A Preserving-Privacy Synchronization Protocol in Actor Models

Mahboubeh Samadi, Fatemeh Ghassemi, and Ramtin Khosravi

University of Tehran, Tehran, Iran
{mbh.samadi,fghassemi,r.khosravi}@ut.ac.ir

Abstract. In actor-based models, sensitive data can be easily inferred from the sequences of messages, and hence, the privacy of data is violated. To protect data privacy against such inferences, a synchronization protocol should be used to coordinate actors in a distributed way to avoid the formation of the sequences violating the privacy. However, the current synchronization mechanisms proposed for asynchronous settings are not sufficient in this regard. We introduce a synchronization protocol based on a set of primitives which are not supported by the classical actor model. We extend the actor-based model with communication mechanisms while each actor is equipped with two queues with different priorities. Then, we automatically convert a given actor-based model to one whose communication primitives are replaced by our synchronization protocol. We illustrate the applicability of our approach through a case study, and prove its correctness.

Keywords: Data privacy, Actor-based models, Coordination, Inference attack

1 Introduction

An actor model is a mathematical model of concurrent computation in distributed systems, which abstract away from the network by considering its effects [11, 12] (see Fig. 1). Actors are known as asynchronous agents of concurrent computations which communicate through message passing to provide the functional requirements of the system. Each actor has a mailbox to maintain messages, received from other actors, and process them based on the order of their delivery. In this model, the order of messages is not preserved due to the different network delays over different paths, but their delivery is guaranteed. For each message type, actors may contain a message handler which specifies how the actor responds to. Each message handler, also called *method*, is executed atomically with the assumption that the computational time is negligible in comparison with the network delay. Upon processing a message, actors can make three types of decisions: updates their local state, create new actors, and send new messages.

In addition to functional requirements, there is often quality of service (QOS) requirements such as security and privacy, which should be preserved in an actor

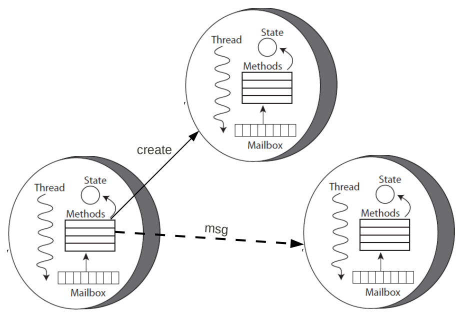

Fig. 1. Each actor has its own thread of control, mailbox, and address [9].

model. By privacy, we aim to preserve the system from the inference attacks of an adversary. In other words, an adversary, which is either honest-but-curious (HBC) [1] or malicious such as Dolev-Yao [2], should not be able to infer sensitive information from non-sensitive information. One source of inferences in the asynchronous setting is the message sequences. For instance, an adversary may conclude from receiving the message sequence $\langle \mathit{requset\_price}, \mathit{inform\_info}, \mathit{buy} \rangle$ that the sender of *requset_price* owns more than X in his account (as he wants to buy) using information of the domain. Although all the participating actors are permissible to send such messages or access data, such inferences cannot prohibited when a legitimate participant acts as an HBC attacker unless we prevent from the sequence formation. There are several approach to overcome this challenge: one may consider a central monitoring actor who is aware of all sequences leading to an inference, and it receives a copy of all messages to control the message flows in a way to avoid a sequence formation. However, this solution is not feasible in actor models due to delay of network: the message which completes a sequence may be received too late by the monitor. Therefore, some *synchronization* mechanism should be used to ensure receipt of a message by the central monitor. This will decrease the efficiency of the system. Another approach is that the senders of messages involved in a sequence, be aware of the sequence and collaborate using the mentioned synchronization mechanism to protect data privacy in a distributed way. However, the current synchronization mechanisms proposed for asynchronous settings are not sufficient in this regard.

We propose a synchronization protocol which requires essential primitives, such as blocking and a special message communication, which are not supported by the classical actor model. Therefore, we extend the classical actor-based model with a communication mechanism while each actor is equipped with two queues with different priorities. The messages in the queue with the lower priority are not processed unless the other queue is empty. Furthermore, actors can wait for receiving a set of messages. A waiting actor can still handle its queue with the higher priority. Then, we automatically convert a given actor-based model to one whose communication primitives are substituted by our synchronization protocols, implemented by our language. To illustrate the applicability of our approach, we implement our actor model in a toy language, so-called PPlang. We show the effectiveness of our synchronization protocol through a case study in smart home, and then prove its correctness.

2 Related Work

There are many work that mention to the inference control problem for data privacy.

In [3, 4] the inference problem is reduced to the access control problem. In [3], accesses to sensitive data are controlled by considering the role of users and inference constraints. A new access control model is proposed in [4] which prevents from inferences about sensitive data caused by semantic relations between data. However, these approaches are not feasible to avoid inferences from the message sequences in asynchronous settings.

One way to coordinate actors in a distributed system is using synchronization mechanisms implemented either by a set of message passing or a special mediator. Two types of commonly used communication patterns for synchronization in asynchronous systems are introduced in [9]: Remote Procedural Call (RPC)-like messaging and local synchronization constraints. In the RPC-like messaging, the sender of a message waits for its reply to arrive before processing other messages [9]. As indicated by [9], the RPC-like communications are not useful to prevent from inferring sensitive information from the sequence of messages. So, the local synchronization constraint, which implicitly implements a synchronization algorithm, is introduced. By a local synchronization constraint, an actor waits until a specific message arrives and then processes it. Meanwhile the received messages are buffered until that specific message is arrived and processed, and then buffered messages will be processed. Another synchronization mechanism is *selector*s, introduced in [8] for asynchronous systems. The selectors are actors extended with multiple guarded mailboxes whose guards can be enabled or disabled to affect the order in which messages are processed [8]. Each actor specifies the intended mailbox to which it is going to send. For a synchronization purpose, messages should be sent to a mailbox whose guard is true. By synchronizing actors through selectors, inferring a private information from the sequence of messages can't be prevented.

3 Problem Statement

Privacy is defined as the claim of individuals, groups or institutions to determine when, how and to what extent information about themselves is communicated to others [5].

People may have some information for disclosure to the public while their sensitive information must be protected from being disclosed to unwanted parties. The unwanted party is usually an attacker or intruder that attempts to retrieve some information illegitimately. However, a HBC adversary [1] is a legitimate participant of the model that attempts to learn more possible information from legitimately received messages. In contrast, the passive Dolev-Yao eavesdrops on communication to glean more information based on all the messages transferred in the model [2].

In this paper, we focus on preserving privacy requirements in the presence of a HBC attacker in actor models through appropriate synchronization mechanisms. We restrict to those privacy requirements which are violated by inference from the sequence of messages. Thus, the privacy requirements can be specified by a set of message sequence, so-called *partial message sequence*s, which should not be formed. Let *Msg* be the set of messages communicated between actors, ranged over by $\mathfrak{m}$.

Definition 1 (Partial Message Sequence). *A partial message sequence, ranged over by ω, is a sequence of the messages and the special character of $*$, representing any sequence of messages, i.e., $\omega \in (Msg \mid *)^*$.*

Due to the asynchronous spirit of communications in actor models, the occurrences of messages from which an attacker can infer, are not consecutive and there will be some other messages between any two messages, denoted by $*$ in Definition 1. Let $\mathbb{P}(Msg)$ denote the powerset of all partial message sequences over *Msg*. Thus, each privacy requirement $r \in \mathbb{P}(Msg)$ constitutes a set of partial message sequences where each sequence specifies an inference attack from the point view of a HBC attacker. We use the notation $\langle \mathfrak{m}_0 \ldots \mathfrak{m}_n \rangle$ to specify a sequence, ω_i to denote the i^{th} element of the sequence, and $len(\omega)$ to show the length of a sequence. Furthermore, we say $\mathfrak{m} \in \omega$, if there exists $j \leq len(\omega)\,(\omega_j = \mathfrak{m})$. We write $\mathfrak{m} \in r$ if there exists $\omega \in r$ such that $\mathfrak{m} \in \omega$.

Each message $\mathfrak{m} \in Msg$ constitutes of three parts, namely the sender identifier, the message type, the receiver identifier. Thus, $Msg = ID \times Mtd \times ID$, where *ID* and *Mtd* are the set of identifiers and message types, respectively. We remark that *Mtd* is the set of the method names of actors. We restrict to those privacy requirements that the partial message sequence of a message $\mathfrak{m}$ is uniquely detectable in terms of the sender of its previous message in the sequence. Let $PreSender(\omega, \mathfrak{m})$ denote the sender identifiers of the messages preceding $\mathfrak{m}$ in the sequence ω:

$$\begin{aligned} PreSender(\omega, \mathfrak{m}) = \{id \mid \forall i > 0\,(\omega_i = \mathfrak{m} \wedge \exists\, j < i\,(\omega_j = (id, m', id') \wedge \\ \forall\, j < k < i\,(\omega_k = *))\} \end{aligned}$$

This notation is lifted to the set of partial message sequences in the usual way. For privacy requirement r, $PreSender(r, \mathfrak{m}) = \bigcup_{\omega \in r} PreSender(\omega, \mathfrak{m})$.

Definition 2 (Uniquely Detectable). *A set of partial message sequences $r \in \mathbb{P}(Msg)$ is called uniquely detectable if for all two distinct message sequences $\omega, \omega' \in r$ and for all $\mathfrak{m} \in \omega$ and $\mathfrak{m} \in \omega'$, then $PreSender(\omega, \mathfrak{m}) \cap PreSender(\omega', \mathfrak{m}) = \emptyset$ and $PreSender(\omega, \mathfrak{m}) \neq PreSender(\omega', \mathfrak{m})$.*

This definition guarantees that every two distinct partial messages sequences in a privacy requirement do not have any subsequence in common. Hence, each actor uniquely recognizes the sequence that will be formed by sending message $\mathfrak{m}$ by getting a confirmation from one of its pre-senders and, consequently it can decide if it can send the message $\mathfrak{m}$ or not.

We also assume that all the messages belong to a workflow of the system and hence, all messages are interrelated. This assumption is easily achieved by stamping related messages with the same number.

4 Motivating Example

As revealed in [6], in the smart grids, customers are equipped with solid-state electronic meters which collect time-based consumption data at daily, hourly, or sub-hourly intervals. These meters then transmit the collected data to the grid operator. Based on the collected data, the grid operator can determine high-consuming customers and penalize them. In some situation, the grid operator can collect some specific data and infer some private information about customers and threats the customer privacy. As it was explained by [7], the smart grid is bringing new nontraditional players into the energy-consumption market. Some players such as law enforcement, marketing, and nefarious individuals are interested in inferring the private information from time-based consumption data. Criminals use such data to facilitate burglary, marketers to initiate targeted advertising based on the activities occurring wholly within the home, and law enforcement to monitor home activities in real-time. In [6], certain algorithms for inferring the activity parameters (such as the absence/presence of the household, the sleep/wake cycle) from power-consumption data were presented.

For the sake of simplicity, we modify the absence/presence algorithm presented in [6]. Suppose each customer's house has three electronic meters corresponding to three lighting lamps located in the kitchen, living room and bedroom. So from the grid operator point of view, it is possible to infer the absence of the household from the sequences of events like: bedroom-lamp-off, kitchen-lamp-off and living room-lamp-off. To model this system, the meters and the grid operator are considered as the actors that communicate through message passing. The grid operator also is modeled as a HBC adversary whose malicious intention is to infer about the presence of people in the house.

5 Synchronization Protocol

We introduce a synchronization protocol to coordinate actors to prevent forming the set of partial message sequence in each execution of the system. In this protocol, it is assumed that messages are either a processing or monitoring one.

Each actor, before sending message $\mathfrak{m}$, finds all the sequences in $r \subseteq \mathbb{P}(Msg)$ which include that message. If the message belongs to one or more sequence in r, then the actor should perform these steps before sending the message:

1. It sends a monitoring message to each pre-sender of $\mathfrak{m}$ and asks them if they has previously send the preceding messages.

2. It waits until their responses are received.
3. If the preceding message of $\mathfrak{m}$ in the sequence $\omega \in r$ had been sent, then it checks this condition:
   – If its message is the last message of the sequence, it should perform an appropriate action.
   – If the message is not the last message of the sequence, it is sufficient to save the message and then sends it.
4. The pre-sender waits until the actor replies to its message. In this situation the pre-sender actor doesn't process any messages except monitoring messages.

The appropriate action in the step 3 depends on the modeler. If the system provides a model of design-time, an error action can be generated. So, by model checking the system, we provide the modeler a scenario which leads to the sequence formation. If the system is a run-time model, the message will not be sent.

The actor which receives a monitoring message performs following steps:

1. It checks its message history to find out if the message indicated by the monitoring message, has been previously sent or not.
2. Based on the message history, it replies to the sender of the monitoring message if it is the legitimate post-sender of the message.
3. The receiver will be blocked until it receives the ack message from the sender.

Following example shows that the privacy requirements will be violated if the receiver is not blocked after responding to the sender.

Example: Consider the actors A, B, C and D and the privacy requirement r:

$$\{\langle (A,1,B) * (C,2,B) * (D,3,B)\rangle, \\ \langle (A,4,D) * (B,5,D) * (C,6,D)\rangle\}$$

Assume that the actor D wants to send a message with method identifier 3 to the actor B (i.e., $(D,3,B)$). Based on the proposed algorithm, the actor D should perform these steps:

1. It sends a monitoring message to its pre-sender (i.e., C) to examine if its preceding message (i.e., $(C,2,B)$) was sent or not.
2. It waits until its response will be received.
3. Suppose that the actor C replies to the actor D that the message $(C,2,B)$ hasn't been sent.

In this situation, the actor D decides to send the message $(D,3,B)$. If the actor C isn't blocked, it is possible that before D resumes its processing to send the message $(D,3,B)$, the actor C sends the message $(C,2,B)$. So the sequence $\langle (A,1,B) * (C,2,B) * (D,3,B)\rangle$ will be appeared in the state space of the system model and so the privacy requirement r will be violated.

6 Extended Actor Model

The proposed synchronization protocol in Section 5 cannot be implemented in the classical model or using the synchronization mechanisms like the local synchronization constraint [9] or selectors [8] (see Section 2). So, we identify a set of essential primitives (e.g., blocking) by which we extend the classical actor model. Then, we introduce a toy actor-based language which acts as an operational interpretation of our extended model.

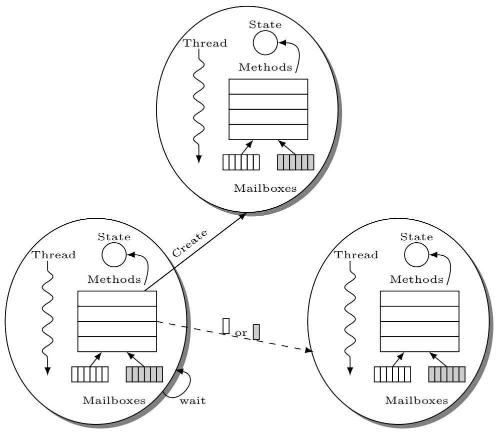

Fig. 2. Extended actor model.

In the extended actor model, each actor is equipped with two queues with different priorities, called *processing* and *monitoring* queues. (see Fig. 2). The monitoring queue has a higher priority in compared to the processing queue. The first identified primitive is sending to the monitoring queue. The other primitive is blocking the actor until it receives a set of message.

Actors may be blocked while they are processing their messages (from each of their queue) if they are not previously blocked. However, they can handle their monitoring queues when they are blocked. The messages of the processing queue is not handled unless the monitoring queue is empty. Therefore, message handlers are not executed atomically if they contain a blocking statement. We remark that when an actor is blocked, other non-blocked actors can be executed. Upon processing a message, actors can make four types of decisions: updates their local state, create new actors, send new messages, wait for a set of messages.

Rebeca is an operational interpretation of the actor model that is to say it is an actor-based language with formal semantics and model checking tools developed to fill the gap between formal methods and software engineering [10]. Palang is a toy actor modeling language used to study various aspects of actor systems. Palang is based on Rebeca, but has omitted a number of features to make it simpler.

We extend Palang to Private Palang (PPalang) to implement our extended actor model. We assume that the distributed system is modeled by Palang. Palang models preserve the functional requirements of the system, but their privacy requirements are not satisfied.

$$
\begin{aligned}
\text{Model} &::= \text{Actor}^{+}\ \mathsf{main}\ \{\text{Send}^{+}\ \} \\
\text{Actor} &::= \mathsf{actor}\ \mathit{ID}\ \{\ (\mathit{Type}\ \ \mathit{Var})^{*}\ \text{Methods}^{*}\ \} \\
\text{Methods} &::= \ \mathit{Mtd}(\text{List}(\mathit{Type}]\ \ \mathit{Var}))\ \{\ \text{Statement}^{*}\ \} \\
\text{List(X)} &::= \langle X, \rangle^{*} X \mid \epsilon \\
\text{Statement} &::= \ \mathit{Var} = \mathit{Expr};\ \mid \text{Conditional} \mid \text{Send} \mid \mathsf{wait}(\text{List}(\mathit{Mtd})); \\
\text{Conditional} &::= \mathsf{if}\ (\mathit{Expr})\ \{\ \text{Statement}^{*}\ \}\ \mathsf{else}\ \{\ \text{Statement}^{*}\ \} \\
\text{Send} &::= \langle \mathsf{self} \mid \mathit{ID} \rangle \langle ! \mid !! \rangle \mathit{Mtd}(\text{List}(\mathit{Expr}));
\end{aligned}
$$

Fig. 3. PPalang language syntax: Angle brackets (⟨ ⟩) are used as meta-parentheses. Superscript + is used for more than one repetition, and * indicates zero or more times repetition. The symbols *ID*, *Type*, *Mtd*, and *Var* denote the set of actor names, types, method and variable names, respectively. The symbol *Expr* denotes an arithmetic expression.

6.1 PPalang: Syntax

We introduce PPlang model which supports the implementation of our privacy-preserving protocol. PPalang syntax is similar to Palang with two differences: PPalang has two separate actions for sending messages, each targets different mailboxes of the receiver; PPlang has a *wait* statement which blocks the actor until a set of specified message are received.

The grammar of PPlang is presented in Fig. 3. It consists of two major parts: actors and main part which specify actor declarations and initial messages to the actors, respectively. Each actor has a name, a set of state variables, and methods. The body of methods is defined by a set statements such as variable assignment, sending statement, and blocking statement. To send message m, its receiver r should be defined by executing $r!m()$ or $r!!m()$, where the former sends to the processing mailbox of the actor and the latter to the monitoring mailbox of the receiver. An actor can send a message to itself by using keyword *self*. Fig. 4 shows our motivating example specified in Palang.

A given PPlang model is called *well-formed* if no state variable is redefined in the scope of a message server, no two state variables, methods or actor names have identical names, identifiers of variables, methods and actors do not clash, and all message communications and variable accesses occur over declared/specified ones and the number and type of actual parameters correctly match the formal ones in their corresponding message server specifications.

```
actor meterB {
 blamp(){
  grid! bloff ();
  meterK!turnoff2();}
}
actor meterK {
 turnoff2(){
  grid! kloff ();
  meterH!turnoff3();}
}
actor meterH {
 turnoff3(){
  grid! hloff ();}
}

actor grid {
 bool blamp;
 bool klamp;
 bool hlamp;

 bloff(){
  blamp=true; }
 kloff() {
  klamp=true; }
 hloff(){
  hlamp=true;
  meterB!blamp();}
}
main {
 meterB!blamp();
}
```

Fig. 4. The Plang model of the smart home example.

6.2 Operational Semantics

The formal semantics of PPalang is described in terms of labeled transition systems. A labeled transition system (LTS), is defined by the quadruple $\langle S, \rightarrow, L, s_0\rangle$ where S is a set of states, $\rightarrow \subseteq S \times L \times S$ a set of transitions, L a set of labels, and s_0 the initial state. For the sake of simplicity, we assume that methods have no parameter and message queues are unbounded.

For a non-empty sequence of elements D like $\rho \in D^*$, we use $\langle d|\rho'\rangle$ to denote that its head and tail are $d \in D$ and $\rho' \in D^*$, respectively. Furthermore, we write $\rho \oplus \rho'$ to denote the sequence achieved by appending ρ' to the end of ρ. We use $set(\ell)$ to convert a list of elements $d_1, d_2, \ldots$ to a set of elements in D, i.e., $\{d_1, d_2, \ldots\}$.

Let ID denote the set of actor identifiers ranged over by x, y, Var the set of variables ranged over by z, Val the set of all possible values for the variables, and Mtd the set of method names ranged over by m.The valuation function $Var \rightarrow Val$, ranged over by v, is used to assign values to the state variables. Let $eval(expr, v)$ denote the value of expression $expr$ in the context of variable evaluation v, and $v[z := e]$ the variable evaluation identical to v except that z is assigned to e. The processing and monitoring queues of an actor are defined by Mtd^*. The auxiliary functions $body(x, m)$ denote the body of method m of an actor with the identifier x. We assume that the set of $Type$ consists of integer, boolean, $Statement^*$, and $\mathbb{P}(Mtd)$ data types. We consider the default value 0, $false$, ϵ and $\emptyset$ for the integer, boolean, sequence and set variables, respectively.

To give semantics of an actor model, we assume each actor is specified by $\langle x, Var_x, Mtd_x\rangle$, where x is the actor identifier, Var_x its set of state variables and Mtd_x its set of methods. We assume that each actor has four variables $blocked : bool$, $wait : \mathbb{P}(Mtd)$, $comd : Statement^*$, and $self$ by default. The variable $blocked$ is used to check if the actor has been blocked in the middle of processing a message. The set *wait* shows the set of the messages an actor is still waiting to receive, and *comd* denotes the set of statements that the actor should execute after resuming its processing.

States : The global state of a PPalang is represented by a function $s : \mathit{ID} \rightarrow (\mathit{Var} \rightarrow \mathit{Val}) \times \mathit{Mtd}^* \times \mathit{Mtd}^* \times \mathit{Statement}^*$, which maps an actor's identifier to its local state. The local state of the actor $\langle x, \mathit{Var}_x, \mathit{Mtd}_x\rangle$ is defined by a quadruple like $(\upsilon, q_p, q_m, \sigma)$ where $\upsilon : \mathit{Var}_x \rightarrow \mathit{Val}$ defines the values of the state variables such that $\upsilon(\mathit{self}) = x$, q_p and q_m specify the processing and monitoring mailboxes of the actor, and finally $\sigma : \mathit{Statement}^*$ contains the sequence of statements the actor is going to execute to finish the service to the message currently being processed.

Transitions : The transitions occur as the results of actors' activities including: taking a message from its mailboxes, and resuming the processing of a message. The transition relation $\rightarrow \subseteq S \times L \times S$ is the least relation satisfying the semantic rules in Table 1.

Initial State: In the initial state, all state variables have their default values, denoted by υ_0, and the messages specified by the send actions in the main block are put in the corresponding actors' mailboxes: $s_0(x) = (\upsilon_0, q_p(x, \sigma), q_m(x, \sigma), \epsilon)$, where $q_p(x, \sigma)$ and $q_m(x, \sigma)$ construct the initial mailbox of actor x from the sequence of send statements σ (from the main block). Function q_p is defined by equations $q_p(x, \epsilon) = \epsilon$, $q_p(x, \langle x!m|\sigma'\rangle) = \langle m|q_p(x, \sigma')\rangle$ and $q_p(x, \langle y!!m(e)|\sigma'\rangle) = q_p(x, \sigma')$ (q_m is defined similarly while considering only the send actions to the monitoring queue).

Label : Each transition will be labeled by taking message m from one of the queues of the actor x or resuming execution of the blocked actor x: $L = \{(x, \mathit{take}(m)), (x, \mathit{resume}) \mid m \in \mathit{Mtd}, x \in \mathit{ID}\}$.

7 Generating Privacy-Preserving PPlang models

A given model specified in Palang and a privacy requirement r, one can substitute our synchronization protocol for its communications to achieve a privacy-preserving model specified in PPlang. This process can be mechanized by a pre-processing step, before executing the Palang model.

To this aim, each actor $\langle x, \mathit{Var}_x, \mathit{Mtd}_x\rangle$ of the given model is modified to $\langle x, \mathit{Var}'_x, \mathit{Mtd}'_x\rangle$ where $\mathit{Var}'_x = \mathit{Var}_x \cup \{\mathit{history\_m\_z}, \mathit{knowledge\_y\_m\_z} \mid y, z \in \mathit{ID} \wedge m \in \mathit{Mtd}\} \cup \{\mathit{formed}, \mathit{end}\}$ and Mtd_x is extended with the set of below methods, as defined in the Fig. 5:

$$\{\mathit{ask\_y\_m\_z} \mid \forall m' \in \mathit{Mtd}_y,\, \forall (x, m, z) \in \mathit{PreSenderMsg}(r, (y, m', z))\}$$
$$\{\mathit{tell\_y\_m\_z} \mid \forall m' \in \mathit{Mtd}_x,\, \forall (y, m, z) \in \mathit{PreSenderMsg}(r, (x, m', z))\}$$
$$\{\mathit{ack\_y} \mid \forall m' \in \mathit{Mtd}_y\, (x \in \mathit{PreSender}(r, (y, m', z')))\}$$

where $\mathit{PreSenderMsg}(r, \mathfrak{m})$ denotes the set of messages preceding $\mathfrak{m}$ in the privacy requirement r:

$$\mathit{PreSenderMsg}(r, \mathfrak{m}) = \{(\mathit{id}, m', \mathit{id}') \mid \forall \omega \in r, \forall i > 0\, (\omega_i = \mathfrak{m} \wedge \exists j < i\, (\omega_j = (\mathit{id}, m', \mathit{id}') \wedge \forall j < k < i\, (\omega_k = *))\}.$$

Table 1. PPlang natural semantic rules: σ, ζ, and T denotes a sequence of statements, a list of method names, and a sequence of methods names, respectively.

$$\mathit{Assign}\quad \frac{s(x) = (\upsilon, q_p, q_m, \langle z = \mathit{expr} | \sigma \rangle)}{s \leadsto s[x \mapsto (\upsilon[z := \mathit{eval}(\mathit{expr}, \upsilon)], q_p, q_m, \sigma)]}$$

$$\mathit{Cond}_1\quad \frac{s(x) = (\upsilon, q_p, q_m, \langle \mathsf{if}\ (\mathit{expr})\ \sigma\ \mathsf{else}\ \sigma' | \sigma'' \rangle) \wedge \mathit{eval}(\mathit{expr}, \upsilon)}{s \leadsto s[x \mapsto (\upsilon, q_p, q_m, \sigma \oplus \sigma'')]}$$

$$\mathit{Cond}_2\quad \frac{s(x) = (\upsilon, q_p, q_m, \langle \mathsf{if}\ (\mathit{expr})\ \sigma\ \mathsf{else}\ \sigma' | \sigma'' \rangle) \wedge \neg \mathit{eval}(\mathit{expr}, \upsilon)}{s \leadsto s[x \mapsto (\upsilon, q_p, q_m, \sigma' \oplus \sigma'')]}$$

$$\mathit{Snd}_m\quad \frac{s(x) = (\upsilon, q_p, q_m, \langle y!!m | \sigma \rangle) \wedge s(y) = (\upsilon', q'_p, q'_m, \sigma')}{s \leadsto s[x \mapsto (\upsilon, q_p, q_m, \sigma)][y \mapsto (\upsilon', q'_p, q'_m \oplus \langle m \rangle, \sigma')]}$$

$$\mathit{Snd}_p\quad \frac{s(x) = (\upsilon, q_p, q_m, \langle y!m | \sigma \rangle) \wedge s(y) = (\upsilon', q'_p, q'_m, \sigma')}{s \leadsto s[x \mapsto (\upsilon, q_p, q_m, \sigma)][y \mapsto (\upsilon', q'_p \oplus \langle m \rangle, q'_m, \sigma')]}$$

$$\mathit{Wait}\quad \frac{s(x) = (\upsilon, q_p, q_m, \langle \mathit{wait}(\zeta) | \sigma \rangle) \wedge \upsilon(\mathit{wait}) = \emptyset}{s \leadsto s[x \mapsto (\upsilon[\mathit{blocked} := \mathit{true}][\mathit{wait} := \mathit{set}(\zeta)][\mathit{comd} := \sigma], q_p, q_m, \epsilon)]}$$

$$\mathit{Take}_p\quad \frac{\begin{array}{c} s(x) = (\upsilon, \langle (m | T \rangle, \epsilon, \epsilon) \wedge \neg \upsilon(\mathit{blocked}) \wedge \\ s'(x) = (\upsilon', q'_p, q'_m, \epsilon) \wedge \forall i \in \mathit{ID} \setminus \{x\} (s(i) = (\upsilon'', q''_p, q''_m, \epsilon) \wedge s'(i) = (\upsilon'', q^*_p, q^*_m, \epsilon)) \\ s[x \mapsto (\upsilon, T, \epsilon, \mathit{body}(x, m))] \leadsto s' \end{array}}{s \xrightarrow{(x, \mathit{take}(m))} s'}$$

$$\mathit{Take}_m\quad \frac{\begin{array}{c} s(x) = (\upsilon, q_p, \langle m | T \rangle, \epsilon) \wedge \\ s'(x) = (\upsilon', q'_p, q'_m, \epsilon) \wedge \forall i \in \mathit{ID} \setminus \{x\} (s(i) = (\upsilon'', q''_p, q''_m, \epsilon) \wedge s'(i) = (\upsilon'', q^*_p, q^*_m, \epsilon)) \\ s[x \mapsto (\upsilon[\mathit{wait} := \upsilon(\mathit{wait}) \setminus \{m\}], q_p, T, \mathit{body}(x, m))] \leadsto s' \end{array}}{s \xrightarrow{(x, \mathit{take}(m))} s'}$$

$$\mathit{Resume}\quad \frac{\begin{array}{c} s(x) = (\upsilon, q_p, \epsilon, \epsilon) \wedge \upsilon(\mathit{blocked}) \wedge \upsilon(\mathit{wait}) = \emptyset \\ s'(x) = (\upsilon', q'_p, q'_m, \epsilon) \wedge \forall i \in \mathit{ID} \setminus \{x\} (s(i) = (\upsilon'', q''_p, q''_m, \epsilon) \wedge s'(i) = (\upsilon'', q^*_p, q^*_m, \epsilon)) \\ s[x \mapsto (\upsilon[\mathit{blocked} := \mathit{false}]), q_p, \epsilon, \upsilon(\mathit{comd})] \leadsto s' \end{array}}{s \xrightarrow{(x, \mathit{resume})} s'}$$

Intuitively, each actor maintains the history of the message type m which it sends to the actor z with the help of the boolean variable *history_m_z*. Before sending a message, the actor x asks from all of its pre-senders to examine if a sequence is formed by sending *ask_x_m_z*. It gathers their results by processing *tell_y_m_z* and the auxiliary boolean variables *knowledge_y_m_z* and *end*, where *knowledge_y_m_z* indicates that the actor y has been sent the message type m to the actor z, and *end* denotes that the current message together with one of its predecessors complete a sequence.

ask_y_m_z is sent by actor y to x when x is the pre-sender of y to ask if the message type m has been previously sent to the actor z. The actor y receives *tell_x_m_z* in respond to *ask_y_m_z* and the actor x receives *ack_y* in respond to *tell_x_m_z*.

```
ask_y_m_z()
{
 y!!tell_x_m_z(history_m_z);
 wait(ack_y);
}
ack_y()
{
 //do nothing
}
tell_y_m_z(bool result)
{
 knowledge_y_m_z = result;
}
```

Fig. 5. Extending methods of actor x.

Being informed that the message (y, m', z) has been previously sent, an appropriate action should be performed if the action $z!m()$ by x completes a sequence. To this aim, we use auxiliary function $PreEndMsg(r, \mathfrak{m})$ to find the preceding messages of $\mathfrak{m}$ that together with $\mathfrak{m}$ complete a sequence:

$$PreEndMsg(r,\mathfrak{m}) = \{(id, m', id') \mid \forall\omega \in r,\, (\omega_{len(\omega)} = \mathfrak{m} \wedge \exists\, j < len(\omega)\,(\\ \omega_j = (id, m', id') \wedge \forall\, j < k < len(\omega)\,(\omega_k = *))\}.$$

For the given privacy requirement r, and for each message $\mathfrak{m} \in r$, where $\mathfrak{m} = (x, m, z)$, each statement $z!m(e)$ in the body of any method of the actor x is replaced with the code of Fig. 6, assuming that $PreSenderMsg(r, \mathfrak{m}) = \{(y1, m1, z), \ldots, (yn, mn, z)\}$ for some $n \geq 1$.

For example, consider the statement $grid!kloff()$ in the method *turnoff2* of the actor *meterK* in the Fig. 4. Its message belongs to the privacy requirement $r = \{\langle(meterB, bloff, grid) * (meterK, kloff, grid) * (meterH, hloff, grid)\rangle\}$. To execute the model in Fig. 4 such that it preserves r,the actor *meterK* is revised as shown in Fig. 7. As we consider the revised model as a run-time model, we do nothing when a sequence is going to be completed. Since the message $(meterK, kloff, grid)$ is not the ending message of any sequence, *end* is set to *false*.

```
y1!!ask_x_m1_z();
 ...
yn!!ask_x_mn_z();
wait(tell_y1_m1_z,...,
          tell_yn_mn_z);
formed = knowlege_y1_m1_z
     or ...
   knowlege_yn_mn_z;
end = /* if knowlege_yk_mk_z
     for any (yk,mk,z) in
     PreEndMsg(r,(x,m,z))*/
if (formed and end) {
//do appropriate action}
else if (formed) {
 history_m_z = true;
 z!m(e);
}
else { z!m(e);}
y1!!ack_x();
 ....
yn!!ack_x();
```

Fig. 6. The code substituted for $z!m(e)$ in the code of actor x.

7.1 Tool Support

We have modified our example specified in Fig. 4 using the algorithm explained in Section 7. Furthermore, the actor *grid* is our HBC actor, so it has an inference engine to infer the presence/absence of people at home. We have added the inference capability to its code. Upon inferring this, it will send the message *infer* to itself. We have implemented the resulting PPlang model using mcrl2 toolset[1] such that its semantic properties are preserved. The mcrl2 modeling language [14] is based on the ACP process algebra [13] and integrates the specification of processes and abstract data types in a unified way. In this approach, each actor is defined as a process with two queues, which can take a message, send new messages or make local decisions and receive messages.[2]

We have exploited the CADP[3] tool to model check the desired property that no privacy violation will be occured in the model. The privacy requirement is that HBC adversary can't infer about the presence/absence of people at home. Our experiment shows that the desired property is not satisfied in the original actor model (i.e., Palang), but the privacy property is satisfied by its the privacy preserving actor model (i.e., PPalang)

8 Correctness of the Synchronization Protocol

Assume that the actor model S is composed of actors $C_1, \ldots, C_n$ where $C'_i = \langle i, Var_i, Mtd_i \rangle$. For a given privacy requirement r, we generate $C_i = \langle i, Var'_i, Mtd'_i \rangle$ using the algorithm of the Section 7 such that the HBC actor sends message *infer* to itself upon inferring a sequence. We claim that semantic model of the generated model satisfies the property "never action $(C_j, take(infer))$ is performed", where C_j plays the role of HBC.
proof by contradiction: Suppose that the property is violated. Hence, before taking the message *infer* from its queue, the sequence $\langle (C_1, a_1, C_j) \ldots (C_{n-1}, a_{n-1},$

[1] `http://www.mcrl2.org/`
[2] File is available at `www.4shared.com/file/Seug1Bqoba/P-Palang_Model.html`.
[3] `http://cadp.inria.fr/`

```
actor meterK{
  bool
       knowledge_meteB_bloff_grid;
  bool history_kloff_grid;
  bool formed;
  bool end;
  turnoff2(){
     meterB!!ask_meteK_bloff_grid();
     wait(tell_meterB_bloff_grid);
     formed =
       knowledge_meterB_bloff_grid;
     end = false;
     if(formed and end) {
      // do nothing}
      else if (formed){
       history_kloff_grid=true;
       grid!kloff();
      }
      else  {
          grid!kloff();  }
      meterB!!ack_meterK();
  }
  tell_meterB_bloff_grid(bool
        result)  {
     knowledge_meteB_bloff_grid
          = result;
  }
}
```

Fig. 7. The revised code of the actor *meterK* in Fig. 4

$C_j) * (C_n, a_n, C_j)\rangle$ was received by C_j. Assume the system was in the state s before the actor C_n sends a_n to C_j as a consequence of processing the message m' in its processing queue. Based on the algorithm of Section 7, C_j sends message $ask\_C_n\_a_{n-1}\_C_j$ to the monitoring queue of C_{n-1} and then waits to receive $tell\_C_{n-1}\_a_{n-1}\_C_j$. So, according to the sos rules $Take_p$ and $Wait$, C_j blocks after processing m' and the state of the system becomes s' such that $s'(C_j) = (\upsilon_j, q_j, \epsilon, \epsilon)$ and $s'(C_{j-1}) = (\upsilon_{j-1}, q_{j-1}\langle ask\_C_n\_a_{n-1}\_C_j\rangle, \epsilon)$, where $\upsilon_j(blocked) = true$ and $tell\_C_{n-1}\_a_{n-1}\_C_j \in \upsilon_j(wait)$. So, according to the sos rule $Resume$, C_j cannot start its processing unless $\upsilon_j(wait) = \emptyset$. And according to the rule $Take_m$, it becomes empty after processing the message $tell\_C_{n-1}\_a_{n-1}$. This message will be send by C_{j-1} after processing its monitoring queue. However, after sending $tell\_C_{n-1}\_a_{n-1}$, it waits to receive ack_C_j. So, using the sos rules $Take_m$ and $Wait$, the state of the system becomes s'' such that $s''(C_j) = (\upsilon_j, q_j, q'_j, \epsilon)$ and $s'(C_{j-1}) = (\upsilon'_{j-1}, q_{j-1}, \epsilon, \epsilon)$, where $tell\_C_{n-1}\_a_{n-1} \in q'_j$, $\upsilon'_{j-1}(blocked) = true$ and $\upsilon_{j-1}(wait) = \{ack\_C_j\}$. After processing all *tell* messages from its pre-senders, the *wait* variable of C_j becomes empty according to the sos rule $Take_m$. Now by $Resume$, C_j can start its processing. According to the algorithm of the Section 7, it sets the *formed* and *end*. Since the sequence was formed, $knowledge\_C_{j-1}\_a_{j-1}\_C_j$ must hold and hence, *formed* becomes true. Furthermore, as $(C_{n-1}, a_{n-1}, C_j) \in PreEndMsg(r, (C_n, a_n, C_j))$, *end* becomes true, and consequently, the message (C_n, a_n, C_j) is not sent and the sequence will not be formed.

9 Conclusion

We addressed data privacy in distributed systems when actors communicate through message passing. The privacy is violated when sensitive data is inferred from the sequences of messages. Such inferences are possible when a legitimate participant of the system acts as a HBC adversary. Thus, such inferences cannot be prohibited by access control mechanisms and the actors should be coordinated in a way to avoid such inferences. For the sake of efficiency, we proposed a synchronization protocol by which actors can be coordinated in a distributed manner. We identified a set of essential primitives by which our protocol can be implemented. Hence, we extended the classical actor-model by these primitives and prove that a system whose commutations are replaced by our synchronization protocol preserves its privacy.

Our synchronization protocol can be used at both design time and runtime. At the design time, if a sequence is formed in the system although the actors are synchronized (found through model checking of the system), the designer should revisit its design using the provided counterexample. At runtime, we assure data privacy is not violated by controlling the behavior of the actor.

In this paper we restricted our privacy requirements to the partial message sequences which are uniquely detectable. Relaxing such a constraint and considering message parameters in inferences are among of our future work. Furthermore, we are going to examine data privacy in the presence of a Dolev-Yao adversary.

References

1. Paverd, A., Martin, A., Brown, I.: Modelling and Automatically Analysing Privacy Properties for Honest-but-Curious Adversaries. Technical report, Available at: https://www.cs.ox.ac.uk , (2014)
2. Hazay, C., Lindell, Y.: A Note on the Relation between the Definitions of Security for Semi-Honest and Malicious Adversaries. In: IACR : 551 (2010)
3. Katos, V., Vrakas, D., Katsaros, P: A Framework for Access Control with Inference Constraints. In: COMPSAC, pp. 289297. IEEE (2011)
4. Paci, F., Zannone, N.: reventing Information Inference in Access Control. In: SACMAT 15 , pp. 8797. ACM, New York (2015)
5. Westin, A.F.: Privacy and Freedom. The Bodley Head Ltd, New York (1967)
6. Lisovich, M.A., Mulligan, D.K., Wicker, S.B.: Inferring Personal Information from Demand-Response Systems. J. IEEE Security and Privacy. 8, 1120 (2010)
7. Zeadally, S., Pathan, A.K., Alcaraz, C., Badra, M.: Towards Privacy Protection in Smart Grid. J. Wireless Personal Communications. 73, 2350 (2013)
8. Imam, S.M., Sarkar, V.: Selectors: Actors with Multiple Guarded Mailboxes. In: AGERE!14 , pp. 114. ACM, New York (2014)
9. Karmani, R.K., Agha, G.: Actors. In Encyclopedia of Parallel Computing, pp. 111. Springer (2011)
10. Sirjani, M., Movaghar, A., Shali, A., de Boe, F.S.: Modeling and Verification of Reactive Systems using Rebeca. J. Fundamenta Informaticae. 63, 385410 (2004)
11. G. Agha. *ACTORS - a model of concurrent computation in distributed systems.* MIT Press series in artificial intelligence. MIT Press, 1990.
12. C. Hewitt. Viewing control structures as patterns of passing messages. *Artif. Intell.*, 8(3):323–364, 1977.
13. J. Bergstra and J.W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37:21–77, 1985.
14. J. F. Groote, A. Mathijssen, M. Reniers, Y. Usenko, and M. van Weerdenburg. The formal specification language mCRL2. In MMOSS, 2006.