Typing One-to-One and One-to-Many Correspondences in Security Protocols

Andrew D. Gordon[1] and Alan Jeffrey[2]

[1] Microsoft Research
[2] DePaul University

Abstract. Both one-to-one and one-to-many correspondences between events, sometimes known as injective and non-injective agreements, respectively, are widely used to specify correctness properties of cryptographic protocols. In earlier work, we showed how to typecheck one-to-one correspondences for protocols expressed in the spi-calculus. We present a new type and effect system able to verify both one-to-one and one-to-many correspondences.

1 Motivation

A common strategy for specifying a security protocol is to enumerate expected correspondences between actions of the principals running the protocol. For example, if Alice sends a series of instructions to her banker Bob, each acceptance by Bob of an instruction apparently from Alice should correspond to an attempt by Alice to issue the instruction to Bob. Ruling out potential attacks on a protocol reduces to ruling out violations of a set of correspondences.

The idea of specifying authenticity properties as correspondences has appeared in many forms [Gol02], but Woo and Lam [WL93] were early pioneers. Woo and Lam formalise the progress of principals through a protocol via labelled events, divided into two kinds, begin-events and end-events. A correspondence assertion specifies that in any run of the system, each end-event has a distinct, preceding begin-event with the same label. Authenticity properties typically do not rely on liveness, so there is no requirement that every begin-event is subsequently followed by a corresponding end-event. Our previous work on typechecking authenticity properties of security protocols [GJ01b,GJ01a,GJ02,GP02] rests on formulating correspondence assertions by annotating programs with begin- and end-events.

Woo and Lam's assertions are *one-to-one* in the sense there is a distinct begin-event for each end-event. Let a *one-to-many* correspondence be the more liberal condition that there exists a preceding—but not necessarily distinct—begin-event for every end-event. One-to-one correspondences are desirable to rule out replay attacks; if Alice sends a single money transfer instruction to Bob, our specification should prevent Bob being duped into acting on it twice. On the other hand, there are at least two situations when it is appropriate to relax the formal one-to-one requirement.

First, security may not actually depend on one-to-one correspondence. For example, if Alice sends an account balance request, and Bob encrypts his reply, it may be harmless if the attacker replaying the request were to cause Bob to send two balance messages to Alice. Replay protection requires some state to be maintained, which in some applications may be more expensive than simply tolerating replays. For example, messages containing nonces cannot usefully be cached by proxies.

Second, even if security depends on a one-to-one correspondence, it may be acceptable to check only a one-to-many correspondence formally, and then to verify the absence of replays outside the formalism. For example, a protocol based on timestamps may prevent replays by relying on a complex infrastructure for securing and synchronising clocks. It may be expedient to check for a one-to-many correspondence formally, and to argue informally that a standard use of timestamps in fact achieves a one-to-one correspondence. In some protocols, such as those involving freshly generated nonces included in event labels, it may be obvious that no two end-events have the same label. If labels are unique, one-to-one and one-to-many correspondences are equivalent, so it may be preferable simply to verify a one-to-many correspondence.

This paper introduces a type and effect system for verifying both one-to-one and one-to-many correspondences. Hence, we generalize our previous work on type and effect systems [GJ01b,GJ01a] for one-to-one correspondences in the spi-calculus [AG99]. There are many prior systems for verifying one-to-many correspondences; this is the first based on typechecking. It shows that our methodology is not limited to the one-to-one case. The main new construct in our type theory are *ok-types*, types whose inhabitation proves it is ok to perform end-events. Moreover, the technicalities of our operational semantics are simpler than in previous work, allowing a corresponding simplification of our type soundness proofs. Using types to embody security invariants fits cleanly into the development cycle, and is becoming increasingly popular [LY97,LLL$^+$02,SM02]. Still, unlike some other protocol analyses, our method requires human intervention to annotate the protocol with type information, and like other type systems, rejects some well-behaved programs.

We verify that correspondences hold in spite of an attacker able to monitor, modify, and replay messages, and, given suitable keys, to encrypt and decrypt messages [DY83]. Hence, like many other formal techniques, our verifications do not rule out attacks that side-step this attacker model, such as cryptanalyses of the underlying cryptographic algorithms. Unlike some other formal techniques, we have no general model of insider attacks, although some specific situations can be modelled.

We can position our work on the spectrum—ranging from extensional to intensional—of security specifications, as follows. At the extensional end, specifications describe the intended service provided by the protocol, for example, in terms of behavioural equivalence [FG94,AFG98,Aba99,AFG00,FGM00]. At the intensional end, specifications describe the underlying mechanism, in terms of states or events [AB01,WL93,Low97,Ros96,Gol02,Mea96,Pau98,Sch98,GT02].

Correspondences are towards the intensional end of the spectrum; our begin- and end-assertions are intermingled with implementation code. Still, we can informally picture correspondences using message-sequence notations, and there are advantages from making specifications executable.

The rest of the paper is structured as follows. Section 2 gives examples of specifying protocols via correspondences. Section 3 explains how to formulate the problem of checking correspondences within a version of the spi-calculus. Section 4 describes how to typecheck one-to-many correspondences in a spi-calculus, by introducing new ok-types. Section 5 summarises and concludes.

2 Specifying Protocols by Correspondences

The goal of this section is to illustrate one-to-many correspondences via some protocol examples, including a couple of classic protocols from the original paper on the BAN logic [BAN89], and to sketch their encoding within our typed spi-calculus.

Example 1: A Basic One-to-Many Correspondence. Suppose two principals, A and B, share a secret symmetric key K_{AB}. The following protocol protocol uses a timestamp to protect encrypted messages sent from A to B.

Event 1	A begins!	$\mathit{Sending}(A, B, M)$
Message 1	$A \rightarrow B:$	$\{M, T_A\}_{K_{AB}}$
Event 2	B ends	$\mathit{Sending}(A, B, M)$

The protocol consists of a single message, Message 1, sent from the initiator A to the responder B. Begin- and end-events mark the progress of the principals through a run of the protocol. Event 1 is a one-to-many begin-event, whose label, $\mathit{Sending}(A, B, M)$ identifies the principals taking part in this run of the protocol, together with the instruction, M, that A wishes to send to B. Event 2 is an end-event by B, whose label matches the earlier begin-event by A. Event 1 occurs immediately before A initiates the protocol. Event 2 occurs only after B is satisfied the protocol run has succeeded, and records the parties B regards as taking part, and the instruction to be acted on.

In general, the assertion induced by begin- and end-events is that each end-event has a corresponding begin-event with the same label. We annotate the begin-event, Event 1, with "!", to indicate that it can match one or more end-events with the same label. This one-to-many correspondence assertion means that every message M accepted by B as coming from A actually came from A, but it does not rule out replays. To specify a one-to-one correspondence, we would omit the "!" to indicate that the begin-event can be matched at most once.

The timestamp serves to narrow the window of opportunity for replay attacks, but it may or may not entirely rule out replays. Typical implementations would reject timestamps older than some limit, for example, five minutes. Stateful servers may also record previously seen timestamps, so as to prevent replays.

Some attention needs to be paid to secure synchronisation of clocks, to prevent, for example, attacks based on fooling a node into issuing requests with future timestamps, for later replay.

Our theory can establish one-to-many correspondences, as specified above, but since it does not deal explicitly with the passage of time, it cannot show that a particular use of timestamps does in fact establish a one-to-one correspondence. We leave this to an informal argument. On the other hand, our earlier papers show how to typecheck the one-to-one correspondences achieved by various kinds of nonce exchanges.

Types for Example 1. We use this basic example to illustrate our use of types and processes to model protocols. Here is the type for the shared key K_{AB}.

$$\mathit{SharedKey} \triangleq \mathsf{Key}(x{:}\mathit{Msg}, t{:}\mathsf{Un}, \mathsf{Ok}(\mathsf{end}\ \mathit{Sending}(A, B, x)))$$

This says K_{AB} is a secret key for encrypting triples (M, T_A, ok), where M has type Msg, T_A has type Un, and ok has type $\mathsf{Ok}(\mathsf{end}\,\mathit{Sending}(A, B, M))$, that is, the type $\mathsf{Ok}(\mathsf{end}\,\mathit{Sending}(A, B, x))$ with the parameter x instantiated to M, the component of the triple labelled x. We leave the type Msg of instructions conveyed by the protocol unspecified. Messages of type Un are public data known to the opponent; we assign timestamps the type Un since they may be published to the opponent. The message ok of type $\mathsf{Ok}(\mathsf{end}\ \mathit{Sending}(A, B, M))$ may be thought of as a certificate testifying that it is justified to perform an end-event labelled $\mathit{Sending}(A, B, M)$.

Ok-types, such as $\mathsf{Ok}(\mathsf{end}\,\mathit{Sending}(A, B, x))$, are one of the main innovations in this paper, compared to our previous type systems for the spi-calculus. The general form is $\mathsf{Ok}(\mathcal{R})$ where $\mathcal{R}$ is a *resource*, a representation of events that may be safely performed. A simple example of a resource is a finite multiset of events, written $\mathsf{end}\ L_1 \mid \cdots \mid \mathsf{end}\ L_n$.

Ok-types are meaningful at compile time—during typechecking—but convey no information at runtime. At compile time, we exploit ok-types to prove that end-events performed in one part of a program are justified by preceding begin-events elsewhere. At runtime, there is exactly one value, written ok, of ok-type. It is the unique inhabitant of every type $\mathsf{Ok}(\mathcal{R})$, and therefore takes no space to represent. We consider $\mathsf{ok} : \mathsf{Ok}(\mathcal{R})$ as simply an annotation, proof that the events in $\mathcal{R}$ are safe to perform. Accordingly, although we formalize Message 1 as an encrypted triple, $\{M, T_A, \mathsf{ok}\}_{K_{AB}}$, as far as runtime behaviour is concerned this is equivalent to the encrypted pair, $\{M, T_A\}_{K_{AB}}$, of our informal description.

Interlude: Resources, Events, and Safety. To describe our process semantics, we suppose that every state of our system includes an imaginary tuplespace, recording those end-events that may be safely performed. It is purely a formal device to record the progress of protocol participants, and cannot be used for actual communication. Accordingly, the opponent cannot read or write the tuplespace. The contents of the tuplespace is a resource, $\mathcal{R}$ (as also found in an ok-type). Here is the full syntax of resources:

Resources:

$\mathcal{R}, \mathcal{S}$ ::=	resource: record of allowed end-events
$\mathsf{end}\ L$	allows one end-event labelled L
$!\mathcal{R}$	allows whatever $\mathcal{R}$ allows, arbitrarily often
$\mathcal{R} \mid \mathcal{S}$	allows whatever R and S allow
$\mathbf{0}$	allows nothing

Syntactically, replication binds tighter than composition; so that $!R \mid R'$ means $(!R) \mid R'$. We identify resources up to an equivalence relation $\mathcal{R} \equiv \mathcal{R}'$, the least congruence relation to satisfy the equations $\mathcal{R} \mid (\mathcal{R}' \mid \mathcal{R}'') \equiv (\mathcal{R} \mid \mathcal{R}') \mid \mathcal{R}''$, $\mathcal{R} \mid \mathcal{R}' \equiv \mathcal{R}' \mid R$, $\mathcal{R} \mid \mathbf{0} \equiv \mathcal{R}$, $!\mathcal{R} \equiv \mathcal{R} \mid !\mathcal{R}$, $!\mathbf{0} \equiv \mathbf{0}$, $!!\mathcal{R} \equiv !\mathcal{R}$, and $!(\mathcal{R} \mid \mathcal{R}') \equiv !\mathcal{R} \mid !\mathcal{R}'$. (Hence, a resource denotes a multiset of end-events, each of which has either finite or infinite multiplicity.) Let $\mathsf{fn}(\mathcal{R})$ be the set of names occurring in any label occurring in $\mathcal{R}$.

We include begin- and end-assertions in the syntax of our spi-calculus to formalize correspondences. These assertions read or write the imaginary tuplespace. A one-to-one begin-assertion $\mathsf{begin}\ L; P$ adds $\mathsf{end}\ L$ to the tuplespace, then runs P. A one-to-many begin-assertion $\mathsf{begin}\,!L; P$ adds the replicated resource $!\mathsf{end}\ L$ to the tuplespace, then runs P. An end-assertion $\mathsf{end}\ L; P$ consumes the resource $\mathsf{end}\,L$ from the imaginary tuplespace. The absence of $\mathsf{end}\,L$, when running $\mathsf{end}\,L; P$, is a deadlock signifying violation of a correspondence assertion. We say a system is *safe* if no reachable state has such a deadlock. Moreover, a system P is *robustly safe* if $P \mid O$ is safe for any process O representing an opponent. We formalize these definitions later; they are similar to those in our previous work, except for the addition of replicated resources and one-to-many begin-assertions, to allow modelling of both one-to-one and one-to-many correspondences.

Processes for Example 1. Returning to our example, the following process represents an attempt by A to send M to B.

$$\begin{aligned} P_A(M) \triangleq\ & \mathsf{begin}\ !\mathit{Sending}(A,B,M);\\ & \mathsf{new}(T_A{:}\mathsf{Un}); \mathsf{out}\ \mathit{net}\langle\{M, T_A, \mathsf{ok}\}_{K_{AB}}\rangle \end{aligned}$$

The process starts with a one-to-many begin-assertion, that adds the replicated resource $!\mathsf{end}\ \mathit{Sending}(A,B,M)$ to the imaginary tuplespace. Since it is replicated, it matches many end-events. The restriction operator $\mathsf{new}(T_A{:}\mathsf{Un})$ generates a fresh timestamp T_A. The process ends by an output of the encrypted triple $\{M, T_A, \mathsf{ok}\}_{K_{AB}}$ on the public channel net.

We postpone a detailed discussion of intuitions behind the type system to Section 4, but note that the message ok can be assigned $\mathsf{Ok}(\mathsf{end}\ \mathit{Sending}(A,B,M))$ only due to the presence of $!\mathsf{end}\ \mathit{Sending}(A,B,M)$ in the imaginary tuplespace.

Next, process P_B represents B receiving a single message from A.

$$\begin{aligned} P_B \triangleq\ & \mathsf{inp}\ \mathit{net}(c{:}\mathsf{Un}); \mathsf{decrypt}\ c\ \mathsf{is}\ \{\mathit{xto}\}_{K_{AB}};\\ & \mathsf{split}\ \mathit{xto}\ \mathsf{is}\ (x{:}\mathit{Msg}, t{:}\mathsf{Un}, o{:}\mathsf{Ok}(\mathsf{end}\ \mathit{Sending}(A,B,x)));\\ & \mathsf{exercise}\ o; \mathsf{end}\ \mathit{Sending}(A,B,x) \end{aligned}$$

The process starts with a blocking receive of a message c off the public channel net. It then attempts to decrypt the message with the key K_{AB}, and split the plaintext into its three components x, t, and o. If c is not encrypted with K_{AB} the process will immediately terminate; we assume there is sufficient redundancy in the ciphertext to detect this situation. At runtime, the process $\mathsf{exercise}\ o$; does nothing; control unconditionally falls through to the end-assertion $\mathsf{end}\ \mathit{Sending}(A, B, x)$ to indicate that B accepts the message x from A. At compile time, the process $\mathsf{exercise}\ o$; exploits the ok-type assigned to o to show that the following end-assertion cannot deadlock.

Finally, we formalize principal A attempting to send a sequence $M_1, \ldots, M_n$ of messages to B as the process $\mathit{Sys}(M_1, \ldots, M_n)$.

$$\mathit{Sys}(M_1, \ldots, M_n) \triangleq \mathsf{new}(K_{AB}{:}\mathit{SharedKey}); (P_A(M_1) \mid \cdots \mid P_A(M_n) \mid !P_B)$$

The process $P_A(M_1) \mid \cdots \mid P_A(M_n)$ is n copies of P_A, one for each of the messages to be sent, running in parallel. The replicated process $!P_B$ is a server run by B that repeatedly runs P_B to try to receive a message from A. The new binder delimits the scope of the key K_{AB} to be the processes representing A and B. The opponent is some process running alongside $\mathit{Sys}(M_1, \ldots, M_n)$ that may send and receive messages on the public net channel. The fact that the opponent O does not know the key K_{AB} is represented by O not being within the scope of the binder for K_{AB}.

This completes our reduction of the one-to-many specification of our protocol to the problem of showing that the process $\mathit{Sys}(M_1, \ldots, M_n)$ is robustly safe. Robust safety follows from the type system of Section 4.

Example 2: The Wide-Mouthed-Frog Protocol. Assume that each of a number of principals A, B, ... shares a key K_{AS}, K_{BS}, ..., with a trusted server S. The Wide-Mouthed-Frog protocol [BAN89] allows one of these principals to create and communicate a key to another, via the server S, using timestamps for replay protection. Here is the protocol and its specification using a one-to-many correspondence.

Event 1	A begins!	$\mathit{Sending}(A, B, K_{AB})$
Message 1	$A \to S:$	$\{\mathit{msg}_1(T_A, B, K_{AB})\}_{K_{AS}}$
Message 2	$S \to B:$	$\{\mathit{msg}_2(T_S, A, K_{AB})\}_{K_{BS}}$
Event 2	B ends	$\mathit{Sending}(A, B, K_{AB})$

Message 1 communicates the new key K_{AB} from A to S, who checks that the timestamp T_A is fresh, and then forwards the key on to B tagged with its own timestamp T_S. The specification says that each time B believes it has received a key K_{AB} from A, by performing Event 2, then in fact A has earlier begun the protocol with B, intending to send K_{AB}, by performing Event 1.

If each principal rejects any message whose timestamp is older than the last message received from the same principal, then the protocol can prevent replays. If so, the specification can be strengthened to a one-to-one correspondence (though this cannot be checked within our type system).

We tag the plaintexts of Messages 1 and 2 to prevent any confusion of the two. This thwarts the “type flaw” attack, reported by Anderson and Needham [AN95], in which an attacker replays Message 2 as Message 1, in order to keep the timestamp fresh.

Types for Example 2. Let variable p range over the principals A, B, ... The longterm key shared between any principal p and the server S has the following type $\mathit{Princ}(p)$, where SKey is the type of session keys, and $\mathit{Payload}$ is the unspecified type of payload data.

$$\begin{aligned}
\mathit{SKey} &\triangleq \mathsf{Key}(\mathit{PayLoad})\\
\mathit{Princ}(p) &\triangleq \mathsf{Key}(\mathsf{Union}(\\
&\qquad\qquad \mathit{msg}_1(\mathit{ta} : \mathsf{Un}, b : \mathsf{Un}, \mathit{kab} : \mathit{SKey}, \mathsf{Ok}(\mathsf{end}\ \mathit{Sending}(p, b, \mathit{kab}))),\\
&\qquad\qquad \mathit{msg}_2(\mathit{ts} : \mathsf{Un}, a : \mathsf{Un}, \mathit{kab} : \mathit{SKey}, \mathsf{Ok}(\mathsf{end}\ \mathit{Sending}(a, p, \mathit{kab})))))
\end{aligned}$$

This says the longterm key for principal p is a secret symmetric key for encrypting two kinds of messages, tagged with msg_1 or msg_2. The first kind are triples (T_A, B, K_{AB}), together with an ok indicating that an end-event labelled $\mathit{Sending}(p, B, K_{AB})$ is safe. Similarly, the second kind are triples (T_S, A, K_{AB}), together with an ok indicating that an end-event labelled $\mathit{Sending}(A, p, K_{AB})$ is safe.

Much as for Example 1, we can encode the protocol behaviour and its specification as a process, but we omit the details.

Example 3: BAN Kerberos. Our final example is a one-to-many specification of an early version of Kerberos [BAN89]. Again, the starting assumption is that there is a set of principals each of which shares a key with an authentication server S. The protocol allows an initiator A to request a fresh session key K_{AB} from S for communication with B, and to have S send the key to both A and B. Here is the protocol and its specification.

Event 1	A begins!	$\mathit{Init}(A, B)$
Message 1	$A \to S :$	A, B
Event 2	S begins!	$\mathit{KeyGenInit}(A, B, K_{AB})$
Event 3	S begins!	$\mathit{KeyGenResp}(A, B, K_{AB})$
Message 2	$S \to A :$	$\{\mathit{msg}_{2a}(T_S, L, B, K_{AB}, \mathit{Tkt})\}_{K_{AS}}$ where $\mathit{Tkt} = \{\mathit{msg}_{2b}(T_S, L, A, K_{AB})\}_{K_{BS}}$
Event 4	A ends	$\mathit{KeyGenInit}(A, B, K_{AB})$
Message 3	$A \to B :$	$\mathit{Tkt}, \{\mathit{msg}_3(A, T_A)\}_{K_{AB}}$
Event 5	B ends	$\mathit{KeyGenResp}(A, B, K_{AB})$
Event 6	B ends	$\mathit{Init}(A, B)$
Event 7	B begins!	$\mathit{Resp}(A, B)$
Message 4	$B \to A :$	$\{\mathit{msg}_4(T_A)\}_{K_{AB}}$
Event 8	A ends	$\mathit{Resp}(A, B)$

Compared to the original presentation, we have added tags to messages, eliminated the decrement of the timestamp in Message 4, and swapped the order of a couple of components of Message 2.

The terms T_A and T_S are timestamps, and L is a lifetime. The specification gives guarantees to the principals performing end-events. At Event 4, A is guaranteed that S generated the session key for use with B, and, symmetrically, at Event 5, B is guaranteed that S generated the session key for use with A. At Event 8, the initiator A is guaranteed that the responder B has run the protocol with A, and, symmetrically, at Event 6, the responder B is guaranteed that the initiator A has run the protocol with B. The server S receives no guarantee that A or B are present or running the protocol.

There are more detailed formal analyses of more recent versions of Kerberos; see the paper [BCJS02], and its bibliography.

Types for Example 3. The longterm key shared between any principal p and the server S has the following type $Princ(p)$, where $SKey(a, b)$ is the type of session keys shared between a and b, and $Payload$ is the unspecified type of payload data.

$$
\begin{aligned}
Princ(p) \triangleq \mathsf{Key}(\mathsf{Union}(&\\
&msg_{2a}(ta : \mathsf{Un}, l{:}\mathsf{Un}, b{:}\mathsf{Un}, kab{:}SKey(p, b), tkt{:}\mathsf{Un},\\
&\qquad \mathsf{Ok}(\mathsf{end}\ KeyGenInit(p, b, kab))),\\
&msg_{2b}(ts{:}\mathsf{Un}, l{:}\mathsf{Un}, a{:}\mathsf{Un}, kab{:}SKey(a, p),\\
&\qquad \mathsf{Ok}(\mathsf{end}\ KeyGenResp(a, p, kab)))))\\
SKey(a, b) \triangleq \mathsf{Key}(\mathsf{Union}(&\\
&msg_3(a'{:}\mathsf{Un}, ta{:}\mathsf{Un}, \mathsf{Ok}(\mathsf{end}\ Init(a, b))),\\
&msg_4(ta{:}\mathsf{Un}, \mathsf{Ok}(\mathsf{end}\ Resp(a, b, kab)))))
\end{aligned}
$$

As with Example 2, we omit the encoding of protocol behaviour and its specification as an actual process.

3 A State-Based Semantics of Correspondences

The previous section illustrates how to reduce an informal protocol specification to the question of whether a process with embedded begin- and end-assertions is robustly safe. This section formalizes this question by precisely defining a spi-calculus and its operational semantics. The next explains how to check robust safety by typechecking.

Messages. The messages and event labels of our calculus are as follows. The syntax is similar to previous versions of spi, except for the addition of ok.

Messages, Event Labels:

ℓ	message tag
a, b, c, x, y, z	names, variables

$L, M, N ::=$	message
x	name: a key or a channel
$\{M\}_N$	message M encrypted with key N
$\ell(M)$	message tagged with ℓ
(M, N)	message pair
ok	an ok to use some resource

As usual, pairs can represent arbitrary tuples; for example, $(L,(M,N))$ can represent the triple (L,M,N). We use analogous, standard abbreviations at the process and type level, but omit the details. Let $\mathsf{fn}(L)$ be the set of names occurring in L.

Processes. The syntax of processes is as follows. Types, ranged over by T, are defined in Section 4. We write $\mathsf{fn}(P)$ and $\mathsf{fn}(T)$ for the sets of names occurring free in the process P and the type T, respectively.

Processes:

$P, Q, R ::=$	process
$\mathsf{out}\ M\langle N\rangle$	asynchronous output
$\mathsf{inp}\ M(x{:}T); P$	input (scope of x is P)
$\mathsf{new}(x{:}T); P$	name generation (scope of x is P)
$!P$	replication
$P \mid Q$	composition
$\mathbf{0}$	inactivity
$\mathsf{begin}\ L; P$	one-to-one begin-assertion
$\mathsf{begin}\ !L; P$	one-to-many begin-assertion
$\mathsf{end}\ L; P$	end-assertion
$\mathsf{decrypt}\ L\ \mathsf{is}\ \{y{:}T\}_N;P$	decryption (scope of y is P)
$\mathsf{case}\ M\ (\ell_i(x_i{:}T_i)P_i\ ^{i\in 1..n})$	union case, $n \geq 0$ (scope of each x_i is P_i)
$\mathsf{split}\ M\ \mathsf{is}\ (x{:}T, y{:}U); P$	pair splitting (scope of x is U, P, of y just P)
$\mathsf{match}\ M\ \mathsf{is}\ (N, y{:}T); P$	pair matching (scope of y is P)
$\mathsf{exercise}\ M; P$	exercise an ok

The first group of constructs forms a typed π-calculus. The middle group consists of the begin- and end-assertions discussed in Section 2. Only one-to-many begin-assertions are new in this paper. The final group is of data manipulation constructs. Only the last, $\mathsf{exercise}\ M; P$, is new, and is described, along with decryption and pair splitting in Section 2. A union case process, $\mathsf{case}\ M\ (\ell_i(x_i{:}T_i)P_i\{x_i\}\ ^{i\in 1..n})$, behaves as $P_j\{N\}$ if M is the tagged message $\ell_j(N)$ for some $j \in 1..n$, and otherwise is stuck, that is, does nothing. (In general, we use the notation $P\{x\}$ to denote the process P and to note that it may have free occurrences of the variable x. In this context, we write $P\{M\}$ for the outcome of substituting the message M for each of those free occurrences of the variable x in P.) A pair match $\mathsf{match}\ M\ \mathsf{is}\ (N, y{:}T); P\{y\}$ behaves as $P\{L\}$ if M is the pair (N, L), and otherwise is stuck.

We write $\vec{x}{:}\vec{T}$ as a shorthand for the list $x_1{:}T_1, \ldots, x_n{:}T_n$ of typed variables, when $\vec{x} = x_1, \ldots, x_n$ and $\vec{T} = T_1, \ldots, T_n$. Moreover, we write $\mathsf{new}(\vec{x}{:}\vec{T}); P$ as shorthand for $\mathsf{new}(x_1{:}T_1); \cdots \mathsf{new}(x_n{:}T_n); P$, and also $\mathsf{end}\ L$ for $\mathsf{end}\ L; \mathbf{0}$.

As in many presentations of the π-calculus, we define a *structural equivalence* relation, that identifies processes up to some structural rearrangements. The following table includes equivalence rules, congruence for the basic structural operators (restriction, parallel composition, and replication), monoid laws for composition, Engelfriet's replication laws [Eng96], and the mobility laws for restriction. As usual, the exact choice of rules of structural equivalence is a little arbitrary.

Structural Equivalence of Processes: $P \equiv Q$

$P \equiv P$	(Struct Refl)
$Q \equiv P \Rightarrow P \equiv Q$	(Struct Symm)
$P \equiv Q, Q \equiv R \Rightarrow P \equiv R$	(Struct Trans)
$P \equiv P' \Rightarrow \mathsf{new}(x{:}T); P \equiv \mathsf{new}(x{:}T); P'$	(Struct Res)
$P \equiv P' \Rightarrow P \mid R \equiv P' \mid R$	(Struct Par)
$P \equiv P' \Rightarrow !P \equiv !P'$	(Struct Repl)
$P \mid \mathbf{0} \equiv P$	(Struct Par Zero)
$P \mid Q \equiv Q \mid P$	(Struct Par Comm)
$(P \mid Q) \mid R \equiv P \mid (Q \mid R)$	(Struct Par Assoc)
$!P \equiv P \mid !P$	(Struct Repl Unfold)
$!!P \equiv !P$	(Struct Repl Repl)
$!(P \mid Q) \equiv !P \mid !Q$	(Struct Repl Par)
$!\mathbf{0} \equiv \mathbf{0}$	(Struct Repl Zero)
$\mathsf{new}(x{:}T); (P \mid Q) \equiv P \mid \mathsf{new}(x{:}T); Q$	(Struct Res Par) (for $x \notin \mathsf{fn}(P)$)
$\mathsf{new}(x_1{:}T_1); \mathsf{new}(x_2{:}T_2); P \equiv$	(Struct Res Res)
$\quad \mathsf{new}(x_2{:}T_2); \mathsf{new}(x_1{:}T_1); P$	(for $x_1 \neq x_2, x_1 \notin \mathsf{fn}(T_2), x_2 \notin \mathsf{fn}(T_1)$)

States. A computation state takes the form $\mathsf{new}(\vec{x}{:}\vec{T}); (\mathcal{R} \parallel P)$, where the typed names $\vec{x}$ are freshly generated, the process P represents the running threads of control, and $\mathcal{R}$ is a resource—the imaginary tuplespace—recording which end-events the process may safely perform.

Structural Equivalence of States: $S \equiv S'$

$S \equiv S$	(Struct Refl)
$S' \equiv S \Rightarrow S' \equiv S$	(Struct Symm)
$S \equiv S', S' \equiv S'' \Rightarrow S \equiv S''$	(Struct Trans)
$S \equiv S' \Rightarrow \mathsf{new}(x{:}T); S \equiv \mathsf{new}(x{:}T); S'$	(Struct Res)
$\mathcal{R} \equiv \mathcal{R}', P \equiv P' \Rightarrow \mathcal{R} \parallel P \equiv \mathcal{R}' \parallel P'$	(Struct Par)
$x \notin \mathsf{fn}(\mathcal{R}) \Rightarrow \mathsf{new}(x{:}T); (\mathcal{R} \parallel P) \equiv \mathcal{R} \parallel \mathsf{new}(x{:}T); P$	(Struct Res State)

State Transitions and Safety. Our operational semantics is a state transition relation, defined by the following rules. The first group are transitions on states of the form $\mathcal{R} \parallel P$, and formalize our intuitive explanations of the various forms of process. The second group includes congruence rules, and the rule (Trans Struct) that allows a state to be re-arranged up to structural equivalence while deriving a transition.

State Transitions: $S \rightarrow S'$

$\mathcal{R} \parallel P \mid \mathsf{out}\ M\langle N\rangle \mid \mathsf{inp}\ M(y{:}T); Q\{y\} \rightarrow$ $\quad\mathcal{R} \parallel Q\{N\}$	(Trans I/O)
$\mathcal{R} \parallel P \mid \mathsf{begin}\ L; Q \rightarrow$ $\quad\mathcal{R} \mid \mathsf{end}\ L \parallel P \mid Q$	(Trans Begin)
$\mathcal{R} \parallel P \mid \mathsf{begin}\ !L; Q \rightarrow$ $\quad\mathcal{R} \mid !\mathsf{end}\ L \parallel P \mid Q$	(Trans Begin!)
$\mathcal{R} \mid \mathsf{end}\ L \parallel P \mid \mathsf{end}\ L; Q \rightarrow$ $\quad\mathcal{R} \parallel P \mid Q$	(Trans End)
$\mathcal{R} \parallel P \mid \mathsf{decrypt}\ \{M\}_N\ \mathsf{is}\ \{y{:}T\}_N; Q\{y\} \rightarrow$ $\quad\mathcal{R} \parallel P \mid Q\{M\}$	(Trans Decrypt)
$\mathcal{R} \parallel P \mid \mathsf{case}\ \ell_j(M)\ (\ell_i(x_i{:}T_i)Q_i\{x_i\}\ ^{i\in 1..n}) \rightarrow$ $\quad\mathcal{R} \parallel P \mid Q_j\{M\} \qquad j \in 1..n$	(Trans Case)
$\mathcal{R} \parallel P \mid \mathsf{split}\ (M, N)\ \mathsf{is}\ (x{:}T, y{:}U); Q\{x, y\} \rightarrow$ $\quad\mathcal{R} \parallel P \mid Q\{M, N\}$	(Trans Split)
$\mathcal{R} \parallel P \mid \mathsf{match}\ (M, N)\ \mathsf{is}\ (M, y{:}U); Q\{y\} \rightarrow$ $\quad\mathcal{R} \parallel P \mid Q\{N\}$	(Trans Match)
$\mathcal{R} \parallel P \mid \mathsf{exercise\ ok}; Q \rightarrow$ $\quad\mathcal{R} \parallel P \mid Q$	(Trans Exercise)
$S \rightarrow S' \Rightarrow \mathsf{new}(x{:}T); S \rightarrow \mathsf{new}(x{:}T); S'$	(Trans Res)
$S \equiv S', S' \rightarrow S'', S'' \equiv S''' \Rightarrow S \rightarrow S'''$	(Trans Struct)

For example, here is a transition sequence in which a one-to-many begin-assertion is matched by a couple of end-events:

$$\begin{aligned} \mathbf{0} \parallel \mathsf{begin}\ !L; \mathsf{end}\ L; \mathsf{end}\ L &\rightarrow\ !\mathsf{end}\ L \parallel \mathsf{end}\ L; \mathsf{end}\ L \\ &\equiv\ !\mathsf{end}\ L \mid \mathsf{end}\ L \mid \mathsf{end}\ L \parallel \mathsf{end}\ L; \mathsf{end}\ L \\ &\rightarrow\ !\mathsf{end}\ L \mid \mathsf{end}\ L \parallel \mathsf{end}\ L \rightarrow\ !\mathsf{end}\ L \parallel \mathbf{0} \end{aligned}$$

On the other hand, a one-to-one begin-assertion matches only one end-event. In the final state $\mathbf{0} \parallel \mathsf{end}\ L; \mathbf{0}$ of the following sequence, the end-assertion is deadlocked, signifying a violation of a correspondence assertion.

$$\mathbf{0} \parallel \mathsf{begin}\ L; \mathsf{end}\ L; \mathsf{end}\ L \rightarrow \mathsf{end}\ L \parallel \mathsf{end}\ L; \mathsf{end}\ L \rightarrow \mathbf{0} \parallel \mathsf{end}\ L$$

We can now formalize the notions of safety (every end-assertion succeeds) and robust safety (safety in the presence of an arbitrary opponent).

- Let $S \rightarrow^* S'$ mean there are $S_1, \ldots, S_n$ with $S \equiv S_1 \rightarrow \cdots \rightarrow S_n \equiv S'$.

- A state S is *fine* if and only if whenever $S \equiv \mathsf{new}(\vec{x}{:}\vec{T}); (\mathcal{R} \parallel \mathsf{end}\ L; P \mid P')$ there is $\mathcal{R}'$ such that $R \equiv \mathsf{end}\ L \mid \mathcal{R}'$.
- A process P is *safe* if and only if, for all S, if $\mathbf{0} \parallel P \rightarrow^* S$ then S is fine.
- An *untyped process* is one in which every type is Un.
- An *opponent* is an untyped process O containing no begin- or end-assertions, and no exercises.
- A process P is *robustly safe* if and only if $P \mid O$ is safe for all opponents O.

The question now is how to check robust safety by typing.

4 Typing One-to-Many Correspondences

Like our previous systems for typechecking correspondences [GJ01a,GJ02], we assign a resource (a kind of *effect* [GL86]) to each process P: an upper bound on the unmatched end-events performed by P. We prove theorems showing that safety and robust safety follow from assigning a process the $\mathbf{0}$ effect. Unlike our previous systems, we also assign a resource (as well as a type) to each message M: an upper bound on the end-events promised by all the ok terms contained within M.

Types:

$T, U ::=$	type
Un	public data
$\mathsf{Key}(T)$	secret key for T plaintext
$\mathsf{Union}(\ell_i(T_i)\ ^{i \in 1..n})$	tagged union, $n \geq 0$
$(x{:}T, U)$	dependent pair (scope of x is U)
$\mathsf{Ok}(\mathcal{R})$	ok to exercise $\mathcal{R}$

Abbreviation: $(x_1{:}T_1, \ldots, x_n{:}T_n, T_{n+1}) \triangleq (x_1{:}T_1, \ldots, (x_n{:}T_n, T_{n+1}))$

Messages of type Un are public data known to the opponent, including ciphertexts known to but indecipherable by the opponent. Messages of type $\mathsf{Key}(T)$ are secret keys for encrypting and decrypting plaintext of type T. Messages of type $\mathsf{Union}(\ell_i(T_i)\ ^{i \in 1..n})$ take the form $\ell_j(N)$ where ℓ_j is one of the tags $\ell_1, \ldots, \ell_n$, and N is a message of type T_j. Messages of type $(x{:}T, U\{x\})$ are pairs (M, N), where M has type T and N has type $U\{M\}$. (The scope of the bound variable x in $(x{:}T, U)$ is U.) Finally, ok is the unique message of type $\mathsf{Ok}(\mathcal{R})$, a pledge that the resource $\mathcal{R}$ is available.

Formation Judgments. Our judgments are defined with respect to an *environment*, E, a list $x_1{:}T_1, \ldots, x_n{:}T_n$ declaring types for those variables in scope. Let $\mathsf{dom}(x_1{:}T_1, \ldots, x_n{:}T_n) = \{x_1, \ldots, x_n\}$. The first three judgments define the correct formation of environments ($E \vdash \diamond$ meaning, roughly, all declarations are distinct) and resources and types ($E \vdash \mathcal{R}$ and $E \vdash T$ meaning, roughly, all free variables are in scope).

Formation Rules:

$$\text{(Env }\varnothing\text{)}\ \frac{}{\varnothing \vdash \diamond} \qquad \text{(Env }x\text{)}\ \frac{E \vdash T \quad x \notin \mathsf{dom}(E)}{E, x{:}T \vdash \diamond} \qquad \text{(Resource Event)}\ \frac{E \vdash \diamond \quad \mathsf{fn}(L) \subseteq \mathsf{dom}(E)}{E \vdash \mathsf{end}\ L}$$

$$\text{(Resource Repl)}\ \frac{E \vdash \mathcal{R}}{E \vdash !\mathcal{R}} \qquad \text{(Resource Par)}\ \frac{E \vdash \mathcal{R} \quad E \vdash \mathcal{S}}{E \vdash \mathcal{R} \mid \mathcal{S}} \qquad \text{(Resource Zero)}\ \frac{E \vdash \diamond}{E \vdash \mathbf{0}} \qquad \text{(Type Un)}\ \frac{E \vdash \diamond}{E \vdash \mathsf{Un}}$$

$$\text{(Type Key)}\ \frac{E \vdash T}{E \vdash \mathsf{Key}(T)} \qquad \text{(Type Union)}\ \frac{E \vdash A_i \quad \forall i \in 1..n}{E \vdash \mathsf{Union}(\ell_i(A_i)\ ^{i \in 1..n})} \qquad \text{(Type Pair)}\ \frac{E, x{:}T \vdash U}{E \vdash (x{:}T, U)} \qquad \text{(Type Ok)}\ \frac{E \vdash \mathcal{R}}{E \vdash \mathsf{Ok}(\mathcal{R})}$$

Typing Messages. The judgment $E \vdash M : T, \mathcal{R}$ means that, given E, the message M has type T and effect $\mathcal{R}$: it needs the resources $\mathcal{R}$ to justify promises made by the ok messages that may be extracted from M. Recall that an extracted message of type $\mathsf{Ok}(\mathcal{R})$ may subsequently be exercised to justify end-assertions allowed by $\mathcal{R}$.

For example, consider again the type

$$\mathit{SharedKey} \triangleq \mathsf{Key}(x{:}\mathit{Msg}, t{:}\mathsf{Un}, \mathsf{Ok}(\mathsf{end}\ \mathit{Sending}(A, B, x)))$$

of longterm keys from Example 1 in Section 2. Let $E = K_{AB}{:}\mathit{SharedKey}$, $x{:}\mathit{Msg}$, $T_A{:}\mathsf{Un}$ and let message M be the ciphertext $\{x, T_A, \mathsf{ok}\}_{K_{AB}}$. Then we have:

$$E \vdash M : \mathsf{Un}, \mathsf{end}\ \mathit{Sending}(A, B, x)$$

The cipher is well-typed because the triple (x, T_A, ok) has type $(x{:}\mathit{Msg}$, $t{:}\mathsf{Un}$, $\mathsf{Ok}(\mathsf{end}\ \mathit{Sending}(A, B, x)))$, and hence can be encrypted with a key of type $\mathit{SharedKey}$. The cipher M has type Un because it may safely be made public. The effect $\mathsf{end}\ \mathit{Sending}(A, B, x)$ arises from the presence of the message $\mathsf{ok} : \mathsf{Ok}(\mathsf{end}\ \mathit{Sending}(A, B, x))$, which may eventually be exercised to justify an end-assertion labelled $\mathit{Sending}(A, B, x)$.

If we have two copies of M, the effect is doubled:

$$E \vdash (M, M) : \mathsf{Un}, \mathsf{end}\ \mathit{Sending}(A, B, x) \mid \mathsf{end}\ \mathit{Sending}(A, B, x)$$

The following table defines message typing. As in earlier work [GJ01a], many constructs have two rules: one for typing data known only to the trusted principals running the protocol, and one for typing Un data known to the opponent. The rules accumulate the effects of components that may be extracted, but ignore the effects of those that cannot. For example, the rules (Msg Encrypt) and (Msg Encrypt Un) assign the effect $\mathcal{R}_1$ of the plaintext M to the ciphertext $\{M\}_N$, since the plaintext may be extracted by decryption, but ignore the effect $\mathcal{R}_2$ of the key N, since there is no primitive to extract it from the ciphertext. Let $\mathcal{R} \leq \mathcal{R}' \triangleq \exists \mathcal{R}''.\mathcal{R} \mid \mathcal{R}'' \equiv \mathcal{R}'$.

Good Messages:

(Msg Subsum)
$$\frac{E \vdash M : T, \mathcal{R} \quad \mathcal{R} \leq \mathcal{R}' \quad E \vdash \mathcal{R}'}{E \vdash M : T, \mathcal{R}'}$$

(Msg x)
$$\frac{E \vdash \diamond \quad x \in \mathsf{dom}(E)}{E \vdash x : E(x), \mathbf{0}}$$

(Msg Encrypt)
$$\frac{E \vdash M : T, \mathcal{R}_1 \quad E \vdash N : \mathsf{Key}(T), \mathcal{R}_2}{E \vdash \{M\}_N : \mathsf{Un}, \mathcal{R}_1}$$

(Msg Encrypt Un)
$$\frac{E \vdash M : \mathsf{Un}, \mathcal{R}_1 \quad E \vdash N : \mathsf{Un}, \mathcal{R}_2}{E \vdash \{M\}_N : \mathsf{Un}, \mathcal{R}_1}$$

(Msg Union) (where $T = \mathsf{Union}(\ell_i(T_i)\ ^{i \in 1..n})$)
$$\frac{E \vdash M : T_j, \mathcal{R} \quad j \in 1..n \quad E \vdash T}{E \vdash \ell_j(M) : T, \mathcal{R}}$$

(Msg Union Un)
$$\frac{E \vdash M : \mathsf{Un}, \mathcal{R}}{E \vdash \ell(M) : \mathsf{Un}, \mathcal{R}}$$

(Msg Tuple)
$$\frac{E \vdash M : T, \mathcal{R}_1 \quad E \vdash N : U\{M\}, \mathcal{R}_2 \quad E \vdash (x{:}T, U\{x\})}{E \vdash (M, N) : (x{:}T, U\{x\}), \mathcal{R}_1 \mid \mathcal{R}_2}$$

(Msg Tuple Un)
$$\frac{E \vdash M : \mathsf{Un}, \mathcal{R}_1 \quad E \vdash N : \mathsf{Un}, \mathcal{R}_2}{E \vdash (M, N) : \mathsf{Un}, \mathcal{R}_1 \mid \mathcal{R}_2}$$

(Msg Ok)
$$\frac{E \vdash \mathcal{R}}{E \vdash \mathsf{ok} : \mathsf{Ok}(\mathcal{R}), \mathcal{R}}$$

(Msg Ok Un)
$$\frac{E \vdash \diamond}{E \vdash \mathsf{ok} : \mathsf{Un}, \mathbf{0}}$$

The following lemma asserts that any message originating from the opponent may be assigned the Un type and $\mathbf{0}$ effect.

Lemma 1. *For any M, if $\mathsf{fn}(M) = \{\vec{x}\}$ then $\vec{x}{:}\mathsf{Un} \vdash M : \mathsf{Un}, \mathbf{0}$.*

Typing Processes. The judgment $E \vdash P : \mathcal{R}$ means that, given E, the effect $\mathcal{R}$ is an upper bound on the resources required for safe execution of P.

The first group of rules concerns the π-calculus fragment of our calculus. The main intuition is that the effect of a process stems from the composition of the effects of any subprocesses, together with the replicated effects of any extractable messages. The effect of a message must be replicated to account for the possibility that the message may be copied many times. In particular, if an ok is copied, it can be exercised many times. For example, suppose that E is the environment and $M = \{M, T_A, \mathsf{ok}\}_{K_{AB}}$ is the ciphertext from our discussion of the message typing rules above. We have $E \vdash M : \mathsf{Un}, \mathsf{end}\ \mathit{Sending}(A, B, x)$ and so we have:

$$E, \mathit{net}{:}\mathsf{Un} \vdash \mathsf{out}\ \mathit{net}\langle M \rangle : !\mathsf{end}\ \mathit{Sending}(A, B, x)$$

Good Processes: Basics

(Proc Subsum)
$$\frac{E \vdash P : \mathcal{R} \quad \mathcal{R} \leq \mathcal{R}' \quad E \vdash \mathcal{R}'}{E \vdash P : \mathcal{R}'}$$

(Proc Output Un)
$$\frac{E \vdash M : \mathsf{Un}, \mathcal{R}_1 \quad E \vdash N : \mathsf{Un}, \mathcal{R}_2}{E \vdash \mathsf{out}\ M\langle N \rangle : !\mathcal{R}_2}$$

(Proc Input Un)

$$\frac{E \vdash M : \mathsf{Un}, \mathcal{R}_1 \quad E, y{:}\mathsf{Un} \vdash P : \mathcal{R}_2 \quad y \notin \mathsf{fn}(\mathcal{R}_2)}{E \vdash \mathsf{inp}\ M(y{:}\mathsf{Un}); P : \mathcal{R}_2}$$

(Proc Res)(where T is *generative*, that is, either Un or $\mathsf{Key}(T')$ for some T')

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

(Proc Par)

$$\frac{E \vdash P_1 : \mathcal{R}_1 \quad E \vdash P_2 : \mathcal{R}_2}{E \vdash P_1 \mid P_2 : \mathcal{R}_1 \mid \mathcal{R}_2}$$

(Proc Repeat)

$$\frac{E \vdash P : \mathcal{R}}{E \vdash !P : !\mathcal{R}}$$

(Proc Zero)

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

The next group of rules concerns begin- and end-assertions. As in our previous work, the effect of a begin-assertion is the effect of its continuation, minus the event, while the effect of an end-assertion is the effect of its continuation, plus the event. What is new here is the use of replication to distinguish one-to-many from one-to-one begin-assertions.

Good Processes: Correspondence Assertions

(Proc Begin)

$$\frac{E \vdash P : \mathcal{R} \mid \mathsf{end}\ L}{E \vdash \mathsf{begin}\ L; P : \mathcal{R}}$$

(Proc Begin!)

$$\frac{E \vdash P : \mathcal{R} \mid !\mathsf{end}\ L}{E \vdash \mathsf{begin}\ !L; P : \mathcal{R}}$$

(Proc End)

$$\frac{\mathsf{fn}(L) \subseteq \mathsf{dom}(E) \quad E \vdash P : \mathcal{R}}{E \vdash \mathsf{end}\ L; P : \mathcal{R} \mid \mathsf{end}\ L}$$

Next, (Proc Exercise) defines that the effect of $\mathsf{exercise}\ M; P$, where M has type $\mathsf{Ok}(\mathcal{R})$, is the effect of P minus $\mathcal{R}$, plus any effect M has itself.

Good Processes: Exercising an Ok

(Proc Exercise)

$$\frac{E \vdash M : \mathsf{Ok}(\mathcal{R}), \mathcal{R}_1 \quad E \vdash P : \mathcal{R} \mid \mathcal{R}_2}{E \vdash \mathsf{exercise}\ M; P : \mathcal{R}_1 \mid \mathcal{R}_2}$$

For instance, looking again at Example 1, we can derive:

$$A, B{:}\mathsf{Un}, x{:}\mathit{Msg}, o{:}\mathsf{Ok}(\mathsf{end}\, \mathit{Sending}(A, B, x)) \vdash \mathsf{exercise}\ o; \mathsf{end}\, \mathit{Sending}(A, B, x) : \mathbf{0}$$

The final group of rules is for typing message manipulation. These rules are much the same as in previous systems [GJ01a,GJ02], except for including the replicated effects of extractable messages.

Good Processes: Message Manipulation

(Proc Decrypt)

$$\frac{E \vdash L : \mathsf{Un}, \mathcal{R}_1 \quad E \vdash N : \mathsf{Key}(T), \mathcal{R}_2 \quad E, y{:}T \vdash P : \mathcal{R}_3 \quad y \notin \mathsf{fn}(\mathcal{R}_3)}{E \vdash \mathsf{decrypt}\ L\ \mathsf{is}\ \{y{:}T\}_N; P : !\mathcal{R}_1 \mid \mathcal{R}_3}$$

(Proc Decrypt Un)

$$\frac{E \vdash L : \mathsf{Un}, \mathcal{R}_1 \quad E \vdash N : \mathsf{Un}, \mathcal{R}_2 \quad E, y{:}\mathsf{Un} \vdash P : \mathcal{R}_3 \quad y \notin \mathsf{fn}(\mathcal{R}_3)}{E \vdash \mathsf{decrypt}\ L\ \mathsf{is}\ \{y{:}\mathsf{Un}\}_N; P : !\mathcal{R}_1 \mid \mathcal{R}_3}$$

(Proc Case)

$$\frac{E \vdash M : \mathsf{Union}(\ell_i(T_i)\ ^{i \in 1..n}), \mathcal{R}_1 \quad E, x_i{:}T_i \vdash P_i : \mathcal{R}_2 \quad x_i \notin \mathsf{fn}(\mathcal{R}_2) \quad \forall i \in 1..n}{E \vdash \mathsf{case}\ M\ (\ell_i(x_i{:}T_i)P_i\ ^{i \in 1..n}) : !\mathcal{R}_1 \mid \mathcal{R}_2}$$

(Proc Case Un)

$$\frac{E \vdash M : \mathsf{Un}, \mathcal{R}_1 \quad E, x_i{:}\mathsf{Un} \vdash P_i : \mathcal{R}_2 \quad x_i \notin \mathsf{fn}(\mathcal{R}_2) \quad \forall i \in 1..n}{E \vdash \mathsf{case}\ M\ (\ell_i(x_i{:}\mathsf{Un})P_i\ ^{i \in 1..n}) : !\mathcal{R}_1 \mid \mathcal{R}_2}$$

(Proc Split)

$$\frac{E \vdash M : (x{:}T, U), \mathcal{R}_1 \quad E, x{:}T, y{:}U \vdash P : \mathcal{R} \quad x, y \notin \mathsf{fn}(\mathcal{R})}{E \vdash \mathsf{split}\ M\ \mathsf{is}\ (x{:}T, y{:}U); P : !\mathcal{R}_1 \mid \mathcal{R}}$$

(Proc Split Un)

$$\frac{E \vdash M : \mathsf{Un}, \mathcal{R}_1 \quad E, x{:}\mathsf{Un}, y{:}\mathsf{Un} \vdash P : \mathcal{R} \quad x, y \notin \mathsf{fn}(\mathcal{R})}{E \vdash \mathsf{split}\ M\ \mathsf{is}\ (x{:}\mathsf{Un}, y{:}\mathsf{Un}); P : !\mathcal{R}_1 \mid \mathcal{R}}$$

(Proc Match)

$$\frac{E \vdash M : (x{:}T, U\{x\}), \mathcal{R}_1 \quad E \vdash N : T, \mathcal{R}_2 \quad E, y{:}U\{N\} \vdash P : \mathcal{R} \quad y \notin \mathsf{fn}(\mathcal{R})}{E \vdash \mathsf{match}\ M\ \mathsf{is}\ (N, y{:}U\{N\}); P : !\mathcal{R}_1 \mid \mathcal{R}}$$

(Proc Match Un)

$$\frac{E \vdash M : \mathsf{Un}, \mathcal{R}_1 \quad E \vdash N : \mathsf{Un}, \mathcal{R}_2 \quad E, y{:}\mathsf{Un} \vdash P : \mathcal{R} \quad y \notin \mathsf{fn}(\mathcal{R})}{E \vdash \mathsf{match}\ M\ \mathsf{is}\ (N, y{:}\mathsf{Un}); P : !\mathcal{R}_1 \mid \mathcal{R}}$$

The following asserts that any opponent process may be assigned the **0** effect.

Lemma 2 (Opponent Typability). *For any opponent O, if $\mathsf{fn}(O) = \{x_1, \ldots, x_n\}$, then $x_1{:}\mathsf{Un}, \ldots, x_n{:}\mathsf{Un} \vdash O : \mathbf{0}$.*

Proof. Recall that an opponent is an untyped process containing no begin- or end-assertions, and no exercises. The result follows by induction on the size of O, with appeal to Lemma 1, (Proc Output Un), (Proc Input Un), (Proc Res), (Proc Par), (Proc Repeat), (Proc Zero), (Proc Decrypt Un), (Proc Case Un), (Proc Split Un), and (Proc Match Un). □

Runtime Invariant. We say that a state $\mathsf{new}(\vec{x}{:}\vec{T}); (\mathcal{R} \parallel P)$ is *good* if and only if the imaginary resource $\mathcal{R}$ can be assigned to P as an effect. Intuitively, this means the available resource is an upper bound of what is actually needed. Recall that (Proc Subsum) allows us to increase the effect assigned to a process.

Good States:

(State Res)
$$\frac{E, x{:}T \vdash S \quad T \text{ generative}}{E \vdash \mathsf{new}(x{:}T); S}$$

(State Base)
$$\frac{E \vdash P : \mathcal{R}}{E \vdash \mathcal{R} \parallel P}$$

The next two propositions assert that being a good state is preserved by the structural equivalence and transition relations.

Proposition 1. *If $E \vdash S$ and $S \equiv S'$ then $E \vdash S'$.*

Proposition 2. *If $E \vdash S$ and $S \rightarrow S'$ then $E \vdash S'$.*

We show that good states are fine, that is, contain no deadlocked end-assertions; theorems establishing safety and robust safety then follow easily.

Proposition 3. *If $E \vdash S$ then S is fine.*

Proof. Suppose that $S \equiv \mathsf{new}(\vec{x}{:}\vec{T}); (\mathcal{R} \parallel \mathsf{end}\ L; P \mid P')$. By Proposition 1, $E \vdash \mathsf{new}(\vec{x}{:}\vec{T}); (\mathcal{R} \parallel \mathsf{end}\ L; P \mid P')$. Therefore, $E, \vec{x}{:}\vec{T} \vdash \mathsf{end}\ L; P \mid P' : \mathcal{R}$. It follows there is $\mathcal{R}'$ such that $R \equiv \mathsf{end}\ L \mid \mathcal{R}'$. Hence S is fine. □

Theorem 1 (Safety). *If $E \vdash P : \mathbf{0}$ then P is safe.*

Proof. By (State Base), $E \vdash \mathbf{0} \parallel P$. Consider any S such that $\mathbf{0} \parallel P \rightarrow^* S$. By Propositions 1 and 2, $E \vdash S$. By Proposition 3, S is fine. So P is safe. □

Theorem 2 (Robust Safety). *If $x_1{:}\mathsf{Un}, \ldots, x_n{:}\mathsf{Un} \vdash P : \mathbf{0}$, P is robustly safe.*

Proof. Combine Theorem 1 and Lemma 2, and a weakening property of the type system. □

We can apply these theorems to verify the one-to-many correspondences of Examples 1–3 of Section 4, but we omit the details. Given processes annotated with suitable types, we claim that typechecking is decidable, though we currently have no implementation for this type system. On the other hand, we rely on human intervention to discover suitable types, such as the types for principal and session keys.

5 Summary and Conclusion

Our previous work [GJ01b,GJ01a,GJ02] shows how to verify one-to-one correspondence assertions by typechecking process calculus descriptions of protocols. This paper shows how to typecheck one-to-many correspondences. Applications include checking security protocols intended only to offer one-to-many guarantees, but also checking protocols that in fact offer stronger one-to-one guarantees, but via mechanisms, such as timestamps, beyond the scope of our type system.

There is little other work we are aware of on typechecking authenticity properties, but there are several other works on types for cryptographic protocols [Aba99,PS00,AB01,Cer01,Dug02], mostly aimed at establishing secrecy properties. Sometimes, of course, authenticity follows from secrecy [Bla02].

The ok-types introduced here are related to the types in our previous work for typing nonce challenges and responses. Like nonce types, they transfer effects by communication, allowing the type system to verify that an end-assertion in one process is justified by a corresponding begin-assertion in another. Unlike values of nonce types, values of ok-type are copyable, and may transfer an effect many times—hence, they are useful for one-to-many but not one-to-one correspondences.

The states—resources paired with processes—and the state transition relation introduced here allow for a smoother technical development than the labelled transitions and other relations used in our earlier papers. Blanchet [Bla02] formalizes correspondence assertions similarly.

The type system of Section 4 can verify only trivial one-to-one correspondences based on straightline code. In a longer version of this paper, we show how to accommodate the nonce types of an earlier paper [GJ01a] and hence to check protocols with interesting combinations of one-to-one and one-to-many correspondences.

In conclusion, the present paper usefully broadens the class of authenticity properties provable by typechecking. Still, our system continues to lack a general treatment of various issues, such as insider attacks and key compromises. We leave these questions, and the experimental evaluation of this type system, as future work.

Acknowledgements. Stimulating conversations with Ernie Cohen and with Dave Walker helped shape the ideas in this paper. Comments from the anonymous reviewers resulted in improvements to the paper.

References

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

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

AFG98. M. Abadi, C. Fournet, and G. Gonthier. Secure communications implementation of channel abstractions. In *13th IEEE Symposium on Logic in Computer Science (LICS'98)*, pages 105–116, 1998.

AFG00. M. Abadi, C. Fournet, and G. Gonthier. Authentication primitives and their compilation. In *27th ACM Symposium on Principles of Programming Languages (POPL'00)*, pages 302–315, 2000.

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

AN95. R. Anderson and R. Needham. Programming Satan's computer. In J. van Leeuwen, editor, *Computer Science Today: Recent Trends and Developments*, volume 1000 of *Lecture Notes in Computer Science*, pages 426–440. Springer, 1995.

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

BCJS02. F. Butler, I. Cervesato, A.D. Jaggard, and A. Scedrov. A formal analysis of some properties of Kerberos 5 using MSR. In *15th IEEE Computer Security Foundations Workshop*, pages 175–190. IEEE Computer Society Press, 2002.

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

Cer01. I. Cervesato. Typed MSR: Syntax and examples. In *First International Workshop on Mathematical Methods, Models and Architectures for Computer Network Security*, volume 2052 of *Lecture Notes in Computer Science*, pages 159–177. Springer, 2001.

Dug02. D. Duggan. Cryptographic types. In *15th IEEE Computer Security Foundations Workshop*, pages 238–252. IEEE Computer Society Press, 2002.

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

Eng96. J. Engelfriet. A multiset semantics for the pi-calculus with replication. *Theoretical Computer Science*, 153:65–94, 1996.

FG94. R. Focardi and R. Gorrieri. A classification of security properties for process algebra. *Journal of Computer Security*, 3(1):5–33, 1994.

FGM00. 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 *Lecture Notes in Computer Science*, pages 258–272. Springer, 2000.

GJ01a. A.D. Gordon and A. Jeffrey. Authenticity by typing for security protocols. In *14th IEEE Computer Security Foundations Workshop*, pages 145–159. IEEE Computer Society Press, 2001.

GJ01b. A.D. Gordon and A. Jeffrey. Typing correspondence assertions for communication protocols. In *Mathematical Foundations of Programming Semantics 17*, volume 45 of *Electronic Notes in Theoretical Computer Science*. Elsevier, 2001.

GJ02. A.D. Gordon and A. Jeffrey. Types and effects for asymmetric cryptographic protocols. In *15th IEEE Computer Security Foundations Workshop*, pages 77–91. IEEE Computer Society Press, 2002.

GL86. D.K. Gifford and J.M. Lucassen. Integrating functional and imperative programming. In *ACM Conference on Lisp and Functional Programming*, pages 28–38, 1986.

Gol02. D. Gollmann. Authentication by correspondence. *IEEE Journal on Selected Areas in Communication*, 2002. To appear.

GP02. A.D. Gordon and R. Pucella. Validating a web service security abstraction by typing. In *ACM Workshop on XML Security*, 2002. To appear.

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

LLL$^+$02. B.A. LaMacchia, S. Lange, M. Lyons, R. Martin, and K.T. Price. *.NET Framework Security*. Addison Wesley Professional, 2002.

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

LY97. T. Lindholm and F. Yellin. *The Java$^{\mathrm{TM}}$ Virtual Machine Specification.* Addison-Wesley, 1997.

Mea96. C. Meadows. The NRL Protocol Analyzer: An overview. *Journal of Logic Programming*, 26(2):113–131, 1996.

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

PS00. B. Pierce and E. Sumii. Relating cryptography and polymorphism. Available from the authors, 2000.

Ros96. A.W. Roscoe. Intensional specifications of security protocols. In *8th IEEE Computer Security Foundations Workshop*, pages 28–38. IEEE Computer Society Press, 1996.

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

SM02. A. Sabelfeld and A.C. Myers. Language-based information-flow security. *IEEE Journal on Selected Areas in Communication*, 2002. To appear.

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