B \rightarrow S: \quad A, B, \{msg_3(B, N_B)\}_{K_{AS}}$$

The resulting protocol can be type-checked, using types:

$$\begin{array}{l} WLMsg(princ) \triangleq \\ \quad (msg_3(WLMsg_3(princ)) \mid msg_5(WLMsg_5(princ))) \\ WLMsg_3(alice) \triangleq \\ \quad (bob{:}Princ, nonce{:}WLNonce(alice, bob)) \\ WLMsg_5(bob) \triangleq \\ \quad (alice{:}Princ, nonce{:}WLNonce(alice, bob)) \end{array}$$

To see that the sender is effect-free, we calculate:

$$\begin{array}{l} FixedSender(net{:}Network, alice{:}Princ, key{:}WLKey(alice)) \triangleq \\ \quad \mathsf{repeat} \\ \quad\quad \mathsf{inp}\ net\ (bob{:}Princ); \\ \quad\quad \left.\begin{array}{l} \mathsf{begin}\ \text{"}alice\ \text{authenticates to}\ bob\text{"}; \\ \mathsf{out}\ net\ (alice) \\ \mathsf{inp}\ net\ (nonce{:}\mathsf{Un}); \\ \left.\begin{array}{l} \mathsf{cast}\ nonce\ \mathsf{is}\ (nonce'{:}WLNonce(alice, bob)); \\ \mathsf{out}\ net\ \{msg_3(bob, nonce')\}_{key} \end{array}\right\} [\mathsf{end}\ \ldots] \end{array}\right\} [] \end{array}$$

To see that the receiver is effect-free, we calculate:

$$
\begin{array}{l}
\mathit{FixedReceiver}(\mathit{net}{:}\mathit{Network}, \mathit{bob}{:}\mathit{Princ}, \mathit{key}{:}\mathit{WLKey}(\mathit{bob})) \triangleq \\
\quad \mathsf{repeat} \\
\quad\quad \left.\begin{array}{l}
\mathsf{inp}\ \mathit{net}\ (\mathit{alice}{:}\mathit{Princ}); \\
\mathsf{new}\ (\mathit{nonce}{:}\mathsf{Un}); \\
\mathsf{out}\ \mathit{net}\ (\mathit{nonce}) \\
\mathsf{inp}\ \mathit{net}\ (\mathit{ctext}{:}\mathsf{Un}); \\
\mathsf{out}\ \mathit{net}\ (\mathit{alice}, \mathit{bob}, \mathit{ctext}) \\
\mathsf{inp}\ \mathit{net}\ (\{\mathit{msg}_5(\mathit{alice}, \mathit{nonce}'{:}\mathit{WLNonce}(\mathit{alice}, \mathit{bob}))\}_{\mathit{key}}); \\
\left.\begin{array}{l}
\mathsf{check}\ \mathit{nonce}\ \mathsf{is}\ \mathit{nonce}'; \\
\mathsf{end}\ \text{“}\mathit{alice}\ \text{authenticates to}\ \mathit{bob}\text{”}\}\ [\mathsf{end}\ \ldots]
\end{array}\right\} [\mathsf{check}\ \mathit{nonce}]
\end{array}\right\} []
\end{array}
$$

To see that the server is effect-free, notice that the server makes no use of any process $\mathsf{check}\ N\ \mathsf{is}\ N'; P$, $\mathsf{cast}\ N\ \mathsf{is}\ (N'); P$ or $\mathsf{end}\ M$, and so is automatically effect-free:

$$
\begin{array}{l}
\mathit{FixedServer}(\mathit{net}{:}\mathit{Network}, \mathit{lookup}{:}\mathit{WLLookup}) \triangleq \\
\quad \mathsf{repeat} \\
\quad\quad \mathsf{inp}\ \mathit{net}\ (\mathit{alice}{:}\mathit{Princ}, \mathit{bob}{:}\mathit{Princ}, \mathit{ctext}{:}\mathsf{Un}); \\
\quad\quad \mathsf{let}\ \mathit{keyA}{:}\mathit{WLKey}(\mathit{alice}) = \mathit{lookup}(\mathit{alice}); \\
\quad\quad \mathsf{let}\ \mathit{keyB}{:}\mathit{WLKey}(\mathit{bob}) = \mathit{lookup}(\mathit{bob}); \\
\quad\quad \mathsf{decrypt}\ \mathit{ctext}\ \mathsf{is}\ \{\mathit{msg}_3(\mathit{bob}, \mathit{nonce}'{:}\mathit{WLNonce}(\mathit{alice}, \mathit{bob}))\}_{\mathit{keyA}}; \\
\quad\quad \mathsf{out}\ \mathit{net}\ \{\mathit{msg}_5(\mathit{alice}, \mathit{nonce}')\}_{\mathit{keyB}}
\end{array}
$$

We define a *Woo and Lam configuration* to be a process of the form:

$$
\begin{array}{l}
\mathsf{new}\ (\mathit{lookup}{:}\mathit{WLLookup}); \\
\mathsf{new}\ (\mathit{princ}_1{:}\mathit{Princ}); \ldots \mathsf{new}\ (\mathit{princ}_n{:}\mathit{Princ}); \\
\mathsf{new}\ (\mathit{key}_1{:}\mathit{WLKey}(\mathit{princ}_1)); \ldots \mathsf{new}\ (\mathit{key}_n{:}\mathit{WLKey}(\mathit{princ}_n)); \\
\mathit{FixedSender}(\mathit{net}, \mathit{princ}_1, \mathit{key}_1) \mid \cdots \mid \mathit{FixedSender}(\mathit{net}, \mathit{princ}_n, \mathit{key}_n) \\
\quad \mid \mathit{FixedReceiver}(\mathit{net}, \mathit{princ}_1, \mathit{key}_1) \mid \cdots \mid \mathit{FixedReceiver}(\mathit{net}, \mathit{princ}_n, \mathit{key}_n) \\
\quad \mid \mathit{FixedServer}(\mathit{net}, \mathit{lookup}) \mid \mathit{KeyDB}(\mathit{lookup}, \mathit{princ}_1, \mathit{key}_1, \ldots, \mathit{princ}_n, \mathit{key}_n)
\end{array}
$$

for any effect-free *KeyDB*. We can then apply the results of this paper to get:

- Any Woo and Lam configuration is effect-free, and hence robustly safe.

This example has shown that in our type system, it is important that all messages contain the names of the principals involved. Our type system enforces Principle 3 of Abadi and Needham [4]:

> If the identity of a principal is essential to the meaning of a message, it is prudent to mention the principal's name explicitly in the message.

This requirement is enforced through the usual requirement for variables in a program to be correctly scoped: violations of Principle 3 may be caught because a variable is used when it is not in scope.

A.3 Otway and Rees's Key Exchange Protocol

Otway and Rees [33] propose a server-based symmetric-key key exchange protocol. We cannot verify their protocol using the type system of this paper, even though (as far as we are aware) it is correct, since it relies on using nonces to stand for principal names, which are kept secret, as well as for freshness. Still, it may be possible to adapt our type system to deal with this use of nonces; we leave this for future work.

Abadi and Needham [4] propose a simplification of the protocol, which we verify here:

Message 1	$A \to B$	A, B, N_A
Message 2	$B \to S$	A, B, N_A, N_B
Event 1	S begins	"initiator A shares K_{AB} with B"
Event 2	S begins	"responder B shares K_{AB} with A"
Message 3	$S \to B$	$\{msg_4(A, B, K_{AB}, N_A)\}_{K_{AS}}$, $\{msg_3(A, B, K_{AB}, N_B)\}_{K_{BS}}$
Event 3	B ends	"responder B shares K_{AB} with A"
Message 4	$B \to A$	$\{msg_4(A, B, K_{AB}, N_A)\}_{K_{AS}}$
Event 4	A ends	"initiator A shares K_{AB} with B"

At the end of this dialogue, Alice and Bob both know that K_{AB} was generated by Sam for their private use. Alice does not know that Bob actually received K_{AB}, since this protocol does not ensure that Alice and Bob actually receive K_{AB}, just that nobody else does.

We can allocate types to this protocol:

$$
\begin{array}{l}
ORKey(princ) \triangleq \\
\quad \mathsf{Key}((msg_3(ORMsg_3(princ)) \mid msg_4(ORMsg_4(princ)))) \\
ORMsg_3(bob) \triangleq \\
\quad (alice{:}Princ, bob'{:}Princ, sKey{:}SKey, \\
\quad\ nonce{:}ORNonce_3(alice, bob, sKey)) \\
ORMsg_4(alice) \triangleq \\
\quad (alice'{:}Princ, bob{:}Princ, sKey{:}SKey, \\
\quad\ nonce{:}ORNonce_3(alice, bob, sKey)) \\
ORNonce_3(alice, bob, sKey) \triangleq \\
\quad \mathsf{Nonce}\ [\mathsf{end}\ \text{"responder } bob \text{ shares } sKey \text{ with } alice\text{"}] \\
ORNonce_4(alice, bob, sKey) \triangleq \\
\quad \mathsf{Nonce}\ [\mathsf{end}\ \text{"initiator } alice \text{ shares } sKey \text{ with } bob\text{"}] \\
ORLookup \triangleq \\
\quad (princ{:}Princ) \to ORKey(princ)
\end{array}
$$

and then type-check Alice:

$$
\begin{array}{l}
\mathit{FixedSender}(\mathit{net}{:}\mathit{Network}, \mathit{alice}{:}\mathit{Princ}, \mathit{key}{:}\mathit{ORKey}(\mathit{alice})) \triangleq \\
\quad \mathsf{repeat} \\
\quad\quad \mathsf{inp}\ \mathit{net}\ (\mathit{bob}{:}\mathit{Princ}); \\
\quad\quad \left.\begin{array}{l}
\mathsf{new}\ (\mathit{nonceA}{:}\mathsf{Un}); \\
\mathsf{out}\ \mathit{net}\ (\mathit{alice}, \mathit{bob}, \mathit{nonceA}); \\
\mathsf{inp}\ \mathit{net}\ (\{\mathit{msg}_4(\mathit{alice}, \mathit{bob}, \mathit{sKey}{:}\mathit{SKey}, \\
\qquad\qquad \mathit{nonceA}'{:}\mathit{ORNonce}_4(\mathit{alice}, \mathit{bob}, \mathit{sKey}))\}_{\mathit{keyA}}); \\
\mathsf{check}\ \mathit{nonceA}\ \mathsf{is}\ \mathit{nonceA}'; \\
\mathsf{end}\ \text{“initiator } \mathit{alice} \text{ shares } \mathit{sKey} \text{ with } \mathit{bob}\text{”}\}\ [\mathsf{end}\ \ldots]
\end{array}\right\}\ [\mathsf{check}\ \mathit{nonceA}]
\end{array}
\left.\begin{array}{l}\\\\\\\\\\\\\end{array}\right\}\ []
$$

type-check Bob:

$$
\begin{array}{l}
\mathit{FixedReceiver}(\mathit{net} : \mathit{Network}, \mathit{bob}{:}\mathit{Princ}, \mathit{key}{:}\mathit{ORKey}(\mathit{bob})) \triangleq \\
\quad \mathsf{repeat} \\
\quad\quad \mathsf{inp}\ \mathit{net}\ (\mathit{alice}{:}\mathit{Princ}, \mathit{bob}, \mathit{nonceA}{:}\mathsf{Un}); \\
\quad\quad \left.\begin{array}{l}
\mathsf{new}\ (\mathit{nonceB}{:}\mathsf{Un}); \\
\mathsf{out}\ \mathit{net}\ (\mathit{alice}, \mathit{bob}, \mathit{nonceA}, \mathit{nonceB}); \\
\mathsf{inp}\ \mathit{net}\ (\mathit{ctext}{:}\mathsf{Un}, \{\mathit{msg}_3(\mathit{alice}, \mathit{bob}, \mathit{sKey}{:}\mathit{SKey}, \\
\qquad\qquad \mathit{nonceB}{:}\mathit{ORNonce}_3(\mathit{alice}, \mathit{bob}, \mathit{sKey}))\}_{\mathit{key}}); \\
\mathsf{check}\ \mathit{nonceB}\ \mathsf{is}\ \mathit{nonceB}'; \\
\mathsf{end}\ \text{“responder } \mathit{bob} \text{ shares } \mathit{sKey} \text{ with } \mathit{alice}\text{”}\}\ [\mathsf{end}\ \ldots]
\end{array}\right\}\ [\mathsf{check}\ \mathit{nonceB}]
\end{array}
\left.\begin{array}{l}\\\\\\\\\\\\\end{array}\right\}\ []
$$

and type-check Sam:

$$
\begin{array}{l}
\mathit{FixedServer}(\mathit{net}{:}\mathit{Network}, \mathit{lookup}{:}\mathit{ORLookup}) \triangleq \\
\quad \mathsf{repeat} \\
\quad\quad \mathsf{inp}\ \mathit{net}\ (\mathit{alice}{:}\mathit{Princ}, \mathit{bob}{:}\mathit{Princ}, \mathit{nonceA}{:}\mathsf{Un}, \mathit{nonceB}{:}\mathsf{Un}); \\
\quad\quad \mathsf{let}\ \mathit{key}_A{:}\mathit{ORKey}(\mathit{alice}) = \mathit{lookup}(\mathit{alice}); \\
\quad\quad \mathsf{let}\ \mathit{key}_B{:}\mathit{ORKey}(\mathit{bob}) = \mathit{lookup}(\mathit{bob}); \\
\quad\quad \mathsf{new}\ (\mathit{sKey}{:}\mathit{SKey}); \\
\quad\quad \left.\begin{array}{l}
\mathsf{begin}\ \text{“initiator } \mathit{alice} \text{ shares } \mathit{sKey} \text{ with } \mathit{bob}\text{”}; \\
\mathsf{begin}\ \text{“responder } \mathit{bob} \text{ shares } \mathit{sKey} \text{ with } \mathit{alice}\text{”}; \\
\mathsf{cast}\ \mathit{nonceA}\ \mathsf{is}\ (\mathit{nonceA}'{:}\mathit{ORNonce}_4(\mathit{alice}, \mathit{bob}, \mathit{sKey})); \\
\mathsf{cast}\ \mathit{nonceB}\ \mathsf{is}\ (\mathit{nonceB}'{:}\mathit{ORNonce}_3(\mathit{alice}, \mathit{bob}, \mathit{sKey})); \\
\mathsf{out}\ \mathit{net}\ (\{\mathit{msg}_4(\mathit{alice}, \mathit{bob}, \mathit{sKey}, \mathit{nonceA}')\}_{\mathit{key}_A}, \\
\qquad \{\mathit{msg}_3(\mathit{alice}, \mathit{bob}, \mathit{sKey}, \mathit{nonceB}')\}_{\mathit{key}_B})
\end{array}\right\}\ []\ \left.\begin{array}{l}\\\\\\\\\end{array}\right\}\ [\mathsf{end}\ \ldots]
\end{array}
\left.\begin{array}{l}\\\\\\\\\\\\\end{array}\right\}\ [\mathsf{end}\ \ldots]
\left.\begin{array}{l}\\\\\\\\\\\\\end{array}\right\}\ []
$$

We can then apply the techniques of this paper to show that this modified protocol is robustly safe. This typing makes it clear that Bob's name is not required in Message 3 and Alice's name is not required in Message 4, and these names could be dropped without compromising the correspondence assertions.

A.4 A Secure Message Stream

In Section 4.4 we showed how we can verify a simple two-message protocol to ensure the authenticity of messages. The protocol relied on Alice to send Bob a nonce

challenge for every message Bob sends:

Event 1	A begins	A sent M
Message 1	$B \to A$:	N
Message 2	$A \to B$:	$\{M,N\}_K$
Event 2	B ends	A sent M

This is rather inefficient, since it requires an acknowledgement message for every message. Instead, we could use *message identifiers* to ensure the freshness of messages without Alice having to send constant acknowledgements. Our language does not support message identifiers directly, but they can be coded in messages of nonces: each time Bob sends Alice a message, he sends *two* nonces: the nonce for the current message, and the nonce for the next message. This is enough for Alice to ensure freshness of messages:

Message 0	$B \to A$:	N_1
Event 1a	A begins	A sent M_1
Message 1	$A \to B$:	$\{M_1,N_2,N_1\}_K$
Event 1b	B ends	A sent M_1
...		
Event na	A begins	A sent M_n
Message n	$A \to B$:	$\{M_n,N_{n+1},N_n\}_K$
Event nb	B ends	A sent M_n

In order to check this protocol, we need to make use of latent nonce effects, since nonce N_n is being used to ensure the freshness of nonce N_{n+1}. The types we use are:

$$\mathit{MidKey} \triangleq \mathsf{Key}((\mathit{msg}{:}\mathit{Msg}, \mathit{nonceB}{:}\mathsf{Un}, \mathit{nonceA}{:}\mathit{MidNonce}(\mathit{msg},\mathit{nonceB})))$$
$$\mathit{MidNonce}(\mathit{msg},\mathit{nonceB}) \triangleq \mathsf{Nonce}\ [\mathsf{end}\ \text{“Sender sent }\mathit{msg}\text{”}, \mathsf{check}\ \mathit{nonceB}]$$

The receiver is type-checked:

$$\left.\begin{array}{l}\mathit{FixedReceiver}(\mathit{net}{:}\mathit{Network}, \mathit{key}{:}\mathit{MidKey}) \triangleq \\ \quad \mathsf{new}\ (\mathit{nonceA}{:}\mathsf{Un}); \\ \quad \mathsf{out}\ \mathit{net}\ \mathit{nonceA} \\ \quad \mathit{FixedReceiver}(\mathit{net},\mathit{key},\mathit{nonceA})\}\ [\mathsf{check}\ \mathit{nonceA}]\end{array}\right\}\ []$$

where we use the recursive function:

$$\begin{array}{l}\mathit{FixedReceiver}(\mathit{net}{:}\mathit{Network}, \mathit{key}{:}\mathit{MidKey}, \mathit{nonceA}{:}\mathsf{Un}) \triangleq \\ \quad \mathsf{inp}\ \mathit{net}\ (\{(\mathit{msg}{:}\mathit{Msg}, \mathit{nonceB}{:}\mathsf{Un}, \mathit{nonceA}'{:}\mathit{MidNonce}(\mathit{msg},\mathit{nonceB}))\}_{\mathit{key}}); \\ \quad \mathsf{check}\ \mathit{nonceA}\ \mathsf{is}\ \mathit{nonceA}'; \\ \quad \mathsf{end}\ \text{“Sender sent }\mathit{msg}\text{”}; \\ \quad \mathit{FixedReceiver}(\mathit{net},\mathit{key},\mathit{nonceB})\end{array}$$

The sender is type-checked similarly. This example shows that it is useful for participants in a protocol to be able to pass nonces and nonce effects, as allowed by our effect system.

A.5 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, tagged unions, and dependent function types:

Syntax sugar for use in types:

$T,U ::=$	type
$\ldots$	as in Sections 4.1 and 4.2
$(x_1{:}T_1, x_2{:}T_2, \ldots, x_n{:}T_n)$	dependent record
$(\ell_1(T_1) \mid \cdots \mid \ell_n(T_n))$	tagged union
$(x{:}T) \to U$	dependent function

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

Syntax sugar for use in messages:

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

In processes, we can make use of function declaration, function call, function return, and pattern-matching:

Syntax sugar for use in processes:

$O,P,Q,R ::=$	process
$\ldots$	as in Sections 2.1, 3.1 and 4.2
$\mathsf{function}\ f\,(X_1) : T_1\ \mathsf{is}\ P_1 \cdots f\,(X_n) : T_n\ \mathsf{is}\ P_n$	function declaration
$\mathsf{let}\ x{:}U = f\,(M); P$	function call
$\mathsf{return}\ M$	function return
$\mathsf{match}\ M\ \mathsf{is}\ X; P$	pattern match
$\mathsf{out}\ M\ P;$	output with residual
$\mathsf{inp}\ M\ (X); P$	pattern matching input
$\mathsf{decrypt}\ M\ \mathsf{is}\ \{X\}_P;$	pattern matching decrypt

where X ranges over a grammar of patterns:

Patterns:

$X,Y,Z ::=$	patterns
$x{:}T$	variable
M	constant
$(X_1, \ldots, X_n)$	tuple
$\ell_i(X)$	tagged union
$\{X\}_M$	cyphertext

We will now give definitions for each of these extensions, beginning with types. Dependent records and tagged unions are routine, since we already have pairs and variants types. Dependent records use a variant of the translation of functions into the π-calculus [31]; this is explored in more detail in [18]).

Abbreviations for types:

$(x_1{:}T_1, x_2{:}T_2, \ldots, x_n{:}T_n) \triangleq (x_1{:}T_1, (x_2{:}T_2, (\ldots (x_n{:}T_n, ()) \ldots)))$
$(\ell_1(T_1) \mid \cdots \mid \ell_n(T_n)) \triangleq (T_1 + (T_2 + (\ldots (T_n + 0) \ldots)))$
$(x{:}T) \to U \triangleq \mathsf{Ch}(x{:}T, \mathsf{Ch}(U))$

The translations of messages are similarly straightforward.

Abbreviations for messages:

$(M_1, M_2, \ldots, M_n) \triangleq (M_1, (M_2, (\ldots (M_n, ()) \ldots)))$
$\ell_i(M) \triangleq \mathsf{in}_i\,(M)$
$\mathsf{in}_1\,(M) \triangleq \mathsf{inl}\,(M)$
$\mathsf{in}_{n+1}\,(M) \triangleq \mathsf{inr}\,(\mathsf{in}_n\,(M))$

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

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

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

We use a variant of Milner's translation of the λ-calculus into the π-calculus, extended to deal with pattern-matching.

Abbreviations for functions, where $f : (x{:}T) \to U$**:**

$\mathsf{function}\ f\ (X_1) : U_1\ \mathsf{is}\ P_1 \cdots f\ (X_n) : U_n\ \mathsf{is}\ P_n \triangleq$
  $\mathsf{repeat\ inp}\ f\ (\mathit{request}{:}(x{:}T, \mathsf{Ch}(U)))$;
    $(\mathsf{match}\ \mathit{request}\ \mathsf{is}\ (X_1, \mathit{return}{:}\mathsf{Ch}(U_1)); P_1 \mid \cdots \mid)$
     $\mathsf{match}\ \mathit{request}\ \mathsf{is}\ (X_n, \mathit{return}{:}\mathsf{Ch}(U_n)); P_n$
$\mathsf{return}\ M \triangleq$
  $\mathsf{out}\ \mathit{return}\ N$
$\mathsf{let}\ x{:}U = f\ (M); P \triangleq$
  $\mathsf{new}\ (k{:}\mathsf{Ch}(U)); \mathsf{out}\ f\ (M,k); \mathsf{inp}\ k\ (x{:}U); P$

where we define pattern-matching as:

Abbreviations for pattern matching:

$\mathsf{inp}\ M\ (X); P \triangleq \mathsf{inp}\ M\ (x); \mathsf{match}\ x\ \mathsf{is}\ X; P$
$\mathsf{decrypt}\ M\ \mathsf{is}\ \{X\}_N; P \triangleq \mathsf{decrypt}\ M\ \mathsf{is}\ \{x\}_N; \mathsf{match}\ x\ \mathsf{is}\ X; P$
$\mathsf{match}\ M\ \mathsf{is}\ x{:}T; P \triangleq P\{x \leftarrow M\}$

$\mathsf{match}\ M\ \mathsf{is}\ ();P \triangleq P$

$\mathsf{match}\ M\ \mathsf{is}\ (N,X_1,\ldots,X_n);P \triangleq \mathsf{match}\ M\ \mathsf{is}\ (N,y);\mathsf{match}\ y\ \mathsf{is}\ (X_1,\ldots,X_n);P$

$\mathsf{match}\ M\ \mathsf{is}\ (X_0,X_1,\ldots,X_n);P \triangleq \mathsf{split}\ M\ \mathsf{is}\ (x,y);$
$\mathsf{match}\ x\ \mathsf{is}\ X_0;\mathsf{match}\ y\ \mathsf{is}\ (X_1,\ldots,X_n);P$

$\mathsf{match}\ M\ \mathsf{is}\ \mathsf{in}_1\ (X);P \triangleq \mathsf{case}\ M\ \mathsf{is}\ \mathsf{inl}\ (x)\ \mathsf{match}\ x\ \mathsf{is}\ X;P\ \mathsf{is}\ \mathsf{inr}\ (x)\ \mathsf{stop}$

$\mathsf{match}\ M\ \mathsf{is}\ \mathsf{in}_{n+1}\ (X);P \triangleq \mathsf{case}\ M\ \mathsf{is}\ \mathsf{inl}\ (x)\ \mathsf{stop}\ \mathsf{is}\ \mathsf{inr}\ (x)\ \mathsf{match}\ x\ \mathsf{is}\ \mathsf{in}_n\ (X);P$

$\mathsf{match}\ M\ \mathsf{is}\ \{X\}_N P; \triangleq \mathsf{decrypt}\ M\ \mathsf{is}\ \{x\}_N;\mathsf{match}\ x\ \mathsf{is}\ X;P$

$\mathsf{match}\ M\ \mathsf{is}\ N;P \triangleq \mathsf{match}\ (M,())\ \mathsf{is}\ (N,x);P$

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

B Formal Semantics of our Typed Spi-Calculus

This appendix develops a formal operational semantics for the spi-calculus. Hence, we make precise the informal definition of process safety stated in Section 3.1, and prove the type safety result, Theorem 1 (Safety), stated in Section 4.3.

We begin in Appendix B.1 by defining a trace semantics for the spi-calculus, and use it to define safety in Appendix B.2. In Appendix B.3, we state and prove a subject reduction property (that is, a type preservation property). Finally, in Appendix B.4 we exploit subject reduction to prove Theorem 1 (Safety).

B.1 A Trace Semantics for our Spi-Calculus

We use a trace semantics based on the Chemical Abstract Machine [7]. First, we define a structural equivalence $P \equiv Q$ on processes, and then we define the trace semantics in terms of structural equivalence. This is the same technique as Milner [31] uses in the presentation of the π-calculus, and Abadi and Gordon [3] use in the presentation of the spi-calculus.

Structural Equivalence: $P \equiv Q$

$P \equiv P$	(Struct Refl)
$Q \equiv P \Rightarrow P \equiv Q$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv Q \Rightarrow \mathsf{new}\ (x{:}T);P \equiv \mathsf{new}\ (x{:}T);Q$	(Struct Res)
$P \equiv Q \Rightarrow P \mid R \equiv Q \mid R$	(Struct Par)
$P \mid \mathsf{stop} \equiv P$	(Struct Par Zero)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$\mathsf{repeat}\ P \equiv P \mid \mathsf{repeat}\ P$	(Struct Repl Par)
$x \notin fn(P) \Rightarrow P \mid \mathsf{new}\ (x{:}T);Q \equiv \mathsf{new}\ (x{:}T);(P \mid Q)$	(Struct Par Res)

$x \neq y, x \notin fn(U), y \notin fn(T) \Rightarrow$ (Struct Res Res)
$\quad \mathsf{new}\ (x{:}T); \mathsf{new}\ (y{:}U); P \equiv \mathsf{new}\ (y{:}U); \mathsf{new}\ (x{:}T); P$

A trace of a process is a finite sequence of events. The set of possible events includes the begin- and end-events defined in Section 3.1, as well as other events representing various actions of processes.

Each process is given a trace semantics, where a trace is a sequence of events performed by the process. Events take the following forms.

Events:

$\alpha, \beta ::=$	events
$\mathsf{begin}\ L$	begin-event labelled with message L
$\mathsf{end}\ L$	end-event labelled with message L
$\mathsf{cast}\ x{:}T$	cast-event of name x to type T
$\mathsf{check}\ x$	check-event for nonce x
$\mathsf{gen}\ x{:}T$	fresh-event for name x
τ	internal-event

Events may contain free names. For example, $fn(\mathsf{end}\ (\text{Sender sent } msg)) = \{msg\}$.

Free names, $fn(\alpha)$, of an event α

$fn(\tau) \stackrel{\Delta}{=} \varnothing$
$fn(\mathsf{cast}\ x{:}T) \stackrel{\Delta}{=} \{x\} \cup fn(T)$
$fn(\mathsf{check}\ x) \stackrel{\Delta}{=} \{x\}$
$fn(\mathsf{begin}\ M) \stackrel{\Delta}{=} fn(M)$
$fn(\mathsf{end}\ M) \stackrel{\Delta}{=} fn(M)$
$fn(\mathsf{gen}\ x{:}T) \stackrel{\Delta}{=} \{x\} \cup fn(T)$

Events may also contain generated names. For example, $gn(\mathsf{gen}\ msg{:}Msg) = \{msg\}$.

Generated names, $gn(\alpha)$, of an event α

$$gn(\alpha) \stackrel{\Delta}{=} \begin{cases} \{x\} & \text{if } \alpha = \mathsf{gen}\ x{:}T \\ \varnothing & \text{otherwise} \end{cases}$$

We interpret events as follows:

- An event $\mathsf{begin}\ L$ arises from a process $\mathsf{begin}\ L; P$, and represents the beginning of a correspondence.
- An event $\mathsf{end}\ L$ arises from a process $\mathsf{end}\ L; P$, and represents the end of a correspondence.
- An event $\mathsf{cast}\ N{:}T$ arises from a process $\mathsf{cast}\ N\ \mathsf{is}\ (x{:}T); P$, and represents the cast of an untrusted message into the type T, which the type system requires to be of the specific form $\mathsf{Nonce}\ es$.

- An event check N arises from a process check N is $N;P$, and represents a successful check for the presence of a nonce.
- An event gen $x{:}T$ arises from a process new $(x{:}T);P$, and represents the generation of a fresh name x.
- An event τ arises from an internal computational step of a process.

For example, in the *FixedSystem*(*net*) example from Section 3.3, one possible sequence of events is:

- gen *nonce*:Un: the receiver generates a fresh untrusted name *nonce*.
- gen *msg*:*Msg*: the sender generates a new message *msg*.
- begin (Sender sent *msg*): the sender begins a correspondence.
- cast *nonce*:*MyNonce* (*msg*): the sender casts the untrusted message *nonce* to the type *MyNonce* .
- check *nonce*: the receiver checks that the received nonce is *nonce*.
- end (Sender sent *msg*): the receiver ends a correspondence.

On the other hand, in the compromised system *FlawedSystem*(*net*) | *Attacker*(*net*) one possible sequence of events is:

- gen *msg*:*Msg*: the sender generates a new message *msg*.
- begin (Sender sent *msg*): the sender begins a correspondence.
- end (Sender sent *msg*): the receiver ends a correspondence.
- end (Sender sent *msg*): the receiver mistakenly ends the same correspondence twice.

Next, we give a formal definition of the events a process is capable of, using a *labelled transition system* semantics $P \xrightarrow{\alpha} P'$, meaning "$P$ can perform event α and become P'".

Labelled transitions: $P \xrightarrow{\alpha} P'$

out $x\ M \mid$ inp $x\ (y{:}T);P \xrightarrow{\tau} P\{y{\leftarrow}M\}$	(Trans Comm)
split (M,N) is $(x{:}T,y{:}U);P \xrightarrow{\tau} P\{x{\leftarrow}M\}\{y{\leftarrow}N\}$	(Trans Split)
match (M,N) is $(M,y{:}U);P \xrightarrow{\tau} P\{y{\leftarrow}N\}$	(Trans Match)
case inl (M) is inl $(x{:}T)\ P$ is inr $(y{:}U)\ Q \xrightarrow{\tau} P\{x{\leftarrow}M\}$	(Trans Case Inl)
case inr (M) is inl $(x{:}T)\ P$ is inr $(y{:}U)\ Q \xrightarrow{\tau} Q\{y{\leftarrow}M\}$	(Trans Case Inr)
decrypt $\{M\}_N$ is $\{x{:}T\}_N;P \xrightarrow{\tau} P\{x{\leftarrow}M\}$	(Trans Decrypt)
cast x is $(y{:}T);P \xrightarrow{\text{cast } x:T} P\{y{\leftarrow}x\}$	(Trans Cast)
check x is $x;P \xrightarrow{\text{check } x} P$	(Trans Check)
begin $M;P \xrightarrow{\text{begin } M} P$	(Trans Begin)
end $M;P \xrightarrow{\text{end } M} P$	(Trans End)

$\mathsf{new}\ (x{:}T); P \xrightarrow{\mathsf{gen}\ x} P$ (Trans Gen)

$gn(\alpha) \cap fn(Q) = \varnothing \Rightarrow P \xrightarrow{\alpha} P' \Rightarrow P \mid Q \xrightarrow{\alpha} P' \mid Q$ (Trans Par)

$x \notin fn(\alpha) \Rightarrow P \xrightarrow{\alpha} P' \Rightarrow \mathsf{new}\ (x{:}T); P \xrightarrow{\alpha} \mathsf{new}\ (x{:}T); P'$ (Trans Res)

$P \equiv Q, Q \xrightarrow{\alpha} Q', Q' \equiv P' \Rightarrow P \xrightarrow{\alpha} P'$ (Trans $\equiv$)

A trace is a sequence of events which the process may perform.

Traces:

$s,t ::= \alpha_1, \ldots, \alpha_n$	trace (written ε if $n = 0$)

We extend the definition of free and generated names to traces:

Free names, $fn(s)$, and generated names, $gn(s)$, of trace s

$fn(a_1, \ldots, a_n) \triangleq fn(a_1) \cup \cdots \cup fn(a_n)$

$gn(a_1, \ldots, a_n) \triangleq gn(a_1) \cup \cdots \cup gn(a_n)$

The traces of a process P are defined using a trace-labelled transition system $P \xrightarrow{s} P'$ meaning 'P performs trace s and becomes P'.'

Traced transitions: $P \xrightarrow{s} P'$

$P \equiv P' \Rightarrow P \xrightarrow{\varepsilon} P'$	(Trace $\equiv$)
$P \xrightarrow{\alpha} P'', P'' \xrightarrow{s} P' \Rightarrow P \xrightarrow{a,s} P''$	(Trace Event) (where $fn(a) \cap gn(s) = \varnothing$)

For example one trace of $FixedSystem(net)$ is (ignoring τ actions):

$\mathsf{gen}\ nonce{:}\mathsf{Un},$
$\mathsf{gen}\ msg{:}Msg,$
$\mathsf{begin}\ (\text{Sender sent}\ msg),$
$\mathsf{cast}\ nonce{:}MyNonce\ (msg),$
$\mathsf{check}\ nonce,$
$\mathsf{end}\ (\text{Sender sent}\ msg)$

One trace of $FlawedSystem(net) \mid Attacker(net)$ is:

$\mathsf{gen}\ msg{:}Msg,$
$\mathsf{begin}\ (\text{Sender sent}\ msg),$
$\mathsf{end}\ (\text{Sender sent}\ msg),$
$\mathsf{end}\ (\text{Sender sent}\ msg)$

B.2 Correspondence Traces and Safe Processes

We now define our notion of safety, through correspondence assertions. To do so, we need to recall some standard notation for unordered collections of data, or *multisets*. If x ranges over elements of some given set, we let xs range over multisets of elements of that set.

Multiset of elements

$xs ::=$	multiset
$\quad [x_1, \ldots, x_n]$	unordered collection of elements

We identify multisets up to permuting elements, so $[x,y] = [y,x]$ but not up to copying elements, so $[x] \neq [x,x]$. We define some standard operations on multisets.

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

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

For example:

- $[x,y] + [y,z] = [x,y,y,z]$.
- $[x,y] \leq [x,y,z]$ but $[x,y,y] \not\leq [x,y,z]$.
- $[x,y,y,z] - [w,x,y] = [y,z]$.
- $x \in [x,y]$ but $z \notin [x,y]$.
- $[x,y] \vee [y,z] = [x,y,z]$.

For example, we use Ms to range over multisets of messages.

Multisets of messages Ms**:**

$Ms ::=$	multiset of messages
$\quad [M_1, \ldots, M_n]$	unordered collection of messages

We define the *beginnings* and *endings* of a trace s as the multiset of event labels begun and ended, respectively, in s.

Beginnings, $begins(s)$**, and endings,** $ends(s)$**, of a trace** s

$$begins(a_1, \ldots, a_n) \stackrel{\Delta}{=} begins(a_1) + \cdots + begins(a_n)$$
$$\text{where } begins(a) \stackrel{\Delta}{=} \begin{cases} [M] & \text{if } a = \mathsf{begin}\ M \\ [\,] & \text{otherwise} \end{cases}$$

$$ends\,(a_1,\ldots,a_n) \triangleq ends\,(a_1) + \cdots + ends\,(a_n)$$
$$\text{where } ends\,(a) \triangleq \begin{cases} [M] & \text{if } a = \mathsf{end}\ M \\ [\,] & \text{otherwise} \end{cases}$$

Next, we say that a trace is a correspondence if its beginnings dominate its endings; that is, for each end-event labelled L, there is a corresponding begin-event labelled L.

Correspondence:

A trace s is a *correspondence* if and only if $ends\,(s) \leq begins\,(s)$.

For the example trace of $FixedSystem(net)$ we have:

$$\begin{array}{l} begins\,(\mathsf{gen}\ nonce{:}\mathsf{Un}, \mathsf{gen}\ msg{:}Msg, \mathsf{begin}\ (\text{Sender sent } msg), \\ \quad \mathsf{cast}\ nonce{:}MyNonce\ (msg), \mathsf{check}\ nonce, \mathsf{end}\ (\text{Sender sent } msg)) \\ \qquad = [\text{Sender sent } msg] \\ ends\,(\mathsf{gen}\ nonce{:}\mathsf{Un}, \mathsf{gen}\ msg{:}Msg, \mathsf{begin}\ (\text{Sender sent } msg), \\ \quad \mathsf{cast}\ nonce{:}MyNonce\ (msg), \mathsf{check}\ nonce, \mathsf{end}\ (\text{Sender sent } msg)) \\ \qquad = [\text{Sender sent } msg] \end{array}$$

Therefore, since this trace s satisfies $ends\,(s) \leq begins\,(s)$, it is a correspondence.

For the example trace of $FlawedSystem(net) \mid Attacker(net)$ we have:

$$\begin{array}{l} begins\,(\mathsf{gen}\ msg{:}Msg, \mathsf{begin}\ (\text{Sender sent } msg), \\ \quad \mathsf{end}\ (\text{Sender sent } msg), \mathsf{end}\ (\text{Sender sent } msg)) \\ \qquad = [\text{Sender sent } msg] \\ ends\,(\mathsf{gen}\ msg{:}Msg, \mathsf{begin}\ (\text{Sender sent } msg), \\ \quad \mathsf{end}\ (\text{Sender sent } msg), \mathsf{end}\ (\text{Sender sent } msg)) \\ \qquad = [\text{Sender sent } msg, \text{Sender sent } msg] \end{array}$$

Since this trace has $ends\,(s) \not\leq begins\,(s)$, it is not a correspondence.

We can now restate, precisely, the notions of safety and robust safety introduced informally in Section 3.1.

Safety and Robust Safety:

A process P is *safe* if and only if for all traces s and processes P',
if $P \xrightarrow{s} P'$ then s is a correspondence.
A process P is *robustly safe* if and only if for all opponent processes O, $P \mid O$ is safe

For example, since $FlawedSystem(net) \mid Attacker(net)$ has a trace that is not a correspondence, it follows that $FlawedSystem(net) \mid Attacker(net)$ is not safe. Since the process $Attacker(net)$ is an opponent process, it follows that $FlawedSystem(net)$ is not robustly safe.

B.3 Proof of Subject Reduction

In this section, we prove a subject reduction property for the labelled transition system, that transitions preserve typings. To do so, however, we need to extend the type system to accommodate the fact that cast-processes can change the type of a name after a transition.

We can illustrate some of the subtleties introduced by casting by considering three processes that are well-typed with respect to the typing environment defined by $E = x{:}\mathsf{Un}, y{:}\mathsf{Un}, z{:}\mathsf{Ch}(\mathsf{Nonce}\ [\mathsf{end}\ y])$.

Firstly, the following example illustrates that a well-typed process can cast the name x, originally of type Un, into the distinct type $\mathsf{Nonce}\ [\mathsf{end}\ y]$.

$$P_1 \triangleq \mathsf{cast}\ x\ \mathsf{is}\ (x'{:}\mathsf{Nonce}\ [\mathsf{end}\ y]); \mathsf{out}\ z\ x'$$
$$E \vdash P_1 : [\mathsf{end}\ y]$$
$$P_1 \xrightarrow{\mathsf{cast}\ x{:}\mathsf{Nonce}\ [\mathsf{end}\ y]} \mathsf{out}\ z\ x$$

Secondly, the following example illustrates that the name y, originally of type Un, can be cast into the type $\mathsf{Nonce}\ [\mathsf{end}\ y]$, that depends on the name y itself.

$$P_2 \triangleq \mathsf{cast}\ y\ \mathsf{is}\ (y'{:}\mathsf{Nonce}\ [\mathsf{end}\ y]); \mathsf{out}\ z\ y'$$
$$E \vdash P_2 : [\mathsf{end}\ y]$$
$$P_2 \xrightarrow{\mathsf{cast}\ y{:}\mathsf{Nonce}\ [\mathsf{end}\ y]} \mathsf{out}\ z\ y$$

Thirdly, the following example illustrates that the name x can be cast to two distinct types, $\mathsf{Nonce}\ [\mathsf{end}\ x]$ and $\mathsf{Nonce}\ [\mathsf{end}\ y]$.

$$P_3 \triangleq \mathsf{cast}\ x\ \mathsf{is}\ (x'{:}\mathsf{Nonce}\ [\mathsf{end}\ x]); \mathsf{cast}\ x\ \mathsf{is}\ (x''{:}\mathsf{Nonce}\ [\mathsf{end}\ y]); \mathsf{out}\ z\ x''$$
$$E \vdash P_3 : [\mathsf{end}\ x, \mathsf{end}\ y]$$
$$P_3 \xrightarrow{\mathsf{cast}\ x{:}\mathsf{Nonce}\ [\mathsf{end}\ y]} \xrightarrow{\mathsf{cast}\ x{:}\mathsf{Nonce}\ [\mathsf{end}\ x]} \mathsf{out}\ z\ x$$

Moreover, the possibility that a name can come to inhabit multiple distinct types arises in the setting of our running example. Recall from Section 3.3 that we have:

$$net{:}\mathsf{Un} \vdash \mathit{FixedSystem}(net) : []$$

Now, consider the attacker:

$$\mathit{Attacker}(net) \triangleq \mathsf{inp}\ net\ (nonce{:}\mathsf{Un}); \mathsf{out}\ net\ (nonce); \mathsf{out}\ net\ (nonce)$$

We can derive

$$net{:}\mathsf{Un} \vdash \mathit{FixedSystem}(net) \mid \mathit{Attacker}(net) : []$$

but $\mathit{FixedSystem}(net) \mid \mathit{Attacker}(net)$ has the trace:

- $\mathsf{gen}\ (nonce)$: receiver generates a nonce (initially $nonce{:}\mathsf{Un}$).
- $\mathsf{gen}\ (msg_1)$: sender generates a message $msg_1{:}Msg$.

- begin (Sender sent msg_1): sender begins correspondence 1.
- cast *nonce*:Nonce [begin (Sender sent msg_1)]: sender casts *nonce* (so now *nonce*:Nonce [begin (Sender sent msg_1)]).
- gen (msg_2): sender generates a message msg_2:*Msg*.
- begin (Sender sent msg_2): sender begins correspondence 2.
- cast *nonce*:Nonce [begin (Sender sent msg_2)]: sender casts *nonce* again (so now *nonce*:Nonce [begin (Sender sent msg_2)]).

At the end of this trace, *nonce* has been given three incompatible types:

- Of an untrusted message, *nonce*:Un.
- Of a nonce for correspondence 1, *nonce*:Nonce [begin (Sender sent msg_1)].
- Of a nonce for correspondence 2, *nonce*:Nonce [begin (Sender sent msg_2)].

If we are going to allow names to have more than one type, we need to extend the definition of an environment to allow this.

To accommodate the possibility that a name of type Un can be cast to additional types of the form Nonce *es*, we allow additional entries of the form $+x$:Nonce *es* to be added to environments.

Extended environments:

$E ::=$	environment
$\cdots$	as before
$E, +x{:}T$	extended entry (T always takes the form Nonce *es*)

For example, we now allow the environment:

$$\begin{array}{l} \mathit{nonce}{:}\mathsf{Un}, \\ \mathit{msg}_1{:}\mathit{Msg}, \\ +\mathit{nonce}{:}\mathsf{Nonce}\ [\mathsf{begin}\ (\text{Sender sent } \mathit{msg}_1)], \\ \mathit{msg}_2{:}\mathit{Msg}, \\ +\mathit{nonce}{:}\mathsf{Nonce}\ [\mathsf{begin}\ (\text{Sender sent } \mathit{msg}_2)] \end{array}$$

which records that *nonce* originally had type Un, but has since been cast to two other nonce types.

We extend the definitions of $dom(E)$ and $fn(E)$:

Free names $fn(E)$ of an extended environment:

$$fn(E, +x{:}T) \triangleq fn(E) \cup \{x\} \cup fn(T)$$

Domain $dom(E)$ of an extended environment:

$$dom(E, +x{:}T) \triangleq dom(E) \cup \{x\}$$

We also extend the rules for typing with extended environments. The rule rule (Env $+x$) allows the formation of an extended environment $E,+x{:}T$ only when x originally had type Un, and now also has nonce type. This matches the type rule (Proc Cast). The rule (Msg $+x$) extracts type information from such extended environments.

Typing with extended environments:

$$\frac{E \vdash x : \mathsf{Un} \quad E \vdash es}{E,+x{:}\mathsf{Nonce}\ es \vdash \diamond}\ (\text{Env } +x) \qquad \frac{E,+x{:}T,E' \vdash \diamond}{E,+x{:}T,E' \vdash x : T}\ (\text{Msg } +x)$$

We now show some standard properties about our extended type and effect system.

Lemma 3 (Environment) *If $E \vdash \jmath$ then $E \vdash \diamond$.*

Proof Show by induction on the derivation of $E,E' \vdash \jmath$ that if $E,E' \vdash \jmath$ then $E \vdash \diamond$. □

Lemma 4 (Weakening) *If $E,E'' \vdash \jmath$ and $E,E',E'' \vdash \diamond$ then $E,E',E'' \vdash \jmath$.*

Proof An induction on the derivation of $E,E'' \vdash \jmath$. □

Lemma 5 (Substitutivity) *If $E,x{:}T,E' \vdash \jmath$ and $E \vdash M : T$ and $x \notin dom(E')$ then we have $E,(E'\{x{\leftarrow}M\}) \vdash \jmath\{x{\leftarrow}M\}$.*

Proof First show by case analysis that if $E \vdash M : T$ and T is $\mathsf{Ch}(U)$, $\mathsf{Key}(U)$ or $\mathsf{Nonce}\ es$ then M is a name. The result then follows by induction on the derivation of $E,x{:}T,E' \vdash \jmath$. □

Lemma 6 (Subsumption Elimination) *If $E \vdash P : es$ then $E \vdash P : fs$ can be derived without rule (Proc Subsum), where $fs \leq es$.*

Proof First show that if $E \vdash es$ and $E \vdash fs$ then $E \vdash es + fs$, $E \vdash es - fs$ and $E \vdash es \vee fs$. Then show by induction on derivation that if $E \vdash P : es$ then $E \vdash es$. The result then follows by induction on the derivation of $E \vdash P : es$. □

Next, we show a standard property of our labelled transition system, that we can move every use of structural equivalence up to top level. To state this lemma, we use the shorthand $\mathsf{new}\ (D);P$ where:

$$\mathsf{new}\ (x_1{:}T_1,\ldots,x_n{:}T_n);P \stackrel{\Delta}{=} \mathsf{new}\ (x_1{:}T_1);\ldots\mathsf{new}\ (x_n{:}T_n);P$$

This construct enjoys the derived type rule:

Derived type rule for $\mathsf{new}\ (D);P$**:**

$$\frac{E,D \vdash P : es \quad D \text{ is generative}}{E \vdash \mathsf{new}\ (D);P : es - checks(D)}\ (\text{Proc Res } D)\ (\text{where } dom(D) \cap fn(es - checks(D)) = \varnothing)$$

where $checks(x_1{:}T_1,\ldots,x_n{:}T_n) = [\mathsf{check}\ x_1,\ldots,\mathsf{check}\ x_n]$
and $x_1{:}T_1,\ldots,x_n{:}T_n$ is *generative* if and only if $T_1,\ldots,T_n$ are all generative.

Lemma 7 ($\equiv$ Elimination) *If $P \xrightarrow{\alpha} P'$ then:*

$$P \equiv \mathsf{new}\ (D); (Q \mid R) \qquad P' \equiv \mathsf{new}\ (D); (Q' \mid R) \qquad \mathit{fn}(\alpha) \cap \mathit{dom}(D) = \varnothing$$

and $Q \xrightarrow{\alpha} Q'$ can be derived without rules (Trans Par) (Trans Res) or (Trans $\equiv$).

Proof An induction on the derivation of $P \xrightarrow{\alpha} P'$. □

We now show that the effect judgment for processes is preserved by structural equivalence.

Proposition 8 (Subject Equivalence) *If $E \vdash P : es$ and $P \equiv Q$ then $E \vdash Q : es$.*

Proof Prove by induction on the derivation of $\equiv$ that if $P \equiv Q$ or $Q \equiv P$ then if $E \vdash P : es$ then $E \vdash Q : es$. □

Next, we state the main result of this section, a subject reduction property for our labelled transition system.

Proposition 9 (Subject Reduction) *Suppose $E \vdash P : es$.*

(1) *If $P \xrightarrow{\tau} P'$ then $E \vdash P' : es$.*

(2) *If $P \xrightarrow{\mathsf{cast}\ x:T} P'$ then either:*

   (a) $E \vdash P' : es$, or

   (b) $E, +x{:}T \vdash P' : es - fs$ where $T = \mathsf{Nonce}\ fs$ and $fs \leq es$.

(3) *If $P \xrightarrow{\mathsf{check}\ x} P'$ then either:*

   (a) $E \vdash P' : es$, or

   (b) $E \vdash x : \mathsf{Nonce}\ fs$, $E \vdash P' : (es + fs) - [\mathsf{check}\ x]$ and $\mathsf{check}\ x \in es$.

(4) *If $P \xrightarrow{\mathsf{begin}\ M} P'$ then $E \vdash P' : es + [\mathsf{end}\ M]$.*

(5) *If $P \xrightarrow{\mathsf{end}\ M} P'$ then $E \vdash P' : es - [\mathsf{end}\ M]$, and $\mathsf{end}\ M \in es$.*

(6) *If $P \xrightarrow{\mathsf{gen}\ x:T} P'$ then (up to appropriate α-conversion of x) either:*

   (a) $E, x{:}T \vdash P' : es$ and T is generative, or

   (b) $E, x{:}T \vdash P' : es + [\mathsf{check}\ x]$ and T is Un.

Proof

(1) *If $P \xrightarrow{\tau} P'$ then such that $E \vdash P' : es$.*

A case analysis on the derivation of $P \xrightarrow{\tau} P'$.

Case (Trans Comm): By Lemma 6 (Subsumption Elimination) and Lemma 7 ($\equiv$ Elimination); and Rules (Proc Par), (Proc Res D), (Proc Output), (Proc Input), (Proc Output Un) and (Proc Input Un), we have:

$$\begin{array}{rcl} P & \equiv & \mathsf{new}\ (D); (\mathsf{out}\ x\ M \mid \mathsf{inp}\ x\ (y{:}T); Q \mid R) \\ P' & \equiv & \mathsf{new}\ (D); (Q\{y{\leftarrow}M\} \mid R) \\ E,D & \vdash & x : U \\ E,D & \vdash & M : T \\ E,D,y{:}T & \vdash & Q : es_Q \\ E,D & \vdash & R : es_R \\ es & \geq & (es_Q + es_R) - \mathit{checks}\,(D) \end{array}$$

where $T = \mathsf{Un}$ and $U = \mathsf{Un}$, or $U = \mathsf{Ch}(T)$; D is generative; $\mathit{dom}(D) \cap \mathit{fn}((es_Q + es_R) - \mathit{checks}\,(D)) = \varnothing$; and $y \notin \mathit{fn}(es_Q)$. Then by Lemma 5 (Substitutivity), and Rules (Proc Par), (Proc Res D) and (Proc Subsum) we have:

$$E \quad \vdash \quad P' : es$$

as required.

Case (Trans Split): By Lemma 6 (Subsumption Elimination) and Lemma 7 ($\equiv$ Elimination); and Rules (Proc Par), (Proc Res D), (Msg Pair), (Proc Split), (Msg Pair Un), (Proc Split Un) we have:

$$\begin{array}{rcl} P & \equiv & \mathsf{new}\ (D); (\mathsf{split}\ (M,N)\ \mathsf{is}\ (x{:}T,y{:}U); Q \mid R) \\ P' & \equiv & \mathsf{new}\ (D); (Q\{x{\leftarrow}M\}\{y{\leftarrow}N\} \mid R) \\ E,D & \vdash & M : T \\ E,D & \vdash & N : U\{x{\leftarrow}M\} \\ E,D,x{:}T,y{:}U & \vdash & Q : es_Q \\ E,D & \vdash & R : es_R \\ es & \geq & (es_Q + es_R) - \mathit{checks}\,(D) \end{array}$$

where D is generative; $\mathit{dom}(D) \cap \mathit{fn}((es_Q + es_R) - \mathit{checks}\,(D)) = \varnothing$; $x \notin \mathit{fn}(es_Q)$; and $y \notin \mathit{fn}(es_Q)$. Then by Lemma 5 (Substitutivity), and Rules (Proc Par), (Proc Res D) and (Proc Subsum) we have:

$$E \quad \vdash \quad P' : es$$

as required.

Case (Trans Match): By Lemma 6 (Subsumption Elimination) and Lemma 7 ($\equiv$ Elimination); and Rules (Proc Par), (Proc Res D), (Msg Pair), (Proc

Match), (Msg Pair Un), (Proc Match Un) we have:

$$\begin{array}{rcl} P & \equiv & \mathsf{new}\ (D); (\mathsf{match}\ (M,N)\ \mathsf{is}\ (M,y{:}U\{x{\leftarrow}M\}); Q \mid R) \\ P' & \equiv & \mathsf{new}\ (D); (Q\{y{\leftarrow}N\} \mid R) \\ E,D & \vdash & M : T \\ E,D & \vdash & N : U\{x{\leftarrow}M\} \\ E,D,y{:}U\{x{\leftarrow}M\} & \vdash & Q : es_Q \\ E,D & \vdash & R : es_R \\ es & \geq & (es_Q + es_R) - checks\,(D) \end{array}$$

where D is generative; $dom(D) \cap fn((es_Q + es_R) - checks\,(D)) = \varnothing$; and $y \notin fn(es_Q)$. Then by Lemma 5 (Substitutivity), and Rules (Proc Par), (Proc Res D) and (Proc Subsum) we have:

$$E \quad \vdash \quad P' : es$$

as required.

Case (Trans Case Inl): By Lemma 6 (Subsumption Elimination) and Lemma 7 ($\equiv$ Elimination); and Rules (Proc Par), (Proc Res D), (Msg Inl), (Proc Case), (Msg Inl Un), (Proc Case Un), we have:

$$\begin{array}{rcl} P & \equiv & \mathsf{new}\ (D); (\mathsf{case}\ \mathsf{inl}\ (M)\ \mathsf{is}\ \mathsf{inl}\ (x{:}T)\ Q\ \mathsf{is}\ \mathsf{inr}\ (y{:}U)\ R \mid S) \\ P' & \equiv & \mathsf{new}\ (D); (Q\{x{\leftarrow}M\} \mid S) \\ E,D & \vdash & M : T \\ E,D,x{:}T & \vdash & Q : es_Q \\ E,D,y{:}U & \vdash & R : es_R \\ E,D & \vdash & S : es_S \\ es & \geq & (es_Q \vee es_R) + es_S \end{array}$$

where D is generative; $dom(D) \cap fn(((es_Q \vee es_R) + es_S) - checks\,(D)) = \varnothing$; $x \notin fn(es_Q)$; and $y \notin fn(es_R)$. Then by Lemma 5 (Substitutivity), and Rules (Proc Par), (Proc Res D) and (Proc Subsum) we have:

$$E \quad \vdash \quad P' : es$$

as required.

Case (Trans Case Inr): As for Case (Trans Case Inl).

Case (Trans Decrypt): By Lemma 6 (Subsumption Elimination) and Lemma 7 ($\equiv$ Elimination); and Rules (Proc Par), (Proc Res D), (Msg Encrypt), (Proc

Decrypt), (Msg Encrypt Un), (Proc Decrypt Un), we have:

$$\begin{array}{rcl} P & \equiv & \mathsf{new}\ (D); (\mathsf{decrypt}\ \{M\}_N\ \mathsf{is}\ \{x{:}T\}_N; Q \mid R) \\ P' & \equiv & \mathsf{new}\ (D); (Q\{x{\leftarrow}M\} \mid R) \\ E,D & \vdash & M : T \\ E,D,x{:}T & \vdash & Q : es_Q \\ E,D & \vdash & R : es_R \\ es & \geq & (es_Q + es_R) - checks\,(D) \end{array}$$

where D is generative; $dom(D) \cap fn((es_Q + es_R) - checks\,(D)) = \varnothing$; and $x \notin fn(es_Q)$. Then by Lemma 5 (Substitutivity), and Rules (Proc Par), (Proc Res D) and (Proc Subsum) we have:

$$E \quad \vdash \quad P' : es$$

as required.

(2) *If* $P \xrightarrow{\mathsf{cast}\ x:T} P'$ *then either:*

(a) $E \vdash P' : es$, *or*

(b) $E, {+}x{:}T \vdash P' : es - fs$ *where* $T = \mathsf{Nonce}\ fs$ *and* $fs \leq es$.

By Lemmas 6 (Subsumption Elimination) and 7 ($\equiv$ Elimination); and Rules (Proc Par), (Proc Cast) and (Proc Cast Un) we have:

$$\begin{array}{rcl} P & \equiv & \mathsf{new}\ (D); (\mathsf{cast}\ x\ \mathsf{is}\ (y{:}T); Q \mid R) \\ P' & \equiv & \mathsf{new}\ (D); (Q\{y{\leftarrow}x\} \mid R) \\ E & \vdash & x : \mathsf{Un} \\ E,D,y{:}T & \vdash & Q : es_Q \\ E,D & \vdash & R : es_R \\ es & \geq & (es_Q + fs + es_R) - checks\,(D) \end{array}$$

where D is generative; $dom(D) \cap fn(\mathsf{cast}\ x{:}T) = \varnothing$; $dom(D) \cap fn((es_Q + fs + es_R) - checks\,(D)) = \varnothing$; $y \notin fn(es_Q)$; and either $T = \mathsf{Un}$ and $fs = []$ or $T = \mathsf{Nonce}\ fs$.

Case ($T = \mathsf{Un}$ **and** $fs = []$**):** By Lemma 5 (Substitutivity), and Rules (Proc Par), (Proc Res D) and (Proc Subsum) we have:

$$E \quad \vdash \quad P' : es$$

as required.

Case ($T = \mathsf{Nonce}\ fs$**):** Since $E,D,y{:}T \vdash Q : es_Q$, by Lemma 3 (Environment) and Rules (Env x) we have $E,D \vdash T$. Since $fn(T) \cap dom(D) = \varnothing$, and $x \notin dom(D)$, by repeated use of Lemma 5 (Substitutivity) we have $E \vdash T$ and

$E \vdash x : \mathsf{Un}$. Then we use Rule (Env $+x$) to get $E, +x{:}T \vdash \diamond$, so we can apply Lemma 4 (Weakening) to get $E, +x{:}T, D, y{:}T \vdash Q : es_Q$. By Rule (Msg $+x$) we have $E, +x{:}T, D \vdash x : T$ and so we can apply Lemma 5 (Substitutivity), and Rules (Proc Par), (Proc Res D) and (Proc Subsum) to get:

$$E, +x{:}T \quad \vdash \quad P' : es - fs$$

and $fs \leq es$ as required.

(3) *If* $P \xrightarrow{\mathsf{check}\ x} P'$ *then either:*

(a) $E \vdash P' : es$, *or*

(b) $E \vdash x : \mathsf{Nonce}\ fs$, $E \vdash P' : (es + fs) - [\mathsf{check}\ x]$ *and* $\mathsf{check}\ x \in es$.

By Lemmas 6 (Subsumption Elimination) and 7 ($\equiv$ Elimination); and rules (Proc Par), (Proc Res D), (Proc Check), and (Proc Check Un) we have:

$$\begin{array}{rcl} P & \equiv & \mathsf{new}\ (D); (\mathsf{check}\ x\ \mathsf{is}\ x; Q \mid R) \\ P' & \equiv & \mathsf{new}\ (D); (Q \mid R) \\ E, D & \vdash & x : T \\ E, D & \vdash & Q : es_Q \\ E, D & \vdash & R : es_R \\ es & \geq & ((es_Q - fs) + fs' + es_R) - checks\,(D) \end{array}$$

where D is generative; $dom(D) \cap fn(\mathsf{check}\ x{:}T) = \varnothing$; $dom(D) \cap fn(((es_Q - fs) + fs' + es_R) - checks\,(D)) = \varnothing$; and either $T = \mathsf{Un}$ and $fs = fs' = []$ or $T = \mathsf{Nonce}\ fs$ and $fs' = [\mathsf{check}\ x]$.

Case ($T = \mathsf{Un}$ **and** $fs = fs' = []$**)** By Lemma 5 (Substitutivity), and Rules (Proc Par), (Proc Res D) and (Proc Subsum) we have:

$$E \quad \vdash \quad P' : es$$

as required.

Case ($T = \mathsf{Nonce}\ fs$ **and** $fs' = [\mathsf{check}\ x]$**)** By Lemma 5 (Substitutivity), and Rules (Proc Par), (Proc Res D) and (Proc Subsum) we have:

$$\begin{array}{rcl} E & \vdash & x : \mathsf{Nonce}\ fs \\ E & \vdash & P' : (es + fs) - [\mathsf{check}\ x] \end{array}$$

and $\mathsf{check}\ x \in es$ as required.

(4) *If* $P \xrightarrow{\mathsf{begin}\ M} P'$ *then* $E \vdash P' : es + [\mathsf{end}\ M]$.

By Lemmas 6 (Subsumption Elimination) and 7 ($\equiv$ Elimination); and Rules (Trans Begin), (Proc Par), (Proc Res D), and (Proc Begin) we have:

$$\begin{array}{rcl} P & \equiv & \mathsf{new}\ (D); (\mathsf{begin}\ M; Q \mid R) \\ P' & \equiv & \mathsf{new}\ (D); (Q \mid R) \\ E, D & \vdash & M : T \\ E, D & \vdash & Q : es_Q \\ E, D & \vdash & R : es_R \\ es & \geq & ((es_Q - [\mathsf{end}\ M]) + es_R) - checks\,(D) \end{array}$$

where D is generative; $dom(D) \cap fn(\mathsf{begin}\ M) = \varnothing$; and $dom(D) \cap fn(((es_Q - [\mathsf{end}\ M]) + es_R) - checks\,(D)) = \varnothing$. Then by Lemma 5 (Substitutivity), and Rules (Proc Par), (Proc Res D) and (Proc Subsum) we have:

$$E \quad \vdash \quad P' : es + [\mathsf{end}\ M]$$

as required.

(5) *If* $P \xrightarrow{\mathsf{end}\ M} P'$ *then* $E \vdash P' : es - [\mathsf{end}\ M]$, *and* $\mathsf{end}\ M \in es$.

By Lemmas 6 (Subsumption Elimination) and 7 ($\equiv$ Elimination); and Rules (Trans End), (Proc Par), (Proc Res D) and (Proc End) we have:

$$\begin{array}{rcl} P & \equiv & \mathsf{new}\ (D); (\mathsf{end}\ M \mid Q) \\ P' & \equiv & \mathsf{new}\ (D); Q \\ E, D & \vdash & M : T \\ E, D & \vdash & Q : es_Q \\ es & \geq & (es_Q + [\mathsf{end}\ M]) - checks\,(D) \end{array}$$

where D is generative; $dom(D) \cap fn(\mathsf{end}\ M) = \varnothing$; and $dom(D) \cap fn((es_Q + [\mathsf{end}\ M]) - checks\,(D)) = \varnothing$. Then by Lemma 5 (Substitutivity), and Rules (Proc Par), (Proc Res D) and (Proc Subsum) we have:

$$E \quad \vdash \quad P' : es - [\mathsf{end}\ M]$$

and $\mathsf{end}\ M \in es$ as required.

(6) *If* $P \xrightarrow{\mathsf{gen}\ x:T} P'$ *then (up to appropriate* α*-conversion of* x*) either:*

(a) $E, x{:}T \vdash P' : es$ *and* T *is generative, or*

(b) $E, x{:}T \vdash P' : es + [\mathsf{check}\ x]$ *and* T *is* Un.

By Lemmas 6 (Subsumption Elimination) and 7 ($\equiv$ Elimination); and Rules (Trans Gen), (Proc Par),(Proc Res D) and (Proc Res) we have:

$$\begin{array}{rcl} P & \equiv & \mathsf{new}\ (D); (\mathsf{new}\ (x{:}T); Q \mid R) \\ P' & \equiv & \mathsf{new}\ (D); (Q \mid R) \\ E, D, x{:}T & \vdash & Q : es_Q \\ E, D & \vdash & R : es_R \\ es & \geq & ((es_Q - [\mathsf{check}\ x]) + es_R) - checks\,(D) \end{array}$$

where D is generative; $dom(D) \cap fn(\mathsf{gen}\, x{:}T) = \varnothing$; $dom(D) \cap fn((es_Q + [\mathsf{end}\, M]) - checks(D)) = \varnothing$; $x \notin fn(es_Q)$; and T is generative.

Case ($T = \mathsf{Un}$) By Lemma 5 (Substitutivity), Lemma 4 (Weakening), and Rules (Proc Par), (Proc Res D) and (Proc Subsum) we have:

$$E, x{:}\mathsf{Un} \quad \vdash \quad P' : es + [\mathsf{check}\, x]$$

as required.

Case ($T \neq \mathsf{Un}$) Since $E, D, x{:}T \vdash Q : es_Q$, we have by Lemma 3 (Environment) $E, D, x{:}T \vdash es_Q$ and so (since $E, D, x{:}T \not\vdash [\mathsf{check}\, x]$) we have $\mathsf{check}\, x \notin es_Q$. By Lemma 5 (Substitutivity), Lemma 4 (Weakening), and Rules (Proc Par) (Proc Res D) and (Proc Subsum) we have:

$$E, x{:}T \quad \vdash \quad P' : es$$

as required. □

B.4 Proof of Safety

The purpose of this appendix is to prove the type safety result, Theorem 1 (Safety). It asserts that a process assigned the empty effect is safe. This theorem is a key fact in the proof of the main result of the paper, Theorem 2 (Robust Safety), in Section 4.3.

To prove Theorem 1 (Safety), we actually prove a stronger invariant, Proposition 18 (Transition Safety), about processes with non-empty effects. To state the invariant we introduce a function $ends(E \vdash es)$ which computes the multiset of end-events represented by an effect es. With this notation, we can roughly state the invariant as follows:

- If $E \vdash P : es$ and $P \xrightarrow{s} P'$ then we can find E' and es' such that $E' \vdash P' : es'$ where $ends(E \vdash es) + begins(s) \geq ends(E' \vdash es') + ends(s)$.

From this, we deduce that every process $E \vdash P : []$ is safe.

However, we have some work ahead of us, in particular in defining the function $ends(es)$. A naïve definition would just be to count all of the $\mathsf{end}\, M$ effects in es, but this ignores the latent effect of nonces. Consider the following typing:

$$x{:}\mathsf{Un}, +x{:}\mathsf{Nonce}\, [\mathsf{end}\, M] \vdash (\mathsf{check}\, x\ \mathsf{is}\ x; \mathsf{end}\, M) : [\mathsf{check}\, x]$$

The process has trace $\mathsf{check}\, x, \mathsf{end}\, M$, which has an unbalanced $\mathsf{end}\, M$, even though the effect of the process only contains a $\mathsf{check}\, x$ effect. So, in addition to counting the end-events, we need also to compute the end-events that may be unleashed by nonce effects.

Another problem is that we have to make sure that nonces are used *linearly*, that is, at most once. For example we need to ban processes such as:

$$x{:}\mathsf{Un}, +x{:}\mathsf{Nonce}\, [\mathsf{end}\, M] \vdash (\mathsf{check}\, x\ \mathsf{is}\ x; \mathsf{check}\, x\ \mathsf{is}\ x; \mathsf{end}\, M; \mathsf{end}\, M) : [\mathsf{check}\, x, \mathsf{check}\, x]$$

which use a nonce more than once, or even worse:

$$x{:}\mathsf{Un}, +x{:}\mathsf{Nonce}\,[\mathsf{end}\,M, \mathsf{check}\,x] \vdash \\ (\mathsf{check}\,x\,\mathsf{is}\,x; \mathsf{end}\,M; \mathsf{check}\,x\,\mathsf{is}\,x; \mathsf{end}\,M; \ldots) : [\mathsf{check}\,x]$$

where we have a self-certifying nonce with the *cyclic* type $x : \mathsf{Nonce}\,[\mathsf{end}\,M, \mathsf{check}\,x]$, which allows an unbounded number of unbalanced assertions.

We first define the *latent effects* of a well-typed message $E \vdash M : \mathsf{Un}$. If M is anything other than a name, then the latent effects are emtpy. Otherwise if $M = x$, we find all the occurrences of $x{:}\mathsf{Nonce}\ es$ in E, and sum them. For example:

$$\mathit{effects}\,(x{:}\mathsf{Un}, +x{:}\mathsf{Nonce}\,[\mathsf{end}\,M], +x{:}\mathsf{Nonce}\,[\mathsf{check}\,N] \vdash x : \mathsf{Un}) \\ = [\mathsf{end}\,M, \mathsf{check}\,N]$$

Effects $\mathit{effects}\,(E \vdash M : \mathsf{Un})$ **of a typed message** $E \vdash M : \mathsf{Un}$**:**

$$\begin{array}{l}
\mathit{effects}\,(E, x{:}T \vdash x : \mathsf{Un}) \triangleq [] \\
\mathit{effects}\,(E, +x{:}\mathsf{Nonce}\ es \vdash x : \mathsf{Un}) \triangleq \mathit{effects}\,(E \vdash x : \mathsf{Un}) + es \\
\mathit{effects}\,(E, x{:}T \vdash y : \mathsf{Un}) \triangleq \mathit{effects}\,(E \vdash y : \mathsf{Un}) \qquad (\text{when } x \neq y) \\
\mathit{effects}\,(E, +x{:}T \vdash y : \mathsf{Un}) \triangleq \mathit{effects}\,(E \vdash y : \mathsf{Un}) \qquad (\text{when } x \neq y) \\
\mathit{effects}\,(E \vdash M : \mathsf{Un}) \triangleq [] \qquad (\text{when } M \text{ is not a name})
\end{array}$$

As discussed above, we maintain an invariant for well-typed systems, which is that they are *nonce linear*, so they only only allow each nonce to be checked once. We define this in terms of a predicate $Ms \trianglelefteq \mathit{checks}\,(E \vdash es)$ which can be read as 'Ms is a lower bound on the nonce checks allowed by $E \vdash es$'. For example:

$$\begin{array}{rcl}
[x] & \trianglelefteq & \mathit{checks}\,(x{:}\mathsf{Un} \vdash [\mathsf{check}\,x]) \\
[x,y] & \trianglelefteq & \mathit{checks}\,(x{:}\mathsf{Un}, y{:}\mathsf{Un} \vdash [\mathsf{check}\,x, \mathsf{check}\,y]) \\
[x,y] & \trianglelefteq & \mathit{checks}\,(x{:}\mathsf{Un}, y{:}\mathsf{Un}, +x{:}\mathsf{Nonce}\,[\mathsf{check}\,y] \vdash [\mathsf{check}\,x]) \\
[x,x] & \trianglelefteq & \mathit{checks}\,(x{:}\mathsf{Un} \vdash [\mathsf{check}\,x, \mathsf{check}\,x]) \\
[x,x] & \trianglelefteq & \mathit{checks}\,(x{:}\mathsf{Un}, +x{:}\mathsf{Nonce}\,[\mathsf{check}\,x] \vdash [\mathsf{check}\,x])
\end{array}$$

When we calculate the lower bound on the nonces allowed by $E \vdash es$, we include the latent effects of es. In particular, the last example shows that we have to be careful about cyclic uses of nonces.

Lower bound $Ms \trianglelefteq \mathit{checks}\,(E \vdash es)$ **of the nonces of a typed effect** $E \vdash es$**:**

$$\frac{}{[] \trianglelefteq \mathit{checks}\,(E \vdash es)}\ (\text{Nonces } [])$$

$$\frac{Ms - [M] \trianglelefteq \mathit{checks}\,(E \vdash es + \mathit{effects}\,(E \vdash M : \mathsf{Un}))}{Ms \trianglelefteq \mathit{checks}\,(E \vdash es + [\mathsf{check}\,M])}\ (\text{Nonces check } M)$$

Having defined $Ms \trianglelefteq \mathit{checks}\,(E \vdash es)$, we can define the nonce linear and nonce acyclic effects:

Nonce linear typed effects $E \vdash es$

A typed effect $E \vdash es$ is *nonce linear* if and only if
there is no M such that $[M, M] \trianglelefteq \mathit{checks}\,(E \vdash es)$.

Nonce acyclic typed effects $E \vdash es$

A typed effect $E \vdash es$ is *nonce acyclic* if and only if
there is no $[M] \trianglelefteq \mathit{checks}\,(E \vdash es)$ such that $[M, M] \trianglelefteq \mathit{checks}\,(E \vdash [\mathsf{check}\ M])$.

For example:

$x{:}\mathsf{Un} \vdash [\mathsf{check}\ x]$	is linear and acyclic
$x{:}\mathsf{Un}, y{:}\mathsf{Un} \vdash [\mathsf{check}\ x, \mathsf{check}\ y]$	is linear and acyclic
$x{:}\mathsf{Un}, y{:}\mathsf{Un}, +x{:}\mathsf{Nonce}\ \mathsf{check}\ y \vdash [\mathsf{check}\ x]$	is linear and acyclic
$x{:}\mathsf{Un} \vdash [\mathsf{check}\ x, \mathsf{check}\ x]$	is acyclic but not linear
$x{:}\mathsf{Un}, +x{:}\mathsf{Nonce}\ \mathsf{check}\ x \vdash [\mathsf{check}\ x]$	is neither linear nor acyclic

We can now show some properties about nonce linear effects, and nonce acyclic effects, in particular that every nonce linear effect is nonce acyclic.

Lemma 10 (Nonce monotonicity) *If* $Ms \trianglelefteq \mathit{checks}\,(E \vdash es)$ *and* $E \vdash fs$ *then* $Ms \trianglelefteq \mathit{checks}\,(E \vdash es + fs)$.

Proof An induction on the proof of $Ms \trianglelefteq \mathit{checks}\,(E \vdash es)$. □

Lemma 11 (Nonce transitivity) *If we have* $[M] \trianglelefteq \mathit{checks}\,(E \vdash es)$ *and also that* $Ms \trianglelefteq \mathit{checks}\,(E \vdash [\mathsf{check}\ M])$ *then* $Ms \trianglelefteq \mathit{checks}\,(E \vdash es)$.

Proof An induction on the proof of $[M] \trianglelefteq \mathit{checks}\,(E \vdash es)$. To get $[M] \trianglelefteq \mathit{checks}\,(E \vdash es)$ we must have used Rule (Nonces $\mathsf{check}\ M$) and so either:

- $\mathsf{check}\ M \in es$, so by Lemma 10 (Nonce monotonicity) we have $Ms \trianglelefteq \mathit{checks}\,(E \vdash es)$, or
- $es = fs + [\mathsf{check}\ N]$ and $[M] \trianglelefteq \mathit{checks}\,(E \vdash fs + \mathit{effects}\,(E \vdash N : \mathsf{Un}))$, so by induction $Ms \trianglelefteq \mathit{checks}\,(E \vdash fs + \mathit{effects}\,(E \vdash N : \mathsf{Un}))$, and so by Rule (Nonces $\mathsf{check}\ M$) $Ms \trianglelefteq \mathit{checks}\,(E \vdash es)$.

The result follows. □

Lemma 12 (Linear implies acyclic) *If* $E \vdash es$ *is nonce linear then* $E \vdash es$ *is nonce acyclic.*

Proof Follows from Lemma 11 (Nonce transitivity). □

We can now define $\mathit{ends}\,(E \vdash es)$ for a nonce acyclic effect $E \vdash es$. This is used to set up the invariant for our type safety result.

Endings $ends\,(E \vdash es)$ **of a nonce acyclic effect** $E \vdash es$**:**

$$ends\,(E \vdash []) \triangleq []$$
$$ends\,(E \vdash es + [\mathsf{end}\ M]) \triangleq ends\,(E \vdash es) + [M]$$
$$ends\,(E \vdash es + [\mathsf{check}\ M]) \triangleq ends\,(E \vdash es) + ends\,(E \vdash effects\,(E \vdash M : \mathsf{Un}))$$

Note that $ends\,(E \vdash es)$ is not well-defined for nonce cyclic effects, for example if:

$$E \quad = \quad x{:}\mathsf{Un}, +x{:}\mathsf{Nonce}\,[\mathsf{check}\ x, \mathsf{end}\ M]$$

then:

$$\begin{array}{l} ends\,(E \vdash [\mathsf{check}\ x]) \\ \quad = \quad ends\,(E \vdash [\mathsf{check}\ x, \mathsf{end}\ M]) \\ \quad = \quad ends\,(E \vdash [\mathsf{check}\ x, \mathsf{end}\ M, \mathsf{end}\ M]) \\ \quad = \quad \cdots \end{array}$$

However, they are well-defined for nonce acyclic effects, which is enough for our purposes.

Lemma 13 (End Definedness) *If $E \vdash es$ is nonce acyclic then $ends\,(E \vdash es)$ is well-defined.*

Proof For any finite set of names X, let $ends\,(E \vdash es)_X$ be defined:

$$\begin{array}{l} ends\,(E \vdash [])_X \\ \quad = \quad [] \\ ends\,(E \vdash es + [\mathsf{end}\ M])_X \\ \quad = \quad ends\,(E \vdash es)_X + [M] \\ ends\,(E \vdash es + [\mathsf{check}\ M])_X \\ \quad = \quad \begin{cases} ends\,(E \vdash es)_X + ends\,(E \vdash effects\,(E \vdash M : \mathsf{Un}))_{X-\{M\}} & \text{if } M \in X \\ ends\,(E \vdash es)_X & \text{otherwise} \end{cases} \end{array}$$

It is routine to see that $ends\,(E \vdash es)_X$ is well-defined, by induction first on X then on es. We then show by induction on the definition of $ends\,(E \vdash es)_X$ that:

$$\text{if } \forall [x] \trianglelefteq checks\,(E \vdash es)\,.\,x \in X \text{ then } ends\,(E \vdash es)_X = ends\,(E \vdash es)$$

In particular, we have that $ends\,(E \vdash es)_{dom(E)} = ends\,(E \vdash es)$, and so $ends\,(E \vdash es)$ is well-defined. □

We can now prove some lemmas, leading up to the type safety results we need to show that effect-free processes are safe.

Lemma 14 (End Homomorphism) $ends\,(E \vdash es + fs) = ends\,(E \vdash es) + ends\,(E \vdash fs)$.

Proof An induction on *es*. □

Lemma 15 (End $+x$) *If* $[x] \not\trianglelefteq \mathit{checks}(E \vdash es)$ *then* $\mathit{ends}(E, +x{:}T \vdash es) = \mathit{ends}(E \vdash es)$.

Proof An induction on *es*. □

Lemma 16 (End Nonce) *If* $E \vdash x : \mathsf{Nonce}$ *es and* $E \vdash x : \mathsf{Un}$ *then* $\mathit{ends}(E \vdash [\mathsf{check}\, x]) \geq \mathit{ends}(E \vdash es)$.

Proof Show by induction on E that $es \leq \mathit{effects}(E \vdash x : \mathsf{Un})$. The result then follows by Lemma 14 (End Homomorphism). □

Lemma 17 (End Add Nonce) *If* $E \vdash es + fs$ *is nonce linear then* $\mathit{ends}(E \vdash es + fs) \geq \mathit{ends}(E, +x{:}\mathsf{Nonce}\ fs \vdash es)$

Proof An induction on *es*. The only interesting case is when:

$$es \quad = \quad es' + [\mathsf{check}\, x]$$

Since $E \vdash es + fs$ is nonce linear, we have $[x] \not\trianglelefteq \mathit{checks}(E \vdash es')$ and $[x] \not\trianglelefteq \mathit{checks}(E \vdash \mathit{effects}(E, +x{:}\mathsf{Nonce}\ fs \vdash x : \mathsf{Un}))$ and so by Lemmas 14 (End Homomorphism) and 15 (End $+x$):

$$\begin{aligned}
&\mathit{ends}(E \vdash es + fs) \\
&\quad = \quad \mathit{ends}(E \vdash es' + [\mathsf{check}\, x] + fs) \\
&\quad = \quad \mathit{ends}(E \vdash es' + [\mathsf{check}\, x]) + \mathit{ends}(E \vdash fs) \\
&\quad = \quad \mathit{ends}(E \vdash es') + \mathit{ends}(E \vdash \mathit{effects}(E \vdash x : \mathsf{Un})) + \mathit{ends}(E \vdash fs) \\
&\quad = \quad \mathit{ends}(E \vdash es') + \mathit{ends}(E \vdash \mathit{effects}(E \vdash x : \mathsf{Un}) + fs) \\
&\quad = \quad \mathit{ends}(E, +x{:}\mathsf{Nonce}\ fs \vdash es') + \\
&\qquad\quad \mathit{ends}(E, +x{:}\mathsf{Nonce}\ fs \vdash \mathit{effects}(E, +x{:}\mathsf{Nonce}\ fs \vdash x : \mathsf{Un})) \\
&\quad = \quad \mathit{ends}(E, +x{:}\mathsf{Nonce}\ fs \vdash es' + [\mathsf{check}\, x]) \\
&\quad = \quad \mathit{ends}(E, +x{:}\mathsf{Nonce}\ fs \vdash es)
\end{aligned}$$

as required.

Proposition 18 (Transition Safety) *If* $E \vdash es$ *is nonce linear,* $E \vdash P : es$ *and* $P \xrightarrow{\alpha} P'$ *then* $E' \vdash P' : es'$ *for some nonce linear* $E' \vdash es'$ *such that* $\mathit{ends}(E \vdash es) + \mathit{begins}(\alpha) \geq \mathit{ends}(E' \vdash es') + \mathit{ends}(\alpha)$.

Proof A case analysis on α:

Case ($\alpha = \mathsf{cast}\ x{:}T$) By Proposition 9 (Subject Reduction), we have one of the following cases:

Subcase ($E \vdash P' : es$) Immediate.

Subcase ($E, +x{:}T \vdash P' : es - fs$ where $T = \mathsf{Nonce}\ fs$ and $fs \leq es$) Then, using Lemma 17 (End Add Nonce) we have:

$$\begin{array}{rcl} & & ends\,(E \vdash es) + begins\,(\alpha) \\ & = & ends\,(E \vdash es) \\ & \geq & ends\,(E, +x{:}T \vdash es - fs) \\ & = & ends\,(E, +x{:}T \vdash es - fs) + ends\,(\alpha) \end{array}$$

and $E, +x{:}T \vdash es - fs$ is nonce linear as required.

Case ($\alpha = \mathsf{check}\ x$) By Proposition 9 (Subject Reduction), we have one of the following cases:

Subcase ($E \vdash P' : es$) Immediate.

Subcase ($E \vdash x : \mathsf{Nonce}\ fs$, $E \vdash P' : (es + fs) - [\mathsf{check}\ x]$, $\mathsf{check}\ x \in es$) Given Lemmas 16 (End Nonce) and 14 (End Homomorphism) we have:

$$\begin{array}{rcl} & & ends\,(E \vdash es) + begins\,(\alpha) \\ & = & ends\,(E \vdash es) \\ & = & ends\,(E \vdash (es - [\mathsf{check}\ x]) + [\mathsf{check}\ x]) \\ & = & ends\,(E \vdash (es - [\mathsf{check}\ x])) + ends\,(E \vdash [\mathsf{check}\ x]) \\ & \geq & ends\,(E \vdash (es - [\mathsf{check}\ x])) + ends\,(E \vdash fs) \\ & = & ends\,(E \vdash (es + fs) - [\mathsf{check}\ x]) \\ & = & ends\,(E \vdash (es + fs) - [\mathsf{check}\ x]) + ends\,(\alpha) \end{array}$$

and $E \vdash (es + fs) - [\mathsf{check}\ x]$ is nonce linear as required.

Case ($\alpha = \mathsf{begin}\ M$) Follows directly from Proposition 9 (Subject Reduction).

Case ($\alpha = \mathsf{end}\ M$) Follows directly from Proposition 9 (Subject Reduction).

Case ($\alpha = \mathsf{gen}\ x{:}T$) Follows directly from Proposition 9 (Subject Reduction).

Case ($\alpha = \tau$) Follows directly from Proposition 9 (Subject Reduction). □

Proposition 19 ($\xrightarrow{s}$ Safety) *If $E \vdash es$ is nonce linear, $E \vdash P : es$ and $P \xrightarrow{s} P'$ then $E' \vdash P' : es'$ for some nonce linear $E' \vdash es'$ such that $ends\,(E \vdash es) + begins\,(s) \geq ends\,(E' \vdash es') + ends\,(s)$.*

Proof An induction on the derivation of $P \xrightarrow{s} P'$.

Case (Trace $\equiv$) We have:

$$\begin{array}{rcl} s & = & \varepsilon \\ P & \equiv & P' \end{array}$$

By Proposition 8 (Subject Equivalence):

$$E \vdash P' : es$$

and so (since $\mathit{begins}(\varepsilon) = \mathit{ends}(\varepsilon) = [\,]$ we have:

$$\mathit{ends}(E \vdash es) + \mathit{begins}(s) \quad \geq \quad \mathit{ends}(E \vdash es) + \mathit{ends}(s)$$

as required.

Case (Trace Event) We have:

$$\begin{array}{rcl} P & \xrightarrow{\alpha} & P'' \\ P'' & \xrightarrow{t} & P' \\ s & = & \alpha, t \end{array}$$

By Proposition 18 (Transition Safety) we can find nonce linear $E'' \vdash es''$ such that:

$$\begin{array}{rcl} E'' & \vdash & P'' : es'' \\ \mathit{ends}(E \vdash es) + \mathit{begins}(\alpha) & \geq & \mathit{ends}(E'' \vdash es'') + \mathit{ends}(\alpha) \end{array}$$

By induction, we can find nonce linear $E' \vdash es'$ such that:

$$\begin{array}{rcl} E' & \vdash & P' : es' \\ \mathit{ends}(E'' \vdash es'') + \mathit{begins}(t) & \geq & \mathit{ends}(E' \vdash es') + \mathit{ends}(t) \end{array}$$

and so:

$$\begin{array}{rcl} & & \mathit{ends}(E \vdash es) + \mathit{begins}(s) \\ & = & \mathit{ends}(E \vdash es) + \mathit{begins}(\alpha) + \mathit{begins}(t) \\ & \geq & \mathit{ends}(E'' \vdash es'') + \mathit{ends}(\alpha) + \mathit{begins}(t) \\ & \geq & \mathit{ends}(E' \vdash es') + \mathit{ends}(\alpha) + \mathit{ends}(t) \\ & = & \mathit{ends}(E' \vdash es') + \mathit{ends}(s) \end{array}$$

as required. □

We have now done all the work required to show our main theorem: any effect-free process is safe.

Proof of Theorem 1 (Safety) *If $E \vdash P : [\,]$ then P is safe.*

Proof If $P \xrightarrow{s} P'$ then we use Proposition 19 ($\xrightarrow{s}$ Safety) to get:

$$\begin{array}{rcl} & & \mathit{begins}(s) \\ & = & \mathit{ends}(E \vdash [\,]) + \mathit{begins}(s) \\ & \geq & \mathit{ends}(E' \vdash es') + \mathit{ends}(s) \\ & \geq & \mathit{ends}(s) \end{array}$$

Thus, P is safe. □

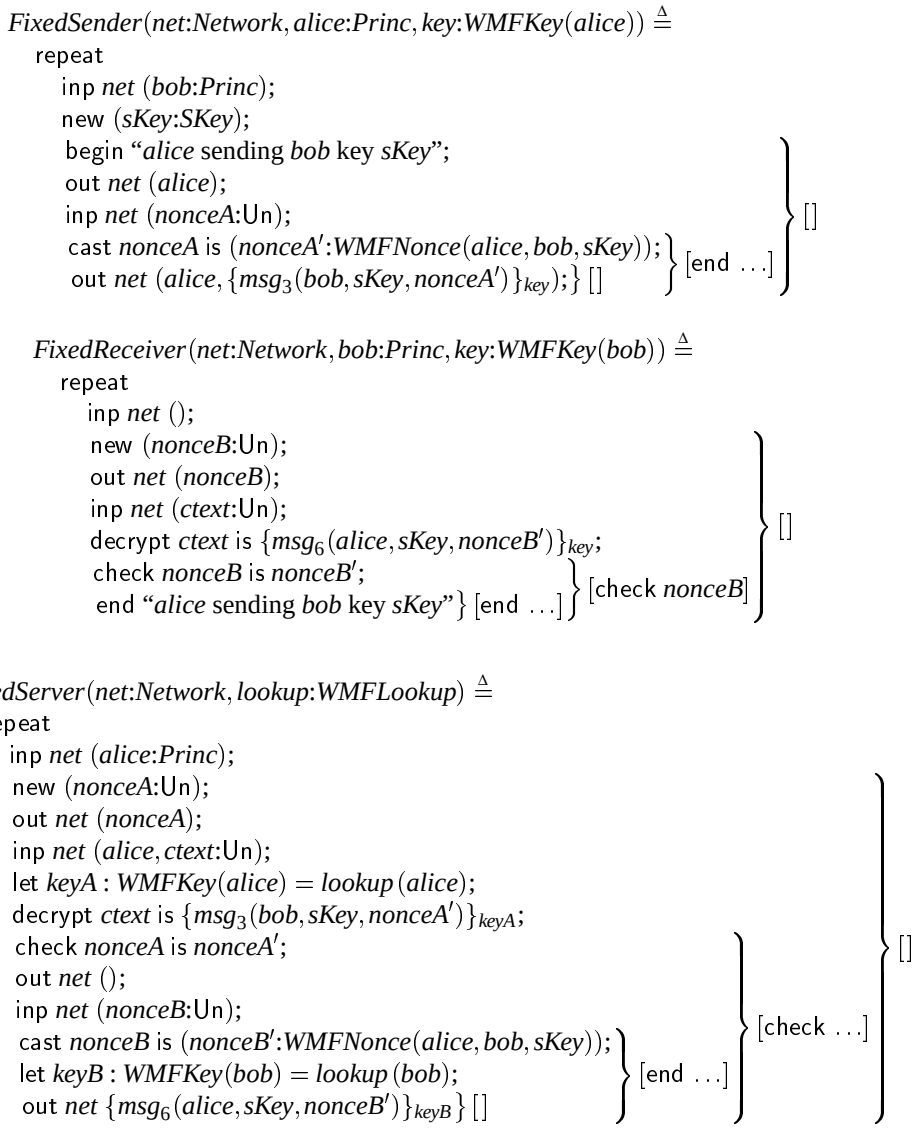

Figure 1: Type checked participants in the Wide Mouth Frog protocol

References

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

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

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

[4] M. Abadi and R. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

[5] R. Anderson and R. Needham. Programming Satan's computer. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lectures Notes in Computer Science*, pages 426–440. Springer, 1995.

[6] M. Bellare and P. Rogaway. Entity authentication and key distribution. In *Advances in Cryptology: CRYPTO'93*, volume 773 of *Lectures Notes in Computer Science*, pages 232–249. Springer, 1994.

[7] G. Berry and G. Boudol. The chemical abstract machine. *Theoretical Computer Science*, 96(1):217–248, April 1992.

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

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

[10] E. Cohen. TAPS: A first-order verifier for cryptographic protocols. In *13th Computer Security Foundations Workshop*, pages 144–158. IEEE Computer Society Press, 2000.

[11] S. Dal Zilio and A.D. Gordon. Region analysis and a π-calculus with groups. In *Mathematical Foundations of Computer Science 2000 (MFCS2000)*, volume 1893 of *Lectures Notes in Computer Science*, pages 1–21. Springer, 2000.

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

[13] A. Durante, R. Focardi, and R. Gorrieri. A compiler for analysing cryptographic protocols. *ACM Transactions on Software Engineering and Methodology*, 2000. To appear.

[14] R. Focardi, R. Gorrieri, and F. Martinelli. Message authentication through non-interference. In *International Conference on Algebraic Methodology And Software Technology (AMAST2000)*, volume 1816 of *Lectures Notes in Computer Science*, pages 258–272. Springer, 2000.

[15] D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on Lisp and Functional Programming*, pages 28–38, 1986.

[16] D. Gollmann. What do we mean by entity authentication? In *1995 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 46–54, 1995.

[17] L. Gong, R. Needham, and R. Yahalom. Reasoning about beliefs in cryptographic protocols. In *1990 IEEE Computer Society Symposium on Research in Security and Privacy*, 1990.

[18] A.D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. In *Mathematical Foundations of Programming Semantics 17*, Electronic Notes in Theoretical Computer Science. Elsevier, 2001. To appear.

[19] A.D. Gordon and D. Syme. Typing a multi-language intermediate code. In *28th ACM Symposium on Principles of Programming Languages (POPL'01)*, pages 248–260, 2001.

[20] O. Grumberg and D.E. Long. Model checking and modular verification. *ACM Transactions on Programming Languages and Systems*, 16(3):843–871, 1994.

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

[22] J. Heather. *'Oh! . . . Is it* really *you?' Using rank functions to verify authentication protocols*. PhD thesis, Royal Holloway, University of London, 2000.

[23] J. Heather, G. Lowe, and S. Schneider. How to prevent type flaw attacks on security protocols. In *13th Computer Security Foundations Workshop*, pages 255–268. IEEE Computer Society Press, 2000.

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

[25] M. Hennessy and J. Riely. Resource access control in systems of mobile agents. In *3rd International Workshop on High-Level Concurrent Languages*, volume 16(3) of *Electronic Notes in Theoretical Computer Science*. Elsevier, 1998.

[26] T. Lindholm and F. Yellin. *The Java*$^{\text{TM}}$ *Virtual Machine Specification*. Addison-Wesley, 1997.

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

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

[29] J.M. Lucassen. *Types and effects, towards the integration of functional and imperative programming*. PhD thesis, MIT, 1987. Available as Technical Report MIT/LCS/TR–408, MIT Laboratory for Computer Science.

[30] 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.

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

[32] G. Necula. Proof-carrying code. In *24th ACM SIGPLAN–SIGACT Symposium on Principles of Programming Languages*, pages 106–119. ACM Press, 1997.

[33] D. Otway and O. Rees. Efficient and timely "mutual authentication". *Operating Systems Review*, 21(1):8–10, 1987.

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

[35] B. Pierce and E. Sumii. Relating cryptography and polymorphism. Available from the authors, 2000.

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

[37] C. Skalka and S. Smith. Static enforcement of security with types. In P. Wadler, editor, *2000 ACM International Conference on Functional Programming*, pages 34–45, 2000.

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

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

[40] T.Y.C. Woo and S.S. Lam. Authentication for distributed systems. *Computer*, 25(1):39–52, 1992.

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