Certified Information Flow Analysis of Service Implementations

Thomas S. Heinze
Institute of Data Science
German Aerospace Center (DLR)
Jena, Germany
thomas.heinze@dlr.de

Jasmin Türker
Institute of Computer Science
Friedrich Schiller University Jena
Jena, Germany
jasmin.tuerker@uni-jena.de

***Abstract*—Process analysis enables the certification of distributed business processes using automated process compliance checks. In such an auditing scenario, analysis correctness is key but is usually taken for granted. We therefore argue in this paper for the idea of certified analysis. As is shown by the example of a static information flow analysis and its accompanying Coq development, certified analysis of distributed business processes is feasible and provides machine-checkable correctness certificates and thus helps in increasing thrustworthiness of automated process compliance audits.**

I. Introduction

For distributed business processes implemented as service choreographies or orchestrations, e.g., in languages like WS-BPEL or BPMN, process compliance is of vital importance since the enacted business processes usually are subject to the various regulations and standards of participating organizations. A large fraction thereof considers process security and data confidentiality in particular. As an example, the *Health Insurance Portability and Accountability Act (HIPAA)* states that each person who maintains or transmits health information shall maintain reasonable and appropriate measures to ensure the confidentiality of the information and prohibit unauthorized uses or disclosures of the information. Process auditing scenarios can substantialy benefit from methods for automatically analyzing processes with respect to compliance rules like confidentiality, be it at design time or runtime.

While the need for automated process compliance analysis has been addressed in the business process and service domain [1], as, e.g., reflected by the notion of *certified processes* [2], analysis correctness plays only a minor role and is usually shown by pen-and-paper proofs, if at all. In this paper, we show how to enrich process auditing scenarios using automated analyses with mechanized correctness guarantees for the analyses themselves. Using the example of an information flow analysis, we argue for the idea of *certified analysis* [3]. A certified analysis is accompanied by a machine-readable proof, which can be seen as a certificate guaranteeing correctness. Thus, a user is not required to trust in the analysis, but can validate its specification in terms of the correctness proof. Furthermore, mechanically proving correctness and generating the analysis implementation from the proof supports formal rigor and helps in avoiding implementation errors.

DOI 10.1109/SOCA.2018.00033

The presented *certified information flow analysis* looks for information leaks in a process model at design time. An information leak happens when sensitive information can be accessed by an untrusted party. Assuming a process model in which activities, where sensitive information originates, and activities, that can be accessed by untrusted parties, are marked by labels H and L, respectively, we check for information flows from an activity with label H to an activity with label L. We focus on the data flow, i.e., the flow of information along variables, messages, and heap-allocated data objects.

More technically, we provide a static analysis for detecting data leaks under a multi-level security model with mandatory access control. We can formulate the problem addressed by the analysis in other words as: Given a stateful service implementation in terms of a business process model and a list of sensitive information sources and untrusted sinks, check whether there exists a flow of information in the process' data flow from a source to a sink. To address the problem, we propose a unified approach based on *points-to* and *taint analysis*. We provide a *Coq development* for the analysis including its correctness proof, proving soundness and termination. The contributions of this paper are thus:

- We study the concept of certified analysis in the business process and service domain and demonstrate its feasibility by way of the example of an information flow analysis and its machine-readable Coq proof.
- We address a heap-based data model for process data as prevalent in state-of-the-art process engines.
- We present a formal development for proving the correctness of unified points-to/taint analysis in line with most recent research [4].

The rest of the paper is structured as follows: In Sect. II, we sketch the approach of certified analysis based upon abstract interpretation. In Sect. III, the information flow analysis is first introduced as a unified points-to/taint analysis. In the more technical part of the paper (Sect. III-B and III-C), the analysis' abstract and concrete semantics are developed and used for proving its correctness. A prototype implementation of the analysis as a Camunda Modeler plugin is presented in Sect. IV. The discussion of related work is contained in Sect. V and Sect. VI eventually concludes the paper.

II. Certified Process Analysis

Data flow analysis has shown to be a utility for the analysis of service implementations, be it for supporting formal verification [5], [6], for analyzing data- or control-flow related properties [7], or for optimization and reengineering [8]. In particular two aspects make it such a useful method: (1) Data flow analysis provides a general framework which can be easily adjusted to new domains and analysis problems, (2) data flow analysis is grounded on well-founded theories like Kildall's lattice-theoretic formulation [9] or Cousot's abstract interpretation [10]. While the latter aspect offers a way to formally show an analysis to be correct, until now, only a small number of data flow analyses in the business process and service domain have been proven correct, using pencil-and-paper proofs and just considering termination [5], [7].

This is suprising considering the recent advances made in the verification of static analysis. State-of-the-art approaches define not only an analysis but rather add a mechanized, i.e., machine-checkable proof of its correctness. The term *certified analysis* [3] has therefore been coined, as a user does not need to trust in such an analysis but can automatically validate its conjoined correctness proof. We therefore argue for the idea of a certified analysis of service implementations. The advantages are manifold. For instance, the complexity of modeling languages for distributed business processes like WS-BPEL and BPMN makes not only the design and implementation of processes, but also of analyses error-prone. An approach for proving analysis correctness helps in regaining confidence. A mechanized correctness proof also augments process auditing scenarios [2] with an orthogonal check that the audit itself can be trusted. Automatically generating an analysis implementation from its correctness proof even relieves the analysis designer from the implementation burden. We next introduce the concept of certified analysis based upon abstract interpretation, which serves as basis for the formal development of the information flow analysis in this paper.

Data flow analysis can be seen as *abstract interpretation* [10], where processes are evaluated using abstract values instead of concrete ones. As an example, instead of performing arithmetic operations on integers when executing a process, an analysis may only track whether an integer has abstract value odd or $even$. In this way, e.g., addition boils down to four cases: $even + even = even$, $even + odd = odd$, $odd + even = odd$, and $odd + odd = even$. The domains of abstract and concrete values are connected by an abstraction relation, which relates concrete values and their abstract representations, e.g., odd approximates $\{1, 3, 5, \dots\}$. In addition, the abstract domain A should form a partially-ordered set, reflecting precision among abstract values such that for all $a, b \in A$, $a \leq b$ iff all concrete values approximated by b are also approximated by a. Consider, e.g., another abstract value int representing all integers, then apparently $int \leq odd$. Executing a process' activities on concrete values can be formalized using standard structural operational semantics [11], which yields the *concrete semantics*. The effect of execution on abstract values is defined by the analysis in terms of a monotone function $f\colon A \to A$, mimicking, e.g., above's addition example, which in this way provides the *abstract semantics*. An analysis based upon abstract interpretation then calculates a fixpoint abstract value, i.e., $f(a) = a$, for a given process by continuously applying the abstract semantics to an initial abstract value until the fixpoint is reached. An analysis can be shown correct, iff:

$$\forall a \in A\colon a \text{ approximates } init \wedge f(a) = a$$
$$\Rightarrow (\forall c'\colon init \to^* c' \Rightarrow a \text{ approximates } c')$$

meaning that a fixpoint a of the abstract semantics, that approximates the initial concrete value $init$, also approximates the concrete values c' in all reachable states, $initial \to^* c'$. Note that correctness here refers to soundness, i.e., the abstract value a is guaranteed to (over-)approximate the concrete value. Analysis termination can though also be shown, using the fact that the abstract semantics forms a montone function on the partially-ordered set of abstract values. In particular, sets with a well-founded order relation guarantee the existence of a fixpoint, which can be calculated with standard algorithms as in monotone data flow analysis [9] (optionally using accelerators like widening/narrowing [10]).

The *Coq proof assistant*[1] has been shown of use for proving analysis correctness [12]. Its inductive types provide a way for encoding programs and its recursive functions for encoding concrete and abstract semantics. Proving analysis correctness in Coq still requires manual work, yet Coq allows for automatic proof validation and even to automatically extract the analysis implementation from the proof, which justifies the notion of a *certified analysis*.

III. Certified Information Flow Analysis

In the following, information flow analysis is first introduced as a feasible candidate for certified analysis in the business process and service domain in Sect. III-A. Afterwards, the Coq development of the information flow analysis is presented in terms of the analysis' concrete and abstract semantics in Sect. III-B. This allows us to discuss the analysis' soundness and termination proofs in Sect. III-C.

A. Information Flow Analysis

An *information flow analysis* [13] checks the propagation of information. The analysis thereby distinguishes information of different sensitivity levels. In the basic case, which is also addressed here, two levels are considered: *low (L)* representing information coming from an insensitive source and *high (H)* representing information coming from a sensitive source. Note that we however could easily adjust the analysis to support

[1]`https://coq.inria.fr/`

more than these two levels. In order to guarantee the absence of information leakage, the analysis needs to check that no information is propagated from a high-level source to a low-level sink, while any flow of information between equal levels or from a low-level source to a high-level sink is allowed.

Example 1: In Fig. 1, a BPMN process model is shown which represents the fictive risk assessment of a health insurance company. In the process, a sensitive medical score (message `score`) for the insurant is automatically used as rating, if the policy shall not exceed a certain limit, or the rating is manually created. The resulting assessment in variable `rating` is then sent to a possibly untrusted external contractor for further processing. Assuming the sensitivity label H for the incoming message `score` and label L for `rating`, there is a path on which information is propagated from a high-level source to a low-level sink and thus a possible information leak in the process. Though, considering label L for variable `reply` instead, anonymization converts the high-level into low-level information, so that information leakage is prevented.

In this paper, we focus on the propagation of information along the data flow in a given process model, as sketched in Ex. 1, and abstract from other side channels like implicit information flow, e.g., the leakage of information about variable `rating` through the branching condition in the example process. Under these requirements, the information flow analysis problem can be reformulated in terms of *static taint checking* [4], [14]. Traditionally, taint checking is used to increase software security by identifying unchecked inputs which allow to insert malicious data into an application, like in SQL injection attacks. Similar to information flow analysis, a taint analysis looks for data flow from untrusted sources, labeled H, to security critical sinks, labeled L. In addition, a taint analysis is also aware of *sanitized data*, i.e., input from untrusted sources which though has been freed from malicious data. Note that sanitized data is also important when analyzing for information leakage, if, e.g., personal information is anonymized as in Ex. 1.

Static taint checking usually relies on information about which data objects a variable may point to as provided by points-to analysis [15]. Note that we here assume a heap-based data model, similar to languages like Java and JavaScript, and typically used in recent process engines, e.g., Camunda[2]. Consequently, points-to information is key as information flow apparently depends on the flow of data along variables, messages, and heap-allocated objects. Until currently, taint checking has been a client of points-to analysis. For our analysis, points-to and taint information is in contrast computed simultaneously, in line with recent research [4].

The following definition states the information flow analysis using a set of constraints according to a process' activities:

Definition 1: For a given process model, each variable or message x is assigned the set of potentially assigned objects, denoted by $\llbracket x \rrbracket \subseteq \mathit{Objects}$, using the constraints:

$$\begin{array}{rc}
\text{Object creation } \mathtt{k\colon Allocate(x)}: & o_k^L \in \llbracket x \rrbracket, o_k \text{ fresh} \\
\text{Sensitive object } \mathtt{k\colon Source(x)}: & o_k^H \in \llbracket x \rrbracket, o_k \text{ fresh} \\
\text{Assignment } \mathtt{x = y}: & \llbracket y \rrbracket \subseteq \llbracket x \rrbracket \\
\text{Sanitizer } \mathtt{x = Sanitize(y)}: & \dfrac{o_k^L \in \llbracket y \rrbracket}{o_k^L \in \llbracket x \rrbracket} \quad \dfrac{o_k^H \in \llbracket y \rrbracket}{o_k^L \in \llbracket x \rrbracket} \\
\text{Field read } \mathtt{x = y.f}: & \dfrac{o_k^L \in \llbracket y \rrbracket}{\llbracket o_k.f \rrbracket \subseteq \llbracket x \rrbracket} \quad \dfrac{o_k^H \in \llbracket y \rrbracket}{\llbracket o_k.f \rrbracket \subseteq \llbracket x \rrbracket} \\
\text{Field write } \mathtt{x.f = y}: & \dfrac{o_k^L \in \llbracket x \rrbracket}{\llbracket x \rrbracket \subseteq \llbracket o_k.f \rrbracket} \quad \dfrac{o_k^H \in \llbracket x \rrbracket}{\llbracket x \rrbracket \subseteq \llbracket o_k.f \rrbracket}
\end{array}$$

The constraints in Def. 1 are used to assign each variable and message the set of potentially assigned data objects. Along the lines of classical ANDERSEN-style subset-based points-to analysis [15], objects $o_k^L, o_k^H \in \mathit{Objects}$ are abstract and distinguished with respect to their static allocation site k. Note that inbound message activities are treated as allocation sites of objects submitted via respective incoming messages and objects created at sensitive sites (`Source`) are distinguished from objects created at ordinary sites (`Allocate`) using object labels H and L. The constraints model the process' data flow by modeling the effect of executing an activity on the sets of assigned objects. For instance, an assignment `x = y` is reflected by the subset constraint $\llbracket y \rrbracket \subseteq \llbracket x \rrbracket$, safely modeling the flow of objects from assignment source to assignment target by the rule that each object assigned to `y` may as well be assigned to `x`. Solving the thus-defined constraint system for a process then results in a safe approximation of assigned objects. Using this information, we can decide for each process variable and message individually whether or not there is information leakage:

Definition 2: There exists a *potential information leak* for a given low-level variable or message x iff $\exists o_i^H \in \llbracket x \rrbracket$.

Example 2: Consider the example process in Fig. 1 and label H for the object submitted via message `score` and L for variable `rating`, respectively. The application of the constraints in Def. 1 results in the following derivation:

$$\dfrac{\dfrac{\dfrac{}{o_1^L \in \llbracket \mathit{request} \rrbracket}\,1}{\llbracket o_1.\mathit{risk} \rrbracket \subseteq \llbracket \mathit{rating} \rrbracket}\,4 \qquad \dfrac{\dfrac{\dfrac{}{o_1^L \in \llbracket \mathit{request} \rrbracket}\,1}{\llbracket \mathit{score} \rrbracket \subseteq \llbracket o_1.\mathit{risk} \rrbracket}\,3 \quad \dfrac{}{o_2^H \in \llbracket \mathit{score} \rrbracket}\,2}{o_2^H \in \llbracket o_1.\mathit{risk} \rrbracket}}{o_2^H \in \llbracket \mathit{rating} \rrbracket}$$

For improving comprehensibility, we have therein labeled constraint applications with the corresponding process activities. As can be seen, our analysis concludes that there is a information leak for variable `rating` as the object contained in the set $\llbracket \mathit{rating} \rrbracket$ is labeled with sensitivity label H.

B. Abstract and Concrete Semantics

A formal basis for the analysis is developed next. The development is done in Coq, which allows for a machine-readable correctness proof. The Coq sources are available online[3] and extend prior work of Adam Chlipala[4] (Coq names for definitions and theorems are given in brackets, [`like this`]).

[2]https://camunda.com/

[3]https://gitlab.com/t.heinze/soca2018.git
[4]http://adam.chlipala.net/itp/coq/src

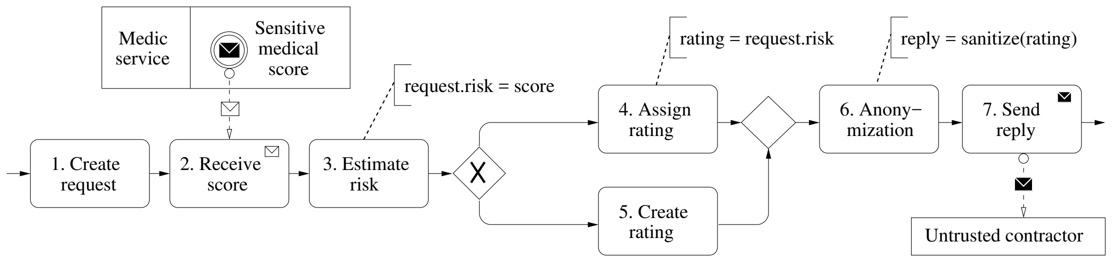

Fig. 1. BPMN service implementation model used as running example.

As outlined in Sect. II, abstract interpretation provides a well-grounded theory for static data flow analysis which will be used for formalizing the information flow analysis. In terms of abstract interpretation, a data flow analysis executes a process, though not on concrete values but rather on abstract values. Execution on concrete values determines the *concrete semantics* and execution on abstract values the *abstract semantics*. In order to establish analysis correctness, we need to show that the abstract semantics is a valid approximation of the concrete semantics and its continuous application on a given process eventually results in a fixpoint solution. For defining the concrete and abstract semantics, the usual approach of structural operational semantics is used, that is, small-step semantics [11], [12]. Note that, in contrast to Sect. III-A, a *field-insensitive analysis* [15] is considered here, to keep the formal development of the analysis and proof relatively simple.

Small-step semantics uses states and a reduction relation to describe the effect of a process activity's execution. For its definition in Coq, another process representation is needed. Instead of using a process model in a language like BPMN or WS-PEL, we therefore use a more structured representation which consists of compound activities, determining a process' control flow and supporting the usual structures, i.e., sequence, choice, loop and fork/join, and basic activities modifying process data. Along the lines of [16], we thus define:

Definition 3 ([`compound_activity`]*):* A *process* is defined inductively as:

- The empty activity `Empty` is a process
- A basic activity `Activity a` is a process, where `a` is:
  - an allocation site `Allocate(x)` or `Source(x)`
  - a field read `x = y.f` or field write `x.f = y`
  - a sanitizer `x = Sanitize(y)`
  - an assignment `x = y`
- If `a` and `b` are processes, then their sequential execution `Sequence(a, b)` is also a process
- If `a` and `b` are processes, then their alternative execution `Choice(a, b)` is also a process
- If `a` and `b` are processes, then their parallel execution `Fork(a, b)` is also a process
- If `a` is a process, then its repeated execution `Loop(a, b)` is also a process

Example 3: Considering the BPMN process model from Ex. 1, the following process representation is used in Coq:

```
Sequence(
  Activity Allocate(request),
  Sequence(
    Activity Source(score),
    Sequence(
      Activity request.risk=score,
      Sequence(
        Choice(
          Activity rating=request.risk,
          Activity Allocate(rating)),
        Sequence(
          Activity reply=Sanitize(rating),
          Empty)))))
```

Furthermore, variables and messages are uniquely mapped to the natural numbers and objects are modelled as pairs (p, q) of their allocation site $p \in \mathbb{N}$ and sensitivity label $q \in \{L, H\}$. Based on this representation, concrete states can be defined:

Definition 4 ([`state`]*):* A *concrete state* c is a triple $(vars, heap, l)$, with functions $heap\colon \mathbb{N} \to (\mathbb{N} \times \{L, H\})$ and $vars\colon \mathbb{N} \to (\mathbb{N} \times \{L, H\})$, assigning objects to object fields, and variables and messages, respectively, and a counter $l \in \mathbb{N}$ used to distinguish the process' dynamic allocation sites.

For better comprehension, we separate the definition of the concrete semantics for compound activities and basic activities. Concrete semantics for the former ones is defined in Fig. 2. The rules are rather unsurprising and follow the usual approach [11], [12]. Each rule $a, s \to a', s'$ transforms a state s and an activity a into the modified state s' and the residual activity a'. Note that, however, the rules have been adapted to Def. 3, such that choices and loops are nondeterministic and fork/join parallelism is supported. In addition, overloading is used for referring to basic activities.

In Fig. 3, the concrete semantics for basic activities is defined. As can be seen, each creation of an object implies an update of a state's $vars$ function, such that the respective variable or message is assigned an object identified by its dynamic allocation site $l + 1$ in the modified state. Objects created at activities `Allocate` are thereby labeled with L and objects created at activities `Source` are labeled with H. An assignment implies the assignment of the object of the assignment's source to the assignment's target in the modified state. The rule for activity `Sanitize` is similar, but additionally changes the respective object's label to L. A field read implies the update of a state's $vars$ function, such that the respective variable or message is assigned the object addressed

$$\frac{\texttt{a},\ (vars, heap, l) \to (vars', heap', l')}{\texttt{Activity a},\ (vars, heap, l) \to \texttt{Empty},\ (vars', heap', l')}$$

$$\frac{\texttt{a}_1,\ (vars, heap, l) \to \texttt{a}_2,\ (vars', heap', l')}{\texttt{Sequence}(\texttt{a}_1, \texttt{a}),\ (vars, heap, l) \to \texttt{Sequence}(\texttt{a}_2, \texttt{a}),\ (vars', heap', l')}$$

$$\texttt{Choice}(\texttt{Empty}, \texttt{a}),\ (vars, heap, l) \to \texttt{a},\ (vars, heap, l)$$

$$\texttt{Choice}(\texttt{a}_1, \texttt{a}_2),\ (vars, heap, l) \to \texttt{a}_1,\ (vars, heap, l)$$

$$\texttt{Choice}(\texttt{a}_1, \texttt{a}_2),\ (vars, heap, l) \to \texttt{a}_2,\ (vars, heap, l)$$

$$\texttt{Loop}(\texttt{a}),\ (vars, heap, l) \to \texttt{Sequence}(\texttt{a}, \texttt{Loop}(\texttt{a})),\ (vars, heap, l)$$

$$\texttt{Loop}(\texttt{a}),\ (vars, heap, l) \to \texttt{Empty},\ (vars, heap, l)$$

$$\frac{\texttt{a}_1,\ (vars, heap, l) \to \texttt{a}_2,\ (vars', heap', l')}{\texttt{Fork}(\texttt{a}_1, \texttt{a}),\ (vars, heap, l) \to \texttt{Fork}(\texttt{a}_2, \texttt{a}),\ (vars', heap', l')}$$

$$\frac{\texttt{a}_1,\ (vars, heap, l) \to \texttt{a}_2,\ (vars', heap', l')}{\texttt{Fork}(\texttt{a}, \texttt{a}_1),\ (vars, heap, l) \to \texttt{Fork}(\texttt{a}, \texttt{a}_2),\ (vars', heap', l')}$$

$$\texttt{Fork}(\texttt{Empty}, \texttt{Empty}),\ (vars, heap, l) \to \texttt{Empty},\ (vars, heap, l)$$

Fig. 2. Structural concrete semantics ([step]).

$$\texttt{k: Allocate(x)},\ (vars, heap, l) \to (vars[x \leftarrow (l+1, L)], heap, l+1)$$

$$\texttt{k: Source(x)},\ (vars, heap, l) \to (vars[x \leftarrow (l+1, H)], heap, l+1)$$

$$\texttt{x = y},\ (vars, heap, l) \to (vars[x \leftarrow vars(y)], heap, l)$$

$$\frac{(p, q) = vars(y)}{\texttt{x = Sanitize(y)},\ (vars, heap, l) \to (vars[x \leftarrow (p, L)], heap, l)}$$

$$\frac{heap(vars(y)) = (p, q) \qquad p \neq 0}{\texttt{x = y.f},\ (vars, heap, l) \to (vars[x \leftarrow heap(p)], heap, l)}$$

$$\frac{heap(vars(y)) = (0, q)}{\texttt{x = y.f},\ (vars, heap, l) \to (vars, heap, l)}$$

$$\frac{vars(x) = (p, q) \qquad p \neq 0}{\texttt{x.f = y},\ (vars, heap, l) \to (vars, heap[p \leftarrow vars(y)], l)}$$

$$\frac{vars(x) = (0, q)}{\texttt{x.f = y},\ (vars, heap, l) \to (vars, heap, l)}$$

Fig. 3. Structural concrete semantics continued ([exec]).

by the read's receiver object's allocation site p in the modified state, if different from zero. Note that we associate objects in the heap based on their allocation sites only and ignore labels, which is a necessary condition for soundness. Eventually, the rule for a field write updates the *heap* function of a state in that the receiver object's allocation site p is assigned the write's source object in the modified state's heap, if unequal zero.

For the abstract semantics, modeling our information flow analysis as introduced in Sect. III-A, abstract states are used. In contrast to a concrete state, which assigns the runtime object to a variable, message, or field, an abstract state assigns a set of objects identified by their static allocation site:

Definition 5 ([abstract_state]*):* An *abstract state* a is a pair $(avars, aheap)$, with functions $aheap\colon \mathbb{N} \to \mathcal{P}(\mathbb{N} \times \{L, H\})$ and $avars\colon \mathbb{N} \to \mathcal{P}(\mathbb{N} \times \{L, H\})$, assigning sets of objects to fields and, messages and variables, respectively.

The abstract semantics is defined in Fig. 4. As our analysis is flow-insensitive, there are no rules for compound activities.

$$\begin{aligned}&\texttt{k: Allocate(x)} :: ls,\ (avars, aheap)\\ &\quad\to ls,\ (avars[x \leftarrow avars(x) \cup \{(k, L)\}], aheap)\end{aligned}$$

$$\begin{aligned}&\texttt{k: Source(x)} :: ls,\ (avars, aheap)\\ &\quad\to ls,\ (avars[x \leftarrow avars(x) \cup \{(k, H)\}], aheap)\end{aligned}$$

$$\begin{aligned}&\texttt{x = y} :: ls,\ (avars, aheap)\\ &\quad\to ls,\ (avars[x \leftarrow avars(x) \cup avars(y)], aheap)\end{aligned}$$

$$\begin{aligned}&\texttt{x = Sanitize(y)} :: ls,\ (avars, aheap)\\ &\quad\to ls,\ (avars[x \leftarrow avars(x) \cup (p, L) \mid (p, q) \in avars(y)], aheap)\end{aligned}$$

$$\begin{aligned}&\texttt{x = y.f} :: ls,\ (avars, aheap)\\ &\quad\to ls,\ (avars[x \leftarrow avars(x) \cup \bigcup_{(p,q) \in avars(y)} aheap(p)], aheap)\end{aligned}$$

$$\begin{aligned}&\texttt{x.f = y} :: ls,\ (avars, aheap)\\ &\quad\to ls,\ (avars, aheap[p \leftarrow aheap(p) \cup avars(y) \mid (p, q) \in avars(x)])\end{aligned}$$

Fig. 4. Abstract semantics ([abstract_exec]).

Instead, the process is transformed into a list of the process' basic activities. The reduction relation $s, a :: ls \to ls', s'$ then relates a state s and the head a of the list to the modified state s' and the list's tail ls. Creation of an object implies again the update of a state's *avars* function, but this time, the static allocation site k is added to the set of assigned objects of the respective variable or message (also cf. constraints in Def. 1). In the same way, an assignment updates a state's *avars* function by merging the sets of objects assigned to the assignment's source and target. The rule for activity Sanitize is similar, though again changing the labels of objects to L. A field read implies the update of a state's *avars* function, such that the variable's or message's set is merged with the union of all sets, which are assigned in the heap to an allocation site p of an object that is included in the set of objects for the read's receiver y. Eventually, the rule for a field write updates a state's *aheap* function by merging the set of the write's source y with the set of each object, which is assigned in the heap to an allocation site p of an object for the write's receiver x.

C. Soundness and Termination

For showing correctness of the analysis, its soundness and termination has to be proven. As explained in Sect. II, soundness is shown by relating concrete and abstract semantics and proving that the resulting abstract values approximate the concrete values, i.e., every possible concrete value is covered by the abstract values. To this end, we first develop a relation between concrete and abstract states and establish how abstract semantics safely approximates the concrete semantics. We then can show that the presented information flow analysis is sound, i.e., for every variable or message which is not assigned a sensitive object in any reachable abstract state, there is no sensitive object assigned in any reachable concrete state.

Relating concrete and abstract states is done based on access paths, which represent lists of field accesses starting at a variable or message and ending at an object:

Definition 6 ([`followPath`], [`abstract_followPath`]*):* An *access path* is an inductively defined as follows:

$$(\mathit{vars}, \mathit{heap}, l) \vdash v :: \mathit{vars}(v)$$

$$\frac{(\mathit{vars}, \mathit{heap}, l) \vdash n :: (p, q) \quad p \neq 0}{(\mathit{vars}, \mathit{heap}, l) \vdash m :: (p, q) :: \mathit{heap}(p)}$$

$$\frac{(\mathit{avars}, \mathit{aheap}) \quad (p, q) \in \mathit{avars}(v)}{(\mathit{avars}, \mathit{aheap}) \vdash v :: (p, q)}$$

$$\frac{(\mathit{avars}, \mathit{aheap}) \vdash m :: (p, q) \quad (p', q') \in \mathit{aheap}(p)}{(\mathit{avars}, \mathit{aheap}) \vdash m :: (p, q) :: (p', q')}$$

In other words, an access path describes for a concrete or an abstract state the possibility to access a certain object via variable or message v. Using this notion allows for defining the abstraction relation relating abstract and concrete states:

Definition 7 ([`compatible`]*):* State $a = (\mathit{avars}, \mathit{aheap})$ *approximates* concrete state $c = (\mathit{vars}, \mathit{heap}, l)$ iff

- $\mathit{heap}(0) = (0, L)$
- $\forall p > l\colon \mathit{heap}(p) = (0, L)$
- $\forall m, p\colon s \vdash (p, q) :: m \Rightarrow p \leq l$
- $\forall v_1, v_2, m_1, m_2, q_1, q_2, p \neq 0\colon$
  $c \vdash v_1 :: m_1 :: (p, q_1) \wedge c \vdash v_2 :: m_2 :: (p, q_2)$
  $\Rightarrow \exists m'_1, m'_2, q'_1, q'_2, p'\colon$
  $a \vdash v_1 :: m'_1 :: (p', q'_1) \wedge a \vdash v_2 :: m'_2 :: (p', q'_2)$
- $\forall v, m, p \neq 0\colon c \vdash v :: m :: (p, H)$
  $\Rightarrow \exists m', p'\colon a \vdash v :: m' :: (p', H)$

An abstract state a is thus an approximation for a concrete state c, if, besides some side conditions, there holds: (1) for two objects (p, q_1), (p, q_2), which share a common dynamic allocation site p and are accessible via access paths under c, there exist similar objects (p', q'_1), (p', q'_2), which share a common static allocation site p' and are accessible via access paths under a, and (2) for a sensitive object (p, H), which is accessible via an access path under c starting at v, there exists an object (p', H), which is accessible via an access path under a also starting at v. Note that the latter condition is necessary for the soundness of the information flow analysis, while the former condition guarantees soundness of the underlying points-to analysis. Using the abstraction relation allows for proving the safety of the abstract semantics, that is, every concrete state s' reachable from another concrete state s that is approximated by abstract state a, is approximated by an abstract state a' reachable by a. Reachability is thereby defined as usual using the star operator on the reduction relation of the abstract or concrete semantics [12]:

Theorem 1 ([`allocation_site_model_conservative`]*):* The abstract semantics in Fig. 4 is a safe approximation for the concrete semantics in Fig. 2 and 3, i.e., for all concrete states c, c' and for all abstract states a, s.t., $c \rightarrow^* c'$ and a approximates c, exists another abstract state a', s.t., $a \rightarrow^* a'$ and a' approximates c'. ■

In other words, the abstract states resulting from the continuous application of the abstract semantics approximate the concrete states resulting from the application of the concrete semantics. This general result is then used to prove the soundness of the information flow analysis. However, we first need to define the initial state for both semantics:

Definition 8 ([`initState`], [`abstract_initState`]*):* The initial concrete state is $i_c = (\emptyset, \emptyset, 0)$ and the initial abstract state is $i_a = (\{\mathbb{N} \mapsto \emptyset\}, \{\mathbb{N} \mapsto \emptyset\})$.

Theorem 2 ([`andersen_sound`]*):* The analysis is sound, i.e., $\forall v\colon (\forall a = (\mathit{avars}, \mathit{aheap})\colon i_a \rightarrow^* a \Rightarrow \nexists (p, H) \in \mathit{avars}(v))$ $\Rightarrow (\forall c = (\mathit{vars}, \mathit{heap}, l)\colon i_c \rightarrow^* c \Rightarrow \nexists (p, H) \in \mathit{vars}(v)))$. ■

Establishing termination of the information flow analysis is done by proving the existence of a fixpoint solution for the continuous application of the abstract semantics to a given process. To this end, we first need to define a partial order on the set of abstract states (cf. Sect. II):

Definition 9 ([`approx`]*):* The partial order on the set of abstract states is: $(\mathit{avars}, \mathit{aheap}) \leq (\mathit{avars}', \mathit{aheap}')$ iff $\forall v : \mathit{avars}'(v) \subseteq \mathit{avars}(v)$ and $\forall p : \mathit{aheap}'(p) \subseteq \mathit{aheap}(p)$.

Using the fact that for a given process, functions *avars*, *aheap* can be limited to finite sets, i.e., the process' variables, messages, fields, and static allocation sites, and thus can also be the set of valid abstract states, we reuse the well-known result on the existence of a fixpoint if the set of abstract states is well-founded and the abstract semantics is monotone [9]:

Theorem 3 ([`monotonic`]*):* The abstract semantics in Fig. 4 is monotone with respect to the partial order in Def. 9, i.e., for all basic activities a and all abstract states a, b, a', b' with $a \leq b$ and $\mathsf{a} :: ls, a \rightarrow ls, a'$ and $\mathsf{a} :: ls, b \rightarrow ls, b'$ follows $a' \leq b'$. A fixpoint abstract state can be therefore computed by continuous application of the abstract semantics. ■

Eventually, we can state the correctness of the analysis:

Theorem 4: The analysis is correct, i.e., sound and always terminates with a fixpoint solution. ■

IV. Prototype Implementation

Besides the Coq development, we have integrated the analysis as a Camunda Modeler plugin, which is also available online[5]. Camunda Modeler[6] is the modeling tool of the commercial Camunda workflow management system. Using our plugin, a service designer can identify flaws in a modeled service design with respect to potential information leaks.

A screenshot of the plugin applied to a variant of our running example from Fig. 1 is shown in Fig. 5. The example process has been slightly modified such that there is now a potential information leak for outgoing message `reply` (Activity 6.`Anonymization` replaced with assignment `reply` = `rating`). Data-manipulating activities are thereby implemented as simple Groovy scripts, attached to Camunda service tasks. As can be seen, the analysis identifies the information leak and labels the activity, where sensitive information flows to the untrusted sink (message `reply`), with a warning. The plugin provides explanatory information via tooltips.

To this end, the plugin translates a BPMN process into the process represenation defined in Def. 3 and generates an

[5]`https://gitlab.com/t.heinze/soca2018.git`
[6]`https://camunda.com/products/modeler/`

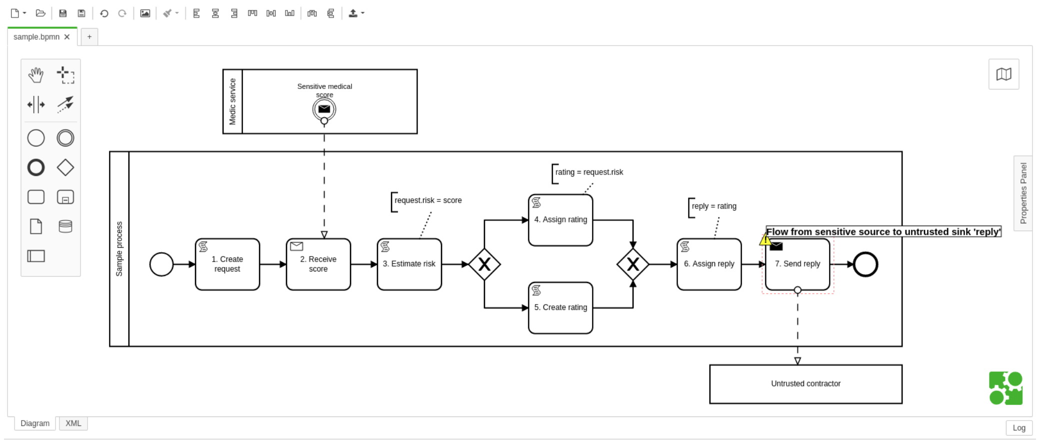

Fig. 5. Prototype implementation as a Camunda Modeler plugin.

OCaml program from the Coq development and the process representation, which is afterwards run. According to its results, the plugin labels activities with warnings for identified potential information leaks. Note that the Coq development is compiled once, when installing the plugin, such that the analysis can be used instantaneously within Camunda Modeler.

V. Related Work

In this section, we discuss related work to the presented analysis, where we consider the state of the art of information flow analysis in the business process and service domain in Sect. V-A. We also provide an overview of certified static analysis with a particular focus on Coq in Sect. V-B.

A. Information Flow Analysis for Business Processes

Information flow has been addressed before in the business process and service domain with varying security models. Models founded on mandatory access control can thereby be distinguished from models using discretionary or role-based access control. We focus on *mandatory access control* with lattice-based security models as in [13], where access is granted using policies based on security classification levels (cf. Sect. III-A). Within these models, in general, access from a lower classified entity to a higher classified entity is prohibited.

Modeling and analysis of stateful service implementations with respect to mandatory access control typically relies on Petri nets as the predominant formalism in this area. The natural choice is to use high-level nets in order to represent security levels in the process model. In [17], colored Petri nets help in the definition of access policies which then can be analyzed with the standard CPN tooling. An extension of workflow nets to algebraic nets is used in [18], to reduce an information flow analysis to a check of the soundness property with the MAUDE model checker. While the previous both approaches focussed on the data flow, other side channels are considered for the more general problem of non-interference in [19]. This work extends previous work on information flow analysis [20]–[22], reducing the problem to reachability in Petri nets. While the prior approaches required a full state space analysis, the more recent tooling in [19] exploits state space reduction techniques implemented in the LoLA model checker. In the same line of research also falls [23].

In spite of the fact that mandatory access control using lattice-based security models has a strong formal foundation, there is, with the exception of the pencil-and-paper proof in [19], a general lack of soundness guarantees for information flow analysis in the domain. Note that this issue is clearly addressed by our certified analysis approach. Furthermore, all the Petri net based techniques rely in one way or another on state space exploration which can raise scalability issues. Additionally, we are not aware of any Petri net approach modeling a heap-like data model as used in modern process engines like Camunda. While a sophisticated choice of abstraction in the used Petri net models may allow for addressing these problems, such an abstraction often requires manual and nontrivial effort. Static points-to/taint analysis instead already includes this abstraction and has a proven feasibility for the analysis of large software systems [4], [15].

B. Certified Static Data Flow Analysis

Proof assistants and theorem provers, in particular Coq, play an increasing role in the area of static analysis and have made their way into the common repertoire of analysis designers. First approaches to verifying static analyses with Coq considered the classical monotone data flow analysis framework, including the groundbreaking work on analyses supporting optimizations in the *CompCert* optimizing compiler [24]. Mechanized verification of the more general abstract interpretation has later been studied [25]–[27]. The presentations in [3], [12] provide introductions into the approach and are accompanied by basic Coq developments for standard analyses, including, e.g., liveness analysis and interval analysis. Among the various Coq developments are also formalizations of points-to analysis [28], the author is however not aware of any Coq development for a unified points-to/taint analysis as presented in this paper.

A general problem of mechanized proofs for static analysis is termination, i.e., the convergence of underlying fixpoint iterations. The corresponding Coq proofs have to be constructive. In monotone data flow analysis, termination is usually guaranteed by the construction of well-founded orderings, which is also used, e.g., in [29] to reason on the existence of fixpoint solutions (cf. Noetherian recursion). Note that we follow the same argument to show termination for our information flow analysis. While well-founded orderings are also used for Coq developments of abstract interpretation [25], they can not always be shown in practice. Instead, abstract interpretation often uses widening accelerators to guarantee termination of fixpoint iteration, e.g., arbitrarily bounding the number of iterations [24]. Some approaches even restrict themselves to only prove partial correctness [26].

Another more recent approach on certified static analysis using Coq, named *a posteriori validation*, does not prove the soundness of static analyzers but rather follows a proof-carrying code approach. To this end, the result of an existing and untrusted analysis is checked for each run by a validator for correctness, which is then verified in Coq instead of the analysis. As the validator is typically of lesser complexity compared to the static analysis, the approach helps in reducing the proof effort, while increasing analysis' expenses. An example for a posteriori validation is slicing analysis in [30].

In previous work [31], [32], the author already argued for the use of the Coq proof assistant in the business process domain and sketched the idea of a certified information flow analysis. However, this work has been preliminary and did not cover aspects like, e.g., termination, sanitizers, and the structural semantics discussed in this paper.

VI. Conclusion

In this paper, we have advocated the use of certified data flow analysis for the analysis of service implementations specified in terms of distributed business processes. In this way, process auditing scenarios using automated process analysis are enriched by mechanically verifiable certificates for analysis correctness. By way of the example of an information flow analysis, we showed the feasibility of the approach, providing a Coq development of the analysis including its soundness and termination proof based upon abstract interpretation.

Our information flow analysis is flow-insensitive [15], which means that the execution order of process activities is basically ignored, possibly resulting in a too coarse approximation of information flow. Note that flow insensitivity is a usual measure for lowering analysis effort. For improving precision, we can though rebase the analysis onto the process representation of *extended workflow graphs* [33], [34]. While the analysis in this way can regain a certain degree of flow sensitivity, a Coq formalization of extended workflow graph and its translation is required, which is subject of future work.

Acknowledgment

I would like to thank Adam Chlipala for providing the Coq development of the points-to analysis underlying this research.

References

[1] M. Leitner and S. Rinderle-Ma, "A Systematic Review on Security in Process-Aware Information Systems - Constitution, Challenges, and Future Directions," *Inf. Softw. Technol.*, vol. 56, no. 3, pp. 273–293, 2014.

[2] R. Accorsi, L. Lowis, and Y. Sato, "Automated Certification for Compliant Cloud-based Business Processes," *BISE*, vol. 3, no. 3, pp. 145–154, 2011.

[3] F. Besson, D. Cachera, T. Jensen, and D. Pichardie, "Certified Static Analysis by Abstract Interpretation," in *FOSAD 2007/08/09*. Springer, 2009, pp. 223–257.

[4] N. Grech and Y. Smaragdakis, "P/Taint: Unified Points-to and Taint Analysis," *PACMPL*, vol. 1, no. OOPSLA, pp. 102:1–102:28, 2017.

[5] T. S. Heinze and W. Amme, "Sparse Analysis of Variable Path Predicates Based upon SSA-Form," in *ISoLA'16 (1)*. Springer, 2016, pp. 227–242.

[6] T. S. Heinze, W. Amme, and S. Moser, "Compiling More Precise Petri Net Models for an Improved Verification of Service Implementations," in *SOCA 2014*. IEEE, 2014, pp. 25–32.

[7] W. Amme, A. Martens, and S. Moser, "Advanced verification of distributed WS-BPEL business processes," *IJBPIM*, vol. 4, no. 1, pp. 47–59, 2009.

[8] O. Kopp, R. Khalaf, and F. Leymann, "Deriving Explicit Data Links in WS-BPEL Processes," in *SCC 2008 (2)*. IEEE, 2008, pp. 367–376.

[9] G. A. Kildall, "A Unified Approach to Global Program Optimization," in *POPL'73*. ACM, 1973, pp. 194–206.

[10] P. Cousot and R. Cousot, "Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs," in *POPL'77*. ACM, 1977, pp. 238–252.

[11] G. D. Plotkin, "A Structural Approach to Operational Semantics," *J. Log. Algebr. Program.*, vol. 60–61, pp. 17–139, 2004.

[12] X. Leroy, "Mechanized semantics," in *Logics and Languages for Reliability and Security*. IOS Press, 2010, pp. 195–224.

[13] D. E. Denning, "A Lattice Model of Secure Information Flow," *Comm. ACM*, vol. 19, no. 5, pp. 236–243, 1976.

[14] V. B. Livshits and M. S. Lam, "Finding Security Vulnerabilities in Java Applications with Static Analysis," in *USENIX Security 2005*, 2005, pp. 271–286.

[15] M. Sridharan, S. Chandra, J. Dolby, S. J. Fink, and E. Yahav, "Alias Analysis for Object-Oriented Programs," in *Aliasing in Object-Oriented Programming*. Springer, 2013, pp. 196–232.

[16] B. Kiepuszewski, A. H. M. ter Hofstede, and C. J. Bussler, "On Structured Workflow Modelling," in *CAiSE 2000*. Springer, 2000, pp. 431–445.

[17] K. Juszczyszyn, "Verifying Enterprise's Mandatory Access Control Policies with Coloured Petri Nets," in *WETICE 2003*. IEEE, 2003, pp. 184–189.

[18] K. Barkaoui, R. B. Ayed, H. Boucheneb, and A. Hicheur, "Verification of Workflow Processes Under Multileven Security Considerations," in *CRiSIS 2008*. IEEE, 2008, pp. 77–84.

[19] R. Accorsi, A. Lehmann, and N. Lohmann, "Information leak detection in business process models: Theory, application, and tool support," *Inf. Sys.*, vol. 47, pp. 244–257, 2015.

[20] R. Accorsi and C. Wonnemann, "InDico: Information Flow Analysis of Business Processes for Confidentiality Requirements," in *STM 2010*. Springer, 2010, pp. 194–209.

[21] ——, "Strong non-leak guarantees for workflow models," in *SAC 2011*. ACM, 2011, pp. 308–314.

[22] R. Accorsi, C. Wonnemann, and S. Dochow, "SWAT: A Security Workflow Analysis Toolkit for Reliably Secure Process-aware Information Systems," in *ARES 2011*. Springer, 2011, pp. 692–697.

[23] S. Frau, R. Gorrieri, and C. Ferigato, "Petri Net Security Checker: Structural Non-interference at Work," in *FAST 2008*. Springer, 2008, pp. 210–225.

[24] X. Leroy, "Formal Verification of a Realistic Compiler," *Comm. ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[25] D. Pichardie, "Building Certified Static Analysers by Modular Construction of Well-founded Lattices," *Electr. Notes Theor. Comput. Sci.*, vol. 212, pp. 225–239, 2008.

[26] J.-H. Jourdan, V. Laporte, S. Blazy, X. Leroy, and D. Pichardie, "A Formally-Verified C Static Analyzer," in *POPL 2015*. ACM, 2015, pp. 247–259.

[27] S. Blazy, V. Laporte, A. O. Maroneze, and D. Pichardie, “Formal Verification of a C Value Analysis Based on Abstract Interpretation,” in *SAS 2013*. Springer, 2013, pp. 324–344.

[28] V. Robert and X. Leroy, “A Formally-Verified Alias Analysis,” in *CPP 2012*. Springer, 2009, pp. 11–26.

[29] D. Cachera, T. P. Jensen, D. Pichardie, and V. Rusu, “Extracting a data flow analyser in constructive logic,” *Theor. Comput. Sci.*, vol. 342, no. 1, pp. 56–78, 2005.

[30] S. Blazy, A. O. Maroneze, and D. Pichardie, “Verified Validation of Program Slicing,” in *CPP 2015*. ACM, 2015, pp. 109–117.

[31] T. S. Heinze, “Towards Certified Data Flow Analysis of Business Processes,” in *ZEUS 2017*. CEUR-WS.org, 2017, pp. 1–3.

[32] ——, “Schritte zu einer zertifizierten Informationsflussanalyse von Geschftsprozessen,” in *ZEUS 2018*. CEUR-WS.org, 2018, pp. 24–31.

[33] T. S. Heinze, W. Amme, and S. Moser, “A Restructuring Method for WS-BPEL Processes Based on Extended Workflow Graphs,” in *BPM 2009*. Springer, 2009, pp. 211–228.

[34] ——, “Static analysis and process model transformation for an advanced business process to Petri net mapping,” *Softw., Pract. Exper.*, vol. 48, no. 1, pp. 161–195, 2018.