Cryptographic Types
(Extended Abstract)

Dominic Duggan*

Abstract

Networks secured by cryptographic techniques can be likened to programming languages where all type-checking is performed at run-time: cryptographic operations such as signature checking and decryption are analogous to run-time dynamic type-checking. Languages such as Java allow a combination of static and dynamic type-checking in distributed programming. Cryptographic types are a way to express cryptographic guarantees (of secrecy and integrity) in a type system for a network programming language. This allows some of these guarantees to be checked statically, before a network program executes. Where dynamic checks are required, these are represented at the source language level as dynamic checking, and are translated by the compiler to lower level cryptographic operations for encryption/decryption and signing/authentication. Static checking avoids the unnecessary overhead of run-time cryptographic operations, and also provides static guarantees of the reliability of a network application. Cryptographic types are described formally using a process calculus, the ec-calculus, allowing the correctness to be verified for a scheme for compiling type operations to cryptographic operations.

1 Introduction

Programming of wide-area networks and the Internet has gained a great deal of interest with the success of the Web. WANs and the Internet introduce or emphasize new issues that receive less priority in programming local area networks. One of these issues is the fact that communication is typically over untrusted networks, and so a security infrastructure is emerging to protect computations over such networks. Cryptography, for protection and for integrity, plays an important rôle in this infrastructure, potentially at various levels in the network protocol stack.

A difficulty with the ubiquitous application of cryptography in network communication is the high runtime overhead of running hashing, encryption and signing algorithms over network traffic. In many cases such a ubiquitous application is unnecessary. Wide-area networks, including the Internet, are not a single flat homogeneous

*Department of Computer Science, Stevens Institute of Technology, Hoboken, New Jersey 07040, USA. E-mail: dduggan@cs.stevens-tech.edu.

namespace, with every host given the same level of trust. Rather, networks are organized into administrative domains, with the boundaries of such domains delineated by firewalls [11]. Such firewalls may be represented by physical machines, or they may be represented by virtual private networks (VPNs) implemented using network-layer authentication techniques such as IPsec.

Different levels of trust may be assigned to different parts of the network, based on the application that is using the network and its relationship to those parts of the network. Blindly applying cryptography to secure all traffic is appropriate if no part of the network is trusted, but potentially incurs unnecessary overhead for traffic that traverses trusted administrative domains. On the other hand, if some communication is to be transmitted in the clear over some domains, it is desirable to have some static guarantee that communication is not accidentally transmitted in cleartext over untrusted domains.

Cryptographic types are a way to replace the ubiquitous use of cryptographic techniques with a combination of static and dynamic techniques for secure communication:

1. Values have types that include *encryption types* and *signature types*. The former denotes a value that has been encrypted, and the type is annotated with the principal whose public key was used for encryption. The latter denotes a value that has been digitally signed, and the type is annotated with the principal whose private key was used for signing.

2. Computations execute under the authorization of a *principal*. A computation executing under the principal a may sign messages with a's key, and may decrypt messages encrypted with a's public key. These provide the static guarantees of integrity and safety for messages that may not actually be signed or encrypted, as long as the message are sent over trusted networks.

3. In the case where a message is sent over an untrusted network, this is reflected by a channel type that masks the identities of principals in cryptographic types in the channel message type. Cryptographic types include an (unchecked) *type widening* rule that allows such masking be done via subtyping. There is a corresponding (checked) *type narrowing* rule that exposes principal identities that have been masked by widening. The check in the narrowing rule is realized concretely by decryption and signature verification.

To reason about security properties of programs making use of cryptographic operations, we use techniques originally developed for reasoning about concurrent programming languages. We cast our approach in the framework of process algebras: we give a small process algebra (an extension of the asynchronous π-calculus [25, 23, 7, 8, 18, 32]) that represents a small kernel programming language with a cryptographic type system, and we use process equivalence techniques to reason about the preservation of secrecy and integrity properties of programs in this language [6, 2, 4, 3, 5].

In the next section we describe the ec-calculus, an extension of the asynchronous π-calculus with cryptographic types. In Sect. 3 we consider the operational semantics of the ec-calculus. Sect. 4 of this extended abstract considers related work and provides conclusions.

2 Cryptographic Types

The π-calculus is a small kernel concurrent programming language, expressive enough to describe distributed programming paradigms such as client-server applications with private reply channels, yet simple enough to support powerful reasoning about process equivalence. These reasoning techniques have in turn been used to verify secrecy and integrity properties of programs in the spi-calculus, an extension of the π-calculus with cryptographic primitives for encrypting and signing messages.

In this section we describe the ec-calculus, a process calculus that extends the π-calculus with cryptographic types. Unlike the π-calculus, processes are organized into *principals*. A process P acting on behalf of the principal a is represented by the process $a[P]$. This gives a two-level syntax, of processes (which may be multi-threaded) and principals (of which there may be several, communicating during their execution).

Unlike cryptographic extensions of process calculi such as the spi-calculus and the sjoin-calculus, keys and key pairs are not explicit in the ec-calculus. The private key of a principal is implicitly available for signing and decryption to code executing within that principal. The public key of a principal is implicitly available for authentication and encryption to any process that "knows" of that principal. "Knowing" just means that that process has obtained the principal's name; principal names are ordinary data values.

The syntax of processes in the ec-calculus is given by:

$p \in$ Param	$::=$	x	Variable
	$\mid$	n	Name
$P \in$ Procs	$::=$	stop	Stopped Process
	$\mid$	$\mathit{receive}\ x\ \mathit{from}\ n\ \mathit{in}\ P$	Message Receive
	$\mid$	$\mathit{send}\ V\ \mathit{on}\ p$	Message Send
	$\mid$	$\mathit{new}\ n : A\ \mathit{in}\ P$	Scoped Name
	$\mid$	$\mathit{repeat}\ P$	Replication
	$\mid$	$(P_1 \mid P_2)$	Parallel Comp
	$\mid$	$\mathit{let}\ \langle x_1, \ldots, x_k \rangle = V\ \mathit{in}\ P$	Access Tuple
	$\mid$	$\mathit{if}\ V_1 = V_2\ \mathit{then}\ P_1\ \mathit{else}\ P_2$	Equality Test
	$\mid$	$\mathit{access}\ V\ \mathit{as}\ \rho\ \mathit{in}\ P$	Access Packet
	$\mid$	$\mathit{hide}\ V\ \mathit{as}\ x : B\ \mathit{in}\ P$	Hide Principals
	$\mid$	$\mathit{expose}\ V\ \mathit{as}\ x : B\ \mathit{in}\ P$	Expose Principals
$V \in$ Value	$::=$	p	Parameter

$$
\begin{array}{rcll}
 & \mid & \langle V_1, \ldots, V_k \rangle & \text{Tuple} \\
 & \mid & packet_B(\overline{p}, V) & \text{Packet} \\
\rho \in \text{Pattern} & ::= & x & \text{Variable Pattern} \\
 & \mid & \_ & \text{Wildcard Pattern} \\
 & \mid & \langle \rho_1, \ldots, \rho_k \rangle & \text{Tuple Pattern}
\end{array}
$$

As usual in such calculi, processes are simple "assembly language" concurrent programs, with operations for asynchronous message-passing. There is also an operation for generating new channels, for replicating processes (this can be used to define recursive process descriptions) and for forming the parallel composition of processes. Data structures are built up in the language using tuples, hence there is a construct for decomposing a tuple. There is also an equality test operation.

This calculus extends the π-calculus with a new type of value, a *packet*. There are three operations associated with packets: an *access* operation that allows authorized access to parts of a packet, a *hide* operation for hiding some of the principals under whose keys some of the packet payload has been encrypted or signed, and an *expose* operation for revealing principals whose identities have been hidden. The access operation is guaranteed to succeed by the type system. Types prevent a process from even attempting to access a field of a packet, unless it has obtained the access right. This restriction is enforced in the type rules for patterns; the process can only match with a pattern that gives it access to the authorized fields.

The second level of the process syntax is the description of networks. A network is the parallel composition of processes acting on behalf of principals. Network descriptions also allow channel and principal names to be "extruded" outside of process descriptions, so that they can be communicated between processes representing different principals. The syntax of networks is given by:

$$
\begin{array}{rcll}
N \in \text{Network} & ::= & stop & \text{Empty Network} \\
 & \mid & a[P] & \text{Located Process} \\
 & \mid & Q & \text{Primitive Process} \\
 & \mid & new\ n : A\ in\ N & \text{Scoped Name} \\
 & \mid & (N_1 \mid N_2) & \text{Wire}
\end{array}
$$

Typed processes execute under a particular principal n, hence at the top level typed processes are located at principals $n[P]$. In addition we allow primitive processes Q at the top level. A primitive process is a process in the spi-calculus, a version of the pi-calculus without cryptographic types but extended with cryptographic primitives. Primitive processes represent potential attackers in our calculus; we make them part of the network descriptions in order to model attacks on typed systems from primitive adversaries.

The form of types is given by:

$$
\begin{array}{rcll}
A \in \text{Name Type} & ::= & \mathit{Chan}[B] & \text{Channel Type} \\
 & | & \mathit{Prin} & \text{Principal Type} \\
B \in \text{Type} & ::= & A & \text{Name Type} \\
 & | & \mathit{Packet}[\overline{x}.T] & \text{Packet Type} \\
 & | & \langle B_1, \ldots, B_k \rangle & \text{Tuple Type} \\
T \in \text{Crypto Type} & ::= & \mathit{Prin} & \text{Principal} \\
 & | & \mathit{Chan}[B] & \text{Channel Type} \\
 & | & \mathcal{E}\{T\}_p & \text{Encrypted Type} \\
 & | & \mathcal{S}\{T\}_p & \text{Signed Type} \\
 & | & \langle T_1, \ldots, T_k \rangle & \text{Tuple Type}
\end{array}
$$

Name types are the types to which new names can be bound, within untrusted processes. Besides name types, we also have packet types and tuple types. Cryptographic types allow the form of packet payload types to specify access and control information and author information. There are two forms of such types:

1. $\mathcal{E}\{T\}_a$ denotes a value encrypted with the public key of the principal a. Only a process executing for that process can access that value.

2. $\mathcal{S}\{T\}_a$ denotes a value signed with the secret key of the principal a. Any process that knows of a can authenticate this signature.

As the syntax suggests, cryptographic types can be nested inside of other cryptographic types. An advantage of exposing the identity of the principals in types, is that access control and integrity can be performed statically, without heavyweight cryptographic operations. These operations are only invoked when static information about principals must be elided, as packets are transmitted over an insecure network. This amounts to a form of existential quantification that elides the identities of some of the principals in a packet type.

Using cryptographic types, we can provide a secure channel of type $\mathit{Chan}[\mathit{Req}]$ that the principal a can use to make requests to the principal b:

$$
\begin{array}{rcl}
\mathit{Req} & = & \mathit{Packet}[\mathcal{S}\{\langle \mathcal{E}\{\tau_{\text{req}}\}_b, \mathit{Chan}[\mathit{Reply}]\rangle\}_a] \\
\mathit{Reply} & = & \mathit{Packet}[\mathcal{E}\{\mathcal{S}\{\tau_{\text{reply}}\}_b\}_a]
\end{array}
$$

A request message is a signed pair, consisting of a request datum and a reply channel. The request payload (of type τ_{req}) is encrypted using the server's public key (the reply channel is transmitted unencrypted for no apparent reason), and the entire message is signed using the client's private key. The reply channel carries messages of payload type τ_{reply}, signed using the server's private key[1] and encrypted using the client's public key.

[1] This does not necessarily restrict us to cryptographic algorithms (such as RSA) where the same keys can be used for both encryption and authentication. For example a public key in our calculus could be realized in a concrete implementation by a pair of keys, one for encryption and one for signature verification.

Suppose we have a packet of the reply type. If the current thread has a as its principal, and it knows about b, then it can access the payload. No authentication or decryption is required for this access; the checking can be done statically. For example, if a and b are in the same administrative domain, then they may establish a secure reply channel[2] $n : \mathit{Chan}[\mathit{Reply}]$. Then a reply can be transmitted securely without encryption:

$$b[\mathit{send\ packet}_{\mathit{Packet}[\mathit{Reply}]}((), V)\ \mathit{on}\ n] \mid a[\mathit{receive}\ x\ \mathit{from}\ n\ \mathit{in}\ P]$$
$$\longrightarrow a[\{\mathit{packet}_{\mathit{Packet}[\mathit{Reply}]}((), V)/x\}P]$$

Suppose on the other hand that it is not possible to establish a secure channel between a and b. The insecurity of a channel is given by the fact that the channel type does not mention one or both of the principals. This is possible because a packet type may encapsulate some of the principals in the payload type. In general a packet type in the ec-calculus has the form

$$\mathit{Packet}[\overline{x}.\sigma]$$

The variables $x_1, \ldots, x_k$ are bound by the packet type, and encapsulate some of the principals in the packet payload type σ. The full form of a packet is

$$\mathit{packet}_{\mathit{Packet}[\overline{x}.T]}(\overline{a}, V)$$

where V has type $\{\overline{a}/\overline{x}\}T$.

Manipulating packets of this form requires operations for hiding principals in packet types, and exposing them. The former operation is unchecked; it only requires access to public keys for encryption and private keys for signing (by the current principal). On the other hand exposure is a checked operation; it requires private keys for decryption and public keys for authentication. For example, if:

$$\mathit{PrivateReply} \quad \equiv \quad \mathit{Packet}[(x,y).\mathcal{E}\{\mathcal{S}\{\tau_{\text{reply}}\}_y\}_x]$$

and $n : \mathit{Chan}[\mathit{PrivateReply}]$ then:

$$b[\mathit{hide\ packet}_{\mathit{Reply}}((), V)\ \mathit{as}\ x : \mathit{PrivateReply}\ \mathit{in\ send}\ x\ \mathit{on}\ n]$$
$$\longrightarrow b[\mathit{send\ packet}_{\mathit{PrivateReply}}((a,b), V)\ \mathit{on}\ n]$$

The hiding of the signing key b is possible because the hiding is done by a process executing for b. This hiding operation is the coercion from a more specific type to a less specific type arising from upcasting, in the example in the introduction (there the upcast was from type `Request`$\langle$`a,thisPrin`$\rangle$ to `Request`).

On the receiving site, the principals must be re-exposed before the contents of the packet can be accessed:

$$b[\mathit{send\ packet}_{\mathit{PrivateReply}}((a,b), V)\ \mathit{on}\ n]$$
$$\mid\ a[\mathit{receive}\ y\ \mathit{from}\ n\ \mathit{in\ expose}\ y\ \mathit{as}\ x : \mathit{Reply}\ \text{in}\ \ldots x \ldots]$$
$$\longrightarrow a[\mathit{expose\ packet}_{\mathit{PrivateReply}}((a,b), V)\ \mathit{as}\ x : \mathit{Reply}\ \text{in}\ \ldots x \ldots]$$
$$\longrightarrow a[\ldots \mathit{packet}_{\mathit{Reply}}((), V) \ldots]$$

In this case the exposure of the encrypting key a is possible because the exposure is done by a process executing for a.

[2]Our oversimplified network model allows secure channels to be established between any two principals. This oversimplification can be fixed with a more complicated network model, along the lines of the ambient and SEAL calculi [13, 14].

3 Operational Semantics for The EC-Calculus

There are two semantics that can be given for the ec-calculus. We can give a direct semantics for the ec-calculus itself, and verify subject reduction: well-typedness is preserved by process execution. Alternatively we can give a translation semantics for the ec-calculus, by providing a translation from it into the spi-calculus, an extension of the pi-calculus with primitive cryptographic operations for signing and encrypting messages [6]. This provides a compilation strategy for the ec-calculus. In either case we need to consider a translation of the ec-calculus types and values into the spi-calculus. This is because ec-calculus processes may communicate with spi-calculus processes, allowing us to reason about security properties of the ec-calculus, that may be composed with attacker processes coded in the spi-calculus.

Processes in the spi-calculus and processes in the ec-calculus need to exchange values, we assume over channels typed in the ec-calculus. We use the following translation of ec-calculus types to spi-calculus types:

$$
\begin{array}{rcl}
[\![Chan[B]]\!] & = & Chan[[\![B]\!]] \\
[\![Prin]\!] & = & Public \\
[\![Packet[\overline{x}.T]]\!] & = & [\![T]\!]\overline{x} \\
[\![\langle B_1,\ldots,B_k\rangle]\!] & = & \langle [\![B_1]\!],\ldots,[\![B_k]\!]\rangle \\
[\![Chan[B]]\!]\overline{x} & = & Chan[[\![B]\!]] \\
[\![Prin]\!]\overline{x} & = & Public \\
[\![\mathcal{E}\{T\}_p]\!]\overline{x} & = & \left\{ \begin{array}{ll} \mathcal{E}\{[\![T]\!]\overline{x}\} & \text{if } p \in \{\overline{x}\} \\ [\![T]\!]\overline{x} & \text{otherwise} \end{array} \right. \\
[\![\mathcal{S}\{T\}_p]\!]\overline{x} & = & \left\{ \begin{array}{ll} \mathcal{S}\{[\![T]\!]\overline{x}\} & \text{if } p \in \{\overline{x}\} \\ [\![T]\!]\overline{x} & \text{otherwise} \end{array} \right. \\
[\![\langle T_1,\ldots,T_k\rangle]\!]\overline{x} & = & \langle [\![T_1]\!]\overline{x},\ldots,[\![T_k]\!]\overline{x}\rangle
\end{array}
$$

Principals in the ec-calculus are translated as public keys in the spi-calculus. This means that keys cannot leak in the ec-calculus, since processes can only access public keys as data values, while the use of private keys is implicit in type widening and narrowing. Packets are erased entirely; the principals encapsulated in a packet are used in the translation of encryption and signature types. If a value has an encryption type in the ec-calculus, and the corresponding principal is encapsulated by a packet containing that value, then the value is translated as an encrypted value, otherwise the value is not encrypted in the translation. Similarly if a value has a signature type in the ec-calculus, and the corresponding principal is encapsulated by a packet containing that value, then the value is translated as a signed value, otherwise the value is not digitally signed in the translation.

For example the value in the ec-calculus:

$$packet_B((a,b,c),\langle 3,4\rangle) \in Packet[(x,y,z).\mathcal{S}\{\langle \mathcal{E}\{int\}_x,\mathcal{E}\{int\}_y\rangle\}_z]$$

is translated into the spi-calculus as the value:

$$\mathcal{S}\{\langle \mathcal{E}\{3\}_{public(a)},\mathcal{E}\{4\}_{public(b)}\rangle\}_c \in \mathcal{S}\{\langle \mathcal{E}\{int\},\mathcal{E}\{int\}\rangle\}.$$

This translation is only conceptual; there is no run-time change of data representation implied by it, and in fact we would expect that ec-calculus programs would be compiled to spi-calculus programs, so that ec-calculus values would be represented at run-time by spi-calculus values.

The following theorem verifies that process execution preserves well-typedness:

Theorem 1 (Subject Reduction) *Suppose Γ contains only bindings of names to name types. Suppose $\Gamma \vdash N$ **net** and $\Gamma \vdash N \longrightarrow N'$. Then $\Gamma \vdash N'$ **net**.*

The translation from the ec-calculus to the spi-calculus is defined by induction on type derivations. The details are provided in the full paper [17]. In the full paper, a notion of typed process equivalence is defined for processes ($\Gamma \vdash N_1 \approx_{be} N_2$ and $\Gamma \vdash Q_1 \approx_{be} Q_2$), and the following is verified:

Theorem 2 (Full Abstraction) *Given $fv(N_1) \cup fv(N_2) \subseteq \{\}$. Given $\Gamma \vdash N_1$ **net** and $\Gamma \vdash N_2$ **net**, and $\Gamma \vdash N_1$ **net**$\Longrightarrow Q_1$ and $\Gamma \vdash N_2$ **net**$\Longrightarrow Q_2$. Then*

$$\Gamma \vdash N_1 \approx_{be} N_2 \Leftrightarrow \Gamma \vdash Q_1 \approx_{be} Q_2.$$

4 Conclusion

The motivation for this work has been the need for proper programming abstractions for developing applications that span wide-area networks and the Internet. Almost all of the work in the area of wide-area languages has focused on security, for example, providing abstractions of secure channels [5, 4, 3], controlling key distribution [14, 12], reasoning about security protocols [6, 1], tracking untrustworthy hosts in the system [22, 31], etc.

Abadi [1] considers a type system for ensuring that secrecy is preserved in security protocols implemented in that type system. For securing communication over untrusted networks he includes a "universal" type `Un` inhabited by encrypted values. Our work separates the type-checking aspects of network communication (performed by unmarshalling, implicit in our work) from the security-checking aspects of network communication (performed by cryptographic operations), whereas both are combined in Abadi's approach. Our type system tracks security aspects of communicated values even after decrypting and signature-checking, through principals in cryptographic types[3]. Whereas Abadi's approach only allows one to consider completely trusted and completely untrusted networks, our approach allows us to consider intermediate levels of trust (e.g. "trusted to contain values produced by a and accessible by b"). As such it provides a refined notion of security via types, analogous to refinement types [19] and refinement kinds [16] for more traditional forms of type systems.

Our work is most closely related to work on compiling high-level programs to run over untrusted networks, where the compiler generates code for securing channels in this environment [5]. The different is the level at which programming abstractions are provided. For example, in Java, distributed communication can be secured at the transport level, by installing a secure sockets layer, or alternatively the application

[3]This is the motivation for our use of dependent types.

itself can take responsibility for security using the `SignedObject` and `SealedObject` classes. The work of Abadi et al takes the former approach (moving some of the security protocol out of the OS protocol stack and into the compiler) whereas this paper takes the latter approach. An advantage of our approach is that the correctness guarantees are weaker and therefore easier to attain; for example we do not need to broadcast network noise to prevent traffic analysis. A disadvantage of our approach is that the correctness guarantees are weaker, and therefore more responsibility is placed on the application to secure communications. This reflects a difference of philosophy between the two approaches.

Sumii and Pierce [34] describe the cryptographic lambda calculus, an extension of the typed lambda calculus with symmetric cryptographic operations along the lines of the spi-calculus. They then use reasoning techniques for the polymorphic lambda calculus (in particular, relational parametricity) to verify properties of programs implemented using cryptographic techniques. They cite as further work the implementation of a polymorphic lambda calculus that could then be translated into the cryptographic lambda calculus. Such a translation is provided in a related unpublished manuscript, but it is not clear how to extend reasoning based on relational parametricity to this approach. In contrast to such a translation, our technique does not provide a new programming model *per se*, but provides a different approach to encrypting and signing values that allows trust to be checked using a combination of static and dynamic techniques. Because of the latter dynamic checks, our approach does not have a parametricity property.

Gordon and Jeffrey [20, 21] have developed a new type-based approach to verifying authentication protocols: first, code the protocol in the spi-calculus, second specify authenticity properties using correspondence assertions [36, 37], third figure out the types of keys, nonces and messages, and fourth use a type and effect system to check that the protocol is well-typed. Gordon and Jeffrey pursue the same approach as Abadi for encrypting values for transmission over an untrusted network. They also develop a dependent type system, but in their case the intention is to maintain a connection in the type system between messages and nonces, while nonce types and an effect type system use correspondence assertions to ensure the freshness of communications. It remains to be seen if there can be useful synergy between the approaches, perhaps in incorporating our notion of tracking levels of trustworthiness into the approach of Gordon and Jeffrey, or extending our approach with guarantees of freshness based on their approach. Since their work is based on symmetric key systems while ours is based on asymmetric key systems, such a combination appears both attractive and challenging.

Other work on security in programming languages has focused on ensuring safety properties of untrusted code [29, 28, 26] and preventing unwanted security flows in programs [15, 27, 35, 30, 33]. A major difference between cryptographic types and this work is that the latter is intended for centralized systems, where some form of centralized access control or information flow control is enforced based on static type-checking. Therefore the applications of these approaches are to ensure safety properties of applets and DLLs. In contrast cryptographic types are explicitly concerned with distributed access control, using cryptographic techniques. Our security concerns have largely been with access control, so it is not clear how relevant the work on information flow is to this. Pottier and Conchon [30] have developed an interesting approach to encoding information flows, and claim it offers a way to modularly combine access control and information flow (while keeping the notions of security levels and principals separate). Because of the operational nature of their work, it appears highly plausible that it could be combined with our work. We would also like to combine cryptographic types with trust management techniques [24, 10, 9], to obtain a much richer notion of principals and authentication, and use this to develop a trust model for this approach.

References

[1] Martin Abadi. Security by typing in security protocols. In *Theoretical Aspects of Computer Science*, pages 611–638, 1997.

[2] Martin Abadi, Cedric Fournet, and Georges Gonthier. Secure implementation of channel abstractions. In *Proceedings of IEEE Symposium on Logic in Computer Science*, 1998.

[3] Martin Abadi, Cedric Fournet, and Georges Gonthier. Secure communications processing for distributed languages. In *IEEE Symposium on Security and Privacy*, 1999.

[4] Martin Abadi, Cedric Fournet, and Georges Gonthier. A top-down look at a secure message. In *Foundations of Software Technology and Theoretical Computer Science*, pages 122–141, 1999.

[5] Martin Abadi, Cedric Fournet, and Georges Gonthier. Authentication primitives and their compilation. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 2000.

[6] Martin Abadi and Andrew Gordon. A calculus for cryptographic protocols: The spi calculus. *Information and Computation*, 148(1):1–70, January 1999.

[7] R. M. Amadio. An asynchronous model of locality, failure and process mobility. In *COORDINATION'97*, volume 1282 of *Lecture Notes in Computer Science*. Springer-Verlag, 1997.

[8] R. M. Amadio, L. Castellani, and D. Sangiorgi. On bisimulations for the asynchronous pi-calculus. *Theoretical Computer Science*, 195(2):291–324, 1998.

[9] Andrew Appel and Edward Felten. Proof-carrying authentication. In *ACM Symposium on Security*, 2000.

[10] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, 1996.

[11] Luca Cardelli. Abstractions for mobile computation. In Jan Vitek and Christian Jensen, editors, *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, volume 1603 of *Lecture Notes in Computer Science*. Springer-Verlag, 1999.

[12] Luca Cardelli, Giorgio Ghelli, and Andrew D. Gordon. Secrecy and group creation. In *Concurrency Theory (CONCUR)*. Springer-Verlag, 2000.

[13] Luca Cardelli and Andrew Gordon. Mobile ambients. In Maurice Nivat, editor, *Foundations of Software Science and Computational Structures*, volume 1378 of *Lecture Notes in Computer Science*, pages 140–155. Springer-Verlag, 1998.

[14] Guiseppe Castagna and Jan Vitek. A calculus of secure mobile computations. In *Internet Programming Languages*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[15] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Communications of the ACM*, 1977.

[16] Dominic Duggan. Dynamic typing for distributed programming in polymorphic languages. *ACM Transactions on Programming Languages and Systems*, 21(1):11–45, January 1999.

[17] Dominic Duggan. Cryptographic types. Technical report, Stevens Institute of Technology, 2001.

[18] C. Fournet and G. Gonthier. A hierarchy of equivalences for asynchronous calculi (extended abstract). In *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 844–855. Springer-Verlag, 1998.

[19] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 268–277. ACM Press, 1991.

[20] Andrew D. Gordon and Alan Jeffrey. Authenticity by typing for security protocols. In *IEEE Computer Security Foundations Workshop (CSFW)*, June 2001.

[21] Andrew D. Gordon and Alan Jeffrey. Typing correspondence assertions for communication protocols. In *Mathematical Foundations of Programming Semantics (MFPS)*, May 2001.

[22] Matthew Hennessy and James Riely. Type-safe execution of mobile agents in anonymous networks. In *Secure Internet Programming: Security Issues for Distributed and Mobile Objects*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[23] Kohei Honda and Mario Tokoro. An object calculus for asynchronous communication. In *European Conference on Object-Oriented Programming*, Lecture Notes in Computer Science, pages 133–147. Springer-Verlag, 1991.

[24] Butler Lampson, Michael Burrows, and Edward Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 1992.

[25] Robin Milner. The polyadic π-calculus: A tutorial. In Friedrich L. Bauer, Wilfried Brauer, and Helmut Schwichtenberg, editors, *Logic and Algebra of Specification*, volume 94 of *Computer and Systems Sciences*, pages 203–246. Springer-Verlag, 1993.

[26] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1998.

[27] A. C. Myers and B. Liskov. Complete, safe information flow with decentralized labels. In *IEEE Symposium on Security and Privacy*, 1998.

[28] George Necula. Proof-carrying code. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1997.

[29] George Necula and Peter Lee. Safe kernel extensions without run-time checking. In *Operating Systems Design and Implementation*, 1996.

[30] Francois Pottier and Sylvain Conchon. Information flow inference for free. In *Proceedings of ACM International Conference on Functional Programming*, 2000.

[31] James Riely and Matthew Hennessy. Trust and partial typing in open systems of mobile agents. In *Proceedings of ACM Symposium on Principles of Programming Languages*, 1999.

[32] Davide Sangiorgi. Asynchronous process calculi: The first-order and higher-order paradigms. *Theoretical Computer Science*, 1999.

[33] Geoffrey Smith. A new type system for secure information flow,. In *IEEE Computer Security Foundations Workshop*, pages 115–125, June 2001.

[34] Eijiro Sumii and Benjamin C. Pierce. Logical relations for encryption. In *Computer Security Foundations Workshop*, June 2001.

[35] D. Volpano and G. Smith. A type-based approach to program security. In *Proceedings of the International Joint Conference on Theory and Practice of Software Development*. Springer-Verlag, 1997.

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

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