Privacy by Design: On the Conformance Between Protocols and Architectures

Vinh-Thong Ta, Thibaud Antignac

Privacy by Design: On the Conformance Between Protocols and Architectures*

Vinh-Thong Ta and Thibaud Antignac

INRIA, University of Lyon, France
{vinh-thong.ta, thibaud.antignac}@inria.fr

Abstract. In systems design, we generally distinguish the architecture and the protocol levels. In the context of privacy by design, in the first case, we talk about privacy architectures, which define the privacy goals and the main features of the system at high level. In the latter case, we consider the underlying concrete protocols and privacy enhancing technologies that implement the architectures. In this paper, we address the question that whether a given protocol conforms to a privacy architecture and provide the answer based on formal methods. We propose a process algebra variant to define protocols and reason about privacy properties, as well as a mapping procedure from protocols to architectures that are defined in a high-level architecture language.

1 Introduction

According to the definition provided in [6], "the architecture of a system is the set of [elements and their relations] needed to reason about the system". In the context of privacy, the elements are typically the privacy enhancing technologies (PETs) themselves and the purpose of the architecture is to combine them to achieve the privacy requirements. Generally speaking, an architecture can be seen as the abstraction of a system since an architecture abstracts away the details provided by PETs (such as message ordering, timing, complex cryptographic algorithms, etc.). Architectures only capture the main functionalities that a system should provide, for instance, which computations and communications are to be performed by the components.

Works in privacy by design mainly focus on PETs rather than architectures. In the position paper [3], the authors addressed the problem of privacy by design at the architecture level and proposed the application of formal methods that facilitate a systematic architecture design. In particular, they provided the idea of *the architecture language and logic*, a dedicated variant of epistemic logics [12], to deal with different aspects of privacy. Basically, an architecture is defined as a set of *architecture relations*, which capture the computations and communications abilities of each component. For instance, a relation $compute_i(x = t)$ specifies that a component i can compute a value t for x. Nevertheless, since [3] is a

* The final publication is available at link.springer.com (URL not yet available).

position paper, the language envisioned in the paper is mainly based on an introductory description. An extended version of this language is detailed in [2].

In this paper, we address the major question that whether the integration or combination of several different PETs conforms to a particular architecture. One challenge we have to face is that due to the diversity of technologies and protocols, their combination can raise a huge number of scenarios. Moreover, architectures are defined in an abstract way, while concrete implementations are more detailed, and it is challenging to define a proper abstraction from a lower to a higher level. The goal of this paper is to provide answers to this question.

Specifically, our main contributions are two-fold: first, we propose a modified variant of the applied π-calculus [13] for specifying the protocols related to PETs, and reasoning about the knowledge of components during the protocol run. Second, we propose a mapping procedure which defines the connection between the protocol specified in the calculus and the architecture defined in the architecture language. This mapping allows us to show whether a protocol (or a combination of protocols) conforms to a given architecture. To the best of our knowledge, this work is the first attempt that examines the connection between the two levels based on formal methods in the context of privacy protection.

The paper is organized as follows: in Section 2, we review the privacy architectures language (PAL) proposed in [2], which is a high-level language for specifying architectures and reasoning about privacy requirements in them. Sections 3 and 4 contain our contributions. The modified applied π-calculus is given in Section 3. Section 4 discusses the connection between each calculus process and relation in PAL, as well as the definitions and properties of the conformance between the two levels. In Section 5 we review the most relevant related works. Finally, we conclude the paper and discuss about the future works in Section 6.

2 Architecture Level

The language we review here is a simplified version of the one in [2]. The functionality of a service is defined by $\Omega = \{\tilde{X} = T\}$, where T is a term and $\tilde{X} \in \mathit{Var}$ represents a variable that can be either indexed (X_K) or unindexed (X). Each $\tilde{X}$ can be a single variable or an array of variables. $F \in \mathit{Fun}$ denotes a function, and $\odot F(X)$ defines the iterative application of F to the variables in the array X (e.g.: $\odot + (X)$ defines the summation of the variables in a given array).

$$T ::= \tilde{X} \mid F(T_1, \ldots, T_n) \mid \odot\, F(X); \qquad \tilde{X} ::= X \mid X_K$$

$$\begin{aligned}
\mathcal{A} &::= \{\mathcal{R}\} \\
\mathcal{R} &::= \mathit{Has}_i^{arch}\left(\tilde{X}\right) \mid \mathit{Receive}_{i,j}\left(\{\mathit{Att}\}, \tilde{X}\right) \mid \mathit{Compute}_i\left(\tilde{X} = T\right) \\
&\quad \mid \mathit{Check}_i\left(T_1 = T_2\right) \mid \mathit{Verif}_j^{Attest}\left(\mathit{Att}\right) \mid \mathit{Trust}_{i,j} \\
\mathit{Att} &::= \mathit{Attest}_i\left(\{\tilde{X} = T\}\right)
\end{aligned}$$

An architecture $\mathcal{A}$ is defined by a set of components C_i, $i \in [1, \ldots, n]$, associated with the set of *relations* $\{\mathcal{R}\}$. Each *relation* $\mathcal{R}$ specifies a capability of the components. Subscripts i and j denote component IDs. $\mathit{Has}_i^{arch}(\tilde{X})$ expresses

the fact that $\tilde{X}$ is a variable that component C_i initially has (i.e., an input variable of C_i). $Receive_{i,j}(\{Att\}, \tilde{X})$ expresses the possibility for C_i to receive the variable $\tilde{X}$ directly from C_j, and optionally an attestation Att related to this variable. An attestation, defined by $Attest_i(\{\tilde{X} = T\})$, captures a statement made by C_i on the set of equations. Each $\tilde{X} = T$ in the set $\{\tilde{X}{=}T\}$ expresses the integrity of $\tilde{X}$, stating that it equals to T. $Compute_i(\tilde{X} = T)$ says that C_i can compute a variable defined by an equation $\tilde{X} = T$. $Check_i\,(T_1 = T_2)$ states that C_i can check the satisfaction of property $T_1 = T_2$. The property $\tilde{X} = T$ in $Attest_i$ is related to the same property in $Compute_i$, namely when C_i computes $\tilde{X} = T$ it can send an attestation on this. $Verif_j^{Attest}(Att)$ says that C_j is able to successfully verify the origin of an attestation. Finally, $Trust_{i,j}$ is used to express the fact that component C_i trusts C_j, and this trust relation does not change during operations. Trust relations are pre-defined, and an attestation sent by C_i will be accepted by C_j after a successful verification only if C_j trusts C_i.

The semantics of an architecture is defined as its sets of compatible traces. A trace is a sequence of possible high-level events occurring in the system. Events can be seen as *instantiated relations* of the architecture.

$$\begin{array}{lcl} \theta & ::= & Seq(\epsilon) \\ \epsilon & ::= & has_i\left(\tilde{X} : V\right) \mid receive_{i,j}\left(\{Att\}, \tilde{X} : V\right) \mid compute_i\left(\tilde{X} = T\right) \\ & & \mid check_i\,(T_1 = T_2) \mid verif_j^{Attest}\,(Att) \end{array}$$

To distinguish events from relations, we let events start with lowercase. For instance, event $has_i\left(\tilde{X} : V\right)$ captures the fact that C_i has the value V for $\tilde{X}$, and $compute_i\left(\tilde{X} = T\right)$ expresses the fact that C_i performes the computation $\tilde{X} = T$. The other events are interpreted based on their corresponding relations (see [2] for details). An event trace θ is *compatible* with an architecture $\mathcal{A}$, if in this trace, only events which are instantiations of components of the architecture can appear in θ – except for the *compute* events. For the case of *compute* events, besides the computation specified explicitly in the architecture, we also take into account the "background" computations (deduction) that can be performed by each component, based on the data it has. This deduction ability of each C_i is captured by its deduction system $\triangleright_i$ [2]. The semantics of events is based on the *component states* and the *global state* of the architecture, given as follows:

$$State = State_V \times State_P; \qquad State_V = (Var \rightarrow Val_\perp); \qquad State_P = \left\{\{\tilde{X} = T\} \cup \{T_1 = T_2\} \cup \{Trust_{i,j}\}\right\}$$

The state of a component ($State$) is composed of a variable state ($State_V$) and a property state ($State_P$). $State_V$ assigns a value (which can be undefined, $\perp$) to each variable. $State_P$ defines the set of properties $\tilde{X} = T$ and $T_1 = T_2$ known by a component.

In the sequel, σ is used to denote the global state of the architecture $\mathcal{A}$ (state of the components $\langle C_1, \ldots, C_n\rangle$) defined on $State^n$. σ_i ($\sigma_i = (\sigma_i^v, \sigma_i^{pk})$) denotes the state of the component C_i, where σ_i^v and σ_i^{pk} represent the variable state and property state of C_i, respectively. The initial state of $\mathcal{A}$, denoted by $Init^{\mathcal{A}}$, contains only the trust properties specified by the architecture. The semantics of an event trace is defined by the function $\mathcal{S}_T$, which specifies the impact of

a trace of events on the states of the components, through the impact of each event on the states (defined by the function S_E).

$$\mathcal{S}_T : \mathit{Trace} \times \mathit{State}^n \rightarrow \mathit{State}^n; \qquad \mathcal{S}_E : \mathit{Event} \times \mathit{State}^n \rightarrow \mathit{State}^n;$$
$$\mathcal{S}_T\ (\epsilon.\theta,\ \sigma) = S_T(\theta,\ S_E(\epsilon,\ \sigma)); \qquad \mathcal{S}_T\ (\langle\rangle,\ \sigma) = \sigma;$$
$$\mathcal{S}_E\ (\mathit{compute}_i\ \left(\tilde{X} = T\right), \sigma) = \sigma[(\sigma_i^v[\mathit{eval}(T, \sigma_i^v)/\tilde{X}],\ \sigma_i^{pk} \cup \{\tilde{X} = T\})\ /\ \sigma_i\ ].$$

Due to lack of space we only present $\mathcal{S}_E$ for the compute event here, the full list can be found in [2]. The notation $\epsilon.\theta$ is used to denote a trace whose first element is ϵ and the rest of the trace is θ, while $\langle\rangle$ denotes the empty trace. Each event modifies only the state σ_i of the component C_i. A modification is expressed by $\sigma[(v, pk)/\sigma_i]$ that replaces σ_i^v and σ_i^{pk} of σ_i by v and pk, respectively. The effect of $\mathit{compute}_i(\tilde{X} = T)$ is to set $\tilde{X}$ to the evaluation of T based on the current variable state σ_i^v, which is denoted by $\mathit{eval}(T, \sigma_i^v)$. Event $\mathit{compute}_i$ also results in adding the knowledge about $\tilde{X} = T$ to the property state σ_i^{pk}.

The semantics of an architecture $\mathcal{A}$ is defined as $\mathcal{S}(\mathcal{A}) = \{\sigma \in \mathit{State}^n \mid \exists\theta \in T(\mathcal{A})$ such that $S_T(\theta, \mathit{Init}^{\mathcal{A}}) = \sigma\}$, where $T(\mathcal{A})$ is the set of compatible traces of $\mathcal{A}$. To reason about the privacy requirements of architectures, the *architecture logic* is proposed in [2], which is based on the architecture language PAL.

$$\phi ::= \ \mathit{Has}_i^{all}\left(\tilde{X}\right) \mid \mathit{Has}_i^{none}\left(\tilde{X}\right) \mid K_i\left(T_1 = T_2\right) \mid \phi_1 \wedge \phi_2$$

This logic involves modality K_i that represents the epistemic knowledge [12] of C_i about $T_1 = T_2$. In the rest of the paper, we refer to ϕ as an architecture property. The semantics $S(\phi)$ of a property ϕ is defined as follows:

1. $\mathcal{A} \in S(\mathit{Has}_i^{all}\left(\tilde{X}\right)) \Leftrightarrow \exists\ \sigma \in \mathcal{S}(\mathcal{A}),\ \sigma_i^v(\tilde{X}) \neq \perp$
2. $\mathcal{A} \in S(\mathit{Has}_i^{none}\left(\tilde{X}\right)) \Leftrightarrow \forall\ \sigma \in \mathcal{S}(\mathcal{A}), \sigma_i^v(\tilde{X}) = \perp$
3. $\mathcal{A} \in S(K_i\left(Eq\right)) \Leftrightarrow \forall\ \sigma' \in \mathcal{S}_i(\mathcal{A}),\ \exists\ \sigma \in \mathcal{S}_i(\mathcal{A}),\ \exists\ Eq'$: $(\sigma \geq_i \sigma') \wedge (\sigma_i^{pk} \triangleright_i Eq') \wedge (Eq' \Rightarrow Eq)$,

where Eq (Eq') represents an equation $T_1 = T_2$ ($T_1' = T_2'$). An architecture satisfies the $\mathit{Has}_i^{all}(\tilde{X})$ property if and only if C_i may obtain the value of all X_k in $\mathit{Range}(X)$ in at least one compatible execution trace. $\mathit{Has}_i^{none}(\tilde{X})$ holds if and only if no execution trace can lead to a state in which C_i gets any value of any X_k. We note that Has_i properties only inform on the fact that C_i can get or derive some values for the variables but they do not bring any guarantee about the correctness of these values. Integrity requirements can be expressed using the property $K_i(T_1 = T_2)$, which states that the component C_i knows the truth of the integrity property $T_1 = T_2$. In $\sigma \geq \sigma'$, compared to σ', σ represents the state at the end of a longer trace. Finally, $\sigma_i^{pk} \triangleright_i Eq'$ and $Eq' \Rightarrow Eq$ capture that Eq' can be deduced from σ_i^{pk} and Eq', respectively.

Example Architecture: Let us consider a very simple smart metering architecture which consists in the communication between two components: the meter (M) and the operator (O). The goal of this architecture is to ensure that the operator will get the consumption fee for a given period and to be convinced that the fee is correct. The privacy requirement says that O must not obtain the consumption data. One possible design solution is that the meter passes directly the consumption data to the operator who will compute the fee:

$$\mathcal{A}_1 = \{\text{for } i \in [1, \ldots, r]\text{: } \mathit{Has}_M^{arch}(X_c);\ \mathit{Compute}_M(X_{m_i} = X_{c_i});$$
$$\mathit{Receive}_{O,M}(\mathit{Attest}_M(X_{m_i} = X_{c_i}),\ X_{m_i});\ \mathit{Verif}_O^{Attest}(\mathit{Attest}_M(X_{m_i} = X_{c_i}));$$
$$\mathit{Compute}_O(X_{tf_i} = F(X_{m_i});\ \mathit{Compute}_O(X_{fee} = \odot + (X_{tf})),\ \mathit{Trust}_{O,M}\}.$$

In the architecture $\mathcal{A}_1$, the meter initially has the (input) variable X_c that represents the array of r consumption data X_{c_i}, $i \in [1, \ldots, r]$. The meter is capable to compute each metered data (X_{m_i}) based on each consumption data (X_{c_i}). Intuitively, in $X_{m_i} = X_{c_i}$, X_{m_i} will get the value of X_{c_i} . Then, the operator will receive the metered data (X_{m_i}), along with the attestation made by M on the integrity property $X_{m_i} = X_{c_i}$. After verifying the received attestation with success, due to $\mathit{Trust}_{O,M}$ the operator knows that $X_{m_i} = X_{c_i}$. Then for each X_{m_i}, O computes the tariff based on the function F. Finally, O computes the summation of the r tariffs (i.e., array X_{tf}) to get the fee for the period. The requirements of the architecture are modeled with the properties of the architecture logic. Namely, Has_O^{all} (X_{fee}) specifies that O has (all) the fee, while Has_O^{none} (X_c) says that O must not have any consumption data. $\mathcal{A}_1$ fulfills the first requirement, but it does not satisfy the privacy requirement because based on $\mathit{Receive}_{O,M}(\mathit{Attest}_M(X_{m_i} = X_{c_i}\ ),\ X_{m_i})$ O can obtain X_{c_i} from X_{m_i}.

3 Protocol Level

To reason about the concrete implementations of an architecture, we propose a modified variant of the applied π-calculus. We decided to modify the basic applied π-calculus [13] because thanks to its expressive syntax and semantics, it is broadly used for security verification of systems and protocols (e.g., [14], [10], [11], [18], [9], [4], [17]). Our main goal is to modify some syntax and semantics elements of the applied π-calculus, making it more convenient to find the connection between the calculus semantics and the interpretation of architecture relations. One of such modifications is the notion of component, which is characterized by three elements: (i) the internal behavior of the component; (ii) the unique ID assigned to the component; and (iii) the set of IDs of the components who are trusted by this component. Another reason why we cannot use the basic applied π-calculus is that it focuses on reasoning about the information a Dolev-Yao attacker (who can eavesdrop on all communications) obtains. However, in our case we reason about the information that components can have, which are only aware of the communications they can take part in.

3.1 Syntax of the Modified Applied π-Calculus

We assume an infinite set of *names* $\mathcal{N}$ and *variables* $\mathcal{V}$, and a finite set of *component identifiers* $\mathcal{L}$, where $\mathcal{V} \cap \mathcal{N} \cap \mathcal{L} = \emptyset$. Terms are defined as follows:

$$t \ ::= \ c \ \mid \ l_i \ \mid \ n, m, k \ \mid \ x, y, z \ \mid \ f(t_1, \ldots, t_p).$$

The meaning of each term is given as follows: c models a communication channel. l_i represents a component ID ($l_i \neq l_j$ if $i \neq j$) that uniquely identifies a

component. n, m and k denote names, which model some kind of data (e.g., a random nonce, a secret key, etc.). Terms x, y, z denote variables that represent any term, namely, any term can be bounded to variables. $f(t_1, \ldots, t_p)$ is a function, which models cryptographic primitives, e.g., digital signature can be modeled by $sign(x_m, x_{sk})$, where x_m and x_{sk} specify the message and the private key, respectively. Moreover, f can also be used to specify verification functions (e.g., the signature check is modeled by function $checksign(sign(x_m, x_{sk}), x_{pk})$, where x_{pk} represents the public key corresponding to the private key x_{sk}).

We rely on the same type system for terms as in the applied π-calculus [13]. Due to lack of space, we omit the unimportant details of this type system, and leave it implicit in the rest of the paper. We assume that terms are well-typed and that substitutions preserve types (see [22] for details).

The internal operation of components is modeled by *processes*. Processes are specified with the following syntax:

$$P, Q, R ::= \overline{c}\langle t\rangle.P \;\|\; \overline{c}\langle t_m, t_{sig}\rangle.P \;\|\; c(x).Q \;\|\; c(x_m, x_{sig}).Q \;\|\; P|Q \;\|\; \nu n.P \;\|\; let\ x = t\ in\ P \;\|\; if\ (t_1 = t_2)\ then\ P \;\|\; \mathbf{0}.$$

Note that for simplicity we left out the infinite replication of processes, $!P$. As a result a protocol/system run consists of a finite number of traces.

Process $\overline{c}\langle t\rangle.P$ sends the term t (where $t \neq (t_m,\ t_{sig})$) on channel c, and continues with the execution of P. Process $\overline{c}\langle t_m, t_{sig}\rangle.P$ models the attestation sending, where t_m and the signature t_{sig} are sent on c.

Process $c(x).Q$ waits for a term on channel c and then binds the received term to x in Q. Process $c(x_m, x_{sig}).Q$ waits for a term x_m and its signature x_{sig} on channel c, which models the attestation reception.

$P|Q$ behaves as processes P and Q running (independently) in parallel. A restriction $\nu n.P$ is a process that creates a new, bound name n, and then behaves as P. The name n is called bound because it is available only to P. Process *let* $x = t$ *in* P proceeds to P and binds every (free occurrence of) x in P to t.

Process *if* $(t_1 = t_2)$ *then* P says that if $t_1 = t_2$ (with respect to the equational theory E, discussed later) then process P is executed, else it stops. Its special case is *if* $x_m = checksign(x_{sig}, x_{l_i}^{pk})$ *then* P, which captures the verification of an attestation (i.e., signature x_{sig} with key $x_{l_i}^{pk}$). For message authentication and integrity protection purposes digital signature and message authentication code (MAC) are used. In this paper we only consider signature.

Finally, the *nil* process $\mathbf{0}$ does nothing and specifies process termination.

Components: To make the connection between calculus processes and architecture relations more straightforward, we introduce the notion of components. $\lfloor P \rfloor_l^\rho$ defines a component with the unique identifier l, who trusts the components whose IDs are in the set ρ, and whose behavior is defined by process P. The trust relation can be either one-way or symmetric, for instance, $\lfloor P \rfloor_{l_1}^{\{l_2\}}$ and $\lfloor Q \rfloor_{l_2}^{\{l_1\}}$ represent components l_1 and l_2 who trust each other. The rationale behind this way of component specification is that the component IDs and the trust relation between them are pre-defined, and do not change during the pro-

tocol run (this is what we assumed at the architecture level). In addition, we assume that a trusted component will not become untrusted.

Systems: A *system*, denoted by S, can be an empty system with no component: $\mathbf{0}_S$; a singleton system with one component: $\lfloor P \rfloor_l^{\emptyset}$; the parallel composition of components: $\lfloor P \rfloor_{l_1}^{\rho_1} \mid \lfloor Q \rfloor_{l_2}^{\rho_2}$, where ρ_1 and ρ_2 may include l_2 and l_1, respectively; or a system with name restriction. To capture more complex systems, we also allow systems to be the parallel composition of sub-systems, $S_1 \mid S_2$.

$$S ::= \mathbf{0}_S \mid \lfloor P \rfloor_l^{\rho} \mid \nu n.S \mid (S_1 \mid S_2).$$

The name restriction $\nu n.S$ represents the creation of new name n, such as secret keys, or a random nonce which are only available to the components in S.

3.2 Semantics of the Modified Applied π-Calculus

In order to check the conformance between protocols and architectures, it suffices to consider the internal reduction rules of the calculus, which model the behavior of the protocol (without contact with its environment). Reduction rules capture the internal operations (e.g., *let* or *if* processes) and communications performed by components. We define and distinguish the following reduction rules:

(Reduction rules)

(Rcv) $\lfloor \overline{c}\langle t \rangle.P \rfloor_{l_i}^{\rho_i} \mid \lfloor c(x).Q \rfloor_{l_j}^{\rho_j} \xrightarrow{rcv(l_j, l_i, x:t)} \lfloor P \rfloor_{l_i}^{\rho_i} \mid \lfloor Q\{t/x\} \rfloor_{l_j}^{\rho_j}$, $t \neq (t_m, t_{sig})$;

(Rcv$_{att}$) $\lfloor \overline{c}\langle t_m, t_{sig} \rangle.P \rfloor_{l_i}^{\rho_i} \mid \lfloor c(x_m, x_{sig}).Q \rfloor_{l_j}^{\rho_j} \xrightarrow{rcv_{att}(l_j, l_i, x_m:t_m)} \lfloor P \rfloor_{l_i}^{\rho_i} \mid \lfloor Q\{t_m/x_m, t_{sig}/x_{sig}\} \rfloor_{l_j}^{\rho_j}$;

(Verif$_{att}$) $\lfloor \mathit{if}\ x_m = \mathit{checksign}(x_{sig}, t_{l_i}^{pk})\ \mathit{then}\ Q' \rfloor_{l_j}^{\rho_j} \xrightarrow{ver_{att}(l_j,\ x_m:t_m)} \lfloor Q' \rfloor_{l_j}^{\rho_j}$, where $\{t_m/x_m, t_{sig}/x_{sig}\}$. Note: l_j accepts the attestation if $l_i \in \rho_j$.

(Check) $\lfloor \mathit{if}\ (t_1 = t_2)\ \mathit{then}\ P \rfloor_{l_j}^{\rho_j} \xrightarrow{check(l_j, t_1:t_2)} \lfloor P \rfloor_{l_j}^{\rho_j}$ $(t_1 = t_2 \in E,\ t_2 \neq \mathit{checksign})$;

(Comp) $\lfloor \mathit{let}\ x = t\ \ \mathit{in}\ P \rfloor_{l_j}^{\rho_j} \xrightarrow{\omega_{comp}} \lfloor P\{t/x\} \rfloor_{l_j}^{\rho_j}$, $(t = x'$ or f, such that $\omega_{comp} = \mathit{comp}(l_j, x:t)$ when $f \notin \{\mathit{sign}, \mathit{checksign}\}$, else $\omega_{comp} = \tau)$;

(Has) $(\nu k.)\ \lfloor \mathit{let}\ x = k\ \mathit{in}\ P \rfloor_{l_j}^{\rho_j} \xrightarrow{has(l_j, x:k)} (\nu k.)\ \lfloor P\{k/x\} \rfloor_{l_j}^{\rho_j}$,

(Error) $\lfloor \mathit{if}\ (t_1 = t_2)\ \mathit{then}\ P \rfloor_{l_j}^{\rho_j} \xrightarrow{error} \lfloor \mathbf{0} \rfloor_{l_j}^{\rho_j}$ $(\mathit{if}\ t_1 = t_2 \notin E)$;

(Par-C) $\lfloor P \rfloor_{l_j}^{\rho_j} \xrightarrow{\omega_c} \lfloor P' \rfloor_{l_j}^{\rho_j}$ then $\lfloor Q \mid P \rfloor_{l_j}^{\rho_j} \xrightarrow{\omega_c} \lfloor Q \mid P' \rfloor_{l_j}^{\rho_j}$;

(Res-C) $\lfloor P \rfloor_{l_j}^{\rho_j} \xrightarrow{\omega_c} \lfloor P' \rfloor_{l_j}^{\rho_j}$ then $\lfloor \nu n.P \rfloor_{l_j}^{\rho_j} \xrightarrow{\omega_c} \lfloor \nu n.P' \rfloor_{l_j}^{\rho_j}$, where $\omega_c \in \{\mathit{comp}(l_j, x:t), \mathit{has}(l_j, x:k), \mathit{check}(l_j, t_1:t_2), \mathit{error}, \mathit{ver}_{att}(l_j, x_m{:}t_m)\}$;

(Par-S) $S_1 \xrightarrow{\omega_s} S_1'$ then $S_2 \mid S_1 \xrightarrow{\omega_s} S_2 \mid S_1'$;

(Res-S) $S \xrightarrow{\omega_s} S'$ then $\nu n.S \xrightarrow{\omega_s} \nu n.S'$, where
ω_s can be ω_c, and $rcv(l_j, l_i, x : t)$, $rcv_{att}(l_j, l_i, x_m : t_m)$.

Before defining the system states, we label each reduction relation (arrow) based on the name of the rule and the terms used in them. We adopt the notion of equational theory E from [13], [22], which contains rules of form $t_1 = t_2 \in E$, that define when two terms are equal. For instance, the equational theory E may include rules for signature verification, decryption, MAC verification, etc. The meaning of each reduction rule is as follows:

- Rule (Rcv) captures the communication between components l_i and l_j. Namely, l_i sends value t for x on channel c, which is received by l_j. As a result, we get $Q\{t/x\}$ that binds t to every free occurrence of x in Q. It is assumed that $t \neq (t_m, t_{sig})$, which is treated as a special case.
- Rule (Rcv_{att}) deals with exchanging the message t_m and the signature t_{sig} on channel c, which models the reception of the attestation $Attest_{l_i}(\{x_m = t_m\})$. As a result, we get Q in which t_m and the signature are bound to x_m and x_{sig}, respectively. The reason that we distiguish (Rcv_{att}) from (Rcv) is because we want to make a clear distinction between the cases of receiving a message with and without an attestation.
- Rule (Verif_{att}) captures the case when after binding t_m and t_{sig} to x_m and the signature x_{sig}, respectively, component l_j successfully verified the signature using the corresponding public key of l_i, $t^{pk}_{l_i}$. We implicitly assume that t_m and t_{sig} contain enough information for the receiver to identify the "type" of the received message (e.g., the consumption fee in smart metering systems). Rules (Rcv_{att}) and (Verif_{att}) together specify the scenario when l_i sends to l_j the value t_m for x_m, with the signature that proves the integrity and the authenticity of this message. Then, in case l_j trusts l_i, it knows the truth about the integrity property $x_m = t_m$.
- Rule (Check) considers the case when two terms are equal in the check (with respect to the equational theory E), which leads to P as result. We assume that t_2 is not the *checksign* function, which is used for the attest verification.
- Rule (Comp) models the computation $x = t$ performed by l_j. As a result, every free occurrence of x in P is given the value t. In this rule we assume that t can be either a variable or a function (except for *sign* and *checksign*, because they are considered as parts of the attestation), but not a name.
- Rule (Has) deals with the case when t is a name. The name k, either bound (with νk) or free (without νk), represents the value of x. Here x is used to model the variable that l_j initially has to capture the input data coming from the environment (e.g., the consumption data in the smart metering).
- Rule (Error) specifies the case when two terms are not equal with respect to E. As a result, the process will continue with the *nil* process.
- Rules (Par-C) and (Res-C) say that the *if* and *let* reductions are closed under parallel composition and restriction within a component, respectively. Rules (Par-S) and (Res-S) capture that all the reductions are closed under parallel composition and restriction on systems, respectively.

Instead of referring to the trace of reductions $\xrightarrow{\omega_s^1} \ldots \xrightarrow{\omega_s^m}$ we will refer to the trace of the corresponding labels $\omega_s^1, \ldots, \omega_s^m$ for the sake of clarity.

States of components and systems: Let us consider a system S with n components. Let $Label_S$ be the set of all labels ($\omega_s \in Label_S$) of the reduction relations defined above, and let $LTrace_S$ be the set of all possible label traces of S. We define the functions $\mathcal{V}_{ST}$, $\mathcal{V}_T$ and $\mathcal{V}_L$ that update the states of the components and the entire system. $\mathcal{V}_T$ and $\mathcal{V}_L$ are similar to $\mathcal{S}_T$ and $\mathcal{S}_E$ in PAL, but they are based on label traces and labels instead of traces of events and events. $\mathcal{V}_{ST}$ takes as input the set of all the possible label traces of S and handle each trace with $\mathcal{V}_T$. Let $State_S^n$ denote the set of global state of S and $\emptyset_{tr}$ an empty set of traces. Finally, in $\omega_s.tr$ the label ω_s is the prefix of the trace tr.

$\mathcal{V}_{ST} : \{LTrace_S\} \times State_S^n \rightarrow State_S^n$;
$\mathcal{V}_T : LTrace_S \times State_S^n \rightarrow State_S^n$;
$\mathcal{V}_{ST}(LTrace_S, \lambda) = \mathcal{V}_{ST}\ (LTrace_S \backslash \{tr\}, \mathcal{V}_T(tr, \lambda))$;
$\mathcal{V}_T(\omega_s.tr, \lambda) = \mathcal{V}_T(tr, \mathcal{V}_L(\omega_s, \lambda))$;
$\mathcal{V}_L : Label_S \times State_S^n \rightarrow State_S^n$;
$\mathcal{V}_{ST}(\emptyset_{tr}, \lambda) = \lambda$;
$\mathcal{V}_T\ (\langle\rangle, \lambda) = \lambda$;
$\mathcal{V}_L(has(l_i, x : k), \lambda) = \lambda[(\lambda_{l_i}^v\{k/x\}, \lambda_{l_i}^{pk})\ /\ \lambda_i]$;
$\mathcal{V}_L(rcv(l_i, l_j, x : t), \lambda) = \lambda[(\lambda_{l_i}^v\{t/x\}, \lambda_{l_i}^{pk})\ /\ \lambda_i]$
$\mathcal{V}_L(rcv_{att}(l_i, l_j, x_m : t_m), \lambda) = \lambda[(\lambda_{l_i}^v\{t_m/x_m\}, \lambda_i^{pk})\ /\ \lambda_{l_i}]$
$\mathcal{V}_L(comp(l_i, x : t), \lambda) = \lambda[(\lambda_{l_i}^v\{\lambda_{l_i}^v t/x\}, \lambda_{l_i}^{pk} \cup \{x = t\})\ /\ \lambda_{l_i}]$
$\mathcal{V}_L(check(l_i, t_1 : t_2), \lambda) = \lambda[(\lambda_{l_i}^v, \lambda_{l_i}^{pk} \cup \{t_1 = t_2\})\ /\ \lambda_{l_i}]$ if $\lambda_{l_i}^v t_1 = \lambda_{l_i}^v t_2 \in E$
$\mathcal{V}_L(ver_{att}(l_i,\ x_m : t_m), \lambda) = \lambda[(\lambda_{l_i}^v, \lambda_{l_i}^{pk} \cup \{\{x_m = t_m\}$ if $Trust_{l_i,l_j} \in \lambda_{l_i}^{pk}\})\ /\ \lambda_i]$.

We let λ_{l_i} and λ denote the state of component l_i and the global state that consists in the state of all components in the system, respectively. Each λ_{l_i} is defined by the pair $(\lambda_i^v, \lambda_{l_i}^{pk})$, which is the variable state and the property state for component l_i, respectively. In our calculus the variable state $\lambda_{l_i}^v$ is defined by the set of substitutions $\{t_1/x_1, \ldots, t_m/x_m\}$, which captures the terms available to l_i, as well as the values of each variable from the perspective of l_i. $\lambda_{l_i}^v\{t/x\}$ is a shorthand for $(\lambda_{l_i}^v \cup \{t/x\}) \setminus \{t'/x\}$, if $\{t'/x\} \in \lambda_{l_i}^v$ for some t'. λ_i^{pk} is the set of integrity properties (e.g., $t_1 = t_2$) that captures the knowledge gained by l_j about these properties. $\lambda_i^v t$ represents the evaluation of t based on λ_i^v, and $\lambda_{l_i}^v t_1 = \lambda_{l_i}^v t_2 \in E$ says that the evaluation of t_1 and t_2 in $\lambda_{l_i}^v$ are equal up to the equational theory E. We also consider the state update that results after a failed check (namely, $\lambda[\lambda_{Err}/\lambda_i]$, where λ_{Err} denotes the error state), though we omit the formal details here to save space.

4 From Protocols to Architectures

In the sequel, we discuss how the corresponding architecture can be extracted based on a given protocol or system. Namely, given a protocol specified in our process calculus we define an extraction procedure that extracts the corresponding architecture relations. The extraction procedure is based on the application of a set of extraction rules that we define below. Each extraction rule specifies the connection between the traces of labels of a system and the corresponding architecture relation. We assume a (initial) system S which consists in the parallel

composition of r components (for a finite r), namely, $S \stackrel{def}{=} \lfloor P_1 \rfloor_{l_1}^{\rho_1} \mid \ldots \mid \lfloor P_r \rfloor_{l_r}^{\rho_r}$. The corresponding architecture relations will be extracted based on the possible traces (i.e., the trace semantics) of S. We emphasize that during the extraction of architectural properties we only consider the reduction traces to capture the communication between components, without considering the activity of the environment (i.e., the Dolev-Yao attacker). Formally, we do not take into account the labelled transitions known in the applied π-calculus [13]. The reason is that the architecture relations focus only on the abilities of the components and the communication between them.

An architecture does not not contain the *Compute* relations for background computations. The situation is similar at the protocol level, where the protocol description specifies the basic computations and communications of the components, without involving the background computations. Hence, when extracting the architecture relations, it is sufficient to consider only the protocol description and its corresponding reduction traces. The background computations will be taken into account when we discuss the mapping to the Has_j architecture logic property for reasoning about the data that can be deduced by a component.

Given a system S and the set $LTrace_S$ of (all) its possible label traces, we define the extraction function $\mathcal{X}_T$ that extracts the corresponding architecture based on $LTrace_S$. $\mathcal{X}_{ST}$ is interpreted similarly as $\mathcal{V}_{ST}$. Rel_S denotes the set of architectural relations of S. Function $\mathcal{X}_L$ extracts a relation based on a label ω_s and put it into α_S. We use α_S to denote the set of the extracted relations so far. We have the following extraction rules:

$\mathcal{X}_{ST} : \{LTrace_S\} \times Rel_S \to Rel_S$;
$\mathcal{X}_T : LTrace_S \times Rel_S \to Rel_S$; $\quad \mathcal{X}_L : Label_S \times Rel_S \to Rel_S$;
$\mathcal{X}_{ST}(LTrace_S, \alpha_S) = \mathcal{X}_{ST}(LTrace_S \backslash \{tr\}, \mathcal{X}_T(tr, \alpha_S))$; $\quad \mathcal{X}_{ST}(\emptyset_{tr}, \alpha_S) = \alpha_S$;
$\mathcal{X}_T(\omega_s.tr, \alpha_S) = \mathcal{X}_T(tr, \mathcal{X}_L(\omega_s, \alpha_S))$; $\quad \mathcal{X}_T\ (\langle\rangle, \alpha_S) = \alpha_S$;
R^{has}: $\mathcal{X}_L(has(l_j, x : k), \alpha_S) = \alpha_S \cup \{Has_{l_j}^{arch}(x)\}$;
R^{recv}: $\mathcal{X}_L(rcv(l_j, l_i, x : t), \alpha_S) = \alpha_S \cup \{Receive_{l_j,l_i}(x)\}$;
R_{att}^{recv}: $\mathcal{X}_L(rcv_{att}(l_j, l_i, x_m : t_m), Compute_{l_i}(x_m = t_m) \in \alpha_S) =$
$\alpha_S \cup \{Receive_{l_j,l_i}(\{Att\}, x_m)\}$, where $Att = Attest_{l_i}(\{x_m = t_m\})$;
R^{comp}: $\mathcal{X}_L(comp(l_j, x : t), \alpha_S) = \alpha_S \cup \{Compute_{l_j}(x = t)\}$, where $t \notin \{sign, checksign\}$;
R^{check}: $\mathcal{X}_L(check(l_j, t_1 : t_2), \alpha_S) = \alpha_S \cup \{Check_{l_j}(t_1 = t_2)\}$ if $t_1 = t_2 \in E$;
R^{attver}: $\mathcal{X}_L(ver_{att}(l_j,\ x_m : t_m), \{Trust_{l_j,l_i}, Receive_{l_j,l_i}(\{Att\}, x_m)\} \subseteq \alpha_S) =$
$\alpha_S \cup \{Verif_{l_j}^{Attest}(Att)\}$, where $Att = Attest_{l_i}(\{x_m = t_m\})$;

All the rules above capture the communication and computation abilities of each component during the protocol run and are defined based on the trace semantics. In contrast, the $Trust_{l_i,l_j}$ relations are extracted based on the syntax. The initial set of relations is $\alpha_S^{init} = \{Trust_{l_i,l_j}$ if $l_j \in \rho_i \mid \forall\ l_i, l_j \in \{l_1, \ldots, l_r\}\}$. The meaning of each rule is defined as follows:

- Rule R^{has} corresponds to the relation $Has_{l_j}^{arch}(x)$, which says that l_j initially has a value for x. The name k represents an input data for x of l_j.
- R^{recv} extracts the relation $Receive_{l_j,l_i}(x)$, and describes the case when l_j receives a value t for x during the protocol run. S' and S'' represent the systems before and after the communication between l_i and l_j. S' involves the possibility for l_j to receive the variable x.

- R_{att}^{recv} extracts the relation $Receive_{l_j,l_i}(Attest_{l_i}(\{x_m = t_m\}), x_m)$, where $\{x_m = t_m\}$ contains $x_m = t_m$, along with all the equations $x_g = x_h$ computed by l_i in order to constitute t_m. Intuitively, besides attesting $x_m = t_m$, l_i attests the integrity of all the computations it performed in order to get x_m. S' includes the possibility for l_j to receive x_m, and its signature x_{sig}. Assumption $Compute_{l_i}(x_m = t_m) \in \alpha_S$ captures the fact that l_i is able to compute $x_m = t_m$, hence, it can make an attestation on this equation.
- Rule R^{comp} corresponds to the relation *Compute*, for the equations $x = f$ or $x = x'$. We do not extract the computations for signature and its verification since these computations are integrated within the *Attest* relation.
- Rule R^{check} extracts the relation *Check*. To be able to check an equation, a component must have the ability to perform the function required in the check and it should possess the required data during the protocol run. This is determined by the equational theory E, which defines the checking abilities of a component. We do not extract *Check* for signature check because it is considered as an attestation verification.
- Rule R^{attver} deals with the case when component l_j successfully verified the attestation sent by component l_i. However, we get the corresponding relation $Verif_{l_j}^{Attest}(Att)$ only in case $l_i \in \rho_j$ (i.e., l_j trusts l_i). The assumption $Receive_{l_j,l_i}(Att, x_m) \in \alpha_S$ captures the fact that l_j has received (Att, x_m).

The extraction procedure starts with the initial system S, then we follow the possible reduction traces from S and apply the extraction rules where possible. Although during the extraction every possible label trace of the system is examined, due to the simplifications we made on the processes (e.g., infinite process replication is leftout), the number of traces is finite, hence, the extraction procedure will terminate. In the sequel, we let $\mathcal{A}_S$ denote the extracted architecture of S (i.e., the set of relations α_S when we have examined all the possible traces).

Definition 1 (***State based semantics***) *The state based semantics of a given system is defined as* $\mathcal{V}(S) = \{\lambda \in State_S^n \mid \exists\ tr \in T(S),\ \mathcal{V}_T(tr, Init^S) = \lambda\}$.

$Init^S$ is the initial state of the system S which contains only the *Trust* relations in λ^{pk}. We adopt the *Has* properties used in PAL (Section 2), and define their semantics based on the semantics of the calculus.

$$\psi ::= \ Has_{l_i}^{all}(x) \mid Has_{l_i}^{none}(x) \mid K_{l_i}(t_1 = t_2) \mid \psi_1 \wedge \psi_2$$

Definition 2 (***Semantics of property*** ψ ***for systems***)

1. $S \in \mathcal{V}(Has_{l_i}^{all}(x)) \Leftrightarrow \exists\ \lambda \in \mathcal{V}(S)$: $\exists\ t'$ *and* t *such that* $(\lambda_{l_i}^v t' = t) \in E$, *where* $BoundTo(t) = x$
2. $S \in \mathcal{V}(Has_{l_i}^{none}(x)) \Leftrightarrow \forall\ \lambda \in \mathcal{V}(S)$ *and* $\forall\ t \in terms(\lambda_{l_i}^v)$: $\nexists\ t'$ *such that* $(\lambda_{l_i}^v t' = t) \in E$, *where* $BoundTo(t) = x$
3. $S \in \mathcal{V}(K_{l_i}(Eq)) \Leftrightarrow \forall\ \lambda' \in \mathcal{V}_{l_i}(S)\ \exists\ \lambda \in \mathcal{V}_{l_i}(S),\ \exists\ Eq'$: $(\lambda \geq_{l_i} \lambda') \wedge (\lambda_{l_i}^{pk} \triangleright_E Eq') \wedge (Eq' \Rightarrow_E Eq)$.

S satisfies the property $Has_{l_i}^{all}(x)$ when during the system run, l_i can deduce or obtain a value t for x. $(\lambda_{l_i}^{v} t' = t) \in E$ means that l_i can deduce t based on $\lambda_{l_i}^{v} t'$ and the equational theory E. $BoundTo(t) = x$ captures the fact that this t has been bounded to x during the reduction trace (i.e., t is the value of x). S satisfies $Has_{l_i}^{all}(x)$ when l_i cannot deduce or obtain any value t for x. Finally, the deduction $\lambda_{l_i}^{pk} \triangleright_E Eq'$ and $Eq' \Rightarrow_E Eq$ are defined on the deduction system based on the equational theory E.

To compare $\mathcal{A}_S$ and $\mathcal{A}$, we define $\mathcal{E}$, the set of *type-preserved* mappings from terms in the calculus to the terms in PAL: $\mathcal{E} = \{l_i \mapsto i;\ x \mapsto \tilde{X};\ f(t_1, \ldots, t_m) \mapsto F(T_1, \ldots, T_m);\ f(x_1, \ldots, x_m) \mapsto \odot F(X),\ X = [X_1, \ldots, X_m]\}$. It is important to emphasize that in each mapping, the result and its preimage must have the same type. Defining an explicit type system for terms is not in the scope of this paper. Here, we only provide general type matching requirements for the mapping rules, giving the reader an intuition about the mapping to understand the definitions given below. In $\mathcal{E}$, each ID l_i can be mapped to an ID i; each x can be mapped to a $\tilde{X}$ of the same type. $f(t_1, \ldots, t_m)$ can be mapped to $F(T_1, \ldots, T_m)$ if each (t_j, T_j) pair has the same type, and the two functions return the same type, too. Similarly, $f(x_1, \ldots, x_m)$ can be mapped to $\odot F(X)$ if they return the same type, and the array X contains m variables, such that each corresponding variable pair has the same type. In the sequel, we let $\mathcal{E}\mathcal{A}_S$ denote the application of the mapping $\mathcal{E}$ to the architecture $\mathcal{A}_S$.

The property 1 discusses the connection between a system S and its extracted architecture $\mathcal{E}\mathcal{A}_S$ with respect to the *Has* and K logical properties (ϕ and ψ).

Property 1 (***Correctness of the mapping***) *Given a system S and its extracted architecture $\mathcal{E}\mathcal{A}_S$, for some $\mathcal{E}$, we have that $\forall\ x, l_i, t_1, t_2$ in S and $\forall\ \tilde{X}, i, T_1, T_2$ in $\mathcal{E}\mathcal{A}_S$, where $\{x \mapsto \tilde{X}, l_i \mapsto i, t_1 \mapsto T_1, t_2 \mapsto T_2\} \in \mathcal{E}$: **1.** $S \in \mathcal{V}(Has_{l_i}^{all}(x))$ iff $\mathcal{E}\mathcal{A}_S \in \mathcal{S}(Has_i^{all}(\tilde{X}))$; **2.** $S \in \mathcal{V}(Has_{l_i}^{none}(x))$ iff $\mathcal{E}\mathcal{A}_S \in \mathcal{S}(Has_i^{none}(\tilde{X}))$; and **3.** $S \in \mathcal{V}(K_{l_i}(t_1 = t_2))$ iff $\mathcal{E}\mathcal{A}_S \in \mathcal{S}(K_i(T_1 = T_2))$.*

The first point of Property 1 says that if the system S satisfies $Has_{l_i}^{all}(x)$, then the extracted architecture $\mathcal{E}\mathcal{A}_S$ of S satisfies $Has_i^{all}(\tilde{X})$, and vice versa. The second point is related to the privacy requirement capturing that when $\mathcal{E}\mathcal{A}_S$ satisfies $Has_i^{none}(\tilde{X})$, the system S satisfies $Has_{l_i}^{none}(x)$, and vice versa. The third point is related to the integrity property stating that if in the system S component l_i knows $t_1 = t_2$, then in the extracted architecture this component knows the corresponding $T_1 = T_2$, and vice versa. The proof is based on the semantics of the architecture and the state based semantics of the systems, as well as the correspondence between the deduction rules of the privacy logic and the equational theory E of the calculus.

We give two conformance definitions between protocol and architecture, a *strong* one and a *weak* one.

Definition 3 (***Strong Conformance***) *Let us consider a system S and an architecture $\mathcal{A}$. We say that S strongly conforms to $\mathcal{A}$ up to $\mathcal{E}$ ($S \models_{\mathcal{E}}^{s} \mathcal{A}$) if $\exists\ \mathcal{E}$ such that $\mathcal{E}\mathcal{A}_S = \mathcal{A}$.*

In the strong case, we require that there exists a mapping $\mathcal{E}$ such that $\mathcal{E}\mathcal{A}_S$ contains exactly the same relations as $\mathcal{A}$.

Definition 4 (***Weak Conformance***) *Let us consider a system S and an architecture $\mathcal{A}$. We say that S weakly conforms to $\mathcal{A}$ up to $\mathcal{E}$ (denoted $S \models^{w}_{\mathcal{E}} \mathcal{A}$) if* (***i.***) *$\exists\ \mathcal{E}$ such that $\mathcal{A} \subset \mathcal{E}\mathcal{A}_S$, and* (***ii.***) *$\forall\ x, \tilde{X}$ such that $\{x \mapsto \tilde{X}\} \in \mathcal{E}$: If $\mathcal{A} \in \mathcal{S}\ (Has^{none}_{i}(\tilde{X}))$ then $S \in \mathcal{V}(Has^{none}_{l_i}(x))$.*

Point (i.) of the weak case requires the more relaxed $\mathcal{A} \subset \mathcal{E}\mathcal{A}_S$. Point (ii.) says that for every $\tilde{X}$ in the privacy requirement $Has^{none}_{j}(\tilde{X})$ of the architecture, l_j cannot have any value t for x in the system S (where $\{x \mapsto \tilde{X}\} \in \mathcal{E}$).

Next, we provide the state simulation and bisimulation definitions in order to formulate Properties 2 and 3 about the relationship between the states of a system and states of an architecture in case of weak and strong conformance.

Definition 5 (***State simulation***)*: Let us consider a system S and an architecture $\mathcal{A}$. We say that $\lambda \in \mathcal{V}(S)$ simulates $\sigma \in \mathcal{S}(\mathcal{A})$ up to $\mathcal{E}$ (denoted by $\lambda \sqsubseteq_{\mathcal{E}} \sigma$), if $\forall\ l_i, x, t_1, t_2$ in S, and $\forall\ i, \tilde{X}, T_1, T_2$ in $\mathcal{A}$, such that $\{l_i \mapsto i, x \mapsto \tilde{X}, t \mapsto T, t_1 \mapsto T_1, t_2 \mapsto T_2\} \in \mathcal{E}$:*

- *if $\exists\ \sigma[(\sigma^{v}_{i}[V/\tilde{X}], \sigma^{pk}_{i})\ /\ \sigma_i] \in \mathcal{S}(\mathcal{A})$, then $\exists\ \lambda[(\lambda^{v}_{l_i} \cup \{t/x\}, \lambda^{pk}_{l_i})\ /\ \lambda_{l_i}] \in \mathcal{V}(S)$*
- *if $\exists\ \sigma[(\sigma^{v}_{i}[eval(T, \sigma^{v}_{i})/\tilde{X}], \sigma^{pk}_{i} \cup \{\tilde{X} = T\})\ /\ \sigma_i] \in \mathcal{S}(\mathcal{A})$, then $\exists\ \lambda[(\lambda^{v}_{l_i} \cup \{\lambda^{v}_{l_i} t/x\}, \lambda^{pk}_{l_i} \cup \{x = t\})\ /\ \lambda_{l_i}] \in \mathcal{V}(S)$, and*
- *if $\exists\ \sigma[(\sigma^{v}_{i}, \sigma^{pk}_{i} \cup \{T_1 = T_2\})\ /\ \sigma_i] \in \mathcal{S}(\mathcal{A})$, then $\exists\ \lambda[(\lambda^{v}_{l_i}, \lambda^{pk}_{l_i} \cup \{t_1 = t_2\})\ /\ \lambda_{l_i}] \in \mathcal{V}(S)$.*

Also, we write $\lambda \sqsubseteq^{(\tilde{X},x)}_{\mathcal{E}} \sigma$ if λ simulates σ up to $\mathcal{E}$, but with respect to only the pair $(\tilde{X}, x)$, where $\{x \mapsto \tilde{X}\} \in \mathcal{E}$.

Each point of Definition 5 captures the state simulation that results from the corresponding architecture relations. For example, the second point says that if $\exists\ Compute_i(\tilde{X} = T) \in \mathcal{A}$ then $\exists\ Compute_{l_i}(\tilde{X} = T) \in \mathcal{E}\mathcal{A}_S$.

Definition 6 (***State bisimulation***)*: Given a system S and an architecture $\mathcal{A}$:*

1. *We say that $\lambda \in \mathcal{V}(S)$ and $\sigma \in \mathcal{S}(\mathcal{A})$ simulate each other up to $\mathcal{E}$ ($\lambda \simeq_{\mathcal{E}} \sigma$), if $\lambda \sqsubseteq_{\mathcal{E}} \sigma$ and $\sigma \sqsubseteq_{\mathcal{E}} \lambda$.*
2. *We say that $\lambda \in \mathcal{V}(S)$ and $\sigma \in \mathcal{S}(\mathcal{A})$ simulate each other up to $\mathcal{E}$ and the variable pair $(\tilde{X}, x)$, $\lambda \simeq^{(\tilde{X},x)}_{\mathcal{E}} \sigma$, if $\lambda \sqsubseteq^{(\tilde{X},x)}_{\mathcal{E}} \sigma$ and $\sigma \sqsubseteq^{(\tilde{X},x)}_{\mathcal{E}} \lambda$.*

Property 2 *Given a system S and an architecture $\mathcal{A}$, where $\lambda \in \mathcal{V}(S)$ and $\sigma \in \mathcal{S}(\mathcal{A})$. We have that $S \models^{s}_{\mathcal{E}} \mathcal{A}$ iff $\lambda \simeq_{\mathcal{E}} \sigma$.*

Property 2 says that when S strongly conforms to $\mathcal{A}$, then the states of l_j in S simulates the states of the corresponding component j in $\mathcal{A}$, and vice versa.

Property 3 *Given a system S and an architecture $\mathcal{A}$, where $\lambda \in \mathcal{V}(S)$ and $\sigma \in \mathcal{S}(\mathcal{A})$. We have that $S \models^{w}_{\mathcal{E}} \mathcal{A}$ iff* (***i.***) *$\lambda \sqsubseteq_{\mathcal{E}} \sigma$ and* (***ii.***) *$\lambda \simeq^{(\tilde{X},x)}_{\mathcal{E}} \sigma$, for all $\tilde{X}$ in $Has^{none}_{j}(\tilde{X})$.*

Property 3 says that in case S weakly conforms to $\mathcal{A}$, then the states of l_j simulates the states of the corresponding component j, and these states are bisimilar for all the variable pair $(x, \tilde{X})$, such that $Has_j^{none}(\tilde{X})$ holds. A consequence of Properties 2 and 3 is that it is sufficient to show the state simulation and bisimulation to prove the weak and strong conformance properties. These two properties also capture the correctness of the mapping with respect to the weak and strong conformance definitions. The proof of Properties 2 and 3 is based on the defined extraction rules and the correspondence between functions $\mathcal{S}_E$ of the architecture and $\mathcal{V}_L$ of the system.

Example Conformance Check: We check the conformance between an example protocol and the architecture $\mathcal{A}_1$ at the end of Section 2, with $r = 1$. Let us consider the protocol description in which there are components l_M and l_O that refer to the meter and operator, respectively. The behavior of the meter is specified by the process R_M. The operator is defined by the process R_O.

$$R_M \stackrel{def}{=} let\ x_{c_1} = k_1\ \ in\ P_1; \qquad P_1 \stackrel{def}{=} let\ x_{m_1} = x_{c_1}\ in\ P_2;$$
$$P_2 \stackrel{def}{=} let\ x_{sig} = sign(x_{m_1}, sk_m)\ in\ P_3; \qquad P_3 \stackrel{def}{=} \overline{c_{mo}}\langle x_{m_1}, x_{sig}\rangle.\ \mathbf{0}.$$
$$R_O \stackrel{def}{=} c_{mo}(x_{m_1}, x_{sig}).\ \mathbf{0}. \qquad S \stackrel{def}{=} \lfloor R_M \rfloor_{l_M} \mid \lfloor R_O \rfloor_{l_O}^{l_M}.$$

Due to lack of space, we use this very simple example to demonstrate the mapping procedure and the conformance check between S and $\mathcal{A}_1$. The initial relations set α_S^{init} is $\{Trust_{l_O,l_M}\}$. The architecture relations corresponding to S can be extracted in the following steps: From the two reductions

$S \stackrel{has(l_M,\ x_{c_1}:k_1)}{\longrightarrow} \lfloor P_1 \rfloor_{l_M} \mid \lfloor R_O \rfloor_{l_O}^{l_M} \stackrel{comp(l_M,\ x_{m_1}:x_{c_1})}{\longrightarrow} \lfloor P_2 \rfloor_{l_M} \mid \lfloor R_O \rfloor_{l_O}^{l_M}$ and rules R^{has}, R^{comp} we have $\alpha_S = \alpha_S^{init} \cup \{Has_{l_M}^{arch}(x_{c_1})\} \cup \{Compute_{l_M}(x_{m_1} = x_{c_1})\}$. The *let*-process in P_2 has no effect on the extraction, while the channel synchronization will result in adding $Receive_{l_O,l_M}(\{x_{m_1} = x_{c_1}\}, x_{m_1})$ to α_S. Since R_O terminates right after receiving the attestation, the two $Compute_{l_O}$ relations and $Verif_{l_O}^{Attest}(Attest_{l_M}(\{x_{m_1} = x_{c_1}\}))$ cannot be extracted. This means that the system S does not conform to the architecture $\mathcal{A}_1$.

5 Related Works

Dedicated languages have been proposed to specify privacy properties (e.g., [5], [7], [16]) but they are complex and not intended to be used at the architectural level. In [2, 3] the authors addressed the idea of applying formal methods to architecture design and proposed a simple privacy architecture language (PAL).

On the other hand, there are also many works focusing mainly on the protocol level, providing formal methods for specifying and verifying protocols, as well as reasoning about the security and privacy properties (e.g., [19], [21], [8]). For this purpose, process algebra languages are the most favoured means in the literature, because they are general frameworks to model concurrent systems.

In addition, among the process algebras, the applied π-calculus ([22], [13]) is one of the most promising language in the sense that its syntax and semantics

are more expressive than the others (e.g., [20], [1], [15]). It also have been used to analyse security and privacy protocols (e.g., in [14], [10], [11], [18], [9], [4], [17]). However, we cannot use it directly for our purpose because for instance, it lacks syntax and semantics for modelling component IDs and trust relations. Some modifications and extensions of the applied π-calculus are required, which we proposed in Section 3.

Finally, the definition of the architecture comes before the definition of the protocol in software development cycles. Therefore, we chose to make it possible to verify the conformance of a protocol described in our language to an architecture. We used the architecture language in [2] for this purpose. Its main advantage is that (i) compared to informal pictorial methods, or semi-formal representations such as UML diagrams, it is more formal and precise, while (ii) compared to process calculi, it is more abstracted. The architecture language PAL enables designers to reason at the level of architectures, providing ways to express properties without entering into the details of specific protocols.

6 Conclusions and Future Works

In this paper, we proposed the application of formal methods to privacy by design. We provided the mapping from the protocol level to the architecture level for checking if a given implementation conforms to an architecture and showed its correctness. For this purpose, we modified the applied π-calculus and defined the connection between the semantics of the calculus and PAL. To the best of our knowledge, this is the first attempt at examining the connection between the protocol and the architecture levels in the privacy protection context.

The calculus version and the mapping procedure we proposed in this paper are based on a simplified version of the architecture language. Indeed, we only consider the attestation on equation $\tilde{X} = T$. Moreover, our proposed calculus (and mapping) does not support the modelling of zero-knowledge proofs, as well as the posibility of spot-checks used in toll pricing systems. Hence, our method can only handle simple architectures and protocols at this stage. One future direction of our work is to extend the calculus to support these such extensions.

Acknowledgements. The authors would like to thank Daniel Le Métayer for his initial idea and valuable comments during this work. This work is partially funded by the European project PARIS/FP7-SEC-2012-1, the ANR project BIO-PRIV, and the Inria Project Lab CAPPRIS.

References

1. Abadi, M., Gordon, A.: A calculus for cryptographic protocols: the Spi calculus. Tech. Rep. SRC RR 149, Digital Equipment Corp., Systems Research Center (1998)
2. Antignac, T., Le Métayer, D.: Privacy architectures: Reasoning about data minimisation and integrity. In: Proc. of The 10th International Workshop on Security and Trust Management, STM'14. pp. 1–16 (2014)

3. Antignac, T., Le Métayer, D.: Privacy by design: From technologies to architectures - (position paper). In: Annual Privacy Forum (APF). pp. 1–17 (2014)
4. Backes, M., Maffei, M., Unruh, D.: Zero-knowledge in the applied pi-calculus and automated verification of the direct anonymous attestation protocol. In: IEEE Symposium on Security and Privacy, Proc. of SSP'08. pp. 202–215 (May 2008)
5. Barth, A., Datta, A., Mitchell, J., Nissenbaum, H.: Privacy and contextual integrity: framework and applications. In: Security and Privacy, 2006 IEEE Symposium on. pp. 15 pp. –198 (may 2006)
6. Bass, L., Clements, P., Kazman, R.: Software Architecture in Practice. SEI series in Software Engineering, Addison-Wesley, 3rd edn. (September 2012)
7. Becker, M.Y., Malkis, A., Bussard, L.: A practical generic privacy language. Information Systems Security 6503, 125–139 (2011)
8. Burrows, M., Abadi, M., Needham, R.: A logic of authentication. ACM Trans. Comput. Syst. 8, 18–36 (February 1990)
9. Delaune, S., Kremer, S., Ryan, M.: Verifying privacy-type properties of electronic voting protocols. Journal of Computer Security 17(4), 435–487 (Dec 2009)
10. Delaune, S., Ryan, M.D., Smyth, B.: Automatic verification of privacy properties in the applied pi-calculus. In: IFIPTM'08: 2nd Joint iTrust and PST Conferences on Privacy, Trust Management and Security. vol. 263, pp. 263–278. Springer (2008)
11. Dong, N., Jonker, H.L., Pang, J.: Analysis of a receipt-free auction protocol in the applied pi calculus. In: Formal Aspects in Security and Trust. pp. 223–238 (2010)
12. Fagin, R., Halpern, J.Y., Moses, Y., Vardi, M.: Reasoning About Knowledge. MIT Press, paperback edn. (jan 2004)
13. Fournet, C., Abadi, M.: Mobile values, new names, and secure communication. In: In Proceedings of the 28th ACM Symposium on Principles of Programming, POPL'01. pp. 104–115 (2001)
14. Fournet, C., Abadi, M.: Hiding names: Private authentication in the applied pi calculus. In: Software Security, Theories and Systems, Lecture Notes in Computer Science, vol. 2609, pp. 317–338. Springer Berlin Heidelberg (2003)
15. Hoare, C.A.R.: Communicating sequential processes. Communications of the ACM 21(8), 666–677 (Aug 1978)
16. Jafari, M., Fong, P.W., Safavi-Naini, R., Barker, K., Sheppard, N.P.: Towards defining semantic foundations for purpose-based privacy policies. In: Proceedings of the first ACM conference on Data and application security and privacy. pp. 213–224. CODASPY '11, ACM, New York, NY, USA (2011)
17. Kremer, S., Ryan, M.: Analysis of an electronic voting protocol in the applied pi calculus. In: In Proc. 14th European Symposium On Programming (ESOP'05), volume 3444 of LNCS. pp. 186–200. Springer (2005)
18. Li, X., Zhang, Y., Deng, Y.: Verifying anonymous credential systems in applied pi calculus. In: Proc. of the 8th International Conference on Cryptology and Network Security. pp. 209–225. CANS '09, Springer-Verlag, Berlin, Heidelberg (2009)
19. Meadows, C.: Formal methods for cryptographic protocol analysis: emerging issues and trends. Selected Areas in Communications, IEEE 21(1), 44 – 54 (jan 2003)
20. Milner, R., Parrow, J., Walker, D.: A calculus of mobile processes, parts i and ii. Information and Computation (September 1992)
21. Paulson, L.C.: The inductive approach to verifying cryptographic protocols. Journal of Computer Security 6(1-2), 85–128 (January 1998)
22. Ryan, M.D., Smyth, B.: Cryptology and Information Security Series, vol. 5, chap. Applied pi calculus, pp. 112–142 (2011)