Mobile Values, New Names, and Secure Communication

Martín Abadi
Bell Labs Research
Lucent Technologies

Cédric Fournet
Microsoft Research

Abstract

We study the interaction of the "new" construct with a rich but common form of (first-order) communication. This interaction is crucial in security protocols, which are the main motivating examples for our work; it also appears in other programming-language contexts. Specifically, we introduce a simple, general extension of the pi calculus with value passing, primitive functions, and equations among terms. We develop semantics and proof techniques for this extended language and apply them in reasoning about some security protocols.

1 A case for impurity

Purity often comes before convenience and even before faithfulness in the lambda calculus, the pi calculus, and other foundational programming languages. For example, in the standard pi calculus, the only messages are atomic names [32]. This simplicity is extremely appealing from a foundational viewpoint, and helps in developing the theory of the pi calculus. Furthermore, ingenious encodings demonstrate that it may not entail a loss of generality: in particular, integers, objects, and even higher-order processes can be represented in the pure pi calculus.

On the other hand, this purity has a price. In applications, the encodings can be futile, cumbersome, and even misleading. For example, in the study of programming languages based on the pi calculus (such as Pict [37] or Jocaml [14]), there is little point in pretending that integers are not primitive. The encodings may also complicate static analysis and preclude careful thinking about the implementations of communication. Moreover, it is not clear that satisfactory encodings can always be found. We may ask, for instance, whether there is a good representation of the spi calculus [5] (a calculus with cryptographic operations) in the standard pi calculus; we are not aware of any such representation that preserves security properties without a trusted central process.

These difficulties are often circumvented through on-the-fly extensions. The extensions range from quick punts ("for the next example, let's pretend that we have a datatype of integers") to the laborious development of new calculi, such as the spi calculus and its variants. Generally, the extensions bring us closer to a realistic programming language or modeling language—that is not always a bad thing.

Although many of the resulting calculi are ad hoc and poorly understood, others are robust and uniform enough to have a rich theory and a variety of applications. In particular, impure extensions of the lambda calculus with function symbols and with equations among terms ("delta rules") have been developed systematically, with considerable success. Similarly, impure versions of CCS and CSP with value-passing are not always deep but often neat and convenient [31].

In this paper, we introduce, study, and use an analogous uniform extension of the pi calculus, which we call the applied pi calculus (by analogy with "applied lambda calculus"). From the pure pi calculus, we inherit constructs for communication and concurrency, and for generating statically scoped new names ("new"). We add functions and equations, much as is done in the lambda calculus. Messages may then consist not only of atomic names but also of values constructed from names and functions. This embedding of names into the space of values gives rise to an important interaction between the "new" construct and value-passing communication, which appears in neither the pure pi calculus nor value-passing CCS and CSP. Further, we add an auxiliary substitution construct, roughly similar to a floating "let"; this construct is helpful in programming examples and especially in semantics and proofs, and serves to capture the partial knowledge that an environment may have of some values.

The applied pi calculus builds on the pure pi calculus and its substantial theory, but it shifts the focus away from encodings. In comparison with ad hoc approaches, it permits a general, systematic development of syntax, operational semantics, equivalences, and proof techniques.

Using the calculus, we can write and reason about programming examples where "new" and value-passing appear. First, we can easily treat standard datatypes (integers, pairs, arrays, etc.). We can also model unforgeable capabilities as new names, then model the application of certain functions to those capabilities. For instance, we may construct a pair of capabilities. More delicately, the capabilities may be pointers to composite structures, and then adding an offset to a pointer to a pair may yield a pointer to its second component (e.g., as in [27]). Furthermore, we can study a variety of security protocols. For this purpose, we represent fresh channels, nonces, and keys as new names, and primitive cryptographic operations as functions, obtaining a simple but useful programming-language perspective on security protocols (much as in the spi calculus). A distinguishing characteristic of the present approach is that we need not craft a special calculus and develop its proof techniques for each choice of cryptographic operations. Thus, we can express and analyze fairly sophisticated protocols that combine several cryptographic primitives (encryptions, hashes, signatures, XORs, ...). We can also describe attacks against the protocols that rely on (equational) properties of some of those primitives. In our work to date, security protocols are our main source of examples.

The next section defines the applied pi calculus. Section 3 introduces some small, informal examples. Section 4 defines semantic concepts, such as process equivalence, and develops proof techniques. Sections 5 and 6 treat two larger examples; they concern a Diffie-Hellman key exchange and message authentication codes, respectively. (The two examples are independent.) Section 7 discusses some related work and concludes.

2 The applied pi calculus

In this section we define the applied pi calculus: its syntax and informal semantics, then its operational semantics (in the now customary chemical style).

2.1 Syntax and informal semantics

A *signature* Σ consists of a finite set of function symbols, such as f, $\mathsf{encrypt}$, and pair, each with an arity. A function symbol with arity 0 is a constant symbol.

Given a signature Σ, an infinite set of names, and an infinite set of variables, the set of *terms* is defined by the grammar:

$L, M, N, T, U, V ::=$	terms
$\quad a, b, c, \ldots, k, \ldots, m, n, \ldots, s$	name
$\quad x, y, z$	variable
$\quad f(M_1, \ldots, M_l)$	function application

where f ranges over the functions of Σ and l matches the arity of f. Although names, variables, and constant symbols have similarities, we find it clearer to keep them separate. A term is ground when it does not have free variables (but it may contain names and constant symbols). We use metavariables u, v, w to range over both names and variables. We also use standard conventional notations for function applications. We abbreviate tuples $u_1, \ldots, u_l$ and $M_1, \ldots, M_l$ to $\widetilde{u}$ and $\widetilde{M}$, respectively.

We rely on a sort system for terms. It includes a set of base types, such as $\mathsf{Integer}$, Key, or simply a universal base type Data. In addition, if τ is a sort, then $\mathsf{Channel}\langle\tau\rangle$ is a sort too (intuitively, the sort of those channels that convey messages of sort τ). A variable can have any sort. A name can have any sort or, in a more refined version of the sort system, any sort in a distinguished class of sorts. We typically use a, b, and c as channel names, s and k as names of some base type (e.g., Data), and m and n as names of any sort. For simplicity, function symbols take arguments and produce results of the base types only. (This separation of channels from other values is convenient but not essential to our approach.) We omit the unimportant details of this sort system, and leave it mostly implicit in the rest of the paper. We always assume that terms are well-sorted and that substitutions preserve sorts.

The grammar for *processes* is similar to the one in the pi calculus, except that here messages can contain terms (rather than only names) and that names need not be just channel names:

$P, Q, R ::=$	processes (or plain processes)
$\quad \mathbf{0}$	null process
$\quad P \mid Q$	parallel composition
$\quad !P$	replication
$\quad \nu n.P$	name restriction ("new")
$\quad \textit{if } M = N \textit{ then } P \textit{ else } Q$	conditional
$\quad u(x).P$	message input
$\quad \overline{u}\langle N\rangle.P$	message output

The null process $\mathbf{0}$ does nothing; $P \mid Q$ is the parallel composition of P and Q; the replication $!P$ behaves as an infinite number of copies of P running in parallel. The process $\nu n.P$ makes a new, private name n then behaves as P. The conditional construct *if* $M = N$ *then* P *else* Q is standard, but we should stress that $M = N$ represents equality, rather than strict syntactic identity. We abbreviate it *if* $M = N$ *then* P when Q is $\mathbf{0}$. Finally, $u(x).P$ is ready to input from channel u, then to run P with the actual message replaced for the formal parameter x, while $\overline{u}\langle N\rangle.P$ is ready to output N on channel u, then to run P. In both of these, we may omit P when it is $\mathbf{0}$.

Further, we extend processes with *active substitutions*:

$A, B, C ::=$	extended processes
$\quad P$	plain process
$\quad A \mid B$	parallel composition
$\quad \nu n.A$	name restriction
$\quad \nu x.A$	variable restriction
$\quad \{^{M}/_{x}\}$	active substitution

We write $\{^{M}/_{x}\}$ for the substitution that replaces the variable x with the term M. Considered as a process, $\{^{M}/_{x}\}$ is like *let* $x = M$ *in* ..., and is similarly useful. However, unlike a "let" definition, $\{^{M}/_{x}\}$ floats and applies to any process that comes into contact with it. To control this contact, we may add a restriction: $\nu x.(\{^{M}/_{x}\} \mid P)$ corresponds exactly to *let* $x = M$ *in* P. The substitution $\{^{M}/_{x}\}$ typically appears when the term M has been sent to the environment, but the environment may not have the atomic names that appear in M; the variable x is just a way to refer to M in this situation. Although the substitution $\{^{M}/_{x}\}$ concerns only one variable, we can build bigger substitutions by parallel composition, and may write

$$\{^{M_1}/_{x_1}, \ldots, ^{M_l}/_{x_l}\} \quad \text{for} \quad \{^{M_1}/_{x_1}\} \mid \ldots \mid \{^{M_l}/_{x_l}\}$$

We write σ, $\{^{M}/_{x}\}$, $\{^{\widetilde{M}}/_{\widetilde{x}}\}$ for substitutions, $x\sigma$ for the image of x by σ, and $T\sigma$ for the result of applying σ to the free variables of T. We identify the empty substitution and the null process $\mathbf{0}$. We always assume that our substitutions are cycle-free. We also assume that, in an extended process, there is at most one substitution for each variable, and there is exactly one when the variable is restricted.

Extending the sort system for terms, we rely on a sort system for extended processes. It enforces that M and N are of the same sort in the conditional expression, that u has sort

$\mathsf{Channel}\langle\tau\rangle$ for some τ in the input and output expressions, and that x and N have the corresponding sort τ in those expressions. Again, we omit the unimportant details of this sort system, but assume that extended processes are well-sorted.

As usual, names and variables have scopes, which are delimited by restrictions and by inputs. We write $fv(A)$, $bv(A)$, $fn(A)$, and $bn(A)$ for the sets of free and bound variables and free and bound names of A, respectively. These sets are inductively defined, using the same clauses for processes as in the pure pi calculus, and using:

$$\begin{array}{rcl} fv(\{^M/_x\}) & \stackrel{\text{def}}{=} & fv(M) \cup \{x\} \\ fn(\{^M/_x\}) & \stackrel{\text{def}}{=} & fn(M) \end{array}$$

for active substitutions. An extended process is *closed* when every variable is either bound or defined by an active substitution. We use the abbreviation $\nu\widetilde{u}$ for the (possibly empty) series of pairwise-distinct binders $\nu u_1.\nu u_2.\ldots.\nu u_l$.

A *frame* is an extended process built up from $\mathbf{0}$ and active substitutions of the form $\{^M/_x\}$ by parallel composition and restriction. We let φ and ψ range over frames. The domain $dom(\varphi)$ of a frame φ is the set of the variables that φ exports (those variables x for which φ contains a substitution $\{^M/_x\}$ not under a restriction on x). Every extended process A can be mapped to a frame $\varphi(A)$ by replacing every plain process embedded in A with $\mathbf{0}$. The frame $\varphi(A)$ can be viewed as an approximation of A that accounts for the static knowledge exposed by A to its environment, but not for A's dynamic behavior. The domain $dom(A)$ of A is the domain of $\varphi(A)$.

2.2 Operational semantics

Given a signature Σ, we equip it with an equational theory, that is, with an equivalence relation on terms that is closed under substitutions of terms for variables. (See for example [33, chapter 3] and its references for background on universal algebra and algebraic data types from a programming-language perspective.) We further require that this equational theory be closed under one-to-one renamings, but not necessarily closed under substitutions of arbitrary terms for names.

We write $\Sigma \vdash M = N$ when the equation $M = N$ is in the theory associated with Σ. Here we keep the theory implicit, and we may even abbreviate $\Sigma \vdash M = N$ to $M = N$ when Σ is clear from context or unimportant. We write $\Sigma \not\vdash M = N$ for the negation of $\Sigma \vdash M = N$.

An equational theory may be generated from a finite set of equational axioms, or even from rewrite rules, but this property is not essential for us. We tend to ignore the mechanics of specifying equational theories.

As usual, a context is an expression (a process or extended process) with a hole. An *evaluation context* is a context whose hole is not under a replication, a conditional, an input, or an output. A context $C[\_]$ *closes* A when $C[A]$ is closed.

Structural equivalence $\equiv$ is the smallest equivalence relation on extended processes that is closed by α-conversion on both names and variables, by application of evaluation contexts, and such that:

$$\begin{array}{llcl}
\textsc{Par-}\mathbf{0} & A & \equiv & A \mid \mathbf{0} \\
\textsc{Par-A} & A \mid (B \mid C) & \equiv & (A \mid B) \mid C \\
\textsc{Par-C} & A \mid B & \equiv & B \mid A \\
\textsc{Repl} & !P & \equiv & P \mid !P \\
\\
\textsc{New-}\mathbf{0} & \nu n.\mathbf{0} & \equiv & \mathbf{0} \\
\textsc{New-C} & \nu u.\nu v.A & \equiv & \nu v.\nu u.A \\
\textsc{New-Par} & A \mid \nu u.B & \equiv & \nu u.(A \mid B) \\
 & & & \text{when } u \notin fv(A) \cup fn(A) \\
\\
\textsc{Alias} & \nu x.\{^M/_x\} & \equiv & \mathbf{0} \\
\textsc{Subst} & \{^M/_x\} \mid A & \equiv & \{^M/_x\} \mid A\{^M/_x\} \\
\textsc{Rewrite} & \{^M/_x\} & \equiv & \{^N/_x\} \quad \text{when } \Sigma \vdash M = N
\end{array}$$

The rules for parallel composition and restriction are standard. Alias enables the introduction of an arbitrary active substitution. Subst describes the application of an active substitution to a process that is in contact with it. Rewrite deals with equational rewriting. In combination, Alias and Subst yield $A\{^M/_x\} \equiv \nu x.(\{^M/_x\} \mid A)$ for $x \notin fv(M)$:

$$\begin{array}{rcll}
A\{^M/_x\} & \equiv & A\{^M/_x\} \mid \mathbf{0} & \text{by } \textsc{Par-}\mathbf{0} \\
 & \equiv & \mathbf{0} \mid A\{^M/_x\} & \text{by } \textsc{Par-C} \\
 & \equiv & (\nu x.\{^M/_x\}) \mid A\{^M/_x\} & \text{by } \textsc{Alias} \\
 & \equiv & \nu x.(\{^M/_x\} \mid A\{^M/_x\}) & \text{by } \textsc{New-Par} \\
 & \equiv & \nu x.(\{^M/_x\} \mid A) & \text{by } \textsc{Subst}
\end{array}$$

Using structural equivalence, every closed extended process A can be rewritten to consist of a substitution and a closed plain process with some restricted names:

$$A \quad \equiv \quad \nu\widetilde{n}.\{^{\widetilde{M}}/_{\widetilde{x}}\} \mid P$$

where $fv(P) = \emptyset$, $fv(\widetilde{M}) = \emptyset$, and $\{\widetilde{n}\} \subseteq fn(\widetilde{M})$. In particular, every closed frame φ can be rewritten to consist of a substitution with some restricted names:

$$\varphi \quad \equiv \quad \nu\widetilde{n}.\{^{\widetilde{M}}/_{\widetilde{x}}\}$$

where $fv(\widetilde{M}) = \emptyset$ and $\{\widetilde{n}\} \subseteq fn(\widetilde{M})$. The set $\{\widetilde{x}\}$ is the domain of φ.

Internal reduction $\rightarrow$ is the smallest relation on extended processes closed by structural equivalence and application of evaluation contexts such that:

$$\begin{array}{lrcl}
\textsc{Comm} & \overline{a}\langle x\rangle.P \mid a(x).Q & \rightarrow & P \mid Q \\
\textsc{Then} & \textit{if } M = M \textit{ then } P \textit{ else } Q & \rightarrow & P \\
\textsc{Else} & \textit{if } M = N \textit{ then } P \textit{ else } Q & \rightarrow & Q \\
 & \multicolumn{3}{l}{\text{for any ground terms } M \text{ and } N} \\
 & \multicolumn{3}{l}{\text{such that } \Sigma \not\vdash M = N}
\end{array}$$

Communication (Comm) is remarkably simple because the message concerned is a variable; this simplicity entails no loss of generality because Alias and Subst can introduce a variable to stand for a term:

$$\begin{array}{rcl}
\overline{a}\langle M\rangle.P \mid a(x).Q & \equiv & \nu x.(\{^M/_x\} \mid \overline{a}\langle x\rangle.P \mid a(x).Q) \\
 & \rightarrow & \nu x.(\{^M/_x\} \mid P \mid Q) \quad \text{by } \textsc{Comm} \\
 & \equiv & P \mid Q\{^M/_x\}
\end{array}$$

(This derivation assumes that $x \notin fv(M) \cup fv(P)$, which can be established by α-conversion as needed.)

Comparisons (Then and Else) directly depend on the underlying equational theory; using Else sometimes requires that active substitutions in the context be applied first, to yield ground terms M and N.

This use of the equational theory may be reminiscent of initial algebras. In an initial algebra, the principle of "no confusion" dictates that two elements are equal only if this is required by the corresponding equational theory. Similarly, *if* $M = N$ *then* P *else* Q reduces to P only if this is required by the equational theory, and reduces to Q otherwise. Initial algebras also obey the principle of "no junk", which says that all elements correspond to terms built exclusively from function symbols of the signature. In contrast, a fresh name need not equal any such term in the applied pi calculus.

3 Brief examples

This section collects several examples, focusing on signatures, equations, and some simple processes. We start with pairs; this trivial example serves to introduce some notations and issues. We then discuss one-way hash functions, encryption functions, digital signatures, and the XOR function [30, 40]. Further examples appear in sections 5 and 6.

Of course, at least some of these functions appear in most formalizations of cryptography and security protocols. In comparison with the spi calculus, the applied pi calculus permits a more uniform and versatile treatment of these functions, their variants, and their properties. Like the spi calculus, however, the applied pi calculus takes advantage of notations, concepts, and techniques from programming languages.

Pairs and other data structures Algebraic datatypes such as pairs, tuples, arrays, and lists occur in many examples. Encoding them in the pure pi calculus is not hard, but neither is representing them as primitive. For instance, the signature Σ may contain the binary function symbol pair and the unary function symbols fst and snd, with the abbreviation (M, N) for $\mathsf{pair}(M, N)$, and the evident equations:

$$\begin{aligned} \mathsf{fst}((x, y)) &= x \\ \mathsf{snd}((x, y)) &= y \end{aligned}$$

(So the equational theory consists of these equations, and all obtained by reflexivity, symmetry, and transitivity and by substituting terms for variables.) The sort system may enforce that fst and snd are applied only to pairs. Alternatively, we may add a boolean function that recognizes pairs. We may also add equations that describe the behavior of fst and snd on other values (e.g., adding a constant symbol wrong, and equations $\mathsf{fst}(M) = \mathsf{snd}(M) = \mathsf{wrong}$ for all appropriate ground terms M). We usually omit such standard variants in other examples.

Using pairs, we may for instance write the process:

$$\nu s.\big(\overline{a}\langle (M, s)\rangle \mid a(x).\mathit{if}\ \mathsf{snd}(x) = s\ \mathit{then}\ \overline{b}\langle \mathsf{fst}(x)\rangle\big)$$

One of its components sends a pair consisting of a term M and a fresh name s on a channel a. The other receives a message on a and, if its second component is s, it forwards the first component on a channel b. Thus, we may say that s serves as a capability (or password) for the forwarding. However, this capability is not protected from eavesdroppers when it travels on a. Any other process can listen on a and can apply snd, thus learning s. We can represent such an attacker within the calculus, for example by the following process:

$$a(x).\overline{a}\langle (N, \mathsf{snd}(x))\rangle$$

which may receive (M, s) on a and send (N, s) on a. Composing this attacker with the program, we may obtain N instead of M on b.

One-way hash functions In contrast, we represent a one-way hash function as a unary function symbol h with no equations. The absence of an inverse for h models the one-wayness of h. The fact that $\mathsf{h}(M) = \mathsf{h}(N)$ only when $M = N$ models that h is collision-free.

Modifying our first example, we may now write:

$$\nu s.\left(\begin{array}{l} \overline{a}\langle (M, \mathsf{h}(s, M))\rangle \mid \\ a(x).\mathit{if}\ \mathsf{h}(s, \mathsf{fst}(x)) = \mathsf{snd}(x)\ \mathit{then}\ \overline{b}\langle \mathsf{fst}(x)\rangle \end{array}\right)$$

Here the value M is signed by hashing it with the fresh name s. Although $(M, \mathsf{h}(s, M))$ travels on the public channel a, no other process can extract s from this, or produce $(N, \mathsf{h}(s, N))$ for some other N using the available functions. Therefore, we may reason that only the intended term M will be forwarded on channel b.

This example is a typical cryptographic application of one-way hash functions. In light of the practical importance of those applications, our treatment of one-way hash functions is attractively straightforward. Still, we may question whether our formal model of these functions is not too strong and simplistic in comparison with the properties of actual implementations based on algorithms such as MD5 and SHA. In section 6, we consider a somewhat weaker, subtler model for keyed hash functions.

Symmetric encryption In order to model symmetric cryptography (that is, shared-key cryptography), we take binary function symbols enc and dec for encryption and decryption, respectively, with the equation:

$$\mathsf{dec}(\mathsf{enc}(x, y), y) = x$$

Here x represents the plaintext and y the key. We often use fresh names as keys in examples; for instance, the (useless) process:

$$\nu k.\overline{a}\langle \mathsf{enc}(M, k)\rangle$$

sends the term M encrypted under a fresh key k.

In applications of encryption, it is frequent to assume that each encrypted message comes with sufficient redundancy so that decryption with the "wrong" key is evident. We could consider incorporating this property for example by adding the equation $\mathsf{dec}(M, N) = \mathsf{wrong}$ whenever M and N are two ground terms and $M \neq \mathsf{enc}(L, N)$ for all L. On the other hand, in modern cryptology, such redundancy is not usually viewed as part of the encryption function proper, but rather an addition. The redundancy can be implemented with message authentication codes. Accordingly, we do not build it in.

Asymmetric encryption It is only slightly harder to model asymmetric (public-key) cryptography, where the keys for encryption and decryption are different. We introduce two new unary function symbols pk and sk for generating public and secret keys from a seed, and the equation:

$$\mathsf{dec}(\mathsf{enc}(x, \mathsf{pk}(y)), \mathsf{sk}(y)) = x$$

We may now write the process:

$$\nu s.\big(\overline{a}\langle \mathsf{pk}(s)\rangle \mid b(x).\overline{c}\langle \mathsf{dec}(x, \mathsf{sk}(s))\rangle\big)$$

The first component publishes the public key $\mathsf{pk}(s)$ by sending it on a. The second receives a message on b, uses the corresponding secret key $\mathsf{sk}(s)$ to decrypt it, and forwards the resulting plaintext on c. As this example indicates, we essentially view ν as a generator of unguessable seeds. In some cases, those seeds may be directly used as passwords or keys; in others, some transformations are needed.

Some encryption schemes have additional properties. In particular, enc and dec may be the same function. This property matters in implementations, and sometimes permits attacks. Moreover, certain encryptions and decryptions commute in some schemes. For example, we have $\mathsf{dec}(\mathsf{enc}(x, y), z) = \mathsf{enc}(\mathsf{dec}(x, z), y)$ if the encryptions and decryptions are performed using RSA with the same modulus. The treatment of such properties is left open in [5]. In contrast, it is easy to express the properties in the applied pi calculus, and to study the protocols and attacks that depend on them.

Non-deterministic ("probabilistic") encryption Going further, we may add a third argument to enc, so that the encryption of a plaintext with a key is not unique. This nondeterminism is an essential property of probabilistic encryption systems [23]. The equation for decryption becomes:

$$\mathsf{dec}(\mathsf{enc}(x, \mathsf{pk}(y), z), \mathsf{sk}(y)) \quad = \quad x$$

With this variant, we may write the process:

$$a(x).(\nu m.\overline{b}\langle \mathsf{enc}(M, x, m)\rangle \mid \nu n.\overline{c}\langle \mathsf{enc}(N, x, n)\rangle)$$

which receives a message x and uses it as an encryption key for two messages, $\mathsf{enc}(M, x, m)$ and $\mathsf{enc}(N, x, n)$. An observer who does not have the corresponding decryption key cannot tell whether the underlying plaintexts M and N are identical by comparing the ciphertexts, because the ciphertexts rely on different fresh names m and n. Moreover, even if the observer learns x, M, and N (but not the decryption key), it cannot verify that the messages contain M and N because it does not know m and n.

Public-key digital signatures Like public-key encryption schemes, digital-signature schemes rely on pairs of public and secret keys. In each pair, the secret key serves for computing signatures and the public key for verifying those signatures. In order to model digital signatures and their checking, we use again the two unary function symbols pk and sk for generating public and secret keys from a seed. We also use the new binary function symbol sign, the ternary function symbol check, and the constant symbol ok, with the equation:

$$\mathsf{check}(x, \mathsf{sign}(x, \mathsf{sk}(y)), \mathsf{pk}(y)) \quad = \quad \mathsf{ok}$$

(Several variants are possible.)

Modifying again our first example, we may now write:

$$\Big(\nu s.\{^{\mathsf{pk}(s)}/_{y}\} \mid \overline{a}\langle(M, \mathsf{sign}(M, \mathsf{sk}(s)))\rangle\Big) \mid$$
$$a(x).\mathit{if}\ \mathsf{check}(\mathsf{fst}(x), \mathsf{snd}(x), y) = \mathsf{ok}\ \mathit{then}\ \overline{b}\langle \mathsf{fst}(x)\rangle$$

Here the value M is signed using the secret key $\mathsf{sk}(s)$. Although M and its signature travel on the public channel a, no other process can produce N and its signature for some other N. Therefore, again, we may reason that only the intended term M will be forwarded on channel b. This property holds despite the publication of $\mathsf{pk}(s)$ (but not $\mathsf{sk}(s)$), which is represented by the active substitution that maps y to $\mathsf{pk}(s)$. Despite the restriction on s, processes outside the restriction can use $\mathsf{pk}(s)$ through y. In particular, y refers to $\mathsf{pk}(s)$ in the process that checks the signature on M.

XOR Finally, we may model the XOR function, some of its uses in cryptography, and some of the protocol flaws connected with it. Some of these flaws stem from the intrinsic equational properties of XOR, such as cancellation property that we may write:

$$\mathsf{xor}(\mathsf{xor}(x, y), y) \quad = \quad x$$

Others arise because of the interactions between XOR and other operations (e.g., [41, 15]). For example, CRCs (cyclic redundancy codes) can be poor proofs of integrity, partly because of the equation:

$$\mathsf{crc}(\mathsf{xor}(x, y)) \quad = \quad \mathsf{xor}(\mathsf{crc}(x), \mathsf{crc}(y))$$

4 Equivalences and proof techniques

In examples, we frequently argue that two given processes cannot be distinguished by any context, that is, that the processes are observationally equivalent. The spi calculus developed the idea that the context represents an active attacker, and equivalences capture authenticity and secrecy properties in the presence of the attacker.

In this section we define observational equivalence for the applied pi calculus. We also introduce a notion of static equivalence for frames, a labeled semantics for processes, and a labeled equivalence relation. We prove that labeled equivalence and observational equivalence coincide, obtaining a convenient proof technique for observational equivalence.

4.1 Observational equivalence

We write $A \Downarrow a$ when A can send a message on a, that is, when $A \rightarrow^* C[\overline{a}\langle M\rangle.P]$ for some evaluation context $C[\_]$ that does not bind a.

Definition 1 Observational equivalence *($\approx$) is the largest symmetric relation $\mathcal{R}$ between closed extended processes with the same domain such that $A\ \mathcal{R}\ B$ implies:*

1. *if $A \Downarrow a$, then $B \Downarrow a$;*
2. *if $A \rightarrow^* A'$, then $B \rightarrow^* B'$ and $A'\ \mathcal{R}\ B'$ for some B';*
3. *$C[A]\ \mathcal{R}\ C[B]$ for all closing evaluation contexts $C[\_]$.*

These definitions are standard in the pi calculus, where $\Downarrow a$ is called a *barb* on a, and where $\approx$ is one of the two usual notions of barbed bisimulation congruence. (See [20] for details.)

For example, when h is a unary function symbol with no equations, we obtain that $\nu s.\overline{a}\langle s\rangle \approx \nu s.\overline{a}\langle \mathsf{h}(s)\rangle$.

4.2 Static equivalence

Two substitutions may be seen as equivalent when they behave equivalently when applied to terms. We write $\approx_s$ for this notion of equivalence, and call it static equivalence. In the presence of the "new" construct, defining $\approx_s$ is somewhat delicate and interesting. For instance, consider two functions f and g with no equations (intuitively, two independent one-way hash functions), and the three frames:

$$\varphi_0 \stackrel{\text{def}}{=} \nu k.\{^{k}/_{x}\} \mid \nu s.\{^{s}/_{y}\}$$
$$\varphi_1 \stackrel{\text{def}}{=} \nu k.\{^{\mathsf{f}(k)}/_{x}, ^{\mathsf{g}(k)}/_{y}\}$$
$$\varphi_2 \stackrel{\text{def}}{=} \nu k.\{^{k}/_{x}, ^{\mathsf{f}(k)}/_{y}\}$$

$$\begin{array}{rcl}
\nu k.\overline{a}\langle \mathsf{enc}(M,k)\rangle.\overline{a}\langle k\rangle.a(z).\mathit{if}\ z=M\ \mathit{then}\ \overline{c}\langle\, \mathsf{oops!}\,\rangle & \xrightarrow{\nu x.\overline{a}\langle x\rangle} & \nu k.(\{^{\mathsf{enc}(M,k)}/_x\} \mid \overline{a}\langle k\rangle.a(z).\mathit{if}\ z=M\ \mathit{then}\ \overline{c}\langle\, \mathsf{oops!}\,\rangle) \\
& \xrightarrow{\nu y.\overline{a}\langle y\rangle} & \nu k.(\{^{\mathsf{enc}(M,k)}/_x\} \mid \{^k/_y\} \mid a(z).\mathit{if}\ z=M\ \mathit{then}\ \overline{c}\langle\, \mathsf{oops!}\,\rangle) \\
& \xrightarrow{a(\mathsf{dec}(x,y))} & \nu k.(\{^{\mathsf{enc}(M,k)}/_x\} \mid \{^k/_y\} \mid \mathit{if}\ \mathsf{dec}(x,y)=M\ \mathit{then}\ \overline{c}\langle\, \mathsf{oops!}\,\rangle) \\
& \rightarrow & \nu k.(\{^{\mathsf{enc}(M,k)}/_x\} \mid \{^k/_y\}) \mid \overline{c}\langle\, \mathsf{oops!}\,\rangle
\end{array}$$

Figure 1: Example transitions

In φ_0, the variables x and y are mapped to two unrelated values that are different from any value that the context may build (since k and s are new). These properties also hold, but more subtly, for φ_1; although $\mathsf{f}(k)$ and $\mathsf{g}(k)$ are based on the same underlying fresh name, they look unrelated. (Analogously, it is common to construct apparently unrelated keys by hashing from a single underlying secret, as in SSL [21].) Hence, a context that obtains the values for x and y cannot distinguish φ_0 and φ_1. On the other hand, the context can discriminate φ_2 by testing the predicate $\mathsf{f}(x) = y$. Therefore, we would like to define static equivalence so that $\varphi_0 \approx_s \varphi_1 \not\approx_s \varphi_2$.

This example relies on a concept of equality of terms in a frame, which the following definition captures.

Definition 2 *We say that two terms M and N are equal in the frame φ, and write $(M = N)\varphi$, if and only if $\varphi \equiv \nu\widetilde{n}.\sigma$, $M\sigma = N\sigma$, and $\{\widetilde{n}\} \cap (fn(M) \cup fn(N)) = \emptyset$ for some names $\widetilde{n}$ and substitution σ.*

For instance, in our example, we have $(\mathsf{f}(x) = y)\varphi_2$ but not $(\mathsf{f}(x) = y)\varphi_1$, hence $\varphi_1 \not\approx_s \varphi_2$.

Definition 3 *We say that two closed frames φ and ψ are* statically equivalent*, and write $\varphi \approx_s \psi$, when $dom(\varphi) = dom(\psi)$ and when, for all terms M and N, we have $(M = N)\varphi$ if and only if $(M = N)\psi$.*

We say that two closed extended processes are statically equivalent, and write $A \approx_s B$, when their frames are statically equivalent.

Depending on Σ, static equivalence can be quite hard to check, but at least it does not depend on the dynamics of processes. Some simplifications are possible in common cases, for example when terms can be put in normal forms.

The next two lemmas state several basic, important properties of $\approx_s$:

Lemma 1 *Static equivalence is closed by structural equivalence, by reduction, and by application of closing evaluation contexts.*

Lemma 2 *Observational equivalence and static equivalence coincide on frames. Observational equivalence is strictly finer than static equivalence on extended processes: $\approx\, \subset\, \approx_s$.*

To see that observational equivalence implies static equivalence, note that if A and B are observationally equivalent then $A \mid C$ and $B \mid C$ have the same barbs for every C, and that they are statically equivalent when $A \mid C$ and $B \mid C$ have the same barb $\Downarrow a$ for every C of the special form $\mathit{if}\ M = N\ \mathit{then}\ \overline{a}\langle s\rangle$, where a does not occur in A or B.

4.3 Labeled operational semantics and equivalence

A labeled operational semantics extends the chemical semantics of section 2, enabling us to reason about processes that interact with their environment. The labeled semantics defines a relation $A \xrightarrow{\alpha} A'$, where α is a label of one of the following forms:

- a label $a(M)$, where M is a term that may contain names and variables, which corresponds to an input of M on a;
- a label $\overline{a}\langle u\rangle$ or $\nu u.\overline{a}\langle u\rangle$, where u is either a channel name or a variable of base type, which corresponds to an output of u on a.

In addition to the rules for structural equivalence and reduction of section 2, we adopt the following rules:

In $\qquad a(x).P \xrightarrow{a(M)} P\{^M/_x\}$

Out-Atom $\qquad \overline{a}\langle u\rangle.P \xrightarrow{\overline{a}\langle u\rangle} P$

Open-Atom $\qquad \dfrac{A \xrightarrow{\overline{a}\langle u\rangle} A' \qquad u \neq a}{\nu u.A \xrightarrow{\nu u.\overline{a}\langle u\rangle} A'}$

Scope $\qquad \dfrac{A \xrightarrow{\alpha} A' \qquad u \text{ does not occur in } \alpha}{\nu u.A \xrightarrow{\alpha} \nu u.A'}$

Par $\qquad \dfrac{A \xrightarrow{\alpha} A' \quad bv(\alpha) \cap fv(B) = bn(\alpha) \cap fn(B) = \emptyset}{A \mid B \xrightarrow{\alpha} A' \mid B}$

Struct $\qquad \dfrac{A \equiv B \qquad B \xrightarrow{\alpha} B' \qquad B' \equiv A'}{A \xrightarrow{\alpha} A'}$

According to In, a term M may be input. On the other hand, Out-Atom permits output only for channel names and for variables of base type. Other terms can be output only "by reference": a variable can be associated with the term in question and output.

For example, using the signature and equations for symmetric encryption, and the new constant symbol $\mathsf{oops!}$, we have the sequence of transitions of Figure 1. The first two transitions do not directly reveal the term M. However, they give enough information to the environment to compute M as $\mathsf{dec}(x, y)$, and to input it in the third transition.

The labeled operational semantics leads to an equivalence relation:

Definition 4 Labeled bisimilarity *($\approx_l$) is the largest symmetric relation $\mathcal{R}$ on closed extended processes such that $A \mathcal{R} B$ implies:*

1. *$A \approx_s B$;*
2. *if $A \rightarrow A'$, then $B \rightarrow^* B'$ and $A' \mathcal{R} B'$ for some B';*
3. *if $A \xrightarrow{\alpha} A'$ and $fv(\alpha) \subseteq dom(A)$ and $bn(\alpha) \cap fn(B) = \emptyset$, then $B \rightarrow^* \xrightarrow{\alpha} \rightarrow^* B'$ and $A' \mathcal{R} B'$ for some B'.*

Conditions 2 and 3 are standard; condition 1, which requires that bisimilar processes be statically equivalent, is necessary for example in order to distinguish the frames φ_0 and φ_2 of section 4.2.

Our main result is that this relation coincides with observational equivalence. Although such results are fairly common in process calculi, they are important and non-trivial.

Theorem 1 *Observational equivalence is labeled bisimilarity: $\approx = \approx_l$.*

One of the lemmas in the proof of this theorem says that $\approx_l$ is closed by application of closing evaluation contexts. However, unlike the definition of $\approx$, the definition of $\approx_l$ does not include a condition about contexts. It therefore permits simpler proofs.

In addition, labeled bisimilarity can be established via standard "bisimulation up to context" techniques [38], which enable useful on-the-fly simplifications in frames after output steps. The following lemmas provide methods for simplifying frames:

Lemma 3 (Alias elimination) *Let A and B be closed extended processes. If $\{^M/_x\} \mid A \approx_l \{^M/_x\} \mid B$, then $A \approx_l B$.*

Lemma 4 (Name disclosure) *Let A and B be closed extended processes. If $\nu s.(\{^s/_x\} \mid A) \approx_l \nu s.(\{^s/_x\} \mid B)$, then $A \approx_l B$.*

In Lemma 3, the substitution $\{^M/_x\}$ can affect only the context, since A and B are closed. However, the lemma implies that the substitution does not give or mask any information about A and B to the context. In Lemma 4, the restriction on s and the substitution $\{^s/_x\}$ mean that the context can access s only indirectly, through the free variable x. Crucially, s is a name of base type. Intuitively, the lemma says that indirect access is equivalent to direct access in this case.

This labeled operational semantics contrasts with a more naive semantics carried over from the pure pi calculus, with output labels $\nu\widetilde{u}.\overline{a}\langle M\rangle$ and rules that permit direct output of any term, such as:

$$\textsc{Out-Term} \qquad \overline{a}\langle M\rangle.P \xrightarrow{\overline{a}\langle M\rangle} P$$

$$\textsc{Open-All} \qquad \frac{A \xrightarrow{\nu\widetilde{u}.\overline{a}\langle M\rangle} A' \quad v \in fv(M) \cup fn(M) \setminus \{a, \widetilde{u}\}}{\nu v.A \xrightarrow{\nu v,\widetilde{u}.\overline{a}\langle M\rangle} A'}$$

These rules lead to a different, finer equivalence relation, which for example would distinguish $\nu k, s.\overline{a}\langle(k, s)\rangle$ and $\nu k.\overline{a}\langle(\mathsf{f}(k), \mathsf{g}(k))\rangle$. This equivalence relation is often inadequate in applications (as in [5, section 5.2.1]), hence our definitions.

We have also studied intermediately liberal rules for output, which permit direct output of certain terms. We explain those rules next.

4.4 Refining the labeled operational semantics

In the labeled operational semantics of section 4.3, the labels for outputs do not reveal much about the terms being output. Except for channel names, those terms are represented by variables. Often, however, more explicit labels can be convenient in reasoning about protocols, and they do not cause harm as long as they only make explicit information that is immediately available to the environment. For instance, for the process $\nu k.\overline{a}\langle(\mathsf{Header}, \mathsf{enc}(M, k))\rangle$, the label $\nu y.\overline{a}\langle(\mathsf{Header}, y)\rangle$ is more informative than $\nu x.\overline{a}\langle x\rangle$. In this example, the environment could anyway derive that $\mathsf{fst}(x) = \mathsf{Header}$. More generally, we rely on the following definition to characterize the information that the environment can derive.

Definition 5 *A variable x can be derived from the extended process A when, for some term M and extended process A', we have $A \equiv \{^M/_x\} \mid A'$.*

In general, when $x \in dom(A)$, there exist $\widetilde{n}$, M, and A' such that $A \equiv \nu\widetilde{n}.\{^M/_x\} \mid A'$. If x can be derived from A, then $\widetilde{n}$ can be chosen empty, so that M is not under restrictions. Intuitively, if x can be derived from A, then A does not reveal more information than $\nu x.A$, because the context can build the term M and use it instead of x. For example, using function symbols for pairs and symmetric encryption, we let:

$$\varphi \stackrel{\text{def}}{=} \nu k.\{^M/_x, ^{\mathsf{enc}(x,k)}/_y, ^{(y,N)}/_z\}$$

The variable y can be derived from φ using $\mathsf{fst}(z)$. Formally, we have:

$$\varphi \equiv \{^{\mathsf{fst}(z)}/_y\} \mid \nu k.\{^M/_x, ^{(\mathsf{enc}(x,k),N)}/_z\}$$

In contrast, x and z cannot be derived from φ in general. However, if k does not occur in N, then z can be derived from φ using (y, N):

$$\varphi \equiv \{^{(y,N)}/_z\} \mid \nu k.\{^M/_x, ^{\mathsf{enc}(x,k)}/_y\}$$

Conversely, if $N = k$, then x can be derived from φ using $\mathsf{dec}(y, \mathsf{snd}(z))$, even if k occurs in M:

$$\varphi \equiv \{^{\mathsf{dec}(y,\mathsf{snd}(z))}/_x\} \mid \nu k.\{^{\mathsf{enc}(M,k)}/_y, ^{(y,k)}/_z\}$$

Relying on Definition 5, we define rules for output that permit composite terms in labels but require that every restricted variable that is exported can be derived by the environment. In the relation $A \xrightarrow{\alpha} A'$, the label α now ranges over the same labels $a(M)$ for input and generalized labels for output of the form $\nu\widetilde{u}.\overline{a}\langle M\rangle$, where M is a term that may contain variables and where $\{\widetilde{u}\} \subseteq fv(M) \cup fn(M)$. The label $\nu\widetilde{u}.\overline{a}\langle M\rangle$ corresponds to an output of M on a that reveals the names and variables $\widetilde{u}$.

We retain the rules for structural equivalence and reduction, and rules In, Par, and Struct. We also keep rule Scope, but only for labels with no extrusion, that is, for labels $\overline{a}\langle M\rangle$ and $a(M)$. As a replacement for the rules Out-Atom and Open-Atom, we use the rules Out-Term and:

$$\textsc{Open-Channel} \qquad \frac{A \xrightarrow{\overline{a}\langle b\rangle} A' \qquad b \neq a}{\nu b.A \xrightarrow{\nu b.\overline{a}\langle b\rangle} A'}$$

$$\textsc{Open-Variable} \qquad \frac{A \xrightarrow{\nu\widetilde{u}.\overline{a}\langle M\rangle} A' \qquad x \in fv(M) \setminus \{\widetilde{u}\} \qquad x \text{ can be derived from } \nu\widetilde{u}.\{^M/_z\} \mid A'}{\nu x.A \xrightarrow{\nu x,\widetilde{u}.\overline{a}\langle M\rangle} A'}$$

Rule Open-Channel is the special of Open-Atom for channel names. Rule Open-Variable filters output transitions whose contents may reveal restricted variables. Only non-derivable subterms have to be replaced with variables before the output. Thus, these rules are more liberal than those of section 4.3. In fact, it is easy to check that the rules of section 4.3 are special cases of these ones.

For instance, consider $A_1 = \nu k.\overline{a}\langle(\mathsf{f}(k), \mathsf{g}(k))\rangle$ and $A_2 = \nu k.\overline{a}\langle(k, \mathsf{f}(k))\rangle$. With the rules of section 4.3, we have:

$$A_i \xrightarrow{\nu z.\overline{a}\langle z\rangle} \nu x, y.\{^{(x,y)}/_z\} \mid \varphi_i$$

where x, y can be eliminated and φ_i is as in section 4.2. With the new rules, we also have:

$$A_i \xrightarrow{\nu x,y.\overline{a}\langle(x,y)\rangle} \varphi_i$$

This transition is adequate for A_1 since x and y behave like fresh, independent values. For A_2, we also have the more informative transition:

$$A_2 \xrightarrow{\nu x.\overline{a}\langle(x,\mathsf{f}(x))\rangle} \nu k.\{^k/_x\}$$

that reveals the link between x and y, but not that x is a name.

In general, for a given message, we may have several output transitions. Each of these transitions may lead to a process with a different frame. However, it suffices to consider any one of the transitions in order to prove that a relation is included in labeled bisimilarity. Hence, a particular label can be chosen to reflect the structure of the protocol at hand, and to limit the complexity of the resulting frame.

The next theorem states that the two semantics yield the same notion of equivalence. Thus, making the labels more explicit only makes apparent some of the information that is otherwise kept in the static, equational part of $\approx_l$.

Theorem 2 *Let $\approx_L$ be the relation of labeled bisimilarity obtained by applying Definition 4 to the semantics of this section. We have $\approx_l = \approx_L$.*

In another direction, we can refine the semantics to permit functions that take channels as arguments or produce them as results (which are excluded in section 2). For example, we can permit a pairing function for channels. Thus, although the separation of channels from other values is frequent in examples and convenient, it is not essential.

For this purpose, we would allow the use of the rule Open-All in the case where v is a channel b. The disadvantages of this rule (indicated above) do not arise if two reasonable constraints are met: (1) channel sorts contain only pairwise-distinct names up to term rewriting; (2) for every term M with a channel variable x, there is a channel term N with free variable y and no free names such that $x = N\{^M/_y\}$.

5 Diffie-Hellman key agreement (example)

The fundamental Diffie-Hellman protocol allows two principals to establish a shared secret by exchanging messages over public channels [17]. The principals need not have any shared secrets in advance. The basic protocol, on which we focus here, does not provide authentication; therefore, a "bad" principal may play the role of either principal in the protocol. On the other hand, the two principals that follow the protocol will communicate securely with one another afterwards, even in the presence of active attackers. In extended protocols, such as the Station-to-Station protocol [18] and SKEME [26], additional messages perform authentication.

We program the basic protocol in terms of the binary function symbol f and the unary function symbol g, with the equation:

$$\mathsf{f}(x, \mathsf{g}(y)) \quad = \quad \mathsf{f}(y, \mathsf{g}(x)) \tag{1}$$

Concretely, the functions are $\mathsf{f}(x, y) = y^x \ mod\ p$ and $\mathsf{g}(x) = \alpha^x \ mod\ p$ for a prime p and a generator α of Z_p^*, and we have the equation $\mathsf{f}(x, \mathsf{g}(y)) = (\alpha^y)^x = \alpha^{y\times x} = \alpha^{x\times y} = (\alpha^x)^y = \mathsf{f}(y, \mathsf{g}(x))$. However, we ignore the underlying number theory, working abstractly with f and g.

The protocol has two symmetric participants, which we represent by the processes A_0 and A_1. The protocol establishes a shared key, then the participants respectively run P_0 and P_1 using the key. We use the public channel c_{01} for messages from A_0 to A_1 and the public channel c_{10} for communication in the opposite direction. We assume that none of the values introduced in the protocol appears in P_0 and P_1, except for the key.

In order to establish the key, A_0 invents a name n_0, sends $\mathsf{g}(n_0)$ to A_1, and A_1 proceeds symmetrically. Then A_0 computes the key as $\mathsf{f}(n_0, \mathsf{g}(n_1))$ and A_1 computes it as $\mathsf{f}(n_1, \mathsf{g}(n_0))$, with the same result. We find it convenient to use the following substitutions for A_0's message and key:

$$\begin{aligned} \sigma_0 &\stackrel{\mathrm{def}}{=} \{^{\mathsf{g}(n_0)}/_{x_0}\} \\ \phi_0 &\stackrel{\mathrm{def}}{=} \{^{\mathsf{f}(n_0,x_1)}/_y\} \end{aligned}$$

and the corresponding substitutions σ_1 and ϕ_1, as well as the frame:

$$\varphi \stackrel{\mathrm{def}}{=} (\nu n_0.\ (\phi_0 \mid \sigma_0)) \mid (\nu n_1.\ \sigma_1)$$

With these notations, A_0 is:

$$A_0 \stackrel{\mathrm{def}}{=} \nu n_0.(\overline{c_{01}}\langle x_0\sigma_0\rangle \mid c_{10}(x_1).P_0\phi_0)$$

and A_1 is analogous.

Two reductions represent a normal run of the protocol:

$$\begin{aligned} A_0 \mid A_1 \quad &\to\to \quad \nu x_0, x_1, n_0, n_1.\ (P_0\phi_0 \mid P_1\phi_1 \mid \sigma_0 \mid \sigma_1) && (2)\\ &\equiv \quad \nu x_0, x_1, n_0, n_1, y.\ (P_0 \mid P_1 \mid \phi_0 \mid \sigma_0 \mid \sigma_1) && (3)\\ &\equiv \quad \nu y.(P_0 \mid P_1 \mid \nu x_0, x_1.\ \varphi) && (4) \end{aligned}$$

The two communication steps (2) use structural equivalence to activate the substitutions σ_0 and σ_1 and extend the scope of the secret values n_0 and n_1. The structural equivalence (3) crucially relies on equation (1) in order to reuse the active substitution ϕ_0 instead of ϕ_1 after the reception of x_0 in A_1. The next structural equivalence (4) tightens the scope for restricted names and variables, then uses the definition of φ.

We model an eavesdropper as a process that intercepts messages on c_{01} and c_{10}, remembers them, but forwards them unmodified. In the presence of this passive attacker, the operational semantics says that $A_0 \mid A_1$ yields instead:

$$\nu y.(P_0 \mid P_1 \mid \varphi)$$

The sequence of steps that leads to this result is similar to the one above. The absence of the restrictions on x_0 and x_1

corresponds to the fact that the eavesdropper has obtained the values of these variables.

The following theorem relates this process to

$$\nu k.(P_0 \mid P_1)\{^k/_y\}$$

which represents the bodies P_0 and P_1 of A_0 and A_1 sharing a key k. This key appears as a simple shared name, rather than as the result of communication and computation. Intuitively, we may read $\nu k.(P_0 \mid P_1)\{^k/_y\}$ as the ideal outcome of the protocol: P_0 and P_1 execute using a shared key, without concern for how the key was established, and without any side-effects from weaknesses in the establishment of the key. The theorem says that this ideal outcome is essentially achieved, up to some "noise". This "noise" is a substitution that maps x_0 and x_1 to unrelated, fresh names. It accounts for the fact that an attacker may have the key-exchange messages, and that they look just like unrelated values to the attacker. In particular, the key in use between P_0 and P_1 has no observable relation to those messages, or to any other left-over secrets. We view this independence of the shared key as an important forward-secrecy property.

Theorem 3 *Let P_0 and P_1 be processes with free variable y where the name k does not appear. We have:*

$$\begin{array}{ll} & \nu y.(P_0 \mid P_1 \mid \varphi) \\ \approx & \nu k.(P_0 \mid P_1)\{^k/_y\} \mid \nu s_0.\{^{s_0}/_{x_0}\} \mid \nu s_1.\{^{s_1}/_{x_1}\} \end{array}$$

The theorem follows from Lemma 2 and the static equivalence $\varphi \approx_s \nu s_0, s_1, k.\{^{s_0}/_{x_0}, ^{s_1}/_{x_1}, ^k/_y\}$, which says that the frame φ generated by the protocol execution is equivalent to one that maps variables to fresh names.

Extensions of the basic protocol add rounds of communication that confirm the key and authenticate the principals. We have studied one such extension with key confirmation. There, the shared secret $\mathsf{f}(n_0, \mathsf{g}(n_1))$ is used in confirmation messages. Because of these messages, the shared secret can no longer be equated with a virgin key for P_0 and P_1. Instead, the final key is computed by hashing the shared secret. This hashing guarantees the independence of the final key.

6 Message authentication codes and hashing (another example)

Message authentication codes (MACs) are common cryptographic operations. In this section we treat MACs and their constructions from one-way hash functions. This example provides a further illustration of the usefulness of equations in the applied pi calculus. On the other hand, some aspects of MAC constructions are rather low-level, and we would not expect to account for all their combinatorial details (e.g., the "birthday attacks"). A higher-level task is to express and reason about protocols treating MACs as primitive; this is squarely within the scope of our approach.

6.1 Using MACs

MACs serve to authenticate messages using shared keys. When k is a key and M is a message, and k is known only to a certain principal A and to the recipient B of the message, B may take $\mathsf{mac}(k, M)$ as proof that M comes from A. More precisely, B can check $\mathsf{mac}(k, M)$ by recomputing it upon receipt of M and $\mathsf{mac}(k, M)$, and reason that A must be the sender of M. This property should hold even if A generates MACs for other messages as well; those MACs should not permit forging a MAC for M. In the worst case, it should hold even if A generates MACs for other messages on demand.

Using a new binary function symbol mac, we may describe this scenario by the following processes:

$$\begin{array}{lcl} A & \stackrel{\text{def}}{=} & !a(x).\overline{b}\langle(x, \mathsf{mac}(k, x))\rangle \\ B & \stackrel{\text{def}}{=} & b(y).\mathit{if}\ \mathsf{mac}(k, \mathsf{fst}(y)) = \mathsf{snd}(y)\ \mathit{then}\ \overline{c}\langle\mathsf{fst}(y)\rangle \\ S & \stackrel{\text{def}}{=} & \nu k.(A \mid B) \end{array}$$

The process S represents the complete system, composed of A and B; the restriction on k means that k is private to A and B. The process A receives messages on a public channel a and returns them MACed on the public channel b. When B receives a message on b, it checks its MAC and acts upon it, here simply by forwarding on a channel c. Intuitively, we would expect that B forwards on c only a message that A has MACed. In other words, although an attacker may intercept, modify, and inject messages on b, it should not be able to forge a MAC and trick B into forwarding some other message.

This property can be expressed precisely in terms of the labeled semantics and it can be checked without too much difficulty when mac is a primitive function symbol with no equations. The property remains true even if there is a function $\mathsf{extract}$ that maps a MAC $\mathsf{mac}(x, y)$ to the underlying cleartext y, with the equation $\mathsf{extract}(\mathsf{mac}(x, y)) = y$. Since MACs are not supposed to guarantee secrecy, such a function may well exist, so it is safer to assume that it is available to the attacker.

The property is more delicate if mac is defined from other operations, as it invariably is in practice. In that case, the property may even be taken as *the* specification of MACs [22]. Thus, a MAC implementation may be deemed correct if and only if the process S works as expected when mac is instantiated with that implementation. More specifically, the next section deals with the question of whether the property remains true when mac is defined from hash functions.

6.2 Constructing MACs

In section 3, we give no equations for one-way hash functions. In practice, one-way hash functions are commonly defined by iterating a basic binary compression function, which maps two input blocks to one output block. Furthermore, keyed one-way hash functions include a key as an additional argument. Thus, we may have:

$$\mathsf{f}(x, y + z) \quad = \quad \mathsf{h}(\mathsf{f}(x, y), z)$$

where f is the keyed one-way hash function, h is the compression function, x is a key, and $y + z$ represents the concatenation of block z to y. Concatenation $(+)$ associates to the left. We also assume other standard operations on sequences and the corresponding equations.

In this equation we are rather abstract in our treatment of blocks, their sizes, and therefore of padding and other related issues. We also ignore two common twists: some functions use initialization vectors to start the iteration, and some append a length block to the input. Nevertheless, we can explain various MAC constructions, describing flaws in some and reasoning about the properties of others.

$$
\begin{array}{lll}
\nu k.(A \mid B) & \xrightarrow{a(M)} & \nu k.(A \mid B \mid \overline{b}\langle(M, \mathsf{mac}(k, M))\rangle) \\
& \xrightarrow{\nu x.\overline{b}\langle x\rangle} & \nu k.(A \mid B \mid \{^{(M, \mathsf{mac}(k,M))}/_{x}\}) \\
& \xrightarrow{b((M+N, \mathsf{h}(\mathsf{snd}(x), N)))} & \nu k.(A \mid \overline{c}\langle M+N\rangle \mid \{^{(M, \mathsf{mac}(k,M))}/_{x}\}) \\
& \xrightarrow{\nu y.\overline{c}\langle y\rangle} & \nu k.(A \mid \{^{(M, \mathsf{mac}(k,M))}/_{x}, {}^{M+N}/_{y}\})
\end{array}
$$

Figure 2: An attack scenario

$$
\begin{array}{lll}
\nu k.(A \mid B) & \xrightarrow{a(M)} & \nu k.(A \mid B \mid \overline{b}\langle(M, \mathsf{mac}(k, M))\rangle) \\
& \xrightarrow{\nu x.\overline{b}\langle(M,x)\rangle} & \nu k.(A \mid B \mid \{^{\mathsf{mac}(k,M)}/_{x}\}) \\
& \xrightarrow{b((M+N, \mathsf{h}(x, N)))} & \nu k.(A \mid \overline{c}\langle M+N\rangle \mid \{^{\mathsf{mac}(k,M)}/_{x}\}) \\
& \xrightarrow{\overline{c}\langle M+N\rangle} & \nu k.(A \mid \{^{\mathsf{mac}(k,M)}/_{x}\})
\end{array}
$$

Figure 3: An attack scenario (with refined labels)

A first, classical definition of a MAC from a keyed one-way hash function f is:

$$\mathsf{mac}(x, y) \stackrel{\text{def}}{=} \mathsf{f}(x, y)$$

For instance, the MAC of a three-block message $M = M_0 + M_1 + M_2$ with key k is $\mathsf{mac}(k, M) = \mathsf{h}(\mathsf{h}(\mathsf{f}(k, M_0), M_1), M_2)$. This implementation is subject to a well-known extension attack. Given the MAC of M, an attacker can compute the MAC of any extension $M + N$ without knowing the MAC key, since $\mathsf{mac}(k, M + N) = \mathsf{h}(\mathsf{mac}(k, M), N)$. We can describe the attack formally through the operational semantics, as done in Figure 2 and in Figure 3, which use the semantics of sections 4.3 and 4.4 respectively. We assume $k \notin fn(M) \cup fn(N)$. In those descriptions, we see that the message M that the system MACs differs from the message $M + N$ that it forwards on c.

There are several ways to address extension attacks, and indeed the literature contains many MAC constructions that are not subject to these attacks. We have considered some of them. Here we describe a construction that uses the MAC key twice:

$$\mathsf{mac}(x, y) \stackrel{\text{def}}{=} \mathsf{f}(x, \mathsf{f}(x, y))$$

Under this definition, the process S forwards on c only a message that it has previously MACed, as desired. Looking beyond the case of S, we can prove a more general result by comparing the situation where mac is primitive (and has no special equations) and one with the definition of $\mathsf{mac}(x, y)$ as $\mathsf{f}(x, \mathsf{f}(x, y))$. Given a tuple of names $\widetilde{k}$ and an extended process C that uses the symbol mac, we write $[\![C]\!]$ for the translation of C in which the definition of mac is expanded wherever a key k_i in $\widetilde{k}$ is used, with $\mathsf{f}(k_i, \mathsf{f}(k_i, M))$ replaced for $\mathsf{mac}(k_i, M)$. The theorem says that this translation yields an equivalent process (so, intuitively, the constructed MACs work as well as the primitive ones):

Theorem 4 *Suppose that the names $\widetilde{k}$ appear only as MAC keys in C. Take no equations for* mac *and the equation* $\mathsf{f}(x, y + z) = \mathsf{h}(\mathsf{f}(x, y), z)$ *for* f. *Then* $\nu\widetilde{k}.C \approx \nu\widetilde{k}.[\![C]\!]$.

7 Related work and conclusions

In this paper, we describe a uniform extension of the pi calculus, the applied pi calculus, in which messages may be compound values, not just channel names. We study its theory, developing its semantics and proof techniques. Although the calculus has no special, built-in features to deal with security, we find it useful in the analysis of security protocols.

Other techniques have been employed for the analysis of these protocols. Some are based on complexity theory; there, principals are basically Turing machines that compute on bitstrings, and security depends on the computational limitations of attackers (e.g., [44, 23, 24, 8, 22]). Others rely on higher-level, formal representations where issues of computational complexity can be conveniently avoided (e.g., [19, 25, 29, 39, 35, 34, 42, 5, 16, 7]). Although some recent work [28, 36, 6] starts to relate these two schools (for example, justifying the soundness of the second with respect to the first), they remain rather distinct. Our use of the applied pi calculus clearly belongs in the second. Within this school, many recent approaches work essentially by reasoning about all possible traces of a security protocol. However, the ways of talking about the traces and their properties vary greatly. We use a process calculus. Its semantics provides a detailed specification for sets of traces. Because the process calculus has a proper "new" construct (like the pi calculus but unlike CSP), it provides a direct account of the generation of new keys and other fresh quantities. It also enables reasoning with equivalence and implementation relations. Furthermore, the process calculus permits treating security protocols as programs written in a programming notation—subject to typing, to other static analyses, and to translations [1, 3, 4, 2, 10, 11, 9, 13].

The applied pi calculus has many commonalities with the original pi calculus and its relatives, such as the spi calculus (discussed above). In particular, the model of communication adopted in the applied pi calculus is deliberately classical: communication is through named channels, and value computation is rather separate from communication. Further, active substitutions are reminiscent of the constraints of the fusion calculus [43]. They are especially close to the substitution environments that Boreale et al. employ in their proof techniques for a variant of the spi calculus with a symmetric cryptosystem [12]; we incorporate substitutions into processes, systematize them, and generalize from symmetric cryptosystems to arbitrary operations and equations.

Famously, the pi calculus is the language of those lively social occasions where all conversations are exchanges of names. The applied pi calculus opens the possibility of more substantial, structured conversations; the cryptic character of some of these conversations can only add to their charm and to their tedium.

Acknowledgements We thank Rocco De Nicola, Andy Gordon, Tony Hoare, and Phil Rogaway for discussions that contributed to this work. Georges Gonthier and Jan Jürjens suggested improvements in its presentation.

References

[1] Martín Abadi. Protection in programming-language translations. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 868–883. Springer, July 1998. Also Digital Equipment Corporation Systems Research Center report No. 154, April 1998.

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

[3] Martín Abadi, Cédric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In *Proceedings of the Thirteenth Annual IEEE Symposium on Logic in Computer Science*, pages 105–116, June 1998.

[4] Martín Abadi, Cédric Fournet, and Georges Gonthier. Authentication primitives and their compilation. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 302–315, January 2000.

[5] Martín Abadi and Andrew D. Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999. An extended version appeared as Digital Equipment Corporation Systems Research Center report No. 149, January 1998.

[6] Martín Abadi and Phillip Rogaway. Reconciling two views of cryptography (The computational soundness of formal encryption). In *Proceedings of the First IFIP International Conference on Theoretical Computer Science*, volume 1872 of *Lecture Notes in Computer Science*, pages 3–22. Springer-Verlag, August 2000.

[7] Roberto M. Amadio and Denis Lugiez. On the reachability problem in cryptographic protocols. In Catuscia Palamidessi, editor, *CONCUR 2000: Concurrency Theory (11th International Conference)*, volume 1877 of *Lecture Notes in Computer Science*, pages 380–394. Springer-Verlag, August 2000.

[8] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In *Advances in Cryptology—CRYPTO '94*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249. Springer-Verlag, 1993.

[9] Chiara Bodei. *Security Issues in Process Calculi*. PhD thesis, Università di Pisa, January 2000.

[10] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Control flow analysis for the pi-calculus. In Davide Sangiorgi and Robert de Simone, editors, *CONCUR '98: Concurrency Theory (9th International Conference)*, volume 1466 of *Lecture Notes in Computer Science*, pages 84–98. Springer, September 1998.

[11] Chiara Bodei, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static analysis of processes for no read-up and no write-down. In Wolfgang Thomas, editor, *Proceedings of the Second International Conference on Foundations of Software Science and Computation Structures (FoSSaCS '99)*, volume 1578 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 1999.

[12] Michele Boreale, Rocco De Nicola, and Rosario Pugliese. Proof techniques for cryptographic processes. In *Proceedings of the Fourteenth Annual IEEE Symposium on Logic in Computer Science*, pages 157–166, July 1999.

[13] Luca Cardelli. Mobility and security. In F. L. Bauer and R. Steinbrueggen, editors, *Foundations of Secure Computation*, NATO Science Series, pages 3–37. IOS Press, 2000.

[14] Sylvain Conchon and Fabrice Le Fessant. Jocaml: Mobile agents for Objective-Caml. In *First International Symposium on Agent Systems and Applications (ASA'99)/Third International Symposium on Mobile Agents (MA'99)*, pages 22–29, October 1999.

[15] Core SDI S.A. ssh insertion attack. Available at `http://www.core-sdi.com/soft/ssh/attack.txt`, July 1998.

[16] Mads Dam. Proving trust in systems of second-order processes. In *Proceedings of the 31th Hawaii International Conference on System Sciences*, volume VII, pages 255–264, 1998.

[17] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, IT-22(6):644–654, November 1976.

[18] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2:107–125, 1992.

[19] Danny Dolev and Andrew C. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, IT-29(12):198–208, March 1983.

[20] Cédric Fournet and Georges Gonthier. A hierarchy of equivalences for asynchronous calculi. In Kim G. Larsen, Sven Skyum, and Glynn Winskel, editors, *Proceedings of the 25th International Colloquium on Automata, Languages and Programming*, volume 1443 of

Lecture Notes in Computer Science, pages 844–855. Springer, July 1998.

[21] Alan O. Freier, Philip Karlton, and Paul C. Kocher. The SSL protocol: Version 3.0. Available at `http://home.netscape.com/eng/ssl3/draft302.txt`, November 1996.

[22] Shafi Goldwasser and Mihir Bellare. Lecture notes on cryptography. Summer Course "Cryptography and Computer Security" at MIT, 1996–1999, August 1999.

[23] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28:270–299, April 1984.

[24] Shafi Goldwasser, Silvio Micali, and Ronald Rivest. A digital signature scheme secure against adaptive chosen-message attack. *SIAM Journal on Computing*, 17:281–308, 1988.

[25] R. Kemmerer, C. Meadows, and J. Millen. Three system for cryptographic protocol analysis. *Journal of Cryptology*, 7(2):79–130, Spring 1994.

[26] Hugo Krawczyk. SKEME: A versatile secure key exchange mechanism for internet. In *Proceedings of the Internet Society Symposium on Network and Distributed Systems Security*, February 1996. Available at `http://bilbo.isu.edu/sndss/sndss96.html`.

[27] Ben Liblit and Alexander Aiken. Type systems for distributed data structures. In *Proceedings of the 27th ACM Symposium on Principles of Programming Languages*, pages 199–213, January 2000.

[28] P. Lincoln, J. Mitchell, M. Mitchell, and A. Scedrov. A probabilistic poly-time framework for protocol analysis. In *Proceedings of the Fifth ACM Conference on Computer and Communications Security*, pages 112–121, 1998.

[29] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Verlag, 1996.

[30] Alfred J. Menezes, Paul C. van Oorschot, and Scott A. Vanstone. *Handbook of Applied Cryptography*. CRC Press, 1996.

[31] Robin Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

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

[33] John C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[34] John C. Mitchell, Mark Mitchell, and Ulrich Stern. Automated analysis of cryptographic protocols using Murϕ. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 141–151, 1997.

[35] Lawrence C. Paulson. The inductive approach to verifying cryptographic protocols. *Journal of Computer Security*, 6(1–2):85–128, 1998.

[36] Birgit Pfitzmann, Matthias Schunter, and Michael Waidner. Cryptographic security of reactive systems (extended abstract). *Electronic Notes in Theoretical Computer Science*, 32, April 2000.

[37] Benjamin C. Pierce and David N. Turner. Pict: A programming language based on the pi-calculus. In Gordon Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language and Interaction: Essays in Honour of Robin Milner*, Foundations of Computing. MIT Press, May 2000.

[38] D. Sangiorgi. On the bisimulation proof method. *Journal of Mathematical Structures in Computer Science*, 8:447–479, 1998.

[39] Steve Schneider. Security properties and CSP. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 174–187, 1996.

[40] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C*. John Wiley & Sons, Inc., second edition, 1996.

[41] Stuart G. Stubblebine and Virgil D. Gligor. On message integrity in cryptographic protocols. In *Proceedings of the 1992 IEEE Symposium on Research in Security and Privacy*, pages 85–104, 1992.

[42] F. Javier Thayer Fábrega, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Why is a security protocol correct? In *Proceedings of the 1998 IEEE Symposium on Security and Privacy*, pages 160–171, May 1998.

[43] Björn Victor. *The Fusion Calculus: Expressiveness and Symmetry in Mobile Processes*. PhD thesis, Dept. of Computer Systems, Uppsala University, Sweden, June 1998.

[44] Andrew C. Yao. Theory and applications of trapdoor functions. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science (FOCS 82)*, pages 80–91, 1982.