Authenticity by Typing for Security Protocols

Andrew D. Gordon
Microsoft Research
Cambridge, United Kingdom

Alan Jeffrey
DePaul University
Chicago, Illinois, U.S.A.

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.

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 [39], 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 [39] describes a formal semantics for correspondence assertions but suggests no verification techniques. Marrero, Clarke, and Jha [28] 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, 14, 25, 12]. 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.

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 [24, 30, 17], for checking access control [23, 35], and, most recently, other type systems for cryptographic primitives [33, 2]. This paper develops some of the constructs in Abadi's system, and proposes a new type and effect system [13, 27] 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 [36] of strand spaces [37], Heather and Schneider's algorithm [22, 20] for computing Schneider's rank functions [34], and Cohen's resolution-based theorem prover TAPS [9]. Non-examples include most approaches based on model-checking [26], which are automatic but require bounds on the size of the opponent or the protocols, and most approaches based on theorem-proving [7, 32], 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 [8, 15]. 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 [16], 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.

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.

Abadi and Gordon's spi-calculus [3] is an extension of Milner, Parrow, and Walker's π-calculus [29] with abstract forms of encryption and decryption, akin to the idealised versions introduced by Dolev and Yao [11]. 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.

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
$\quad x$	$\quad$ name
$\quad (M,N)$	$\quad$ pair
$\quad ()$	$\quad$ empty tuple
$\quad \mathsf{inl}\ (M)$	$\quad$ left injection
$\quad \mathsf{inr}\ (M)$	$\quad$ 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.

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
$\mathsf{case}\ M\ \mathsf{is\ inl}\ (x{:}T)\ P\ \mathsf{is\ inr}\ (y{:}U)\ Q$	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 $\mathsf{case}\ M\ \mathsf{is\ inl}\ (x{:}T)\ P\ \mathsf{is\ inr}\ (y{:}U)\ Q$ checks the tagged union M. If M is $\mathsf{inl}\ (L)$, the process behaves as $P\{x \leftarrow L\}$. If M is $\mathsf{inr}\ (N)$ it behaves as $Q\{y \leftarrow N\}$. Otherwise, it deadlocks.
- A process $\mathsf{decrypt}\ M\ \mathsf{is}\ \{x{:}T\}_N;P$ decrypts M using 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.

The free and bound names of a process are defined in the normal way. 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.

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.

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

These are:

- The process $\mathit{FlawedSender}(\mathit{net},\mathit{key})$ 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.

- The process $\mathit{FlawedReceiver}(net,key)$ 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.

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

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

3 Specifying Protocols

Woo and Lam [39] 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. We formalize these ideas by adding begin- and end-event annotations to spi processes.

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
$\quad \ldots$	as in Section 2.1
$\quad \mathsf{begin}\ L;P$	begin-assertion
$\quad \mathsf{end}\ L;P$	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.

Given this informal semantics, we give an informal definition of process safety. (We formalize these definitions in the full version of the paper.)

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 [18]). 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 descriptions 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.

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

This attacker carries out a replay attack on the system, causing the receiver to assert end *msg* twice, even though the sender has only asserted begin *msg* once.

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	$B \rightarrow A:$	N
Message 2	$A \rightarrow B:$	$\{M,N\}_K$
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 $\mathit{FixedSystem}(\mathit{net})$. 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.

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. Other base types such as int or $\mathsf{boolean}$ 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

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 begin $L;P$ and end $L;P$ are essentially:

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

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

- end $L : [\mathsf{end}\ L]$
- begin $L;$end $L : [\,]$
- end $L;$end $L : [\mathsf{end}\ L, \mathsf{end}\ L]$
- begin $L;$end $L;$end $L : [\mathsf{end}\ L]$
- begin $L;$begin $L;$end $L;$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 [10, 16].

- If $M : \mathsf{Ch}(T)$ and $N : T$ then out $M\ N : [\,]$.
- If $M : \mathsf{Ch}(T)$ and $P : es$ then 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. 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* [19]. 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.

(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 Nonce *es*, where *es* is an effect. The sender casts the nonce $N : \mathsf{Un}$ to this trusted type using the new process cast N is $(x{:}\mathsf{Nonce}\ es);P$. At runtime, this process simply binds the message N to the variable x of type 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 check N is $N';Q$ to check for the nonce $N : \mathsf{Un}$ which it made public earlier. Only a cast can populate the type Nonce *es*. So the presence of the message $N' :$ Nonce *es* proves there was a preceding execution of a cast process.

(4) To guarantee that each nonce N is the subject of no more than one name-check, we introduce a new atomic effect, written check N. We include check N in the effect of a name-check check N is $N';Q$ on a nonce N. When checking name generation new $(N{:}\mathsf{Un});P$, we check that 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 check N is $N';Q$, where $N' :$ Nonce *es*, there is a distinct preceding execution of a process cast N 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 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 cast N is $(x{:}\mathsf{Nonce}\ es);P : (es_P + es)$.
- If $N : \mathsf{Un}$ and $N' : \mathsf{Nonce}\ es$ and $Q : es_Q$
  then check N is $N';Q : ((es_Q - es) + [\mathsf{check}\ N])$.
- If $P : es_P$ then new $(N);P : (es_P - [\mathsf{check}\ N])$.

In Section 4.4 we give an example of these type rules, showing that the *FixedSystem*(net) is robustly safe.

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

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

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.

Types:

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

The free names of a type are defined in the usual way, where the only binder is x being bound in U in the type $(x{:}T, U)$. For example, x is free in $\mathsf{Nonce}\ [\mathsf{check}\ x]$ but not in $(x{:}\mathsf{Un}, \mathsf{Nonce}\ [\mathsf{check}\ x])$. 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	
$\quad \ldots$		as in Sections 2.1 and 3.1
$\quad \mathsf{cast}\ M\ \mathsf{is}\ (x{:}T); P$		cast to nonce type
$\quad \mathsf{match}\ M\ \mathsf{is}\ (N, y{:}U); P$		pair pattern matching

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

- The process $\mathsf{cast}\ M\ \mathsf{is}\ (x{:}T); P$ casts the message M to the type T, by binding the variable x to M, and then running P. (This process can only be typed by our type system if T is of the form $\mathsf{Nonce}\ es$.)
- The process $\mathsf{match}\ M\ \mathsf{is}\ (N, y); P$ is similar to $\mathsf{split}\ M\ \mathsf{is}\ (x, y); P$ 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 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	
$\quad \varnothing$		empty
$\quad E, x{:}T$		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 $\varnothing$) (Env x) (where $x \notin dom(E)$)

$$\frac{}{\varnothing \vdash \diamond} \qquad \frac{E \vdash T}{E, x{:}T \vdash \diamond}$$

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 $\varnothing$) (Effect End) (Effect Check)

$$\frac{E \vdash \diamond}{E \vdash \varnothing} \qquad \frac{E \vdash es \quad E \vdash L : T}{E \vdash es + [\mathsf{end}\ L]} \qquad \frac{E \vdash es \quad E \vdash N : \mathsf{Un}}{E \vdash es + [\mathsf{check}\ N]}$$

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 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) $\dfrac{E', x{:}T, E'' \vdash \diamond}{E', x{:}T, E'' \vdash x : T}$

Rules for Messages of Trusted Type:

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

(Msg Inl) $\dfrac{E \vdash M : T \quad E \vdash U}{E \vdash \mathsf{inl}\ (M) : T + U}$ (Msg Inr) $\dfrac{E \vdash T \quad E \vdash N : U}{E \vdash \mathsf{inr}\ (N) : T + U}$

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

Rules for Messages of Untrusted Type:

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

(Msg Inl Un) $\dfrac{E \vdash M : \mathsf{Un}}{E \vdash \mathsf{inl}\ (M) : \mathsf{Un}}$ (Msg Inr Un) $\dfrac{E \vdash N : \mathsf{Un}}{E \vdash \mathsf{inr}\ (N) : \mathsf{Un}}$

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

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* $x_1{:}\mathsf{Un},\ldots,x_n{:}\mathsf{Un} \vdash M : \mathsf{Un}$.

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

(Proc Par) $\dfrac{E \vdash P : es \quad E \vdash Q : fs}{E \vdash P \mid Q : es + fs}$ (Proc Repeat) $\dfrac{E \vdash P : []}{E \vdash \mathsf{repeat}\ P : []}$

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

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

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. 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 $es \le es'$ 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 $es \le es''$ and $es' \le es''$.

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

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

(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 typecheck 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 $x_1{:}\mathsf{Un}, \ldots, x_n{:}\mathsf{Un} \vdash O : []$.*

The following theorem, proved in the full version of this paper, 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.*

4.4 Typing the Example

Our example *FixedSystem*(*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.

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 Nonce [end *msg*]. Therefore, the shared key has type Key(*msg*:*Msg*, *nonce*:Nonce [end *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 end *msg* event.

Therefore, we introduce the types

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

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

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

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

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

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 *TypedSystem*(*net*:*Network*) is robustly safe. This proves the following authenticity property by typing.

Authenticity: The process *TypedSystem*(*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 [8]. 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 [38] 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 [31] 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, 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

Abbreviations Used in Examples

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

- Dependent record types $(x_1{:}T_1,\dots,x_n{:}T_n)$, rather than just pairs.
- Tagged union types $(\ell_1(T_1) \mid \cdots \mid \ell_n(T_n))$ rather than just binary choice $T+U$.

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

For reasons of length, we will not provide full spi implementations of each of these protocols, and instead just provide the typings. In each case it is fairly routine to reconstruct the spi code. The full specifications are provided in the full version of this paper.

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 [8] 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 α.1	A begins	"A sending B key K_{AB}"
Message α.1	$A \to I$	A
Message β.4	$I \to A$	$()$
Message β.5	$A \to I$	N_A
Message α.2	$I \to A$	N_A
Message α.3	$A \to I$	$A, \{B, K_{AB}, N_A\}_{K_{AS}}$
Message β.6	$I \to A$	$\{B, K_{AB}, N_A\}_{K_{AS}}$
Event β.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 $\mathsf{begin}\ M$ and $\mathsf{end}\ M$ statements to specify the desired correspondence property. Then we can try to define the types appropriately. For most of the types, it is fairly routine:

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

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:

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

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

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

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 [21] 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:

$$WMFMsg_3(alice) \triangleq \\ (bob{:}Princ, sKey{:}SKey, nonce{:}WMFNonce(alice, bob, sKey))$$

and the type for Message 6:

$$WMFMsg_6(bob) \triangleq \\ (alice{:}Princ, sKey{:}SKey, nonce{:}WMFNonce(alice, bob, sKey))$$

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

$$WMFMsg(princ) \triangleq \\ (msg_3(WMFMsg_3(princ)) \mid msg_6(WMFMsg_6(princ)))$$

We can then check that the safe versions of the principals are effect-free. Applying the results of this paper, we get:

- The Wide Mouth Frog protocol is effect-free, and hence robustly safe.

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.

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 [38] 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:

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

and Anderson and Needham [5] observe that Message 3 should be changed to:

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

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:

$$\begin{aligned}
&\mathit{WLMsg}(\mathit{princ}) \triangleq \\
&\quad (\mathit{msg}_3(\mathit{WLMsg}_3(\mathit{princ})) \mid \mathit{msg}_5(\mathit{WLMsg}_5(\mathit{princ}))) \\
&\mathit{WLMsg}_3(\mathit{alice}) \triangleq \\
&\quad (\mathit{bob}{:}\mathit{Princ}, \mathit{nonce}{:}\mathit{WLNonce}(\mathit{alice}, \mathit{bob})) \\
&\mathit{WLMsg}_5(\mathit{bob}) \triangleq \\
&\quad (\mathit{alice}{:}\mathit{Princ}, \mathit{nonce}{:}\mathit{WLNonce}(\mathit{alice}, \mathit{bob}))
\end{aligned}$$

It is routine to rewrite this protocol in the syntax of the spi-calculus. We can then apply the results of this paper to get:

- The Woo and Lam protocol 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 [31] 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	$S \to B$	$\{\mathit{msg}_4(A, B, K_{AB}, N_A)\}_{K_{AS}}$, $\{\mathit{msg}_3(A, B, K_{AB}, N_B)\}_{K_{BS}}$
Event 3	B ends	"responder B shares K_{AB} with A"
Message 4	$B \to A$	$\{\mathit{msg}_4(A, B, K_{AB}, N_A)\}_{K_{AS}}$
Event 4	A ends	"initiator A shares K_{AB} with B"

We can allocate types to this protocol:

$$\begin{aligned}
&\mathit{ORKey}(\mathit{princ}) \triangleq \\
&\quad \mathsf{Key}((\mathit{msg}_3(\mathit{ORMsg}_3(\mathit{princ})) \mid \mathit{msg}_4(\mathit{ORMsg}_4(\mathit{princ})))) \\
&\mathit{ORMsg}_3(\mathit{bob}) \triangleq \\
&\quad (\mathit{alice}{:}\mathit{Princ}, \mathit{bob}'{:}\mathit{Princ}, \mathit{sKey}{:}\mathit{SKey}, \\
&\quad\ \mathit{nonce}{:}\mathit{ORNonce}_3(\mathit{alice}, \mathit{bob}, \mathit{sKey})) \\
&\mathit{ORMsg}_4(\mathit{alice}) \triangleq \\
&\quad (\mathit{alice}'{:}\mathit{Princ}, \mathit{bob}{:}\mathit{Princ}, \mathit{sKey}{:}\mathit{SKey}, \\
&\quad\ \mathit{nonce}{:}\mathit{ORNonce}_3(\mathit{alice}, \mathit{bob}, \mathit{sKey})) \\
&\mathit{ORNonce}_3(\mathit{alice}, \mathit{bob}, \mathit{sKey}) \triangleq \\
&\quad \mathsf{Nonce}\ [\mathsf{end}\ \text{“responder } \mathit{bob} \text{ shares } \mathit{sKey} \text{ with } \mathit{alice}\text{”}] \\
&\mathit{ORNonce}_4(\mathit{alice}, \mathit{bob}, \mathit{sKey}) \triangleq \\
&\quad \mathsf{Nonce}\ [\mathsf{end}\ \text{“initiator } \mathit{alice} \text{ shares } \mathit{sKey} \text{ with } \mathit{bob}\text{”}] \\
&\mathit{ORLookup} \triangleq \\
&\quad (\mathit{princ}{:}\mathit{Princ}) \to \mathit{ORKey}(\mathit{princ})
\end{aligned}$$

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.

References

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

[2] M. Abadi and B. Blanchet. Secrecy types for asymmetric communication. In *Foundations of Software Science and Computation Structures (FoSSaCS'01)*, 2001. To appear.

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

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

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

[10] S. Dal Zilio and A. 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.

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

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

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

[14] 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.

[15] 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.

[16] A. 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.

[17] A. 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.

[18] O. Grumberg and D. Long. Model checking and modular verification. *ACM Trans. Prog. Lang. Syst.*, 16(3):843–871, 1994.

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

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

[21] 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.

[22] 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.

[23] 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.

[24] T. Lindholm and F. Yellin. *The Java™ Virtual Machine Specification*. Addison-Wesley, 1997.

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

[26] 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.

[27] J. 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.

[28] W. Marrero, E. 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.

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

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

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

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

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

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

[35] 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.

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

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

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

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