Verification of Privacy-Enhanced Collaborations

Sara Belluccini
Rocco De Nicola
sara.belluccini@imtlucca.it
rocco.denicola@imtlucca.it
IMT School for Advanced Studies
Lucca, Italy

Marlon Dumas
marlon.dumas@ut.ee
University of Tartu
Tartu, Estonia

Pille Pullonen
pille.pullonen@cyber.ee
Cybernetica AS
Tallinn, Estonia

Barbara Re
Francesco Tiezzi
barbara.re@unicam.it
francesco.tiezzi@unicam.it
University of Camerino
Camerino, Italy

ABSTRACT

In a distributed scenario it is possible to find systems consisting of independent parties that collaboratively execute a business process, but cannot disclose a subset of the data used in this process to each other. Such systems can be modelled using the PE-BPMN notation: a privacy-enhanced extension of the BPMN process modeling notation. Given a PE-BPMN model, we address the problem of verifying that the content of certain data objects is not leaked to unauthorized parties. To this end, we formalise the semantics of PE-BPMN collaboration diagrams via a translation into process algebraic specifications. This formalisation enables us to apply model checking to detect unintended data leakages in a PE-BPMN model. We specifically consider data leakages in the context of secret sharing technology. The approach has been implemented on top of the mCRL2 toolset, and integrated into the Pleak toolset supporting privacy analysis of business processes. The proposal has been evaluated using real scenarios.

KEYWORDS

Privacy-Enhanced Collaboration Models, Model Checking Verification, PE-BPMN, mCRL2, Data Leakage.

ACM Reference Format:
Sara Belluccini, Rocco De Nicola, Marlon Dumas, Pille Pullonen, Barbara Re, and Francesco Tiezzi. 2020. Verification of Privacy-Enhanced Collaborations. In *8th International Conference on Formal Methods in Software Engineering (FormaliSE '20), October 7–8, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3372020.3391553

FormaliSE '20, October 7–8, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7071-4/20/05…$15.00
https://doi.org/10.1145/3372020.3391553

1 INTRODUCTION

The design of collaborative distributed systems is made complex by the need of coordinating the interactions of various components while satisfying at the same time privacy requirements [10, 32]. In such a distributed scenario, it results that organisations need to focus on the way they manage internal activities as well as on the way they exchange information during the execution of everyday business processes, being sure that privacy requirements about data are not violated due to internal accesses and message exchanges. Critical scenarios are those where sensitive data, e.g., patient records in an hospital or financial information in a company, are exchanged among parties, belonging to the same or different organisations, having different access rights. Nowadays, this issue is even more important considering the adoption by the European Union of the General Data Protection Regulation (GDPR for short).

To face this issue, modelling languages for distributed systems need to be extended to include security and privacy aspects [16]. Focusing on the BPMN notation, which is widely adopted by industry and academia, various extensions are indeed available in the literature (e.g., [35, 37]). Among others, in this paper we rely on PE-BPMN [32, 33], specifically devised to express data privacy features and mechanisms in the model. However, even if from the design point of view this enhanced model permits to specify privacy requirements, there is still a lack of techniques to ensure at design time that these are not violated. In particular, a methodological approach is missing, and related supporting tools, for detecting *data leakages*, i.e., situations where a party of the collaborative system can infer secret information or has illegal access to it [5].

In this paper, we face this challenge by defining a **novel verification methodology for data leakage detection in PE-BPMN models**. It differentiates from other approaches for BPMN-based models available in the literature [19, 20, 26], as they permit to check only general correctness properties (e.g., safeness [42] and soundness [41, 43]) without considering data handling concerns and, hence, privacy issues.

The paper specifically studies the question of verifying collaborative business processes enhanced with *secret sharing* technology. Secret sharing is a technique that splits a secret among a set of parties, by giving to each party a randomised share. The secret can be reconstructed by combining all or some qualified subset of the shares [16]. In this setting, an important privacy violation property that we would like to check is as follows: *"Is it possible to leak all pieces of a secret data such that an unauthorised party can reconstruct the secret?"*.

To achieve this goal we rely on a **formalisation of PE-BPMN in terms of the process algebra mCRL2** [12]. We use a process algebraic approach in order to take advantage of its intrinsic compositional nature. This is, indeed, particularly effective in collaboration models, as the behaviours of the distributed parties can be separately rendered as process algebra specifications, which then can be simply composed in parallel to obtain the overall collaboration behaviour. Our choice of using mCRL2 is motivated by *(i)* the suitability of the operators and features provided by this formalism to easily express both control-flow, messages-flow and data aspects of PE-BPMN models; and *(ii)* the availability of the advanced model checking capabilities provided by the mCRL2 toolset. Regarding this latter point, the mCRL2 model checker takes as input properties expressed as μ-calculus formulae [1], which can conveniently formalise privacy requirements.

To validate the feasibility and applicability of the proposed approach, **we have implemented it as a tool** that takes as input a PE-BPMN model, computes its corresponding mCRL2 specification, and checks for data leakages due to misuse of secret sharing technology. Specifically, the tool checks if an unauthorized party may gain access to a qualified subset of the shares required to reconstruct a data object, given the processing and interaction logic in the PE-BPMN model. The tool also provides an example pathway for each detected data leakage. **We have integrated the tool as a plug-in in the Pleak privacy analysis toolset** [2, 40] and **we have used it to encode and analyze various realistic scenarios**.

The rest of the paper is organised as follows. Sec. 2 provides an overview of PE-BPMN and mCRL2. Sec. 3 presents the proposed approach. Sec. 4 discusses the tool implementation and illustrates its use. Finally, Sec. 5 reviews related works and Sec. 6 concludes and discusses directions for future work.

2 BACKGROUND

The BPMN notation allows one to capture processes performed within an organization and across organizations. The latter type of process is called a *collaborative process* and is represented by a *BPMN collaboration diagram* (or *BPMN collaboration* for short). A BPMN collaboration consists of a set of processes, each performed by an independent *party* (e.g., buyer and seller). These processes are executed in parallel and synchronize via message exchanges (dashed arcs). Each process in a BPMN collaboration is captured as a separate pool (denoted as a rectangle). A process consists of tasks (rounded rectangles), events (circles) and gateways (diamonds). A task represents a logical unit of work. An event represents something triggered by the environment (e.g., a message). A gateway is used to capture a choice (XOR gateways, marked by a "×") or the parallel execution or synchronization of multiple branches (AND gateways, marked with a "+"). These three types of elements (tasks, events, gateways) are connected via sequence flows (directed arcs). A sequence flow indicates that the source element must be executed before the target element. To capture data manipulation, each task may be associated (via directed dotted arcs) to one or more input or output data objects. The intended meaning is that when the task is executed, it reads the current state of each input object, and when it completes it writes into the output data objects. These concepts are summarized in Fig. 1.

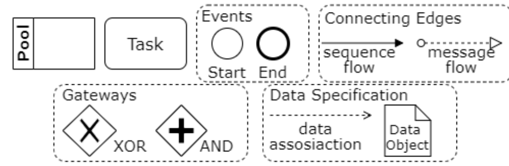

Figure 1: Elements of the BPMN Notation.

PE-BPMN [32] is a conservative extension of BPMN that allows designers to annotate tasks with stereotypes corresponding to different types of *privacy enhancing technologies (PETs)*, e.g. encryption, secret-sharing, secure enclaves. In this paper, we specifically consider secret sharing technology. With respect to secret sharing, the main stereotypes supported in PE-BPMN are *SSSharing*, which indicates that a task splits a data object into multiple secret shares, *SSComputation*, which indicates that multiple tasks with the same name but in separate pools perform a common secure multi-party computation, and *SSReconstruction*, which indicates that a task reconstructs a data object from multiple shares.

To formalise PE-BPMN collaborations, we use mCRL2 [12, 21], a specification language that extends the Algebra of Communicating Processes (ACP for short, [9]) with features for modeling data. The subset of mCRL2 **processes** used in this paper is given by the following grammar:

$$
\begin{aligned}
P ::= {} & act \;\mid\; \cdot_{i\in I} P_i \;\mid\; +_{i\in I}\, P_i \;\mid\; ||_{i\in I} P_i \;\mid\; allow(ActSet, P) \\
& \mid\; comm(CommSet, P) \;\mid\; hide(ActSet, P) \;\mid\; K \\
& \mid\; sum\ p_1, ..., p_n : sortName\,.\,P
\end{aligned}
$$

where: *act* denotes an **action** either of the form a, with no parameter (including the silent action tau), or of the form $a(d_1, \ldots, d_n)$, with data expression parameters d_i; *sortName* identifies a **sort**, which can be predefined or defined in a data specification; *ActSet* denotes a set of actions; and *CommSet* denotes a set of **communication expressions**, each one defining the renaming of multi-actions (i.e., communicating actions that occur simultaneously) to a single action. Let us comment on process syntax. We denote with $\cdot_{i\in I} P_i$ the **sequence** of processes, with $+_{i\in I} P_i$ the **choice** among processes, and with $||_{i\in I} P_i$ the **interleaving** among processes. The **sum** operator $sum\ p_1, \ldots, p_n : sortName\,.\,P$ is a generalisation of the choice operator that permits to express in a concise way the choice between a (possibly infinite) number of processes, by instantiating in P the placeholders $p_1, \ldots, p_n$ with values of type *sortName* (e.g., $sum\ n : Nat\,.\,a(n)$ is equivalent to the process $a(0) + a(1) + a(2) + \ldots$). The **allow** operator $allow(ActSet, P)$ defines the set of actions *ActSet* that the process P can execute; all other actions, except for tau, are blocked. The **communication** operator $comm(CommSet, P)$ permits synchronising actions in P according to the communication expressions *CommSet*; for example, $comm(\{a|b \text{ -> } c\}, (a\,||\,b))$ says that the parallel actions a and b must communicate, resulting in a c action. The **hide** operator $hide(ActSet, P)$ hides those actions produced by P that are in *ActSet*, i.e. it turns these actions into tau actions. Finally, K permits to **call** a process definition of the form $K = P$, where K is a unique process identifier.

The mCRL2 specification language is supported by a toolset that provides equivalence and model checking functionalities. The properties to be checked are specified in a first-order modal μ-calculus extended with data-dependent formulae [1].

3 A METHODOLOGY FOR DATA LEAKAGE VERIFICATION IN PE-BPMN COLLABORATIONS

We present in this section our methodology for verification of data leakages, referring in particular to secret sharing technology, in PE-BPMN models.

3.1 Overview

The input model of our verification methodology is a PE-BPMN collaboration diagram. To simplify the formal treatment, we make two assumptions on the input model: *(i)* the model is *well-structured* [22] (also known as *block-structured*), imposing gateways in each process to form single-entry-single-exit fragments; and *(ii)* each task can send/receive at most one message. The first assumption is not overly restrictive, as it has been shown in previous works that a large class of process models can be re-written as block-structured process models [28]. The second assumption comes without loss of generality, as it is always possible to safely transform a complex task with multiple outgoing/incoming message flows in a sequence of separate tasks, each of which with at most a message flow, with exactly the same meaning. This simplification is also aligned with generally accepted modelling guidelines [14, 25].In fact, it helps to avoid misunderstandings in the execution order among the send/receive actions performed within a task, thus allowing the designer to get a clear understanding of what is happening in the model execution. Our methodology (cf. Fig. 2) consists of three steps: *(1) control-flow transformation*, *(2) data-object and message flow transformation*, and *(3) verification*.

In the first step, the process in each pool of the PE-BPMN model is transformed into a process algebra specification. This step focuses on the control-flow perspective of the model, i.e. we abstract from data objects and message flows. More specifically, the data-abstracted structure of each process is represented as a *process tree*, which is an intermediate representation that can be then easily transformed into a mCRL2 process specification.

In the second step, the specification of each task is enhanced to capture interactions with data objects and exchange of messages, while each data and message communication is encoded via a *buffer*. The generated terms for processes and data handling are then combined via parallel composition, resulting in an overall data-aware specification of the collaboration.

Finally, in the third step, a set of "no-leakage" properties are generated and checked against the mCLR2 specification. In particular, we check the following properties, and if a property is not satisfied, we generate a counter-example:

(1) *Is it possible that a task T can read a set of data D?*, i.e. is there a path in the model leading to a state where D is part of the T's knowledge?

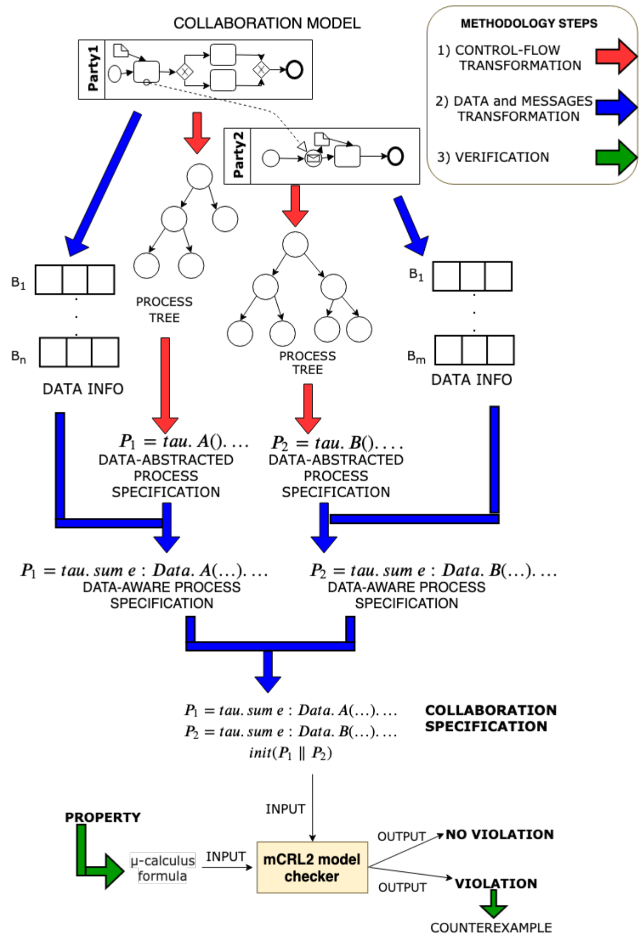

Figure 2: Methodological overview.

(2) *Is it possible that a participant P can read a set of data D?*, i.e. is there a reachable state in which P has knowledge of every element in D?

3.2 Control-flow transformation

The first step of our methodology consists of generating, via a transformation function, a mCRL2 specification from a PE-BPMN collaboration. The result is a coarse-grain specification, as it only considers the control-flow structure of the PE-BPMN model. To simplify the formal definition of the transformation, as well as its implementation, we resort to a tree-based representation of PE-BPMN models. In particular, we have defined a structure, called **process tree**, which is a variant of RPST (Refined Process Structure Tree) introduced in [29, 44].

Definition 3.1 (Process Trees). The syntax of *process trees* is as follows.

$$t ::= start \ \mid \ end \ \mid \ task(n) \ \mid \ seq(t_1, ..., t_k)$$
$$\mid xor(t_1, ..., t_k) \ \mid \ and(t_1, ..., t_k) \ \mid \ while(t)$$

where n denotes a unique task identifier.

PE-BPMN Element	Process tree
	start
	end
Task	$task(n)$
1 → .... → N	$seq(t_1, \ldots, t_n)$
1 .... N	$xor(t_1, \ldots, t_n)$
1 .... N	$and(t_1, \ldots, t_n)$
t	$while(t)$
	$t_1 \ \ldots \ t_n$

Table 1: Correspondence between the PE-BPMN block-structures and process tree elements.

The correspondence between the graphical representation of a PE-BPMN model and its process tree representation is straightforward, as shown in Table 1. The generation of the process tree corresponding to a process of a PE-BPMN collaboration is significantly simplified by the well-structuredness assumption. We refer to the literature about RPST, in particular to [29], for details on the procedure for the generation of the tree-based representation. We explain it by means of an example. Let us consider the collaboration model in Fig. 3. The process trees corresponding to the processes

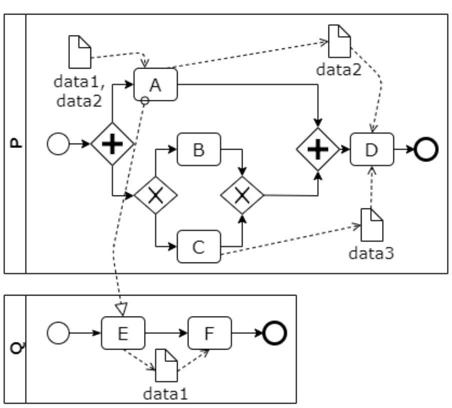

Figure 3: Example of a PE-BPMN collaboration.

of the two parties, which are graphically depicted in Fig. 4 and 5, are as follows:

P : $seq(start, and(task(A), xor(task(B), task(C))), task(D), end)$

Q : $seq(start, task(E), task(F), end)$

We can notice that there is a direct correspondence between tree nodes and (blocks of) elements in the PE-BPMN model. Notably, while the children order does not matter for *and* and *xor* nodes, it is relevant for *seq* nodes (as one may expect, the execution order is from left to right).

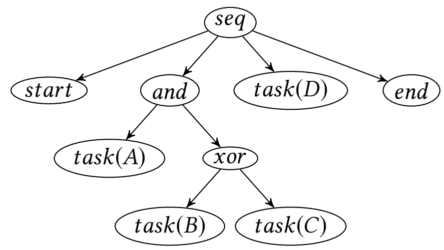

Figure 4: Process tree of party P from Fig. 3

We can now formalise the control-flow transformation step by means of the **translation function** $\mathcal{T} : \mathbb{P} \rightarrow \mathbb{M}$, where $\mathbb{P}$ is the set of process trees and $\mathbb{M}$ the set of mCRL2 terms.

Definition 3.2 (Translation function). Function $\mathcal{T}$ is inductively defined as follows:

$$\mathcal{T}(start) = tau$$

$$\mathcal{T}(end) = tau$$

$$\mathcal{T}(task(n)) = n(\{\ \})$$

$$\mathcal{T}(seq(t_1, \ldots, t_n)) = \mathcal{T}(t_1). \ \ldots \ .\mathcal{T}(t_n)$$

$$\mathcal{T}(xor(t_1, \ldots, t_n)) = \mathcal{T}(t_1) + \ldots + \mathcal{T}(t_n)$$

$$\mathcal{T}(and(t_1, \ldots, t_n)) = \mathcal{T}(t_1) \,||\, \ldots \,||\, \mathcal{T}(t_n)$$

$$\mathcal{T}(while(t)) = P \quad \text{with } P = tau \ + (\mathcal{T}(t).P)$$

where P is a fresh process identifier for the specification.

We comment on salient points. *start* and *end* elements are rendered as silent actions, as they do not have any observable effect. A $task(n)$ is translated into a visible action $n(\{\ \})$, where n is the task identifier and $\{\ \}$ represents the initial knowledge of the task (which is empty at this stage, since we do not take into account data yet). Sequence ($seq(t_1, ..., t_n)$), exclusive($xor(t_1, ..., t_n)$) and parallel ($and(t_1, ..., t_n)$) blocks are expressed by means of sequential, choice and interleaving operators, respectively. The while block ($while(t)$) is rendered as a process call; the called process has a fresh identifier and is a recursive process that non-deterministically can stop the iteration or executing the loop body and restart.

We have seen so far how to transform the process of a given party. Let us consider now a collaboration among multiple parties.

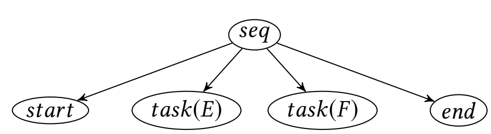

Figure 5: Process tree of party Q from Fig. 3.

Definition 3.3 (Collaboration translation). Given a collaboration involving n parties, let $t_1, \dots, t_n$ be the process trees generated from each of them, the overall specification can be defined as:

$$Party_1 = \mathcal{T}(t_1); \quad \dots \quad Party_n = \mathcal{T}(t_n);$$
$$init\ (Party_1\ ||\ \dots\ ||\ Party_n)$$

where *init* is a keyword in mCRL2 that defines the initial behaviour.

For example, the model in Fig. 3 is transformed into the following specification:

$$P \;=\; tau\,.\,(A(\{\,\})\,||\,(B(\{\,\}) \,+\, C(\{\,\})))\,.\,D(\{\,\})\,.\,tau;$$
$$Q \;=\; tau\,.\,E(\{\,\})\,.\,F(\{\,\})\,.\,tau;$$
$$init\ (P\ ||\ Q)$$

Processes P and Q inherit the name from the participant of the collaboration that they are specifying.

3.3 Data-object and message flow transformation

In the previous step, we have defined a mCRL2 specification dealing with the control-flow of the model, but without taking into account the data handling. We fill this gap in the second step of our methodology.

In PE-BPMN the exchange of data is asynchronous and can take place either *intra-pool*, via data-object connections between tasks of the same process, or *inter-pool*, via message flows connecting tasks of separate processes. Both forms of communication rely on buffers and involve a non-blocking sending task. They differ, instead, on the behaviour of the receiving task: in the inter-pool interaction the receive is blocking if the buffer is empty, while in this case in the intra-pool one the execution of the receiving task can continue as an empty message will be received. These two behaviours are captured by means of two buffer specifications. In addition, a task can have incoming data-objects that are not produced by other tasks; the information they bring is called *prior knowledge*. This information is hence directly inserted inside the task specification.

Therefore, in the second step of the methodology, the PE-BPMN model is analysed to extract all information concerning communication, which is then used to enrich the mCRL2 specification produced in the previous step. We illustrate below how the task specifications are enhanced with data information and how the two kinds of buffers are defined.

Definition 3.4 (Data-aware specification of tasks). Given a task with label n, j incoming data links/message flows, r outgoing data links/message flows, and data $e'_1, ..., e'_p$ as prior knowledge, its mCRL2 specification becomes the following one:

$$sum\ e_{11}, ..., e_{1k_1} : Data.i_1(e_{11}, ..., e_{1k_1}).$$
$$\dots\ .$$
$$sum\ e_{j1}, ..., e_{jk_j} : Data.i_j(e_{j1}, ..., e_{jk_j}).$$
$$n(union(\{e'_1, ..., e'_p\}, \{e_{11}, ..., e_{1k_1}, ..., e_{j1}, ..., e_{jk_j}\})).$$
$$o_1(e''_{11}, ..., e''_{1h_1}).\ \dots\ .o_r(e''_{r1}, ..., e''_{rh_r})$$

Using the . operator among incoming/outgoing message flows we impose an arbitrary order among how a task is receiving/sending the data that is not given in the BPMN model. This decision does not really affect the behaviour of the specification, as we will see later, because the result of this interactions are going to be hidden in the final specification. As an example, let us consider the task in Fig. 6. Its data-aware translation is as follows:

$$sum\ e_1 : Data.i(e_1).$$
$$A\big(union(\{data2, data3\}, \{e_1\}\big).\ o(e_1, data2)$$

where e_1 is a placeholder for a data to be received, say $data1$, by means of the input action i. The prior knowledge $\{data2, data3\}$ will be then extended with the new element e_1 using the *union* function. Finally, data are transmitted in output via action o.

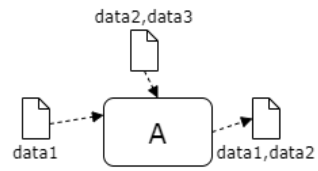

Figure 6: Example of a task.

Every communication between two tasks internal to a pool is realised by means of a dedicated buffer.

Definition 3.5 (Intra-communication buffer). The intra-communication buffer is a process of the following form:

$$B(d_1, ..., d_n : Data) = \quad sum\ e_1, ..., e_n : Data.\ i(e_1, ..., e_n).B(e_1, ..., e_n)$$
$$+\ o(d_1, ..., d_n).B(d_1...d_n)$$

where B is a fresh name for a process with n parameters, i the input channel for writing in the buffer and o the output channel for reading from it.

Notably, the buffer is defined as a recursive process in order to deal with more than one communication in case of loops in the model. Every intra-communication buffer is put in parallel with the other processes at top level of the specification, and is initialized as $B(eps_1, ..., eps_n)$, where eps represents the empty parameter. This is indeed a non-blocking buffer: if no data is written in the buffer, it provides an empty information. Notice that, for the sake of simplicity, we have used a 1-position buffer that, each time it receives new data, it rewrites the current one.

Definition 3.6 (Intra-communication protocol). In a collaboration with k participants, let be T_1 and T_2 tasks in the same pool. T_1 is sending a set of data $D = \{d_1, \dots, d_n\}$ to T_2, then an intra-communication buffer B exist. T_1, T_2 and B defined as following:

$$P_{T_1} = t_1(\{d_1, \dots, d_n\}).o_{t_1}(d_1, \dots, d_n)$$
$$P_{T_2} = sum\ e_1, \dots, e_n : Data.i_{t_2}(\{d_1, \dots, d_n\}).t_2(\{d_1, \dots, d_n\})$$
$$B(d_1, \dots, d_n : Data) = sum\ e_1, \dots, e_n : Data.i_{(}e_1, ..., e_n).B(e_1, ..., e_n)$$
$$+\ o_b(d_1, ..., d_n).B(d_1...d_n)$$

Then the communication among these elements is specified as follows:

$$init\ hide(\{sendread\}, allow(\{sendread, t_1, t_2\} \cup Act,$$
$$comm(\{o_{t_1}|i_{t_2} -> sendread\},$$
$$P_1||P_2||...||P_k||B(eps_1, \dots, eps_n))))$$

where *sendread* it is a keyword action used to represent the result of the communication and $P_1, P_2, \dots, P_k$ are the processes representing the parties in the collaboration.

From the definition it is clear that for every communication a buffer process B will be part of the initial behaviour of the specification in order to receive at any time in the execution a set of data. Then, a communication function $(o_{t_1}|i_{t_2}->sendread)$ between the output channel of the sending task (o_{t_1}) and the input channel of the receiving task (i_{t_2}) is generated, and their synchronization is forced by allowing only the execution of the communication function. Finally, since we are not interested in observing the *sendread* synchronisation actions, they will be transformed into *tau* actions using an enclosing *hide* operator.

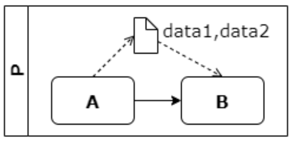

Figure 7: Example of intra-communication.

Let us consider the minimal example in Fig. 7 showing the role of the buffer. The communication between task A and B is specified as follows:

$$\begin{array}{rcl} P & = & P_A.P_B \\ P_A & = & A(\{data1, data2\}).o_A(data1, data2) \\ B_{AB}(d_1, d_2) & = & sum\ e_1, e_2 : Data.i_{AB}(e_1, e_2).B_{AB}(e_1, e_2) \\ & & + o_{AB}(d_1, d_2).B_{AB}(d_1, d_2) \\ P_B & = & sum\ e_1, e_2 : Data.i_B(e_1, e_2).B(\{e_1, e_2\}) \end{array}$$

$$\begin{array}{l} init\ hide(\{sendread\}, allow(\{sendread, A, B\}, \\ \quad comm(\{o_A|i_{AB} \rightarrow sendread, o_{AB}|i_B \rightarrow sendread\}, \\ \quad\quad P \mid\mid B(eps, eps)))) \end{array}$$

Also every communication between two tasks not in the same pool has its own buffer.

Definition 3.7 (Inter-communication buffer). A buffer for an inter-communication between two tasks is defined as follows:

$$\begin{array}{r} B(d_1, ..., d_n : Data) = sum\ e_1, ..., e_n : Data.i(e_1, ..., e_n).B(e_1, ..., e_n) \\ + ((!empty(\{d_1, ..., d_n\})) \rightarrow o(d_1, ..., d_n) \\ .B(eps_1, ..., eps_n)) \end{array}$$

where *empty* is a function that, given a parameter of type *Memory* (i.e., a set of parameters of type *Data*), returns *true* when the memory is empty, i.e. all d_i are *eps*, $false$ otherwise. The buffer is initialized again with empty data.

This is a blocking buffer, because when it is empty the output along o is not provided (due to the condition operator $Cond \rightarrow P$, meaning "if $Cond$ then do process P"), hence the receiving task has to wait.

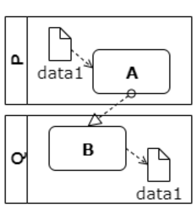

Figure 8: Example of inter-communication.

Definition 3.8 (Inter-communication protocol). In a collaboration with k participants, let be T_1 and T_2 two tasks in different pools where T_1 is sending a set of data $D = \{d_1, \ldots, d_n\}$ to T_2, then an inter-communication buffer B exist and their communication is specified as in Def. 3.6.

Hence, the only difference between *"intra"* and *"inter"* communication is the used buffer. As an example, the communication between tasks A and B in Fig. 8 is specified as follows:

$$\begin{array}{rcl} P_A & = & A(\{data1\}).o_A(data1) \\ B_{AB}(d_1) & = & sum\ e_1 : Data.i_{AB}(e_1).B_{AB}(e_1) \\ & & + ((!empty(\{d_1\})) \rightarrow o_{AB}(d_1).B_{AB}(eps)) \\ Q_B & = & sum\ e_1 : Data.i_B(e_1).B(\{e_1\}) \end{array}$$

$$\begin{array}{l} init\ hide(\{sendread\}, allow(\{sendread, A, B\}, \\ \quad comm(\{o_A|i_{AB} \rightarrow sendread, o_{AB}|i_B \rightarrow sendread\}, \\ \quad\quad P_A \mid\mid Q_B \mid\mid B(eps)))) \end{array}$$

Finally, to show how the pieces of the puzzle fall into place, we refer again to the example in Fig. 3. We report in Listing 1 an excerpt of the mCRL2 code[1] resulting from the execution of the second step of the methodology (the full mCRL2 specification is reported in the Appendix). We distinguish among two data types, namely *Data* and *Memory*. The former is an enumeration of the data objects used inside the model, like e.g. *data*1, *data*2 and *eps*. *Memory*, instead, is a set of elements of type *Data*. Moreover, we define functions *union* and *empty* (lines 9 and 11 of Listing 1). Function *union* makes the union of two variables of type *Memory* and is used to add knowledge to tasks; e.g., $E(union\{\}\{e7\})$ (line 20) means that we add to the empty knowledge of E the value of $e7$. Function *empty* takes as input a variable of type *Memory* and returns false if it exists an element inside the memory different from *eps*, i.e. if there is at least one data inside the message. The function is used to avoid the sending of empty messages (line 31).

```
1  sort Memory =  Set(Data);
2  sort Data = struct data3|data2|data1|eps;
3  map
4  union : Memory # Memory -> Memory;
5  empty : Memory -> Bool;
6  var
7  m0,m1: Memory;
8  eqn
9  union(m0,m1) = m0+m1;
10 %* is the operator for intersection among sets
11 empty(m0) = {d :Data | ({d}*m0!={})&&(d!=eps)} == {};
12 act
13 sendread,t0,t2: Bool#Memory;
14 A,memory1,E,...: Memory;
15 o5,o6,i11,o9,i4,i16,sendread,o18,...: Data;
16 proc
17 %Starting process of party Q
18 P36 = (P10.P11.t2(true,{eps}));
19 %Task E
20 P10 = sum e7: Data.i19(e7).E(union({},{e7}))
21        .o14(e7).t2(false,{e7});
22 %Task A
23 P1 = A(union({},{data1,data2})).o18(data1)
24       .o10(data2).t0(false,{data1,data2});
25 %Memory of Q participant
26 P27(e5:Memory)= sum e3: Bool.sum e4: Memory.t2(e3,e4).
27                 (!e3)->P27(union(e4,e5))<>memory3(e5).delta;
28 %Non-blocking buffer
29 P47(e10:Data) = (sum e8: Data.i8(e8).P47(e8))+(o9(e10).P47(e10));
30 %Blocking buffer
31 P63(e12:Data) = (sum e7: Data.i16(e7).P63(e7))+
```

[1]We show here the actual code taken as input by the mCRL2 tools, which also includes the required language keywords and the data specification. To improve readability, we also create a process definition for each task, we use task names instead of identifiers (e.g. A instead of $Task\_0hcmawl$), and we omit irrelevant tau actions.

```
32                    (!empty({e12})->o17(e12).P63(eps));
33 init hide ({t22,sendread,t23},
34        allow ({memory3,A,E,sendread,...},
35         comm ({o18|i16->sendread, t2|t2->sendread,...},
36           P36||P63(eps)||P47(eps)||P27({})...)));
```

Listing 1: *mCRL2* specification of the example in Fig. 3.

3.4 Verification

The last step of our methodology covers the verification of privacy-related properties over the obtained specification of the model. These properties are expressed using a first-order modal μ-calculus supported by the mCRL2 toolset, which extends the standard μ-calculus to include features for dealing with data. As already mentioned in Sec. 3, we are interested in automatically generating formulae modelling two types of properties: *Pro1* focusses on the task knowledge, and *Pro2* focusses on the participant knowledge.

Definition 3.9 (Task Formula). Given a task T with a knowledge set of dimension m and a set of data $D = \{d_1, ..., d_n\}$, property *"does T knows about D* can be expressed as

$$< true^*.exists\ e_1, ..., e_{m-n} : \\ Data.T(\{e_1, ..., e_{m-n}, d_1, ..., d_n\}) > true$$

Formula $< f > true$ corresponds to the diamond modality, which is satisfied whenever there exists a path where the formula f is satisfied. $true^*$ means that any sequence of actions can be performed before T. $exists\ e_1, ..., e_{m-n} : Data$ defines placeholders for parameters of type $Data$, which are used to simulate the value of the other elements inside the $Memory$. As usual, we consider as an example, the model in Fig. 3, and we want to answer to the following question:

Is it ever possible that task F knows about $data1$?

This property can be translated as $< true^*.F(\{data1\}) > true$.

In the example, the following path reaches a state where this property holds: $B \longrightarrow A \longrightarrow E \longrightarrow F$, where B is the action representing task B and is the first one executed to reach that state, followed by A, E and finally F. The above formula is an "eventually" formula, so it is evaluated to $true$ if there exists at least one path for which the formula is satisfied.

In order to define the second property we need to introduce a new concept (see process $P27$ at line 25 in Listing 1).

Definition 3.10 (Participant memory). Given a party in a collaboration diagram, its knowledge (i.e., the set of elements of type $Data$ that are gained by the execution of its tasks) is stored in a *memory* defined as follows:

$$M(m : Memory) =$$
$$sum\ b{:}Bool.\ sum\ mem : Memory.\ t(b, mem).$$
$$(!b) \rightarrow M(union(m, mem)) <> memory_M(m).delta$$

where $!b$ is the negation of the boolean variable b, $delta$ is a special process in mCRL2 that cannot perform any action (i.e., it is the deadlock process), while $memory_M$ is the visible action performed by M.

Every time that a task in a party is executed, it synchronises with the action t of the corresponding memory (unique for each party's memory) sending a boolean and a memory value. When the boolean parameter is equal to $false$, the memory is updated with the new data, then the process is called again to receive new information. Only when the party terminates its execution, i.e. the boolean is $true$, the memory action is performed and the execution stops (with $delta$).

Definition 3.11 (Participant formula). Given a participant P and its associated memory process M with a knowledge set of dimension m and a set of data $D = \{d_1, ..., d_n\}$, property *"does P knows about D"* can be expressed as

$$< true^*.exists\ e_1, ..., e_{m-n} : \\ Data.memory_M(\{e_1, ..., e_{m-n}, d_1, ..., d_n\}) > true$$

Participants are not identified by identifiers, but by their actions in the memory processes. As an example, we can instantiate the second property for the model in Fig. 3 as follows:

Is it ever possible that participant Q know about $data2$?

This property is expressed as $< true^*.memory1(\{data2\}) > true$, where $memory1$ is the action connected to party Q that contains all the data gained in its execution. The result of the verification is $false$.

We can now focus on the formalisation of properties concerning the *Secret Sharing* privacy technology. It can be implemented using three different stereotypes: **SSsharing**, **SScomputation**, and **SSreconstruction** [32]. *SSsharing* tasks decide how the input data is split into *shares* and which ones are necessary for computation and reconstruction (1 input data, 2 or more shares as output). *SScomputation* defines a common script that will be executed over the data by all the participants (1 or more share/data as input, 1 share as output) and *SSreconstruction* puts together the shares in order to get the result (2 or more shares as input, 1 data as output). Recall that every SSsharing task has a parameter, called *threshold*, which defines how many shares are needed to reconstruct the secret. Essentially a secret sharing protocol with threshold t is violated when:

- A party P has $n >= t$ shares of a secret and it is not either the one that creates it (it has the SSsharing task) nor the one that has to reconstruct it (it has the Reconstruction task).
- A party P has $n >= t$ computed shares (i.e. output coming from different SScomputation tasks) and the conditions are the same as above (no SSsharing and no SSreconstruction task).

The formula previously defined above are suitable to verify properties in PE-BPMN models that use the secret sharing task stereotypes.

Definition 3.12 (Secret sharing violation formula). Given a PE-BPMN model with $n + 1$ parties, where party p_0 creates shares S with a threshold t, $|C| = |S|$ results from each computation made by each party, the violation formula is defined as disjunction of participant formulae as follows:

$$< true^*.exists\ e_1, ..., e_{m-n} : Data.memory_{p_1}(\{e_1, ..., e_m, S'\}) > true\ ||$$
$$...$$
$$< true^*.exists\ e_1, ..., e_{m-n} : Data.memory_{p_n}(\{e_1, ..., e_m, S'\}) > true\ ||$$
$$< true^*.exists\ e_1, ..., e_{m-n} : Data.memory_{p_1}(\{e_1, ..., e_m, C'\}) > true\ ||$$
$$...$$
$$< true^*.exists\ e_1, ..., e_{m-n} : Data.memory_{p_n}(\{e_1, ..., e_m, C'\}) > true$$

where S' and C' are the shares and computation's output with the minimum combination such that $|S'| = |C'| = t$.

If the formula is satisfied, then a violation of the property exists in the model.

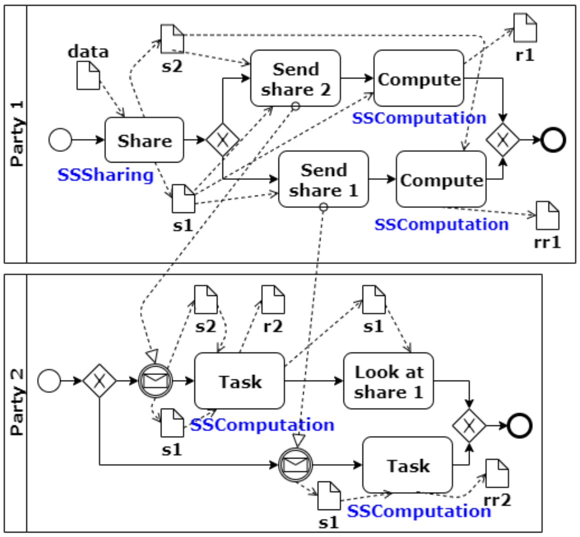

Figure 9: PE-BPMN model where the secret sharing property is violated.

For example, the secret sharing property for the model in Fig. 9 is defined by the following formula ϕ:

$$< true^*.exists\ p_0, p_1 : Data.memory_{Party2}(\{p_0, p_1, s1, s2\}) > true\ ||$$
$$< true^*.exists\ p_0, p_1 : Data.memory_{Party2}(\{p_0, p_1, r1, r2\}) > true\ ||$$
$$< true^*.exists\ p_0, p_1 : Data.memory_{Party2}(\{p_0, p_1, rr1, rr2\}) > true$$

Notably, we do not verify if Party 1 knows about $s1$ and $s2$, i.e. the shares, because it generates them.

The *mCRL2* framework provides a toolchain allowing to obtain the answers:

```
mcrl22lps spec.mcrl2 spec.lps
lps2lts spec.lps spec.lts
ltsconvert ltsconvert --etau-star spec.lts spec.lts
lts2pbes -c -f formula.mcf spec.lts spec.pbes
pbessolve --file=spec.lts spec.pbes
ltsconvert -eweak-trace spec.evidence.fsm
```

In the above sequence of instructions, *"spec.mcrl2"* is the file containing our specification. From that, we generate an LPS model that we will transform into an LTS. We apply a *tau*-reduction over the LTS to reduce the size of the state space. This is useful because every communication, internal or external, generates a *tau*. Then, the new LTS combined with the *"formula"* will create the *pbes* to be solved. *pbessolve* solves the set of equations and can give as a counter-example another LTS. To extract information from it, we transform it into an FSM (finite state machine) operating a minimisation based on the weak-trace equivalence. In such a way, we can extract a path that represents the ordered sequence of actions that leads to the solution of the formula.

4 TOOL IMPLEMENTATION AND VALIDATION

In this section, we present our verification tool and provide details about its integration in Pleak. Then, we show how it can be applied in practice.

Tool Support. The proposed methodology has been implemented as a command-line Java tool that takes as input a PE-BPMN model (in XML format) and produces a leakage diagnosis. The jar is available at https://github.com/pleak-tools/pleak-leakage-detection-analysis. The tool consists of a parser and a verification module. The parser works in two steps: first, it takes in input a PE-BPMN model and generates a set of process trees, than the process trees are transformed in a specification following the mCRL2 input language. The first step is supported by the open-source Java library jBPT [3] that we have integrated, while the second step is fully supported by our implementation. The verification module checks the secret sharing properties over the mCRL2 specification. Our tool has been integrated as a plug-in in the PE-BPMN editor of Pleak. This plug-in allows a designer to model the diagram and invokes our analyzer from Pleak's PE-BPMN editor to obtain a list of secret sharing leakages, including an example pathway for each detected leakage.

Model	# pool	# task	# sss	# msg	Translat. Time (ms)	Verif. Time (ms)	Violation
Model1	4	16	1	24	709	435	Yes
Model2	2	8	1	10	644	1	Yes
Model3	2	8	1	11	630	<1	Yes
Model4	2	3	1	6	598	<1	Yes
Model5	4	12	1	20	858	9	No
Model6	3	12	1	19	741	80	No
Model7	2	6	1	8	698	<1	Yes
Model8	2	7	1	8	721	<1	No
Model9	2	3	1	6	706	<1	No
Model10	2	8	2	11	766	<1	No

Table 2: Experiment results of checking PE-BPMN models using our tool.

Checking Secret Sharing on PE-BPMN Models. We have used our tool to analyze a set of PE-BPMN models[2] defined by designers with Pleak. Table 2 summarizes the characteristics of these models, including the number of secrets, the time necessary to parse and verify them, and the results of the verification (secret sharing violation or not). These tests were performed on a laptop running Windows 10 Pro 64 bits with an Intel(R) Core(TM) i7-5500U CPU and 8 GB of RAM (but only 256MB allocated to the Java heap). As expected, the translation time is not affected by the structure of the model, while the verification time appears to increase with the increasing complexity of the model. Specifically, the execution time seems greatly affected by three factors: the size of the checked formula, the numbers of elements in the set *Data* and the size of

[2]These models are available at: https://github.com/pleak-tools/pleak-leakage-detection-analysis/tree/master/pe-bpmn%20models

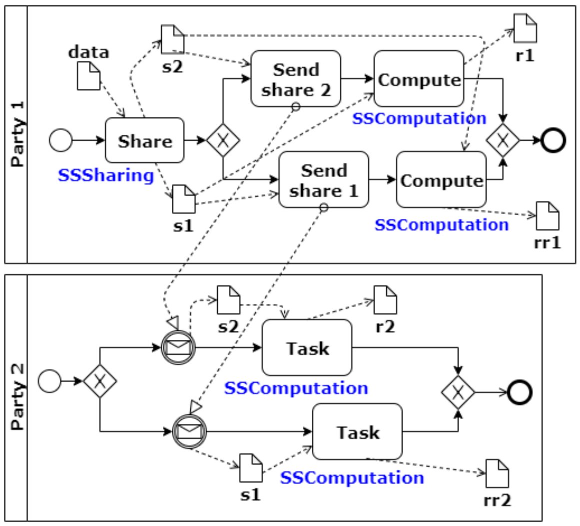

Figure 10: PE-BPMN model where the secret sharing property is preserved.

each participant's memory (i.e., the whole set of data of each participant up to the task execution). Empirically, we noticed that the translation of a formula and LTS in a pbes by recursively evaluating the guards in the LPS leads to a state space explosion.

With reference to the example[3] in Fig. 9, we observe that the formula ϕ is violated, because there is a path in which party 2 gets to know both secret shares ($s1, s2$), although this party is not the one that is expected to reconstruct the secret (it does not have the *reconstruction* task). One of the possible paths that leads to this error is: Share → Send share → Intermediate Message Event → Task. In the resulting state, both $s1$ and $s2$ are part of the knowledge of party 2. On the other hand, in Fig. 10 it seems at a first glance that party 2 will receive both the shares, but this is not the case because there is a XOR gateway both on the sending and the receiving side. A less trivial scenario is shown in Fig. 11. Here, there is a path in which party C makes a computation over $ss1$, but it also receives the result that party D computed ($result2$) when it is selected (instead of B) to make this operation.

5 RELATED WORK

In this section, we discuss the most relevant attempts in formalising BPMN models without and with data, and we compare our work with other verification approaches.

On Formalising BPMN. Several formalisations have been proposed in order to disambiguate the semi-formal semantics of BPMN. The most common formalisations of BPMN are given via mappings to various formalisms focusing on core elements of the notation, such as Petri Nets [4, 7, 17, 23, 34], and process calculi [15, 27, 30, 31, 45]. Some approach also translate processes into a model checker input language, e.g. Masalagiu et. al. [24] verify BPMN by translating it (via a Petri Net intermediate model) into the model checker input language TLA+. Considering process algebras,

[3]Notably, in addition to the PE-BPMN elements discussed in the previous sections, our tool also supports the *message intermediate event*. From the formal point of view, this does not require any extension, as this element can be safely dealt with as a receiving task.

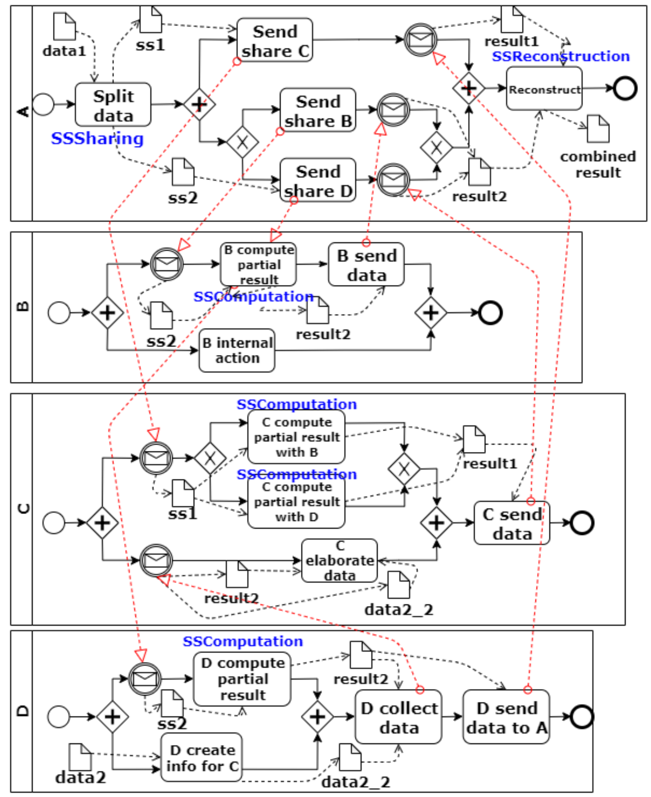

Figure 11: PE-BPMN model where the secret sharing property is violated.

in [30] a translation to COWS is proposed in order to reason about qualitative and quantitative behaviour of the business process. However, the support for specifying and handling data is missing in the verification method. In [45] a formalization from BPMN to CSP is proposed, and also in this case data objects are not considered and the refinement ordering used as verification method makes difficult to construct behavioural properties like the one for verifying a sssharing violation. This kind of formalisations are influenced by the constructs of the used language and the features of the related verification techniques. None of these approaches supports the management of data, which represents a barrier on the verification of data related properties.

Focusing on BPMN with data, only few formalisations are available in the literature (e.g., [11, 18]). In [11] the authors propose a semantic framework for BPMN with data. This approach is based on BPMN 1.0 and has a one-process view, while our focus is on the communication among multiple processes, as we are interested in exchange of data including secrets among multiple collaboration parties. While in [18], BPMN models with data objects are formalised in terms of rewriting logic. Also in this case, collaboration scenarios are not considered, while they are of main importance in our approach.

On Verification for Leakage Detection. Much effort has been devoted to the formalisation and verification of business processes (e.g., [19, 20, 26, 39]). Nevertheless, less attention has been paid to the security perspective over data of the models. Considering privacy issues and data leakage detection, some attempts have been already made using Petri Nets [5], process graphs [38] and also session types [13] to detect if a leakage exists and where. Unfortunately, these approaches are coarse-grained verification techniques that do not consider more advanced features, like the notion of data, instead of tokens, and they do not take into account and make difficult to represent security policies, like PETs. In [6] the authors focus on solving the problem of data privacy by implementing, at design time, GDPR (General Data Protection Regulation) patterns without introducing new BPMN elements, but neither a way to apply verification nor validation is proposed. Regarding GDPR, in [8] the authors propose a systematic approach to operationalize it; in this respect, our proposal could be used in the last step to automatize the way of evaluating the solution, if PETs are used. In [36], an extension of BPMN with security policies expressed as queries is proposed, together with a way to analyse them. In this case, however, the user should learn two languages: the one for modelling, using the new elements, and the one to apply verification, to manually write the queries. In addition, the framework is not able to give a counter-example of a violation, which is an important hint to correct errors occurring at design time as fast as possible.

6 CONCLUSION AND FUTURE WORK

We have proposed a methodology for verification of privacy-enhanced BPMN collaborations. Our methodology permits to detect situations where a participant in a collaboration can reconstruct a data object he/she is not authorised to access, by gathering a sufficient subset of secret shares of this object. The methodology is based on a formalisation of PE-BPMN collaborations in terms of mCRL2 specifications, complemented with an encoding of data leakages as properties in mCRL2's property specification language. The methodology has been implemented as a tool available both as a stand-alone application and as a plug-in in the Pleak toolset for business process privacy analysis. The tool takes as input a PE-BPMN collaboration and returns a list of detected leakages and a sample pathway leading to a state where each leakage occurs.

In this paper, we focused on detecting leakages arising from misuse of secret sharing technology. The proposed methodology however opens the door to verifying other related security properties. For example, the same formalisation could be used to detect situations where an unauthorised party gets access both to an encrypted data object and to the corresponding private key. We also foresee that the proposed technique can be used to check liveness properties such as "will a given participant, who needs to reconstruct a data object, eventually get all the secret shares required for this reconstruction?". Extending the proposed approach to address these and potentially other privacy and security-related properties is an avenue for future work.

Another plan of future work is to encode the PET's underlying protocols directly into the generated mCRL2 specification. This approach would potentially enable us to reduce the computation time needed to check a property, since it would result in a reduction of the state space exploration in case of a violation.

Finally, the proposed transformation from PE-BPMN collaborations is currently restricted to collaborations where each party's process is block-structured. While the class of block-structured BPMN process models is relatively expressive [28], lifting this restriction would be desirable. A challenge here is how to lift this restriction while still taking advantage of the compositionality of the process algebraic approach in order to obtain manageable mCRL2 specifications.

This research was funded by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agency (DARPA) under contract FA8750-16-C-0011. The views expressed are those of the author(s) and do not reflect the official policy or position of the Department of Defense or the U.S. Government. The work was supported by the PRIN projects "SEDUCE" n. 2017TWRCNB, "Fluidware" n. 2017KRC7KT, "IT-MaTTerS" n. 2017FTXR7S and ERDF via the DoRa Pluss programme. The authors thank Aivo Toots from Cybernetica for his valuable help with the Pleak.io toolset.

REFERENCES

[1] [n.d.]. *mCRL2 - analysing system behaviour.* https://www.mcrl2.org/

[2] [n.d.]. *PLEAK - Privacy Leakage Analysis Tools.* https://pleak.io/home

[3] [n.d.]. *The jBPT library.* https://github.com/jbpt/codebase

[4] 2010. Towards Trustworthy Composite Service Through Business Process Model Verification. 422–427.

[5] Rafael Accorsi, Andreas Lehmann, and Niels Lohmann. 2015. Information leak detection in business process models: Theory, application, and tool support. *Information Systems* 47 (2015), 244–257.

[6] Simone Agostinelli, Fabrizio Maria Maggi, Andrea Marrella, and Francesco Sapio. 2019. Achieving GDPR Compliance of BPMN Process Models. In *Information Systems Engineering in Responsible Information Systems - CAiSE Forum 2019, Rome, Italy, June 3-7, 2019, Proceedings.* 10–22.

[7] Ahmed Awad, Gero Decker, and Niels Lohmann. 2010. Diagnosing and Repairing Data Anomalies in Process Models. In *Business Process Management Workshops.* Springer, 5–16.

[8] Vanessa Ayala-Rivera and Liliana Pasquale. 2018. The Grace Period Has Ended: An Approach to Operationalize GDPR Requirements. In *26th IEEE International Requirements Engineering Conference, RE 2018, Banff, AB, Canada, August 20-24, 2018.* 136–146.

[9] Jan A Bergstra and Jan Willem Klop. 1984. Process algebra for synchronous communication. *Information and control* 60, 1-3 (1984), 109–137.

[10] Chiara Bodei, Pierpaolo Degano, Riccardo Focardi, Roberto Gorrieri, and Fabio Martinelli. 2002. Techniques for security checking: non-interference vs control flow analysis. *ENTCS* 62 (2002), 211–228.

[11] Egon Börger and Bernhard Thalheim. 2008. A method for verifiable and validatable business process modeling. In *Advances in Software Engineering.* LNCS, Vol. 5316. Springer, 59–115.

[12] Olav Bunte, Jan Friso Groote, Jeroen JA Keiren, Maurice Laveaux, Thomas Neele, Erik P de Vink, Wieger Wesselink, Anton Wijs, and Tim AC Willemse. 2019. The mCRL2 Toolset for Analysing Concurrent Systems. In *TACAS (LNCS)*, Vol. 11428. Springer, 21–39.

[13] Sara Capecchi, Ilaria Castellani, Mariangiola Dezani-Ciancaglini, and Tamara Rezk. 2010. Session Types for Access and Information Flow Control. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings.* 237–252.

[14] Flavio Corradini, Alessio Ferrari, Fabrizio Fornari, Stefania Gnesi, Andrea Polini, Barbara Re, and Giorgio Oronzo Spagnolo. 2018. A Guidelines framework for understandable BPMN models. *Data Knowl. Eng.* 113 (2018), 129–154.

[15] Flavio Corradini, Fabrizio Fornari, Andrea Polini, Barbara Re, Francesco Tiezzi, and Andrea Vandin. 2017. BProVe: a formal verification framework for business process models. In *ASE.* IEEE Computer Society, 217–228.

[16] George Danezis, Josep Domingo-Ferrer, Marit Hansen, Jaap-Henk Hoepman, Daniel Le Metayer, Rodica Tirtea, and Stefan Schiffner. 2015. Privacy and data protection by design-from policy to engineering. *arXiv preprint arXiv:1501.03726* (2015).

[17] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. 2008. Semantics and analysis of business process models in BPMN. *Information and Software Technology* 50, 12 (2008), 1281–1294.

[18] Nissreen A. S. El-Saber and Artur Boronat. 2014. BPMN Formalization and Verification using Maude. In *Proceedings of the 2014 Workshop on Behaviour Modelling - Foundations and Applications, BM-FA 2014, York, United Kingdom, July 22-22, 2014.* 1.

[19] Michael Fellman and Andrea Zasada. 2014. State of the Art of Business Process Compliance Approaches: A Survey. In *Information Systems.*

[20] Heerko Groefsema and Doina Bucur. 2013. A survey of formal business process verification: From soundness to variability. In *Business Modeling and Software Design.* 198–203.

[21] Jan Friso Groote and Mohammad Reza Mousavi. 2014. *Modeling and analysis of communicating systems.* MIT press.

[22] Bartek Kiepuszewski, Arthur Harry Maria ter Hofstede, and Christoph J. Bussler. 2000. On structured workflow modelling. Springer, 431–445.

[23] Ryszard Koniewski, Andrzej Dzielinski, and Krzysztof Amborski. 2006. Use of Petri Nets and Business Processes Management Notation in Modelling and Simulation of Multimodal Logistics Chains. In *20th European Conference on Modeling and Simulation.* Warsaw, 28–31.

[24] Cristian Masalagiu, Wei-Ngan Chin, Ştefan Andrei, and Vasile Alaiba. 2009. A rigorous methodology for specification and verification of business processes. *Formal Aspects of Computing* 21, 5 (2009), 495–510.

[25] Jan Mendling, Hajo A. Reijers, and Wil MP van der Aalst. 2010. Seven process modeling guidelines. *Information and Software Technology* 52, 2 (2010), 127–136.

[26] Shoichi Morimoto. 2008. A Survey of Formal Verification for Business Process Modeling. In *Computational Science (LNCS)*, Vol. 5102. Springer, 514–522.

[27] Andrea Polini, Andrea Polzonetti, and Barbara Re. 2012. Formal Methods to Improve Public Administration Business Processes. *RAIRO - Theor. Inf. and Applic.* 46, 2 (2012), 203–229.

[28] Artem Polyvyanyy, Luciano García-Bañuelos, and Marlon Dumas. 2012. Structuring acyclic process models. *Information Systems* 37, 6 (2012), 518–538.

[29] Artem Polyvyanyy, Jussi Vanhatalo, and Hagen Völzer. 2010. Simplified computation and generalization of the refined process structure tree. In *International Workshop on Web Services and Formal Methods.* Springer, 25–41.

[30] Davide Prandi, Paola Quaglia, and Nicola Zannone. 2008. Formal Analysis of BPMN Via a Translation into COWS. In *COORDINATION (LNCS)*, Vol. 5052. 249–263.

[31] Davide Prandi, Paola Quaglia, and Nicola Zannone. 2008. Formal Analysis of BPMN Via a Translation into COWS. In *Coordination Models and Languages.* Springer, 249–263.

[32] Pille Pullonen, Raimundas Matulevičius, and Dan Bogdanov. 2017. PE-BPMN: privacy-enhanced business process model and notation. In *BPM (LNCS)*, Vol. 10445. Springer, 40–56.

[33] Pille Pullonen, Jake Tom, Raimundas Matulevicius, and Aivo Toots. 2019. Privacy-enhanced BPMN: enabling data privacy analysis in business processes models. *Software and Systems Modeling* 18, 6 (2019), 3235–3264.

[34] Mohamed Ramadan, Hicham G. Elmongui, and Riham Hassan. 2011. BPMN formalisation using coloured petri nets. In *International Conference on Software Engineering & Applications.*

[35] Alfonso Rodríguez, Eduardo Fernández-Medina, and Mario Piattini. 2007. A BPMN extension for the modeling of security requirements in business processes. *IEICE transactions on information and systems* 90, 4 (2007), 745–752.

[36] Mattia Salnitri, Fabiano Dalpiaz, and Paolo Giorgini. 2017. Designing secure business processes with SecBPMN. *Software and Systems Modeling* 16, 3 (2017), 737–757.

[37] Koh Song Sang and Bo Zhou. 2015. BPMN security extensions for healthcare process. In *CIT/IUCC/DASC/PICom.* IEEE, 2340–2345.

[38] Alexander Seeliger, Timo Nolle, Benedikt Schmidt, and Max Mühlhäuser. 2016. Process compliance checking using taint flow analysis. In *ICIS.* Association for Information Systems.

[39] Alireza Souri, Nima Jafari Navimipour, and Amir Masoud Rahmani. 2018. Formal verification approaches and standards in the cloud computing: a comprehensive and systematic review. *Computer Standards & Interfaces* 58 (2018), 1–22.

[40] Aivo Toots, Reedik Tuuling, Maksym Yerokhin, Marlon Dumas, Luciano García-Bañuelos, Peeter Laud, Raimundas Matulevicius, Alisa Pankova, Martin Pettai, Pille Pullonen, and Jake Tom. 2019. Business Process Privacy Analysis in Pleak. In *FASE (LNCS)*, Vol. 11424. Springer, 306–312.

[41] W.M.P. van der Aalst. 1999. Process-oriented architectures for electronic commerce and inter-organizational workflow. *Information Systems* 24, 8 (1999), 639–671.

[42] W.M.P. van der Aalst. 2000. Workflow Verification: Finding Control-Flow Errors Using Petri-Net-Based Techniques. In *Business Process Management, Models, Techniques, and Empirical Studies.* LNCS, Vol. 1806. Springer, 161–183.

[43] W.M.P. van der Aalst, K. van Hee, A ter Hofstede, N. Sidorova, H. Verbeek, M. Voorhoeve, and M. Wynn. 2011. Soundness of workflow nets: classification, decidability, and analysis. *FAC* 23, 3 (2011), 333–363.

[44] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. 2009. The refined process structure tree. *Data & Knowledge Engineering* 68, 9 (2009), 793–818.

[45] Peter Y. H. Wong and Jeremy Gibbons. 2011. Formalisations and applications of BPMN. *Sci. Comput. Program.* 76, 8 (2011), 633–650.

A COMPLETE MCRL2 SPECIFICATION OF THE MODEL IN FIG. 3

```
sort Memory =  Set(Data);
sort Data = struct data3|data2|data1|eps;
map
union : Memory # Memory → Memory;
empty : Memory → Bool;
var
m0,m1: Memory;
eqn
union(m0,m1) = m0+ m1 ;
empty(m0) = {d :Data | ({d}* m0!={})&&(d!=eps)} == {};
act
t21 ,t20 ,t23 ,t22;
sendread ,t0 ,t2: Bool#Memory;
Task_07m8xua,memory1,Task_102audm,Task_0d860z3,
IntermediateThrowEvent_0h7g12c,Task_0kra3lu,memory3,Task_0mj268e:
        Memory;
o5,o6,i11,o9,i4,i12,i15,i7,i16,o10,i8,o13,i19,o14,o17,sendread,o18:
        Data;
proc
P10 = sum e7:
        Data.i19(e7).IntermediateThrowEvent_0h7g12c(union({},{e7}))
        .o14(e7).t2(false,{e7});
P39(e9:Data) = (sum e6: Data.i4(e6).P39(e6))+(o5(e9).P39(e9));
P4 = (P2+P3);
```

```
P36 = (P10.P11.t2(true,{eps}));
P27(e5:Memory)= sum e3: Bool.sum e4: Memory.t2(e3,e4).
                    (!e3)->P27(union(e4,e5))<>memory3(e5).delta;
P2 = Task_102audm({});
P72 = (t20.P1.t21.P6.t0(true,{eps}));
P6 = sum e6: Data.i7(e6).sum e8:
       Data.i11(e8).Task_07m8xua(union({},{e6,e8})).
      t0(false,{e6,e8});
P75 = (t20.P4.t21);
P34 = (P72||P75);
P17(e2:Memory)= sum e0: Bool.sum e1: Memory.t0(e0,e1).
                    (!e0)->P17(union(e1,e2))<>memory1(e2).delta;
P47(e10:Data) = (sum e8: Data.i8(e8).P47(e8))+(o9(e10).P47(e10));
P1 = Task_0kra3lu(union({},{data1,data2})).o18(data1).o10(data2).
      t0(false,{data1,data2});
P11 = sum e7:
       Data.i15(e7).Task_0d860z3(union({},{e7})).t2(false,{e7});
P55(e11:Data) = (sum e7:
       Data.i12(e7).P55(e7))+(o13(e11).P55(e11));
P63(e12:Data) = (sum e7:
       Data.i16(e7).P63(e7))+(!empty({e12})->o17(e12).P63(eps));
P3 = Task_0mj268e(union({},{data3})).t0(false,{data3}).o6(data3);
init hide ({t22,sendread,t23}, allow({Task_102audm,memory3,
     Task_07m8xua,Task_0d860z3,memory1,Task_0kra3lu,t22,Task_0mj268e,
     sendread,t23,IntermediateThrowEvent_0h7g12c},comm({o9|i11->sendread,
     o18|i16->sendread,o17|i19->sendread,t0|t0->sendread,o6|i4->sendread,
     o5|i7->sendread,t20|t20->t22,t2|t2->sendread,o13|i15->sendread,
     t21|t21->t23,o10|i8->sendread,o14|i12->sendread},
P34||P39(eps)||P36||P63(eps)||P55(eps)||P47(eps)||P27({})||P17({}))));
```