Static semantics of secret channel abstractions

Marco Giunti

CRACS/INESC-TEC
Universidade do Porto

Abstract. The secret π-calculus extends the π-calculus by adding an *hide* operator that permits to declare channels as secret. The main aim is confidentiality, which is gained by restricting the access of the object of the communication. Communication channels protected by *hide* are more secure since they have static scope and do not allow the context's interaction, and can be implemented as dedicated channels. In this paper, we present static semantics of secret channel abstractions by introducing a type system that considers two type modalities for channels (scope): static and dynamic. We show that secret π-calculus channels protected by *hide* can be represented in the π-calculus by prescribing a static type modality. We illustrate the feasibility of our approach by introducing a security *API* for message-passing communication which works for a standard (π-calculus) middleware while featuring secret channels. Interestingly, we just require the programmer to declare which channels are meant to be secret, leaving the burden of managing the security type abstractions to the API compiler.

1 Introduction

The proliferation of interacting computer devices and the growing complexity of the software components continuously cause security issues; see [21] for the list of vulnerabilities discovered at Google in the last years, which includes the recent Heartbeat OpenSSL bug. Secrecy and confidentiality are major concerns in most systems of communicating agents. Either because some of the agents are untrusted, or because the communication uses insecure channels, there may be the risk of sensitive information being leaked to potentially malicious entities. The price to pay for such security breaches may also be very high. It is not surprising, therefore, that secrecy and confidentiality have become central issues in the formal specification and verification of communicating systems. Formal methods have been indeed advocated as an effective tool to analyze and deploy secure communicating programs and protocols [14]. Process calculi, in particular, allow to study prototypal security analysis techniques that could be embedded into next-generation compilers for distributed languages, and to investigate high-level security abstractions that can be effectively deployed into lower-level languages, thus providing for *APIs* for secure process interaction (e.g. [10]).

The π-calculus and especially its variants enriched with mechanisms to express cryptographic operations, the spi calculus [5] and the applied π-calculus [4],

have become popular formalisms for security applications. They all feature the operator new (restriction) and make crucial use of it in the definition of security protocols. The prominent aspects of new are the capability of creating a new channel name, whose use is restricted within a certain scope, and the possibility of enlarging its scope by communicating it to other processes. The latter property is central to the most interesting feature of the π-calculus: the *mobility* of the communication structure. Although in principle the restriction aspect of new should guarantee that the channel is used for communication within a secure environment only, the capability of extruding the scope leads to security problems. In particular, it makes it unnatural to implement the communication using dedicated channels, and non-dedicated channels are not secure by default. The spi calculus and the applied π-calculus do not assume, indeed, any security guarantee on the channel, and implement security by using cryptographic encryption.

By way of motivation, consider the π-calculus process below, which describes a protocol to exchange a confidential information.

$$P = (\mathsf{new}\, c)((\mathsf{new}\, s)(\overline{c}\langle s\rangle.\overline{s}\langle \mathrm{ATMpwd}\rangle) \mid c(x).x(y).\overline{p}\langle x\rangle) \tag{1}$$

This protocol is composed by two threads communicating over a restricted channel c to exchange a password to access the ATM: the thread on the left generates a (secure) channel s and sends it over c, and later sends the password over s; the thread on the right waits to receive from c a channel (that will instantiate) x, waits on x to receive a value, and subsequently releases x over a public channel p to allow further use of the channel. Still, implementing this solution in untrusted environments is difficult, since we cannot rely on dedicated channels for communication on names created by the new operator. One natural approach to cope with this problem is to map the private communication within the scope of the new into open communications protected by cryptography.

For instance, we may resort to the spi calculus and map the command $(\mathsf{new}\, s)$ into the generation of two cryptographic keys, noted $(\mathsf{new}\, s^+, s^-)$, to be sent over the network through the crypto-packet $\{s^+, s^-\}_{c^+}$. The packet is encrypted with the public key c^+ and can be only open a process that knows the decryption key c^-, that is the (spi calculus representation of the) receiver on the right:

$$net(z).\mathsf{decrypt}\ z\ \mathsf{as}\ \{x^+, x^-\}_{c^-}\ \mathsf{in}\ net(w).\mathsf{decrypt}\ w\ \mathsf{as}\ \{y\}_{x^-}\ \mathsf{in}\ \overline{p}\langle x^+, x^-\rangle$$

The aim is to protect the exchange of the confidential information by encrypting the password with the cryptographic key s^+, noted $\{\mathrm{ATMpwd}\}_{s^+}$. Unfortunately, the naive protocol above suffers from a number of problems, among which the most serious is the lack of forward secrecy [1]: the content of the crypto-packet encrypted with s^+ can be retrieved by a spi calculus context that first buffers the encrypted message and later receives the decryption key s^-.

Stated differently, the spi calculus protocol above does not preserve the semantics of P, which can be formalized by means of the behavioural equivalence in (2), where we assume p free in P:

$$P \cong (\mathsf{new}\, s)(\overline{p}\langle s\rangle) \tag{2}$$

This equation establishes a well-known fact, that is that in the π-calculus communication on channels restricted with new is invisible: this is clearly false for the naive spi calculus protocol above, as the context can retrieve the content of the secret exchange. While a solution to recover the behavioral theory of π-calculus is available [10], the price to pay is a complex cryptographic protocol that relies on a set of trusted authorities acting as proxies.

Based on these considerations, in [19] we argue that the restriction operator of π-calculus does not adequately ensure confidentiality, and we introduce an operator to program explicitly secret communications, called hide. The operator is static: that is, we assume that the scope of hidden channels can not be extruded. The motivation is that all processes using a private channel shall be included in the scope of its hide declaration; processes outside the scope represent another location, and must not interfere with the protocol. Since the hide cannot extrude the scope of secret channels, we can use it to directly build specifications that preserves forward secrecy. In contrast, we regard the restriction operator of the π-calculus, new, as useful to create a new channel for message passing with scope extrusion, and which does not provide secrecy guarantees. Still, this approach assumes specialized semantics for communicating processes, that is: to enforce static scope for secret channels, we rely on a special-purpose middleware that checks the content of each exchange and only allows interactions that do not cause a security break.

In this paper, we show that the static scope mechanism can be enforced in a π-calculus enjoying standard semantics (cf. [24]) by using strong static typing. The hide security operator is accessible as a macro of an idealized *API* for secure programming: programs using secret channels are transformed into typed π-calculus processes and checked: when type-checking succeeds, the scope of channels protected by hide cannot be enlarged during the computation. The API language inherits the the syntax of [19] and allows to patch the protocol P in (1) by re-programming the secure channel with hide: note that the scope delimited by the square parentheses crucially includes the receiver, otherwise the protocol would be rejected.

$$H = (\mathsf{new}\, c)([\mathsf{hide}\, s][H']) \qquad H' = \overline{c}\langle s\rangle.\overline{s}\langle \mathrm{ATMpwd}\rangle \mid c(x).x(y).\overline{p}\langle x\rangle \tag{3}$$

User programs are mapped into a typed π-calculus where channel types are *decorated* with modality qualifiers: d for dynamic channels and s for static (secret) channels. The hide declaration in (3) is compiled into a restriction decorated with a static type, where we guess the channel type of s (cf. [27]), which is $\mathsf{chan}\langle\top\rangle$:

$$[\![[\mathsf{hide}\, s][H']]\!] = (\mathsf{new}\, s\colon \mathsf{s}\,\mathsf{chan}\langle\top\rangle)(H')$$

The encoding of H is obtained similarly by guessing the channel type of c, which is $\mathsf{chan}\langle\mathsf{chan}\langle\top\rangle\rangle$. Note that the compilation assigns a dynamic type to the payload of c: the type system allows to send s having a static type over c by upcasting the type of the payload from dynamic to static.

$$[\![H]\!] = (\mathsf{new}\, c\colon \mathsf{d}\,\mathsf{chan}\langle\mathsf{d}\,\mathsf{chan}\langle\top\rangle\rangle)[\![[\mathsf{hide}\, s][H']]\!] \tag{4}$$

Types and Processes

T	::=	Types:	P	::=	Processes:
	$m\,\mathsf{chan}\langle T\rangle_i$	channel		$x(y \div B).P$	input
	$\top$	top		$\overline{x}\langle v\rangle.P$	output
m	::=	Modalities		$(\mathsf{new}\,x : T)(P)$	restriction
	s	static		$P \mid Q$	composition
	d	dynamic		$\mathbf{0}$	inaction
i	::=	Identifiers		$!P$	replication
	$\forall$	universal	B	::=	Blocked entries:
	n	unique number		$\emptyset$	empty
				$B \cup \{T\}$	type

Fig. 1. π-calculus: syntax

Contribution.

- We introduce a type system to enforce a static scope for π-calculus channels by decorating channel types with *static* and *dynamic* type modalities
- We show that a fragment of the secret π-calculus [19] can be encoded into the typed calculus while preserving an operational correspondence. The compilation requires the type of the payload of channels to be guessed (when possible), while type modalities are inferred automatically. An upcast mechanism allows to send static channels over channels exchanging dynamic variables.
- We discuss possible applications of the proposed technique, which we interpret as an abstract API for secure message-passing.

Structure of the paper. Section 2 introduces the syntax and the static and dynamic semantics of the typed π-calculus. Section 3 reviews the secret π-calculus, and presents a semantics-preserving compilation of the secret π-calculus into the typed π-calculus. In Section 4 we discuss some applications of our technique. We conclude in Section 5 by envisioning future work and by discussing a few related papers.

2 Typed π-calculus

In this section we introduce the syntax of the typed π-calculus, and its static and dynamic semantics. We use x, y, v to range over variables, and n to range over unique numbers (identifiers). The syntax of types in Figure 1 include *channel* types decorated with static (s) and dynamic (d) type modalities, and the *top* type, noted $\top$. The type modalities, ranged by m, are the core of the security mechanism of the typed π-calculus and allow to enforce constraints on the mobility of channels. Type identifiers, ranged by i, are used to identify static types and

to disallow their disclosure by means of structural rearrangement of processes. Variables of type top can be passed around but cannot be used in input/output. We assume that input processes are decorated with a set of *blocked types* and that this set is managed automatically through structural congruence, leaving the details of the mechanism to implementations. For the purpose of the presentation, we make type identifiers explicit and assume two forms for identifiers: *universal*, noted $\forall$, and unique numbers n produced by a clash-free generator, noted $\mathsf{gen}()$; in implementations, type identification would be managed transparently by the compiler.

The syntax of processes is standard, but for input. The input process $x(y \div B).P$ includes a set of (blocked) types B in its definition. When B is the empty set, the input process is the standard one of the π-calculus, otherwise, B contains types T that cannot be received by the input process, for any type assignment of x. Notably, this has impact only on the static semantics, while the dynamic semantics of the language is unaffected. Process $(\mathsf{new}\, y\colon T)(P)$ is the restriction process, and introduces a new variable y of type T with scope in P.

The binders of processes appear in parenthesis: $(\mathsf{new}\, y\colon T)(P)$ and $x(y \div B).P$ bind the free occurrences of y in P. Considering the usual notions of free and bound variables, α-conversion, as well as of substitution, cf. [24], we use $\mathrm{fv}(P)$ and $\mathrm{bv}(P)$ to indicate respectively the set of free and bound variables of P, which we assume disjoint by following Barendregt's variable convention [7]. We will often avoid trailing nils and write $\overline{x}\langle v\rangle$ and $x(y)$ to indicate respectively processes $\overline{x}\langle v\rangle.\mathbf{0}$ and $x(y).\mathbf{0}$, and write $x(y).P$ to indicate the input process $x(y \div \emptyset).P$. Structural congruence is the smallest relation on processes including the rules in Figure 2. We embed the type block mechanism in the rules for structural congruence through the block binary function, noted $\uplus$, defined in the same figure. *Blocked* types could indeed be introduced both statically and dynamically, i.e. when structural congruence is performed during the computation. We leave the time when the system blocks explicitly the type in components as an implementation detail. The axioms of structural congruence are below in Figure 2. Most rules are standard but the axiom in the second line, which deal with restriction of a variable having a static type. The scope of x having type T is enlarged to $Q \uplus T$: all inputs in Q are forbidden to receive values of type T by means of the block function defined above, which instructs the typing system. Note that, due to variable convention, x bound in $(\mathsf{new}\, x\colon T)(P)$, cannot be free in Q, and that an input of the form $x(y \div m\,\mathsf{chan}\langle T\rangle_i).P$ can receive values of type $m\,\mathsf{chan}\langle T\rangle_j$ whenever $i \neq j$. The reduction is the binary relation on processes defined by the rules in Figure 2. The [R-Com] rule communicates variable v from an output prefixed process $\overline{x}\langle v\rangle.P$ to an input prefix $x(y \div B).Q$; the result is the parallel composition of the continuation processes, where, in the input process, the bound variable y is replaced by v. The presence of the blocked types B in the input, as introduced, allows to instruct the type checker and has no impact on communication. The rules in the last line allow reduction to happen underneath scope restriction and parallel composition, and incorporate structural congruence into reduction.

$\boxed{P \uplus T = P}$ *Type blocking*

$$(x(y \div B).P) \uplus T \stackrel{\text{def}}{=} x(y \div B \cup \{T\}).(P \uplus T)$$
$$((\mathsf{new}\, x\colon T)(P)) \uplus T' \stackrel{\text{def}}{=} (\mathsf{new}\, x\colon T)(P \uplus T')$$
$$(\overline{x}\langle v\rangle.P) \uplus T \stackrel{\text{def}}{=} \overline{x}\langle v\rangle.(P \uplus T) \qquad (P \mid Q) \uplus T \stackrel{\text{def}}{=} P \uplus T \mid Q \uplus T$$
$$(!P) \uplus T \stackrel{\text{def}}{=} !(P \uplus T) \qquad \mathbf{0} \uplus T \stackrel{\text{def}}{=} \mathbf{0}$$

$\boxed{P \equiv P}$ *Rules for structural congruence*

$$P \mid Q \equiv Q \mid P \qquad (P \mid Q) \mid J \equiv P \mid (Q \mid J) \qquad !P \equiv P \mid !P \qquad P \equiv P \mid \mathbf{0}$$
$$(\mathsf{new}\, x\colon \mathsf{s\,chan}\langle T\rangle_n)(P) \mid Q \equiv (\mathsf{new}\, x\colon \mathsf{s\,chan}\langle T\rangle_n)(P \mid Q \uplus \mathsf{s\,chan}\langle T\rangle_n)$$
$$(\mathsf{new}\, x\colon T)(P) \mid Q \equiv (\mathsf{new}\, x\colon T)(P \mid Q) \qquad T \neq \mathsf{s\,chan}\langle T'\rangle_n$$
$$(\mathsf{new}\, x\colon T)(\mathbf{0}) \equiv \mathbf{0} \qquad (\mathsf{new}\, x\colon T_1)(\mathsf{new}\, y\colon T_2)(P) \equiv (\mathsf{new}\, y\colon T_2)(\mathsf{new}\, x\colon T_1)(P)$$

$\boxed{P \to P}$ *Rules for reduction*

$$\overline{x}\langle v\rangle.P \mid x(y \div B).Q \;\to\; P \mid Q\{v/y\} \qquad \text{[R-Com]}$$
$$\frac{P \;\to\; Q}{(\mathsf{new}\, x\colon T)(P) \;\to\; (\mathsf{new}\, x\colon T)(Q)} \qquad \frac{P \;\to\; P'}{P \mid Q \;\to\; P' \mid Q} \qquad \frac{P \equiv P' \quad P' \;\to\; Q' \quad Q' \equiv Q}{P \;\to\; Q}$$
[R-Res] [R-Par] [R-Struct]

Fig. 2. π-calculus: type-based blocking and reduction semantics

Static semantics We consider typing judgments for processes of the form $\Gamma \vdash P \triangleright \Delta$ with $\mathrm{fv}(P) \subseteq \mathrm{dom}(\Gamma)$ and $\mathrm{dom}(\Delta) = \mathrm{dom}(\Gamma)$. *Type environments* or contexts Γ are a map from variables to types T; *return type environments* or return contexts Δ are map from variables to *return types* U, which include types T in Figure 1 and the *void* type, noted $\bullet$.

$$U \;::=\; T \mid \bullet$$

The void type is managed transparently by the type system and is not used to decorate restricted channels. Return contexts are used to convey the actual use of channels, which can differ from the one described by the type environment. In particular, channels having a void return type cannot be accessed by processes running in parallel, while (dynamic) channels that are not used by the process can be promoted to type $\top$. In order to specify the typing system, in Figure 3 we introduce auxiliary operations on types, type environments and return type environments. The *upcast* partial operation, noted $\uparrow$, is used to upcast the payload of a channel type from dynamic to static. The *downcast* operation, noted $\downarrow_i^\bullet$, transforms dynamic types exchanging a static channel identified by i into the void type, disallowing further interaction of the context. This is enforced in the rule to type a parallel composition by means of a *composition* partial binary op-

$\boxed{T{\uparrow} = T}$ *Type upcast*

$$\mathsf{d\,chan}\langle T\rangle_\forall{\uparrow} = \mathsf{s\,chan}\langle T\rangle_n \qquad n = \mathsf{gen}()$$

$\boxed{\Delta{\downarrow_i^\bullet} = \Delta}$ *Return context downcast*

$$U{\downarrow_i^\bullet} = \begin{cases} \bullet & U = \mathsf{d\,chan}\langle \mathsf{s\,chan}\langle T\rangle_n\rangle_\forall \text{ and } i = n \\ U & \text{else} \end{cases}$$

$$\emptyset{\downarrow_i^\bullet} = \emptyset \qquad \frac{\Delta{\downarrow_i^\bullet} = \Delta_1}{(\Delta, x\colon U){\downarrow_i^\bullet} = \Delta_1, x\colon U{\downarrow_i^\bullet}}$$

$\boxed{\Delta \otimes \Delta = \Delta}$ *Return context composition*

$$U \otimes \top = U \qquad \top \otimes U = U \qquad T \otimes T = T$$

$$\emptyset \otimes \emptyset = \emptyset \qquad \frac{\Delta_1 = \Delta_3, x\colon U_1 \qquad \Delta_2 = \Delta_4, x\colon U_2}{\Delta_1 \otimes \Delta_2 = \Delta_3 \otimes \Delta_4, x\colon U_1 \otimes U_2}$$

Fig. 3. Type system: auxiliary operations

eration over return contexts, noted $\otimes$, which only allows to compose void types with top types.

The typing system in Figure 4 introduces typing rules for values and processes. We illustrate the most interesting rules. Rule [T-Par] allows to type a parallel composition $P_1 \mid P_2$ by composing the return contexts Δ_1 and Δ_2 produced respectively by typing P_1 and P_2, when the $\otimes$ operation is defined. In rule [T-Repl] we accept replicated processes that do not send static channels over dynamic channels, to disallow instances of P to enlarge the scope of a static channel. Rule [T-Res-S] allows to type a restricted variable having a static type identified by a unique number n. To this aim, the continuation must be typed by adding to the context the new entry for the variable and return a context Δ that does not change the type of the restricted variable. The top-level return context is built by pruning the restricted variable and by downcasting Δ. We have two rules for input, [T-In] and [T-In-Up], and to rules for output, [T-Out] and [T-Out-Up], which correspond respectively to input, upcast in input, output, and upcast in output. We allow to upcast the payload of channels from dynamic to static in input (a) and output (b), given that: a) the upcasted type is not blocked; b) the type of the channel's object is equal to the upcasted type. Note that in rules [T-In-Up],[T-Out-Up] the return type of the channel is different from the type in the type environment.

Rule [T-In] allows to type an input process $x(y \div B).P$ with a channel of the form $m\,\mathsf{chan}\langle T'\rangle_i$, given that $T' \notin B$ and that the continuation can be typed by adding the entry $y\colon T'$ to the context. The return type of the bound variable y must not change in the continuation: to upcast the type of a variable bound by an input, one has to use rule [T-In-Up]. The top-level call returns the environ-

$\boxed{\Gamma \vdash v\colon T}$ *Typing rule for variables*

$$\Gamma, v\colon T \vdash v\colon T \qquad \text{[T-Var]}$$

$\boxed{\Gamma \vdash P \triangleright \Delta}$ *Typing rules for processes*

$$\frac{\Gamma \vdash P_1 \triangleright \Delta_1 \qquad \Gamma \vdash P_2 \triangleright \Delta_2}{\Gamma \vdash P_1 \mid P_2 \triangleright \Delta_1 \otimes \Delta_2} \qquad \frac{\Gamma \vdash P \triangleright \Delta \qquad \bullet \notin \mathrm{range}(\Delta)}{\Gamma \vdash !P \triangleright \Delta} \qquad \text{[T-Par] [T-Repl]}$$

$$\frac{\Gamma, y\colon T \vdash P \triangleright \Delta, y\colon U}{\Gamma \vdash (\mathsf{new}\, y\colon T)(P) \triangleright \Delta} \qquad \frac{\Gamma, y\colon T \vdash P \triangleright \Delta, y\colon T}{\Gamma \vdash (\mathsf{new}\, y\colon \mathsf{s}\,\mathsf{chan}\langle T\rangle_n)(P) \triangleright \Delta{\downarrow}^{\bullet}_n} \qquad \text{[T-Res],[T-Res-S]}$$

$$\frac{T = m\,\mathsf{chan}\langle T'\rangle_i \qquad T' = m'\,\mathsf{chan}\langle T''\rangle_j \qquad T' \notin B \qquad \Gamma, x\colon T, y\colon T' \vdash P \triangleright \Delta, x\colon T, y\colon T'}{\Gamma, x\colon T \vdash x(y \div B).P \triangleright (\Delta{\downarrow}^{\bullet}_j), x\colon T} \qquad \text{[T-In]}$$

$$\frac{T{\uparrow} = \mathsf{s}\,\mathsf{chan}\langle T'\rangle_n \qquad T{\uparrow} \notin B \qquad \Gamma, x\colon m\,\mathsf{chan}\langle T{\uparrow}\rangle_i, y\colon T{\uparrow} \vdash P \triangleright \Delta, x\colon m\,\mathsf{chan}\langle T{\uparrow}\rangle_i, y\colon T{\uparrow}}{\Gamma, x\colon m\,\mathsf{chan}\langle T\rangle_i \vdash x(y \div B).P \triangleright (\Delta{\downarrow}^{\bullet}_n), x\colon m\,\mathsf{chan}\langle T{\uparrow}\rangle_i} \qquad \text{[T-In-Up]}$$

$$\frac{T = m\,\mathsf{chan}\langle T'\rangle_i \qquad \Gamma \vdash v\colon T' \qquad \Gamma, x\colon T \vdash P \triangleright \Delta, x\colon T \qquad \Delta \vdash v\colon T'}{\Gamma, x\colon T \vdash \overline{x}\langle v\rangle.P \triangleright \Delta, x\colon T} \qquad \text{[T-Out]}$$

$$\frac{T{\uparrow} = \mathsf{s}\,\mathsf{chan}\langle T'\rangle_n \qquad \Gamma \vdash y\colon T{\uparrow} \qquad \Gamma, x\colon m\,\mathsf{chan}\langle T{\uparrow}\rangle_i \vdash P \triangleright \Delta, x\colon m\,\mathsf{chan}\langle T{\uparrow}\rangle_i \qquad \Delta \vdash y\colon T{\uparrow}}{\Gamma, x\colon m\,\mathsf{chan}\langle T\rangle_i \vdash \overline{x}\langle y\rangle.P \triangleright \Delta, x\colon m\,\mathsf{chan}\langle T{\uparrow}\rangle_i} \qquad \text{[T-Out-Up]}$$

$$\emptyset \vdash \mathbf{0} \triangleright \emptyset \qquad \frac{U = T \text{ or } (T = \mathsf{d}\,\mathsf{chan}\langle T'\rangle_i \text{ and } U = \top) \qquad \Gamma \vdash \mathbf{0} \triangleright \Delta}{\Gamma, x\colon T \vdash \mathbf{0} \triangleright \Delta, x\colon U} \qquad \text{[T-Inact-E],[T-Inact]}$$

Fig. 4. π-calculus: type checking

ment $\Delta{\downarrow}^{\bullet}_j, x\colon T$, where Δ is obtained by the call for the continuation, and j is the identifier of the payload type T': this disallows attempts to declassify secret channels by means of forwarding (cf. Section 4). Rule [T-In-Up] allows to type an input process $x(y \div B).P$ with a type of the form of the form $m\,\mathsf{chan}\langle T\rangle_i$, given that $T{\uparrow}$ is defined and that $T{\uparrow} \notin B$, and that the continuation can be typed by both changing the type of x to $m\,\mathsf{chan}\langle T{\uparrow}\rangle_i$, and by adding the entry $y\colon T{\uparrow}$. Note that the return type of x cannot change, and that the return context Δ is downcasted: this operation set the type of channels that are exchanging y to void, disallowing further interaction of the context. Rule [T-Out] allows to type an output process and is standard, while we require that the type of the sent variable does not change in the return type environment, to enforce a consistent use of the variable in the context. Rule [T-Out-Up] is specular to [T-In-Up], while there is no need to downcast the return type environment since the object of the output has already a static type: as in [T-Out], we enforce that the object type does not change in the return type environment. We have two rules for

inaction, [T-Inact-E] and [T-Inact], corresponding to empty and non-empty contexts. In rule [T-Inact], we allow to promote each return type of the form $\mathsf{d}\,\mathsf{chan}\langle T\rangle_\forall$ to type $\top$: this permits compositions with processes that are not using i/o channel capabilities (cf. [20]).

The subject reduction theorem ensures that the static semantics of the π-calculus agrees with its dynamic semantics. As usual, the proof relies on two auxiliaries results: type preservation under structural congruence, and a substitution lemma. See [18] for all details.

Theorem 1 (Subject reduction). *Let Γ be balanced. If $\Gamma \vdash P \triangleright \Delta$ and $P \to Q$ then there is Δ' such that one of the following hold:*

1. $\Gamma \vdash Q \triangleright \Delta'$;
2. $\Gamma = \Gamma_1, x\colon m\,\mathsf{chan}\langle T\rangle_i$ *and* $\Gamma_1, x\colon m\,\mathsf{chan}\langle T\uparrow\rangle_i \vdash Q \triangleright \Delta'$, *for some* $x \in \mathrm{dom}(\Gamma)$.

The last result of this section establishes the soundness of the types analysis, namely: well-typed processes do not try to enlarge the scope of a channel decorated as static. The theorem below formalizes this intuition.

Theorem 2 (Soundness). *If $\Gamma \vdash P \triangleright \Delta$ and P reduces in zero or more steps to $(\mathsf{new}\, x_1\colon T_1)\cdots(\mathsf{new}\, x_n\colon T_n)(Q \mid R)$ then none of the following cases happen:*

1. $Q = (\mathsf{new}\, y\colon \mathsf{s}\,\mathsf{chan}\langle T\rangle_n)(\overline{x}\langle y\rangle.Q_1 \mid Q_2) \mid x(z \div B).Q_3$
2. $Q = (\mathsf{new}\, y\colon \mathsf{s}\,\mathsf{chan}\langle T\rangle_n)(\overline{x}\langle y\rangle.Q_1 \mid Q_2 \mid x(z \div B \cup \mathsf{s}\,\mathsf{chan}\langle T\rangle_n).Q_3)$

3 An API for secure programming

Rather than ask to programmers to use the security type abstractions presented in the previous section, we want to provide an high-level language featuring secret channels and transparently compile it in the typed π-calculus. In the remainder of the section we present the language, which is inspired by the secret π-calculus [19], and its compilation in the typed π-calculus of Section 2. We conclude by proving that the static and dynamic semantics of the translated programs agree with the dynamic semantics of the secret π-calculus, thus showing that our approach is sound. This provides an abstract *API* for secure programming: the programmer writes the security protocol in the high-level language, the protocol is compiled into the typed π-calculus, type-checking is performed before execution. When the protocol is well-typed, our main result is that direct information flows on secure channels are not allowed (while attacks based on indirect flows are still possible, cf. [26]).

Programmer language The syntax[1] in Figure 5 is inspired by the secret π-calculus [19]. To depict channel-based communication we consider an infinite

[1] The original formulation of the secret π-calculus is untyped and also features a form of trusted input, which is outside the scope of the paper.

Programmer syntax

H	::=		Programs:	$!H$	replication
		$\overline{x}\langle v\rangle.H$	output	$H \mid K$	composition
		$x(y).H$	input	$\mathbf{0}$	inaction
		$[\mathsf{hide}\, x][H]$	secret channel	$(\mathsf{new}\, x)(H)$	channel

Extended syntax

A	::=		Types	$[\mathsf{hide}\, x\colon A][M]$	secret channel
		$\mathsf{chan}\langle A\rangle$	channel	$(\mathsf{new}\, x\colon A)(M)$	channel
		$\top$	top	$!M$	replication
M	::=		Processes:	$M \mid N$	composition
		$\overline{x}\langle v\rangle.M$	output	$\mathbf{0}$	inaction
		$x(y \div \mathcal{B}).M$	input		

Fig. 5. Secret π-calculus

set $\mathcal{N}$ of channels or *variables* ranged over by x, y, z and v. We use $\mathcal{A}, \mathcal{B}, \mathcal{C}$ to denote subsets of $\mathcal{N}$. The programmer syntax includes, in addition to the standard operators of the π-calculus [24], a *secret channel* process $[\mathsf{hide}\, x][H]$, which is a process that creates an invisible channel x and continue as H. The programmer must be carefully include in the scope delimited by the square parentheses all processes that are meant to communicate over a secret channel: (compiled) processes that try to open the scope of a secret channel are rejected by the typed analysis. The extended syntax in the same figure introduces two modifications, which are transparent to the programmer: a set of blocked variables in input, noted $\mathcal{B}$, and channel type decorations in channel and secret channel creation. The binders of the extended language appear in parenthesis: $x(y \div \mathcal{B}).M$, $(\mathsf{new}\, y)(M)$ and $[\mathsf{hide}\, y][M]$ bind the free occurrences of y in M. The set $\mathcal{B}$ allows to embed the block mechanism in the axioms of structural congruence, which is required by the secret π-calculus semantics. For space limitations, we illustrate this mechanism with an example, and refer to [19] for all the details. Consider the secret π-calculus process M_1 below, where we assume x, z different from y and w.

$$M_1 = [\mathsf{hide}\, y\colon \top][\overline{x}\langle z\rangle.\overline{x}\langle y\rangle] \mid x(w \div \emptyset).\overline{x}\langle w\rangle \tag{5}$$

The reduction $M_1 \to [\mathsf{hide}\, y\colon \top][\overline{x}\langle y\rangle \mid \overline{x}\langle z\rangle]$ is enforced in two steps: first, by the structural congruence axioms, we establish $M_1 \equiv [\mathsf{hide}\, y\colon \top][\overline{x}\langle z\rangle.\overline{x}\langle y\rangle \mid x(w \div \{y\}).\overline{x}\langle w\rangle]$, then we allow the inner composition to interact since z sent in output is different from y blocked in input.

Compilation The compilation of programs H into typed π-calculus processes P is in two steps. User programs H are transformed into secret π-calculus pro-

Assignment of type modalities

$$[\![\mathsf{chan}\langle A\rangle]\!]_{\mathsf{d}} = \mathsf{d}\,\mathsf{chan}\langle [\![A]\!]_{\mathsf{d}}\rangle_{\forall} \qquad [\![\mathsf{chan}\langle A\rangle]\!]_{\mathsf{s}} = \mathsf{s}\,\mathsf{chan}\langle [\![A]\!]_{\mathsf{d}}\rangle \mathsf{gen}_{()}$$
$$[\![\top]\!]_m = \top \qquad m = \mathsf{s}, \mathsf{d}$$

Encoding processes

$$\begin{aligned}
[\![[\mathsf{hide}\, x\colon A][M]]\!]_\Gamma &= (\mathsf{new}\, x\colon [\![A]\!]_{\mathsf{s}})([\![M]\!]_{\Gamma, x\colon [\![A]\!]_{\mathsf{s}}}) \\
[\![(\mathsf{new}\, x\colon A)(M)]\!]_\Gamma &= (\mathsf{new}\, x\colon [\![A]\!]_{\mathsf{d}})([\![M]\!]_{\Gamma, x\colon [\![A]\!]_{\mathsf{d}}}) \\
[\![\overline{x}\langle y\rangle.M]\!]_\Gamma &= \overline{x}\langle y\rangle.[\![M]\!]_\Gamma \\
[\![x(y \div \mathcal{B}).M]\!]_\Gamma &= x(y \div B).[\![M]\!]_\Gamma \qquad \mathcal{B} = \{x_1, \ldots, x_n\},\ B = \{\Gamma(x_1), \ldots, \Gamma(x_n)\} \\
[\![M \mid N]\!]_\Gamma &= [\![M]\!]_\Gamma \mid [\![N]\!]_\Gamma \\
[\![!M]\!]_\Gamma &= ![\![M]\!]_\Gamma \\
[\![\mathbf{0}]\!]_\Gamma &= \mathbf{0}
\end{aligned}$$

Fig. 6. Compilation of secret π-calculus

cesses M by means of a function $\langle\!\langle \cdot \rangle\!\rangle_I$ that guesses[2] the channel types of the restricted channels of H (when possible), where $I = A_1 \cdots A_n$ is a stack of types such that A_1 is on top of the stack. The encoding is below: the remaining cases are homomorphic.

$$\begin{aligned}
\langle\!\langle x(y).H \rangle\!\rangle_I &= x(y \div \emptyset).\langle\!\langle H \rangle\!\rangle_I \qquad & \langle\!\langle (\mathsf{new}\, x)H \rangle\!\rangle_{A \cdot I} &= (\mathsf{new}\, x\colon A)\langle\!\langle H \rangle\!\rangle_I \\
\langle\!\langle [\mathsf{hide}\, x]H \rangle\!\rangle_{A \cdot I} &= [\mathsf{hide}\, x\colon A]\langle\!\langle H \rangle\!\rangle_I \qquad & \langle\!\langle H \mid K \rangle\!\rangle_{I_1 \cdot I_2} &= \langle\!\langle H \rangle\!\rangle_{I_1} \mid \langle\!\langle K \rangle\!\rangle_{I_2}
\end{aligned}$$

The encoding from secret π-calculus processes M to π-calculus processes P, noted $[\![\cdot]\!]_\Gamma$, in Figure 3 is parametrized by a type environment Γ. We use Γ to transform blocked variables of a secret π-calculus input process into blocked types of a typed π-calculus process, while constructing the type environment from the program code. Standard types A are encoded into types T of Figure 1 by means of the compilation $[\![\cdot]\!]_m$ defined in the same Figure. Note that the payload of types is qualified as dynamic since the typing system allows to upcast it to a static type, as introduced in Section 2. The encoding of restricted channels assign a dynamic type modality to channels programmed with new, and a static type modality to channels programmed with hide, as expected.

The main result of this section is that compiled processes that type-check preserve the dynamic semantics of the secret π-calculus, in the following sense.

Theorem 3. *Let M be a secret π-calculus process. If there are Γ, Γ_1 and Δ_1 such that $\Gamma_1 \vdash [\![M]\!]_\Gamma \triangleright \Delta_1$, then the following hold.*

1. *If $M \to M'$ then $[\![M]\!]_\Gamma \to [\![M']\!]_\Gamma$*

[2] In practice, channel types would be inferred by using techniques based on constraint systems, e.g. [29, 6].

2. If $[\![M]\!]_\Gamma \to Q$ then there is M' such that $M \to M'$ and $[\![M']\!]_\Gamma = Q$

A simple counter-example is the secret π-calculus process $N = [\mathsf{hide}\, y\colon \top][\overline{x}\langle y\rangle.N'] \mid x(z \div \emptyset).N''$: the compilation $[\![N]\!]_\Gamma$ does not type check, for any Γ, since the type system rejects the attempt of reading from a channel sending a secret variable. Other counter-examples are the secret π-calculus processes $\overline{x}\langle x\rangle$ and $\overline{x}\langle y\rangle \mid \overline{y}\langle x\rangle$, because we do not consider recursive types (for simplicity), as well as processes that are decorated with the wrong types.

4 Applications

To illustrate possible usages of the API, we start by drawing an example based on a process that potentially attempts to declassify a secret channel by means of forwarding it on a public channel x, where we assume w, y, x, z all distinct.

$$M_2 = [\mathsf{hide}\, y\colon \mathsf{chan}\langle\top\rangle][\overline{w}\langle y\rangle \mid w(z \div \emptyset).\overline{x}\langle z\rangle] \tag{6}$$

The encoding of M_2 is process P_2 below, which type-checks: types T_1, T_2 describe the free channels of P, while type T_4 is a return type of P.

$$\begin{aligned} P_2 &= (\mathsf{new}\, y\colon T_3)(\overline{w}\langle y\rangle \mid w(z \div \emptyset).\overline{x}\langle z\rangle) \\ T_1 &= \mathsf{d}\,\mathsf{chan}\langle T_2\rangle_\forall \qquad T_2 = \mathsf{d}\,\mathsf{chan}\langle\top\rangle_\forall \\ T_3 &= \mathsf{s}\,\mathsf{chan}\langle\top\rangle_1 \qquad T_4 = \mathsf{d}\,\mathsf{chan}\langle T_3\rangle_\forall \end{aligned}$$

Indeed this process, taken in isolation, is safe, while its interaction can entail a security break, as we discuss at end of the paragraph.

Take the type environment $\Gamma = w\colon T_1, x\colon T_1$. Informally, the type system allows to send the static channel y over w and x by upcasting the type of the payload of w and x, and by downcasting their return type. We first outline a derivation for the left thread of P_2: note that we change the return type of x through (three applications of) [T-Inact], since x is not used.

$$\dfrac{\dfrac{}{w\colon T_4, x\colon T_1, y\colon T_3 \vdash \mathbf{0} \triangleright w\colon T_4, x\colon \top, y\colon T_3}\ ([\text{T-Inact}])}{\Gamma, y\colon T_3 \vdash \overline{w}\langle y\rangle \triangleright w\colon T_4, x\colon \top, y\colon T_3}\ ([\text{T-Out-Up}])\ (*)$$

A derivation for the right thread is below; note that the return type of x is set to void, since the secret variable z is sent over x.

$$\dfrac{\dfrac{\dfrac{}{w\colon T_4, x\colon T_4, y\colon T_3, z\colon T_3 \vdash \mathbf{0} \triangleright w\colon T_4, x\colon T_4, y\colon T_3, z\colon T_3}\ ([\text{T-Inact}])}{w\colon T_4, x\colon T_1, y\colon T_3, z\colon T_3 \vdash \overline{x}\langle z\rangle \triangleright w\colon T_4, x\colon T_4, y\colon T_3, z\colon T_3}\ ([\text{T-Out-Up}])}{\Gamma, y\colon T_3 \vdash w(z).\overline{x}\langle z\rangle \triangleright w\colon T_4, x\colon \bullet, y\colon T_3}\ ([\text{T-In-Up}])\ (**)$$

We glue together the two derivations by using [T-Par], and finish by applying [T-Res-S]: the final effect is to set the return type of w to void.

$$
\begin{aligned}
\{\!|\,()\,|\!\}_z &= \overline{z}\langle \bot, \bot, \bot \rangle \\
\{\!|\,(\langle a_0, b_0\rangle, \ldots, \langle a_n, b_n\rangle)\,|\!\}_z &= (\mathsf{new}\, z')(\overline{z}\langle a_0, b_0, z'\rangle \mid \{\!|\,(\langle a_1, b_1\rangle, \ldots, \langle a_n, b_n\rangle)\,|\!\}_{z'}) \\
\textsc{Add}(x, y, z) &= z(h_1, h_2, z').((\mathsf{new}\, z'')(\overline{z}\langle x, y, z''\rangle \mid \overline{z''}\langle h_1, h_2, z'\rangle) \mid \\
&\qquad\qquad \overline{\mathrm{port88}}\langle h_1, h_2\rangle) \qquad \%\% \text{ Suspicious}
\end{aligned}
$$

Fig. 7. A malicious list handler

$$
\dfrac{\dfrac{(*) \qquad (**)}{\Gamma, y\colon T_3 \vdash \overline{w}\langle y\rangle \mid w(z).\overline{x}\langle z\rangle \triangleright w\colon T_4, x\colon \bullet, y\colon T_3}\ ([\text{T-Par}])}{\Gamma \vdash P_2 \triangleright w\colon \bullet, x\colon \bullet}\ ([\text{T-Res-S}])
$$

A void return type acts as a protection against contexts trying to leak secrets from the process. For instance, a composition of the form $P_2 \mid x(u).P'$ does not type check, i.e. $\Gamma \not\vdash P_2 \mid x(u).P' \triangleright \Delta$, for any P'. This holds because the composition $(w\colon \bullet, x\colon \bullet) \otimes (w\colon U_1, x\colon U_2)$ is undefined whenever $U_1 \neq \top, U_2 \neq \top$, and because if $\Gamma \vdash x(u).P' \triangleright \Delta'$ then $\Delta'(x) \neq \top$. Similarly, $P_2 \mid Q$ is ill-typed if w appears free as subject of an input or output in Q, for any Q.

Safe programming with third-party libraries The abstract security API can be useful to protect against malicious behaviour of third-party libraries. The malicious code that we consider in Figure 7 is an implementation of a linked list in a polyadic extension of the programmer language in Figure 5: the code is inspired by [11]. A list of paired names $(\langle a_0, b_0\rangle, \ldots, \langle a_n, b_n\rangle)$ is programmed through the encoding $\{\!|\,\cdot\,|\!\}$ as a list of processes linked by pointers: process $\overline{z_i}\langle x_i, y_i, z_{i+1}\rangle$ represents the entry $\langle x_i, y_i\rangle$, where z_i is the reference to the next pair. A pair $\langle x, y\rangle$ can be added to a list z by means of the meta-process Add, noted $\textsc{Add}(x, y, z)$. The question we face is: how to detect if the library provides a backdoor by forwarding the content of the list on some port, as in the last line of process Add? Rather than checking the code of the library, programmers may trust the list implementation and use it to store secure channels in a data structure that allows to customize operations, e.g. searches.

$$
\textsc{StoreSecCh}(N, y) = [\mathsf{hide}\, x][N \mid (\mathsf{new}\, z)(\{\!|\,()\,|\!\}_z \mid \textsc{Add}(x, y, z))]
$$

To ensure that the secure channel is not disclosed when composing $\textsc{StoreSecCh}(N, y)$ with some N', we may compile the composition and run the type-checker. If the result is positive, Theorem 3.2 ensures that the translation preserves the invariant prescribed by the dynamic semantics of the secret π-calculus, that is that process N' will not receive channel x during the computation. This allows to use the secret π-calculus as an abstract API language for secure protocols.

Enforcing mandatory access control We review an example discussed in [19]. D-Bus [25] is an IPC system for software applications that is used in many desktop environments. Applications of each user share a private bus for asynchronous

message-passing communication; a system bus permits to broadcast messages among applications of different users. Versions smaller than 0.36 contain an erroneous access policy for channels which allows users to send and listen to messages on another user's channel if the address of the socket is known. The code for the attack is synthesized below.

```
[marco@localhost]# echo $DBUS_SESSION_BUS_ADDRESS > Public/address
[guest@localhost]# dbus-monitor --address /home/marco/Public/address
```

The correct policy, subsequently released by Fedora, restricts the access to the user's bus to applications with the same user-id. The policy is mandatory and cannot be changed by users, otherwise security is broken: this is enforced directly in the (untyped) Unix-like access control method. Our interpretation of this vulnerability is that the user's bus can be abstracted as an hidden channel, that is a channel that must not be disclosed by means of internal attacks or Trojan horses. To ensure this invariant, we can program the D-Bus protocol in the secret π-calculus, translate it in the π-calculus, and run the type-checker: this will ensure that, when type-checking succeeds, the mandatory policy is enforced, without the need of relying on explicit access control methods. We refer to [19] for a possible implementation of the D-Bus protocol in the secret π-calculus.

5 Discussion

We introduce a type system to enforce static scope for channels in the π-calculus, and defend that the static analysis can help in devise programs featuring secret channels by providing a semantics-preserving translation of a fragment of the secret π-calculus [19] into our typed π-calculus.

While we analyze some simple application of our technique, which we interpret as an abstract API for secure message passing, we leave for future work a precise comparison with calculi and frameworks for secret protocols (e.g. [5, 4, 2, 15]) and π-calculus dialects featuring static channels (e.g. [16, 30]), as well as a (typed) behavioural theory to establish secrecy equations (cf. [17, 19]). Other extensions we are interested in consist in study the integration among static and dynamic type qualifiers and session types [20], develop type-inference techniques á la [29, 6] to fully automatize the process compilation, and devise a type checking algorithm to resolve the sources of non-determinism in the typing system (cf. the rules for input and for inaction).

Many analysis and programming techniques for security have been developed for process calculi. Among these, we would mention the security analysis enforced by means of static and dynamic type-checking (e.g. [12, 22, 9]), the verification of secure implementations and protocols that are protected by cryptographic encryption (e.g. [8, 3, 10]), and programming models that consider a notion of location (e.g. [23, 28, 13, 19]). The most related papers are [19, 12]. The paper in [19] introduces the secret π-calculus, its behavioural theory, and a characterization based on bisimulation semantics. The presence of a *spy* context allows to break some of the standard observational equivalences for restriction, which

can be recovered by using the secret channel operator. It would be interesting to investigate whether the untyped theory of [19] would match a typed behavioural theory based on static and dynamic type qualifiers. The work in [12] introduces a π-calculus featuring a group creation operator, and a typing system that disallows channels to be sent outside of the group. Programmers must declare which is the group type of the payload: the typing system rules out processes of the form $(\mathsf{new}\, p\colon U)(P \mid (\mathsf{new}\, G)(\mathsf{new}\, x\colon G[])(\overline{p}\langle x\rangle))$ since the type U of the public channel p cannot mention the secret type G, which is local. Differently, we accept processes of this form and do not require such effort to programmers: instead, we automatically infer the "group types" of processes declared with the hide macro, and allow secret channels to be sent over "untyped" channels: i.e. we type-check (the compilation of) process $(\mathsf{new}\, p)(p'(y) \mid [\mathsf{hide}\, x][\overline{p}\langle x\rangle])$ whenever $p' \neq p$, and reject it otherwise. Our main motivation is to shift the middleware support for secret channels (cf. [19]) to a software support in a transparent way: we show how this can be achieved, thus establishing an operational correspondence among untyped and typed semantics of secret channels. From the API language design point of view, we share some similarity with the ideas behind the boxed π-calculus [28]. A box in [28] acts as wrapper where we can confine untrusted processes; communication among the box and the context is subject to a fine-grained control that prevents the untrusted process to harm the protocol. Our hide macro is based on the symmetric principle, but requires stronger conditions, because we map the macro in a restriction process of the π-calculus: for a process to be (type-)checked, we require the context outside the scope of an hide to do not listen on channels exchanging secrets.

Acknowledgements This work is supported by the North Portugal Regional Operational Programme under contract NORTE-07-0124-FEDER-000062, by the European Regional Development Fund through the COMPETE Programme under contract FCOMP-01-0124-FEDER-037281 (PEST), and by national funds through FCT - Fundacão para a Ciência e a Tecnologia. I warmly thank PEST for travel support; I also thank the Center for Informatics and Information Technologies (CITI, `citi.di.fct.unl.pt`) for the support of facilities. The final version of this paper has been improved thanks to the detailed comments and useful criticism of the anonymous reviewers, to whom I am especially grateful.

References

1. Abadi, M.: Protection in programming-language translations. In: ICALP. LNCS, vol. 1443, pp. 868–883. Springer (1998)
2. Abadi, M., Blanchet, B.: Analyzing security protocols with secrecy types and logic programs. J. ACM 52(1), 102–146 (2005)
3. Abadi, M., Blanchet, B., Fournet, C.: Just fast keying in the pi calculus. ACM Trans. Inf. Syst. Secur. 10(3) (2007)
4. Abadi, M., Fournet, C.: Mobile values, new names, and secure communication. In: POPL. pp. 104–115. ACM press (2001)

5. Abadi, M., Gordon, A.D.: A calculus for cryptographic protocols: The spi calculus. Inf. Comput. 148(1), 1–70 (1999)
6. et al., M.L.: Typing component-based communication systems. In: FMOODS/-FORTE. LNCS, vol. 5522, pp. 167–181. Springer (2009)
7. Barendregt, H.: The Lambda Calculus - Its Syntax and Semantics. North-Holland (1981 (1st ed), revised 1984)
8. Boreale, M., De Nicola, R., Pugliese, R.: Proof techniques for cryptographic processes. SIAM J. Comput. 31(3), 947–986 (2001)
9. Bugliesi, M., Giunti, M.: Typed processes in untyped contexts. In: TGC. LNCS, vol. 3705, pp. 19–32. Springer (2005)
10. Bugliesi, M., Giunti, M.: Secure implementations of typed channel abstractions. In: POPL. pp. 251–262. ACM (2007)
11. Cai, X., Fu, Y.: The λ-calculus in the π-calculus. Math. Struct. Comp. Sci. 21(5), 943–996 (2011)
12. Cardelli, L., Ghelli, G., Gordon, A.D.: Secrecy and group creation. Inf. Comput. 196(2), 127–155 (2005)
13. Castagna, G., Vitek, J., Nardelli, F.Z.: The seal calculus. Inf. Comput. 201(1), 1–54 (2005)
14. Cortier, V., Kremer, S. (eds.): Formal Models and Techniques for Analyzing Security Protocols, Cryptology and Information Security, vol. 5. IOS Press (2011)
15. Cortier, V., Rusinowitch, M., Zalinescu, E.: Relating two standard notions of secrecy. Logical Methods in Computer Science 3(3) (2007)
16. Fournet, C., Gonthier, G.: The reflexive cham and the join-calculus. In: POPL. pp. 372–385. ACM Press (1996)
17. Giunti, M.: Secure Implementations of Typed Channel Abstractions. PhD Thesis TD-2007-1, Department of Informatics, Ca' Foscari University of Venice (2007)
18. Giunti, M.: Static semantics of secret channel abstractions (2014), technical report, available at `tinyurl.com/n14-report`
19. Giunti, M., Palamidessi, C., Valencia, F.D.: Hide and New in the Pi-Calculus. In: EXPRESS/SOS. EPTCS, vol. 89, pp. 65–79 (2012)
20. Giunti, M., Vasconcelos, V.T.: Linearity, session types and the pi calculus. Math. Struct. Comp. Sci. (2013), to appear, available at `tinyrurl.com/mscs2013`
21. Google: Application security., `google.com/about/appsecurity/research`. Accessed April 2014
22. Hennessy, M.: The security pi-calculus and non-interference. J. Log. Algebr. Program. 63(1), 3–34 (2005)
23. Hennessy, M.: A Distributed Pi-calculus. Cambridge University Press (2007)
24. Milner, R.: Communicating and mobile systems - the Pi-calculus. Cambridge University Press (1999)
25. Pennington, H., Carlsson, A., Larsson, A., Herzberg, S., McVittie, S., Zeuthen, D.: D-Bus specification., `dbus.freedesktop.org`
26. Sabelfeld, A., Myers, A.C.: Language-based information-flow security. IEEE Journal on Selected Areas in Communications 21(1), 5–19 (2003)
27. Sangiorgi, D., Walker, D.: The pi-calculus, a theory of mobile processes. Cambridge University Press (2001)
28. Sewell, P., Vitek, J.: Secure composition of untrusted code: Box pi, wrappers, and causality. J. Comp. Sec. 11(2), 135–188 (2003)
29. Vasconcelos, V.T., Honda, K.: Principal typing schemes in a polyadic pi-calculus. In: CONCUR. LNCS, vol. 715, pp. 524–538. Springer (1993)
30. Vivas, J.L., Dam, M.: From higher-order pi-calculus to pi-calculus in the presence of static operators. In: CONCUR. pp. 115–130. LNCS, Springer (1998)