Data-Dependent Confidentiality in DCR Graphs

Eduardo Geraldo
NOVA University Lisbon,
NOVA LINCS
Caparica, Portugal
e.geraldo@campus.fct.unl.pt

João Costa Seco
NOVA University Lisbon,
NOVA LINCS
Caparica, Portugal
joao.seco@fct.unl.pt

Thomas Hildebrandt
Copenhagen University
Copenhagen, Denmark
hilde@di.ku.dk

Abstract

We present *DCRSec*, a confidentially aware declarative process language with data that employs data-dependent security levels and an information flow monitor that prevents the violation of information flow policies. Data-dependent security levels have been used to shape precise information flow policies and properly identify security compartments. We use an illustrative example to show that it also models process instances in a flexible but precise way.

The semantics of the language is based on a version of the Dynamic Condition Response Graph language, which allows for declaring data-aware, event-based processes with finitary and infinitary computations subject to liveness properties and dynamically spawned sub-processes. The key technical contribution is to provide a termination-insensitive information flow monitor and prove non-interference, a soundness property, and transparency in all traces of *DCRSec* processes.

ACM Reference Format:
Eduardo Geraldo, João Costa Seco, and Thomas Hildebrandt. 2023. Data-Dependent Confidentiality in DCR Graphs. In *International Symposium on Principles and Practice of Declarative Programming (PPDP 2023), October 22–23, 2023, Lisboa, Portugal.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3610612.3610619

1 Introduction

Confidentiality is a key property of software systems and one of the most difficult to ensure. The digitalisation of society and the ensuing digital processing of sensitive and personal data raises the importance of confidentiality and privacy in information systems. This reflects in the general data protection regulation (GDPR) requirement for data protection by design and purpose-specific use of personal data.

There are many strategies for enforcing data confidentiality. However, information flow control is the only one that effectively maintains information secrecy in a system. Information flow control goes back to the seminal work of Denning [17], where she proposes a fixed lattice model, base for type systems [52], monitors [7], and hybrid approaches [27, 49] aiming to ensure data confidentiality. Some approaches trade the fixed-lattice model for more flexible mechanisms such as the decentralized label model [39] or value-dependent security lattices [25, 28, 38]. Independently of the means of enforcement, information flow control is used in many programming languages like Java [26, 39, 47], JavaScript [44], and OCaml [46], or tools like JOANA [47], FlowDroid [5], TaintDroid [20], and Snitch [27]. However, the digitalisation of businesses and public services processes and the tightening of data protection regulations call for high-level treatments of information flow control that acknowledge the business processes and workflows employed.

A place to anchor such treatment is the field of *process aware information systems* [50], which include explicit representations of the business or workflow processes they manage. Many formalisms and notations support the specification of business and workflow processes, both imperative and declarative. Some formalisms support access control, but few allow for information flow control. A notable exception is the analysis of non-interference in Petri Nets [11], applied to business processes in [3, 37]. This work, however, focuses on processes' control flow, abstracting from the data. There is also work applying the declarative specification and verification of information flow control in process calculi; some consider only the control flow and the sequence of messages [1, 35], while others integrate imperative languages in the calculus, expanding the verification scope [34].

The formal connection between processes and data is a well-known problem addressed by different approaches in the literature [13, 24, 51]. The process-data connection may help overcome one of the disadvantages of traditional information flow control approaches for imperative languages. Existing tools can be too strict, making it too difficult to adapt to real-world scenarios. In this context, we highlight the contribution of data-dependent information flow control [38], which is a promising approach to defining security compartments that capture the essence of realistic software.

In this paper, we define *DCRSec* to join data-dependent information flow control with flexible and declarative data-aware Dynamic Condition Response (*DCR*) graphs. *DCRSec* derives from DCR Graphs [32], including sub-processes [15, 16] and data [13, 24], and allows for input and computation

PPDP 2023, October 22–23, 2023, Lisboa, Portugal

ACM ISBN 979-8-4007-0812-1/23/10.
https://doi.org/10.1145/3610612.3610619

events. We assign each event a role for the event executor [31], implementing an access control layer, and a security level, defining the secrecy of events and their data. Information flow policies and access control policies may be intertwined depending on the intended security guarantees. For instance, role-based access control roles easily transpose to a security lattice, allowing to define information flow policies. [10]. In information flow control, security levels compartmentalize data and computations [38, 39], ensuring the preservation of confidentiality (or integrity) [17]. We use value-dependent security levels [38] to define parametrised security compartments in a precise way.

We define the semantics of the language with an inlined information flow monitor that enforces termination-insensitive non-interference [6, 9]. Non-interference ensures that information is only observable with the appropriate security clearance. We can also observe that the use of data-dependent security compartments is a clever form of defining what is a process instance. Instead of artificially separating different process instances, we may use a projection function that combines events that belong to a common security compartment defined by a given set of security labels and parameters.

We present a motivating example in Section 2, introducing the definition of a security lattice and the constructs of *DCRSec*. In Section 3 we illustrate how a process evolves and how information is projected onto the different security compartments. Section 4 presents the semantics of *DCRSec* in a formal setting. We follow with a presentation and proof of non-interference in Section 5.1. We conclude by discussing some related work and enumerating topics for future work.

2 Motivating Example

We now present a running example, introduce the constructs of *DCRSec*, and showcase its confidentiality-aware features. We resort to a simple medical appointment process where clerks, doctors and patients perform different activities, and where sensitive data must be kept confidential.

Before we start, we define a value-dependent security lattice and its value-dependent security levels to support the rich security policies required for the example.

2.1 Security Lattice

We start by defining a set of security levels, each matching a distinct security clearance and security compartment, a conceptual information container aggregating all equally sensitive information. Security compartments are identical to security clearances, except that the former applies to data while the latter applies to entities accessing data. The security lattice establishes a hierarchy between security compartments, detailing how information may flow between them; compartment hierarchies reflect in clearance hierarchies. In the context of the example presented here, we define security levels for clerks, doctors, and patients. A classic lattice does not let us place information about separate patients in separate security compartments, a basic requirement in such a scenario; a patient must not access information about other patients. We define a value-dependent security lattice [38], resorting to parameterised security levels better suited to specify the intended information flow policies.

Information flow control and access control are different techniques, relying on distinct models and mechanisms. However, for the sake of simplicity, we use designations for security levels on par with equally parameterised system roles used to define access control policies [12]. A principal may have many roles and any security clearance. We adopt a simple access control model, that can be used in future developments as the basis for an integrity check. It is also important to understand that integrity and confidentiality must co-exist. For instance, confidentiality-based information flow control does not prevent a patient from changing another patient's information. That is an issue that can be addressed by the (dual) integrity-based information flow analysis [36], or simply by access control techniques [2, 12, 23].

We define several non-parameterised security labels like *Admin* and *Public*, reflecting the most and least confidential compartments in the lattice. Among the non-parameterised security labels is *Clerk*, meaning that users with security clearance *Clerk* share the same security compartment. We define both the security labels for doctors and patients as parametrised labels. For instance, the label $Doctor(d)$ defines the security compartment visible to the principal representing a doctor with identifier d. Generally, with parametrised labels, information flows from a security level $Doctor(d)$, for any d, to an implicitly defined compartment $Doctor(\top)$ representing the information of all doctors. Similarly, information flows from the less secret, implicitly defined, compartment $Doctor(\bot)$, containing information accessible to all doctors to $Doctor(d)$. Given a set of doctor identifiers $\mathcal{D}$ (*e.g.* database identifiers), and patient identifiers $\mathcal{P}$ (*e.g.* names), the resulting security lattice, $\mathcal{L} = (\mathcal{SC}, \rightarrow, \sqcup, \sqcap)$ is defined with the security labels

$$\begin{aligned}\mathcal{SC} \triangleq &\{\top, \bot, Admin, Clerk, Public\} \\ &\cup \{Doctor(d) \mid d \in \mathcal{D}\} \cup \{Doctor(\top), Doctor(\bot)\} \\ &\cup \{Patient(p) \mid p \in \mathcal{P}\} \cup \{Patient(\top), Patient(\bot)\}\end{aligned}$$

and the relation *flows* ($\rightarrow$) as illustrated in Figure 1. The least upper bound ($\sqcup$) and greatest lower bound ($\sqcap$) denote, respectively, the smallest security level greater or equal to and the greatest security level that is lesser or equal to both arguments. We introduce the order relation $\sqsubseteq$ as the transitive, and reflexive closure of the *flows* ($\rightarrow$) relation.

This lattice is not yet expressive enough to represent the intended policies. According to the lattice, the security labels *Doctor*, *Patient*, and *Clerk* are not hierarchically related, and information cannot flow between them. However, such strict separation is impractical; there are many cases in real systems where such containers should share information. For

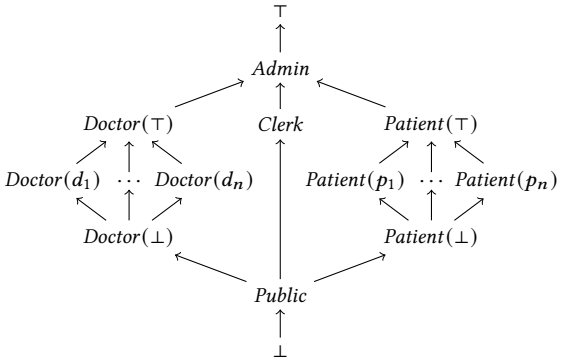

Figure 1. Lattice for the medical appointment process.

instance, a prescription must be known to the doctor prescribing it and the patient who is the subject of the prescription. For this reason, the actual lattice used for the example consists of the powerset [18] of the value-dependent security lattice illustrated in Figure 1 while respecting the existing flow relation. Our security lattice is therefore defined by:

$$\begin{array}{rcl} \mathcal{L}' & = & (\mathcal{P}(\mathcal{SC}'), \rightarrow', \sqcup', \sqcap') \text{ with} \\ \mathcal{SC}' & = & \mathcal{SC} \backslash \{\top, \bot\} \\ A \rightarrow' B & = & \forall\, b \in B.\ \exists\, a \in A.\ a \rightarrow b \\ A \sqcup' B & = & \{\delta \in \mathcal{SC} \mid \forall\, a \in A.\ \forall\, b \in B.\ a \sqcup b = \delta\} \\ A \sqcap' B & = & \{\delta \in \mathcal{SC} \mid \forall\, a \in A.\ \forall\, b \in B.\ a \sqcap b = \delta\} \end{array}$$

We exclude $\top$ and $\bot$ from the set of security levels, as the powerset implicitly includes matching sets: the empty set for $\top$ and the set of all security levels for $\bot$.

Hereafter, we tag each event in the example with a set of security labels from the extended security lattice, *i.e.*, the powerset lattice. To help on this task, we define the following abbreviations for sets of security labels in the extended lattice, where the x marks the component that is parametrised:

$$\begin{array}{rcl} CDP_x(p) & \triangleq & \{Clerk, Doctor(\bot), Patient(p)\} \\ CD_xP_x(d,p) & \triangleq & \{Clerk, Doctor(d), Patient(p)\} \\ CD_x(d) & \triangleq & \{Clerk, Doctor(d)\} \\ CP_x(p) & \triangleq & \{Clerk, Patient(p)\} \\ D_xP_x(d,p) & \triangleq & \{Doctor(d), Patient(p)\} \end{array}$$

Notice that the notion of confidentiality presented above is preserved in this setting. For instance, the security label $\{Clerk, Doctor(d), Patient(p)\}$ is used to tag data visible to clerks and a pair doctor/patient (d, p). This compartment flows to a compartment with level $\{Doctor(d), Patient(p)\}$, where it is visible to the same doctor and patient, but not to clerks; transitively flows to $\{Doctor(\top)\}$, visible to a principal that can see the information of all doctors.

2.2 *DCRSec* Step by Step

Scenario. Consider a process for a medical office to manage its appointments, prescriptions, and the associated payment. To model this scenario, we use a concrete text-based syntax similar to the one used in [13, 24].

Process Definition. A *DCRSec* process is defined by a set of events and the control-flow relations between them (syntactically separated by a semicolon). For simplicity, we leave out data relations from ReGraDa [24], orthogonal to our purpose in this paper. Also, nested process definitions appear in spawn relations, creating data and control flow.

Following the line of ReSeDa [13] and ReGraDa [24], events represent data items. Events store state (*c.f.* Petri nets), forming the persistent portion of a process. Events may be data, input events, or computations events. Input events represent user interactions that feed values into the process and computation events are persistent data items computed from existing events in context. In ReGraDa [24], processes are compiled down to a graph database (data and code) and the events are accessible through a REST API for the user to interact with. We extend the syntax of [13, 24], with security label annotations. The developments in Section 4 will then use an equivalent but more convenient, declarative formalism similar to the one used in [16].

Data or Computation Events. We define three data or computation events to represent entities in our example as

$$d : \mathsf{Doctor}_{(Admin,\ Public)}\ [\{\mathsf{name} = \text{“John”}\}],$$

$$p : \mathsf{Patient}_{(Admin,\ CDP_x(\text{“Jane”}))}\ [\{\mathsf{name} = \text{“Jane”}\}],$$

$$a : \mathsf{Appointment}_{(CD_x(d),\ CD_xP_x(d,\text{“Jane”}))} \left[\left\{\begin{array}{l} \mathsf{patient} = p, \\ \mathsf{doctor} = d, \\ \mathsf{date} = \text{“2021/01/01”} \end{array}\right\}\right]$$

using identifiers d, p, and a (to refer to them in events in the same scope), labels Doctor, Patient, and Appointment (to select them as collections or types), access control levels *Admin*, and $CD_x(\mathrm{d})$ (to define who can execute them), security levels *Public*, $CDP_x(\text{“Jane”})$, and $CD_xP_x(\mathrm{d}, \text{“Jane”})$ (to define the security levels of events in relation to the values in scope), and expressions to define the values in the events {name = “John”}, {name = “Jane”}, and {patient = p, doctor = d, date = “2021/01/01”}. Notice how the security level and expressions in a refer to p and d and their values.

Events store data and a marking, meta-data about the execution. The marking of an event e consists of the boolean values *executed*, *included*, and *pending*, and a value *value*:

- *executed* - tells if e has executed at least once.
- *included* - tells if e is available for execution.
- *pending* - tells if e has yet to execution. If true, e must execute for the *DCRSec* graph to reach a stable state.
- *value* - holds the value of e; provided as input if e is an input event. Results from the last execution otherwise.

The marking is not shown in the syntax of the events above as they have the default values for fields executed (false), included (true) and pending (false).

Security Levels and Access Control. The access control level *Admin*, the first element of the pair in the events above, indicates that only a system administrator can execute them,

while the second element specifies who can see the result. For instance, only an administrator can add a new doctor to the system, but everyone can see the system's doctors. A patient's data should only be visible to the patient and all doctors and clerks in the system. Thus, we use security level CDP_x("Jane") to tag the data of the patient with the name "Jane". For the sake of simplicity, we use the patient's name as their "unique" identifier. Information about an appointment concerns clerks, the named doctor and the named patient, thus we use the level $CD_xP_x(d,$ "Jane"). Label parameters can be values, strings, or events. Finally, a doctor's name is visible to all users. Since events are seen as data, for a user to see an event, the user's clearance must be higher or equal to the event's access control level.

Executing events. A process achieves progress by executing events. In computation events, this amounts to evaluating the expressions in them. Input events include providing the requested values and storing them in the corresponding markings. The given or computed values are cached in the field *value* of the corresponding event's marking. The execution of events has an effect on the state of the process according to the existing relations.

Information flow. Based on their security levels, we can compartmentalise events. The system preserves confidentiality if the effects of executing an event are only visible to the current compartment and all security compartments greater or equal in the security lattice, *i.e.* "more secret" security compartments. Direct dependencies between events establish explicit information flows and are legitimate if referenced events have lower or equal security levels. For instance, appointment a can reference p since the appointment's label ($CD_xP_x(d,$ "Jane")) is higher than the patient's security label (CDP_x("Jane")). Intuitively, in the opposite direction, it would be illegal to create an event with security level *Public* referring to a or any other appointment record.

Implicit flows arise from the control-flow graph revealing the confidential values of the conditions of the guards. The effects of evaluating guards or changing the marking of events can only be visible in allowed security compartments according to the security lattice.

Input Events. The other kind of event that exists in a *DCRSec* process is that of input events. They are defined in the following examples:

r : !reportAppointment$_{(Doctor(d),\ CD_xP_x(d,\text{"Jane"}))}$ [? : {diagnosis : String}],
s : %prescription$_{(Doctor(d),\ D_xP_x(d,p.\texttt{name}))}$ [? : {prescription : String}],
i : %invoice$_{(Clerk,\ CP_x(\text{"Jane"}))}$ [? : {amount : Number}],
y : %payment$_{(Patient(\text{"Jane"}),\ CP_x(\text{"Jane"}))}$ [? : Boolean]

Like data events, input events have a marking and are defined by an identifier ($r, s, i,$ or y) and a label (reportAppointment, prescription, invoice, or payment). Additionally, they include a question mark and a datatype specifying the kind of values they accept. Types include basic datatypes (String, Number, Boolean), records, and lists. Enabled input events are visible to execute at the border of the runtime system (*c.f.* an API).

The syntactic mark % denotes the exclusion of events prescription, invoice, and payment from the *DCR* graph; excluded events cannot execute or have any impact on existing constraints (*c.f.* [32]). The notation ! used above in event reportAppointment marks it as pending, meaning its execution is mandatory for the graph to reach a stable state.

DCR Relations. A *DCRSec* process also supports relations between events, which define the control flow of the process. There are 6 different relations: $\rightarrow+$, $\rightarrow\%$, $\bullet\rightarrow$, $\rightarrow\bullet$, $\rightarrow\diamond$, and $\twoheadrightarrow$. The first three relations in the sequence are used to change the state of the process. Consider the syntactic declarations using the relation $\rightarrow+$ (include):

r : reportAppointment $\rightarrow+$ s : prescription,
r : reportAppointment $\rightarrow+$ i : invoice,
r : reportAppointment $\rightarrow+$ y : payment,

the relations above mean that the execution of event r includes the s, i, and y and enables their execution. For a better understanding of the example, relations include event labels.

The relation $\rightarrow\%$ (exclude) has the opposite outcome

r : reportAppointment $\rightarrow\%$ r : reportAppointment,
s : prescription $\rightarrow\%$ s : prescription,
i : invoice $\rightarrow\%$ i : invoice,
y : payment $\rightarrow\%$ y : payment

here, the exclude relation ensures that the events above only execute once.

Consider now the response relation $\bullet\rightarrow$ in the relations

r : reportAppointment $\bullet\rightarrow$ i : invoice,
i : invoice $\bullet\rightarrow$ y : payment

This means whenever event r with label reportAppointment executes, event i with label invoice becomes pending, *i.e.* selected for mandatory execution. Similarly, whenever i executes, event y with label payment must eventually execute.

The next two relations do not change the materialized state of the process but define the control flow of the process. Consider the condition relation ($\rightarrow\bullet$)

i : invoice $\rightarrow\bullet$ y : payment

it prevents the execution of event y with label payment before the execution of event i with label invoice. Similarly, the milestone relation ($\rightarrow\diamond$) prevents the execution of the event referred to by the right-hand side of the relation while the event on the left-hand side is pending.

Like data dependencies, control-flow relations must follow the information flow policies of the process. Any transgression regarding control flow would lead to (illegal) indirect flows in the process. For example, event r with label reportAppointment and security level $CD_xP_x(d,$ "Jane") may

flow to, or affect, event s with label prescription and, the "more secret" security level $D_xP_x(d, \text{"Jane"})$. The opposite relation is not allowed as it would allow clerks to see events not visible before, thus leaking information.

Finally, the spawn relation, $\twoheadrightarrow$, creates new events and relations in a process. A spawn relation includes a (sub)process definition as in the fragment

c : createDoctor$_{(Admin,\ Public)}$[? : {name : String}]
;
$c \twoheadrightarrow$ {
  d : Doctor$_{(Admin,\ Public)}$[{name = @trigger.name}]
  c : createAppointment$_{(Clerk,\ CD_xP_x(d,\bot))}$[? : {...}]
  ...
}

Whenever event c with label createDoctor executes, it creates events with labels Doctor and createAppointment, placing them at the top level of the process. We assign fresh identifiers to the new events, allowing the same relation to create a multiplicity of events (*e.g.*, multiple doctors). The relations in the sub-process are included in the process referencing the newly created events and their identifiers.

The reserved identifier @trigger represents the event triggering the spawn relation and its value and marking are available to the new events. In the example above, the value of field name of event c with label createDoctor is used and stored in the new event d with label Doctor.

New events must follow the same information flow control policies by having a security level equal to or higher than that of the triggering event. For instance, both *Public* and $CD_xP_x(d, \bot)$ from Doctor and createAppointment conform to this requirement. The creation of a "less secret" or unrelated event would have effects outside the original security compartment and is deemed as an information leak.

2.3 Medical Appointment Process

We specify the initial state of our example, joining all features above, in the process syntactically depicted in Figure 2. The complete *DCRSec* graph is defined by the input events cd and cp, with labels createDoctor and createPatient respectively, and a set of *DCR* relations [14, 15]. We say that cd and cp are locally defined in relation to this process level. Notice that only a principal with *Admin* access control level may execute the events. The result of creating a doctor belongs to the security compartment with security level *Public* since it is observable to anyone. The result of creating a patient has the security level $CDP_x()$, which means that only clerks, doctors and the patient themselves can see the result. Notice that before executing an event, its value is $\bot$ [13, 24], and therefore the security level is pushed downward to the corresponding versions of the parametrised security labels.

Sub-processes and Spawn. After the definition of local events, separated via a (;), we can find the *DCR* relations of the top-level graph. In Figure 2 there are two spawn ($\twoheadrightarrow$) relations at the top level. The execution of event cp : createPatient triggers the first spawn relation, which takes the sub-process definition, on the right-hand side of the relation and instantiates it at the top level, alpha renaming existing local identifiers. The sub-process in the example defines a computation event with label Patient and local identifier p. The expression in p refers to an event with the reserved identifier @trigger; using @trigger on an input event allows access to the value provided in each interaction. Expression evaluation occurs upon event creation or execution.

That said, event p with label Patient has the security level CDP_x(@trigger.name), meaning its value is visible to clerks, doctors, and one patient identified by the name provided as input. The value of the triggering event is used to instantiate the new event and its data and meta-data, including security levels. In this case, the security level of the value of the event gets indexed with the name of the patient.

Our information flow monitor prevents an event from triggering effects on another event at a non-higher or non-equal security level; secret events must not lead to changes in visible, "less secret" events. The information flow monitoring goes beyond event relation, extending to data computations used, for instance, in the guards of control-flow relations. In such cases, the monitor has to consider the security levels of events and that of the guard. So, an event cannot trigger an effect on another event whose security compartment is not equal to or higher than the combined levels of the triggering event and the events used in its guard.

The second spawn relation repeats the same structure, with the event cd having label createDoctor on the left-hand side pointing to a sub-process defining a data event d with label Doctor, to represent a new doctor, and an input event with label createAppointment to receive patient data and the time of a new appointment. Different instances of computation events with the same label work like nodes or rows in a database and, thus, labels are analogous to labels in graph databases or the name of a table in a relational database. As a convention, we use capitalized labels for computation events and lower-case labels for input events, or user actions. By scoping identifiers, the events ca with the label createAppointment created when each sub-graph is instantiated, are bound to the current value of identifier d, which denotes the doctor in the local scope.

The new events and relations created by a spawn relation with ca are bound to a concrete event denoted by identifier d, representing the doctor in the process. In our semantics, the new relations produced are closures tied to the local values from the context where they were instantiated. The remainder of the inner process contains events a : Appointment, r : reportAppointment, s : prescription, i : invoice, and y : payment, some of which excluded initially (with the syntactic mark %). The sub-process also includes the control-flow relations shown above in the example.

```
cd : createDoctor(Admin, Public) [? : {name : String}]
cp : createPatient(Admin, CDP_x(value.name)) [? : {name : String}]
;
cp : createPatient ⇻ { p : Patient(Admin, CDP_x(@trigger.name)) [{name = @trigger.name}]}
cd : createDoctor ⇻ {
    d : Doctor(Admin, Public) [{name = @trigger.name}]
    c : createAppointment(Clerk, CD_x P_x(d,⊥)) [? : {  patient : Patient, doctor : Doctor, date : DateTime  }]
    ;
    createAppointment ⇻ {
        a : Appointment(CD_x(d), CD_x P_x(d,@trigger.value.p.name)) [@trigger.value],
        r : !reportAppointment(Doctor(d), CD_x P_x(d,@trigger.value.p.name)) [? : {diagnosis : String}],
        s : %prescription(Doctor(d), D_x P_x(d,@trigger.value.p.name)) [? : {prescription : String}],
        i : %invoice(Clerk, CP_x(@trigger.value.p.name)) [? : {amount : Number}],
        y : %payment(Patient(@trigger.value.p.name), CP_x(@trigger.value.p.name)) [? : Boolean]
        ;
        r : reportAppointment →+ s : prescription,
        r : reportAppointment →+ i : invoice,
        i : reportAppointment •→ y : invoice,
        r : reportAppointment →+ y : payment,
        i : invoice →• y : payment,
        i : invoice •→ y : payment,
        r : reportAppointment →% r : reportAppointment,
        s : prescription →% s : prescription,
        i : invoice →% i : invoice,
        y : payment →% y : payment
    }
}
```

Figure 2. The medical appointment process in *DCRSec*

The security compartments of new events are indexed by the doctor and patient in context, creating separate compartments where each doctor only sees the diagnosis of their patients, the patient and clerks see accounting information, and the doctor and the patient sees the prescription. This structure is seen in the static definition of the relations in the sub-process and is monitored by the language semantics.

Process Trace. We now explore a sample trace to illustrate the semantics of *DCRSec*. We start with the process defined in Figure 2 and finish with the graph in Figure 3. We omit the existing control flow relations, as they are the static part of the process. The initial state is defined by the input events

cd : createDoctor$_{(Admin,\ Public)}$
cp : createPatient$_{(Admin,\ CDP_x(value.\mathtt{name}))}$

When a principal with role *Admin* executes event *cd* with the input {name = "John"}, we reach a state with the events,

d_0 : Doctor$_{(Admin,\ Public)}$ [{name = "John"}]
c_0 : createAppointment$_{(Clerk,\ CD_x P_x(d,\bot))}$

Notice the new identifiers obtained by alpha renaming the spawned sub-process under the triggered spawn relation. By considering the static binding of identifiers, when triggered, c_0 will spawn events associated with d_0 (*c.f.* a closure); illustrated by a dashed line between d_0 and c_0. Also, the execution of event c_0 has an effect only visible in the security compartment that pertains to clerks, the doctor in scope, and the patient whose name was given as input. Now, a principal with role *Admin*, executing event *cp* with label createPatient and a value {name = "Jane"} creates one more event

p_0 : Patient$_{(Admin,\ CDP_x(\text{"Jane"}))}$ [{name = "Jane"}]

Now that we have our subjects, a principal with role *Clerk* can execute the event c_0 with the input value {p = p_0, d = d_0, t = "2024-01-01"}.

the following events are added to the graph, alpha renaming local identifiers to avoid clashes and preserve dependencies.

$$a_0 : \text{Appointment} \left[\left\{ \begin{array}{l} \text{patient} = p_0, \\ \text{doctor} = d_0, \\ \text{date} = \text{"2024-01-01"} \end{array} \right\} \right]$$

r_0 : !reportAppointment$_{(Doctor(d_0),\ CD_x P_x(d_0,p_0))}$
s_0 : %prescription$_{(Doctor(d_0),\ D_x P_x(d_0,\text{"}Jane\text{''}))}$
i_0 : %invoice$_{(Clerk,\ CP_x(\text{"Jane"}))}$
y_0 : %payment$_{(Patient(\text{"Jane"}),\ CP_x(\text{"Jane"}))}$

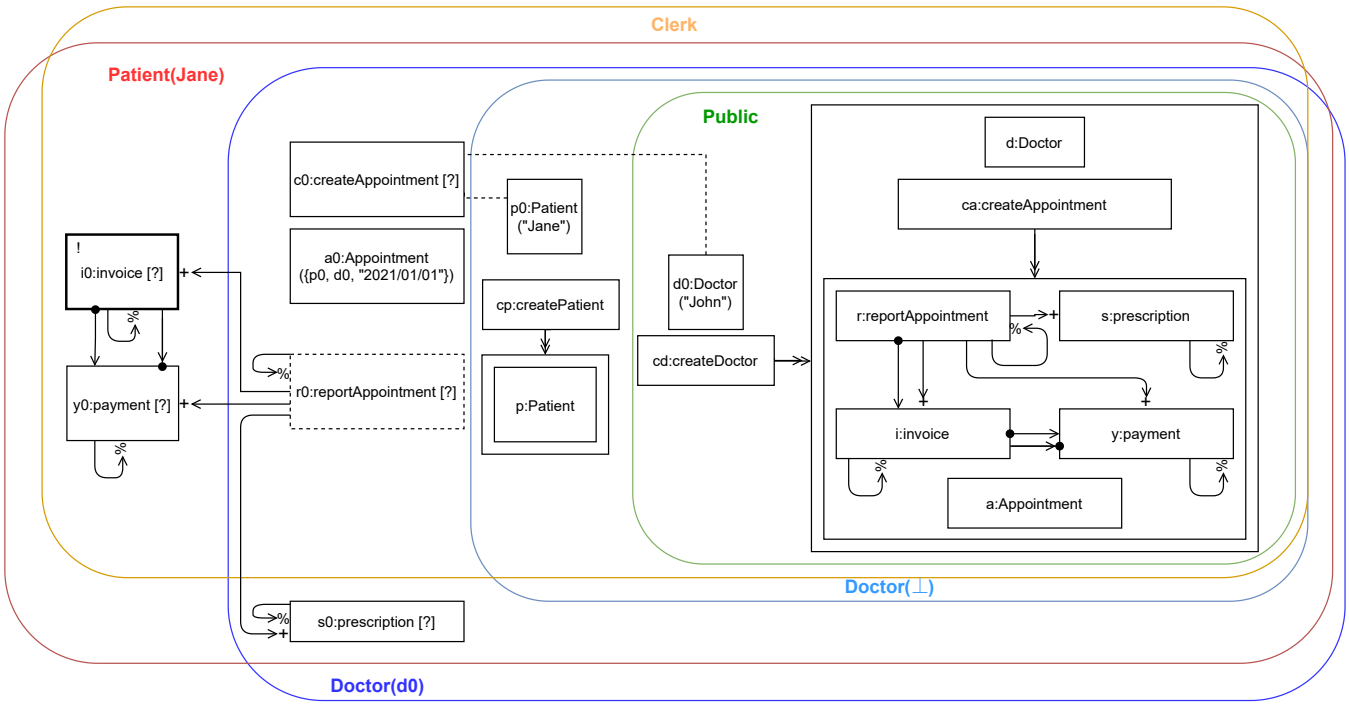

Figure 3. Security compartments for the medical appointment process.

Notice the indexing of the new events with the given patient name. Now, we have all data for the process to proceed and for the appointment to happen. The next task is for the doctor to create a report for it by executing event r_0, with a value for the diagnosis. For instance, {diagnosis = "flu"}.

The principal with role $Doctor(d_0)$ can execute event r_0, statically bound to the appointment a_0 and patient p_0. The modified state contains the following events

$$r_0 : \%\text{reportAppointment}_{(Doctor(d_0),\ CD_x P_x(d_0,p_0))}\ \{\text{diagnosis} = \text{"flu"}\}$$
$$s_0 : \text{prescription}_{(Doctor(d_0),\ D_x P_x(d_0,\text{"}Jane\text{"}))}$$
$$i_0 : !\text{invoice}_{(Clerk,\ CP_x(\text{"Jane"}))}$$
$$y_0 : \text{payment}_{(Patient(\text{"Jane"}),\ CP_x(\text{"Jane"}))}$$

Using the include and exclude relations in the process (see Figure 2), event r_0, reportAppointment, is no longer pending and no longer available for execution (it is excluded), and the previously excluded events, s_0, i_0, and y_0 are now included with i_0 pending. The process can then continue with the payment part of the process. From this sequence of events, we can observe the evolution of the security compartments of the events and data. As an invariant, we impose that one event can only have an effect at the same security level or higher. For instance, the event with label createAppointment whose result is visible to clerks, a doctor, and a patient generates effects (creates events) at a level where only the doctor and the patient can see them. Although the structure of data is flat and organized with labels, we define the sets of visible events so that all other patients and doctors cannot see data generated in the process of a specific appointment.

Security compartments. We depict the security compartments of the process in the final snapshot of the example in Figure 3. We use encircled areas to represent each security compartment and use inclusion to represent the order in the lattice. The intersection of areas represents the sharing of security compartments in the spirit of our powerset-based lattice $\mathcal{L}'$. Notice for instance, that event prescription is visible to one doctor (d_0) and one particular patient (p_0), while event Appointment is visible also to clerks. The dashed event reportAppoitment is excluded in this snapshot, and the stronger border event invoice with a "!" symbol is pending. Intuitively, if an event is created from one area to an inner area, or affects an event in an inner area, an information leak occurs. We want to avoid that. Notice that this graph is well formed, *i.e.*, does not exhibit illegal information flows, and all relations and all spawning of events occur in the outer direction of the security compartments. Furthermore, we place the sub-processes in the security compartment of the triggered event as the spawned events are defined to be in parametrised security compartments which are not yet instantiated. When instantiated, they are placed in the same or outer security compartments.

3 *DCRSec* graphs

We now present a declarative definition and a labelled transition system to denote the semantics of *DCRSec* graphs. The

definition is based on DCR graphs with time and data introduced in [33], but disregarding time and adding dynamically spawned sub-processes, access control and security labels.

Definition 3.1 (*DCRSec* graphs)**.** A *DCRSec* graph is a confidentiality-aware DCR Graph with data, inductively defined by $(E, F, D, L, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\!\!\!\!\rightarrow, w, \text{pc})$ where

1. E is a finite set of events,
2. $F \subseteq E$ is a set of local events,
3. D: $F \rightarrow \text{Exp} \uplus \{?\}$ is the data function
4. L: $E \rightarrow \mathit{Labels}$ is the label function
5. $M = (\text{Ex}, \text{Re}, \text{In}, \text{Va}) \in (\mathcal{P}(F) \times \mathcal{P}(F) \times \mathcal{P}(F) \times (F \rightharpoonup \text{Val}))$ is the marking with data of the local events,
6. $\rightarrow\bullet, \bullet\rightarrow, \rightarrow+, \rightarrow\%, \rightarrow\diamond \subseteq E \times \text{BExp} \times E$ are the condition, response, include, exclude, and milestone guarded relations respectively,
7. $\rightarrow\!\!\!\!\rightarrow$: $E \rightharpoonup \text{BExp} \times \mathcal{G}$ is the guarded spawn relation, where $\mathcal{G}$ is the set of all DCRSec graphs.
8. $w : E \rightarrow \mathcal{P}(\mathcal{SC}' \times E)$ is the access control function
9. $\text{pc} : E \rightarrow \mathcal{P}(\mathcal{SC}' \times E)$ is the security label function

A graph is defined with relation to a set of globally known events E and a set of locally defined events $F \subseteq E$. Local events, $e \in F$, are categorized as computation events when $D(e)$ denotes an expression or as input events when $D(e) = ?$. The marking M denotes the initial/current state of the graph and consists of three sets of events, subsets of F and a function from F to values. Set Ex contains the events that were already executed at least once. Set Re contains the events that are currently pending, meaning that their execution is mandatory, if they are included, to reach a stable state of the graph. Set In is the set of all included events, i.e., events that are eligible and only depend on control-flow relations and conditions to be executed. Function Va maps events to values in Val denoting their current value tagged with security labels, if defined. If $e \in \text{Ex}$ then $\text{Va}(e)$ is defined.

The *DCR* relations, condition, response, include, exclude and milestone consist of a pair of events and a boolean expression that is the guard of the relation. The spawn relation is modelled by a partial function that assigns a boolean expression (a guard) and a DCRSec graph (a sub-process, also referred to as a proto-graph) to some of the events. The elements of proto-graphs are not active elements of the overall process. They represent a static definition standing by to spawn and integrate the top-level graph when the relation is triggered by executing an event e that maps to the proto-graph by the spawn relation. When convenient and without losing generality, we use the spawn relation interchangeably as a set of triples in the domain $E \times \text{BExp} \times \mathcal{G}$. The invariant of *DCRSec* graphs is that the marking of events of all proto-graphs, depicting the initial state of locally defined events, is joined with the locally defined events at the top-level graph when the triggering event is spawned. The set of globally known events in a graph contains all events that are visible in their expressions and relations. The set of locally defined events is the set of events defined at the present level. Thus, at the top level, both sets coincide, but in the nested graph definitions, the locally defined events are a subset of E.

We define the access control function w that maps events to a set of security levels in $\mathcal{SC}$. For convenience, we use the same set of labels as roles for access control and information flow control. We also define a function pc to map events to security levels and define the security compartment in which the triggering of an event can have an effect.

We consider a sub-language of expressions (Exp) that are able to read the marking and values associated with the events in E. This work is parametric in the language of expressions Exp, that in our example and prototype ranges over a set of basic datatypes and operators, and is able of expressing basic, terminating computations. We consider $\text{BExp} \subseteq \text{Exp}$ for boolean expressions to be used in the guards of the events. We define the inductive confidentiality-aware evaluation of expressions over the marking M as $[\![f]\!]_M$ with $f \in \text{Exp}$, yielding a value $v_\delta \in \text{Val}$, in a straightforward way. The security level δ of a value is obtained by recursively computing the least upper bound between the security levels of the operands for each operation in expression f. The set Val contains the usual literals, i.e., booleans, integers, strings, records, the usual arithmetic and logic operations, and events ($E \subseteq \text{Val}$). Expressions may also refer to the meta-data of events using the functions in the marking M given at evaluation time. In the code of the example, we use dot notation as the concrete syntax of such accesses. Notice that the triggering of events can only occur at the top level of the graph, *i.e.*, proto-graphs in the spawn relations are only used to extend the definition of the top-level graph when they are triggered.

Notation. We write $\text{locals}(G)$ to denote the local events of graph G, and $\text{marking}(G)$ to denote its marking.

4 Semantics

In this section, we define the semantics of *DCRSec* graphs. We start by defining when an event is enabled for execution.

Definition 4.1 (Event enabling)**.** Let $G = (E, F, D, L, M, \rightarrow\bullet, \bullet\rightarrow, \rightarrow\%, \rightarrow+, \rightarrow\diamond, \rightarrow\!\!\!\!\rightarrow, w, \text{pc})$ be a *DCRSec* Graph with a marking $M = (\text{Ex}, \text{Re}, \text{In}, \text{Va})$. An event $e \in E$ is enabled in G if and only if:

1. $e \in \text{In}$, and
2. $\forall e' \in \text{In}. e' \xrightarrow{g}\bullet\ e \wedge [\![g]\!]_M = \text{true} \implies e' \in \text{Ex}$, and
3. $\forall e' \in \text{In}. e' \xrightarrow{g}\diamond\ e \wedge [\![g]\!]_M = \text{true} \implies e' \notin \text{Re}$

For an event e to be enabled, (1) it must be included; (2) whenever e has a condition relation pointing to it from an included event e', with a guard that evaluates to true, event e' has been executed; (3) whenever e has a milestone relation pointing to it from an included event e', with a guard that evaluates to true, event e' must not be pending [33].

Notation. We write $\mathit{enabled}(e)$ to denote if an event is enabled and EN_G to denote the set of enabled events in G.
Notation. When referring to all relations $\rightarrow\bullet$, $\bullet\rightarrow$, $\rightarrow\%$, $\rightarrow+$, $\rightarrow\diamond$, $\rightarrow\!\!>$ simultaneously, we write $\rightarrow*$ as the union of all relations and $e \xrightarrow{g} * e'$ to denote pairs in it, with g being the predicate of guarded relations. In the case of the spawn relation, $(e \xrightarrow{g} \!\!> \{G\})$, the relation $e \xrightarrow{g} * e'$ denotes a pair of event e and an event e', locally defined in G, or used as target event in G or any if its sub-processes.

Definition 4.2 (Substitution of identifiers in graphs)**.** We define the capture avoiding substitution on graphs by straightforwardly replacing identifiers in all components of the graph, but alpha-renaming the locally defined names in proto-graphs in spawn relations to avoid clashes.

We now define the plain execution of an event and how it affects the marking of a graph. We then extend this to include the local verification of security policies responsible for the absence of information leaks in the process, as shown by the non-interference result in Section 5.

Definition 4.3 (Effect of an event)**.** Let *DCRSec* Graph G be defined by the tuple $(E, F, D, L, M, \rightarrow*, w, \mathrm{pc})$ with a marking M defined by a tuple $(\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$.

If defined, consider the proto-graph H spawned by event e with guard g evaluating to true in the marking of G (after updating the value of e). We define H' by replacing the identifiers of locally defined events in H by fresh identifiers:

$$H' \triangleq H\{^{\overline{x'}}/_{\overline{x}}\} \text{ with } \overline{x} = \mathsf{locals}(G),\ \overline{x'} \text{ fresh},\ \#\overline{x} = \#\overline{x'}$$

We identify the components of H' as follows

$$H' = (E'', F'', D'', L'', M'', \rightarrow*'', w'', \mathrm{pc}'')$$

The effect of executing an **enabled event** e with a given value v on G, written $G \cdot e$ is the new graph

$$G \cdot e = (E', F', D', L', M', \rightarrow*', w', \mathrm{pc}') \text{ where,}$$

$$E' = E \ \cup \ E''$$
$$F' = F \cup F''$$
$$D' = D \cup D''$$
$$L' = L \cup L''$$
$$M' = (\mathsf{Ex}', \mathsf{Re}', \mathsf{In}', \mathsf{Va}') \text{ where}$$
$$\mathsf{Ex}' = \mathsf{Ex} \cup \{e\} \cup \mathsf{Ex}''$$
$$\mathsf{Re}' = \mathsf{Re}\backslash\{e\} \cup \{e' \in E \mid \exists e \xrightarrow{g}\bullet\ e' \wedge [\![g]\!]_{M''} = \mathsf{true}\} \cup \mathsf{Re}''$$
$$\mathsf{In}' = (\mathsf{In}\backslash\{e' \in E \mid e \xrightarrow{g}\% \ e' \wedge [\![g]\!]_{M''} = \mathsf{true}\})$$
$$\cup \{e' \mid e \xrightarrow{g}+ \ e' \wedge [\![g]\!]_{M''} = \mathsf{true}\} \cup \mathsf{In}''$$
$$\mathsf{Va}' = \mathsf{Va}[e \mapsto \{v\}] \cup \mathsf{Va}''$$
$$\rightarrow*' = \rightarrow* \cup \rightarrow*''$$
$$w' = w \cup w''$$
$$\mathrm{pc}' = \mathrm{pc} \cup \mathrm{pc}''$$

Where v is an input value in case of an input event or the result of evaluating expression e, $[\![D(e)]\!]_M = v$, in case of a computation event, and $M'' = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va}[e \mapsto \{v\}])$

Definition 4.4 (Effect of a monitored event)**.** Let *DCRSec* Graph G be defined by the tuple $(E, F, D, L, M, \rightarrow*, w, \mathrm{pc})$ with a marking M defined by a tuple $(\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$. We define the effect of the monitored execution of an enabled event e, written $G \hat{\cdot} e$ as:

$$G \hat{\cdot} e \ \triangleq \ G \cdot e \text{ if } \forall e \xrightarrow{g} * e'.\big(\delta' \sqsubseteq \mathrm{pc}(e) \wedge \mathrm{pc}(e) \sqcup \delta \sqsubseteq \mathrm{pc}(e')$$
$$\wedge \forall e''.\ e' \rightarrow\bullet\ e'' \vee e' \rightarrow\diamond\ e'' \implies \mathrm{pc}(e) \sqsubseteq \mathrm{pc}(e'')\big),$$

with $[\![g]\!]_{M''} = b_\delta$, and $[\![D(e)]\!]_M = v_{\delta'}$ when $D(e) \neq ?$ and $M'' = (\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va}[e \mapsto \{v\}])$

Definition 4.5 (Labelled Transition System)**.** Let G be a *DCRSec* Graph defined by the tuple $(E, F, D, L, M, \rightarrow*, w, \mathrm{pc})$ with a marking M defined by the tuple $(\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$. We define a labelled transition system with two kinds of transitions. An output or computation transition by executing an event e, written $G \xrightarrow{e} G'$, and an input transition by executing event e with a value v as input, written $G \xrightarrow{e(v)} G'$. In both cases, the transition is defined if $G' = G \hat{\cdot} e$ is defined by applying the effect of executing the event e in G (Definition 4.3 and Definition 4.4). We write $G \longrightarrow G'$ when referring to either kind of transition.

Notice that, for an event violating the security check, the monitored effect and the corresponding transition are not defined, and its execution halts. All termination-insensitive mechanisms for information flow control may leak information through termination channels [6, 9]. The monitor introduces a termination channel from preemptively terminating the execution of an event, which is observable to an attacker. As an illustration, consider the *DCRSec* process below with events e, initially pending, and e', both tagged as secret ($\top$) events and related through a cycle of response relations, from e to e' and from e' to e. Consider a secret condition g as a guard to the relation from e to e'.

$$!e_{(\top,\top)},\ e'_{(\top,\top)};\ e \xrightarrow{g}\bullet\ e',\ e' \bullet\rightarrow e$$

Since e is pending, it must execute, which in turn makes e' pending, which, in turn, makes e pending again. To stop this loop-like structure, g must evaluate to false. This process emulates a loop, often exploited to obtain sensitive information through termination channels. In the case of *DCRSec*, the non-deterministic execution of events makes it harder for an attacker to perceive if a process is in a loop or not.

Monitor-induced termination channels are an acceptable trade-off between security and program acceptance [6]. Consider the secret events e and e' and the public ($\bot$) event e''. Further, consider that e' and e'' are initially excluded, and the events relate through include relations as in the process,

$$e_{(\top,\top)}, \%e'_{(\top,\top)}, \%e''_{(\bot,\bot)};\ e \rightarrow+\ e',\ e' \rightarrow+\ e''$$

The execution of e includes e', which, in turn, includes e'' when executed. Since e'' is public and e' is secret, the monitor halts the execution of e'. Upon the termination of e' an attacker can infer that e' was included, thus, that the secret event e executed. An alternative is to not halt the process and ignore violating events, but that would change the semantics of the program and would lead to undefined behaviours.

5 Properties of the IF Monitor

In this section, we prove the non-interference and transparency properties of the monitor defined in Section 4. We first need some preliminary supporting definitions.

Definition 5.1 (Π_{δ_o} - Projection)**.** Let *DCRSec* Graph G be defined by the tuple $(E, F, D, M, L, \rightarrow*, w, r)$ with a marking M defined by a tuple $(\mathsf{Ex}, \mathsf{Re}, \mathsf{In}, \mathsf{Va})$. The projection of G, written $\Pi_{\delta_o}(G)$, at an observation level δ_o is defined as:

$$\begin{array}{rcl}\Pi_{\delta_o}(G) & = & (E', F', D|_{E'}, L, M', S, \rightarrow*', w|_{E'}, \mathrm{pc}|_{E'}) \text{ with} \\ E' & = & \{e \in E \mid \mathrm{pc}(e) \sqsubseteq \delta_o\} \\ F' & = & \{e \in F \mid \mathrm{pc}(e) \sqsubseteq \delta_o\} \\ M' & = & (\mathsf{Ex} \cap E', \mathsf{Re} \cap E', \mathsf{In}' \cap E', \mathsf{Va}|_{E'}) \\ \rightarrow*' & = & \{e \xrightarrow{g}* e' \mid \mathrm{pc}(e') \sqsubseteq \delta_o\}\end{array}$$

Where $f|_D$ is the restriction of function f to domain D.

Definition 5.2 ($\sim_{\delta_o}$ - Indistinguishability)**.** Let G and G' be two *DCRSec* graphs. G and G' are said to be indistinguishable at observation level δ_o ($G \sim_{\delta_o} G'$) if and only if their projections and enabled sets with respect to δ_o are identical:

$$G \sim_{\delta_o} G' \iff H = H' \wedge EN_H = EN_{H'},$$
$$\text{with } H = \Pi_{\delta_o}(G) \text{ and } H' = \Pi_{\delta_o}(G').$$

Definition 5.3 ($\longrightarrow_{\delta_o}$ - High transition)**.** Let G and G' be two *DCRSec* graphs with an event e and an observation level δ_o. G makes a high transition to G' through an event e, written $G \xrightarrow{e}_{\delta_o} G'$, if the security level of e is not lower than the observation level δ_o:

$$G \xrightarrow{e}_{\delta_o} G' \triangleq G \xrightarrow{e} G' \text{ if } \mathrm{pc}(e) \not\sqsubseteq \delta_o$$

Definition 5.4 ($\longrightarrow_{\delta_o}$ - Low transition)**.** Let G and G' be two *DCRSec* graphs with an event e and an observation level δ_o. G makes a low transition to G' through an event e, written $G \xrightarrow{e}_{\delta_o} G'$, if the security level of e is lower or equal than the observation level δ_o:

$$G \xrightarrow{e}_{\delta_o} G' \triangleq G \xrightarrow{e} G' \text{ if } \mathrm{pc}(e) \sqsubseteq \delta_o$$

Lemma 5.5 (Confinement)**.** *Let G and G' be two DCRSec graphs with an event e and an observation level δ_o. If G makes a high transition to G' through an event e then G and G' are indistinguishable ($G \sim_{\delta_o} G'$).*

Proof. The proof follows by inspection of Definition 4.5 (Transition) and Definition 4.4 (Monitored Effect) where a transition only occurs if the effect of executing an event e is defined and an effect is defined if all affected events, i.e., those whose marking or enabledness change, are visible in a security compartment with a label higher or equal to the execution context $\mathrm{pc}(e)$. Considering an observation level δ_o, we know, by Definition 5.3 (High Transition), that $\mathrm{pc}(e) \not\sqsubseteq \delta_o$ (1.). Furthermore, the effects of e, as well as the events whose enabledness indirectly changes because of e, will have a label δ such that $\mathrm{pc}(e) \sqsubseteq \delta$. We have that $\delta \not\sqsubseteq \delta_o$ because if $\delta \sqsubseteq \delta_o$ we would have, by transitivity, a contradiction with (1.). Thus, the effects of e are not visible at the security compartment defined by δ_o. All changed events in G' have at least a non-lower security level in relation to δ_o, thus, by Definition 5.1 (Projection), they are not present in $\Pi_{\delta_o}(G)$. So, we have $\Pi_{\delta_o}(G) = \Pi_{\delta_o}(G')$, and therefore, by the Definition 5.2, we have the indistinguishability $G \sim_{\delta_o} G'$. □

5.1 Monitor Non-interference

Lemma 5.6 (Low one-step non-interference)**.** *Let G_i and G_i' be two indistinguishable DCRSec graphs at an observation level δ_o. If G_i and G_i' make a low transition to graphs G_f and G_f' respectively, then G_f and G_f' are also indistinguishable:*

$$\frac{G_i \sim_{\delta_o} G_i' \quad G_i \xrightarrow{e}_{\delta_o} G_f \quad G_i' \xrightarrow{e}_{\delta_o} G_f'}{G_f \sim_{\delta_o} G_F'}$$

Proof. Let G_i and G_i' be two indistinguishable *DCRSec* graphs, that make a transition through event e to G_f and G_f' respectively, and *lobs* a given observation level.

By Definitions 5.4, 4.5, and 4.4, we know that $G_f = G_i \cdot e$ and $G_f' = G_i' \cdot e$. The execution of e on G_i and G_i' leads to the addition of events though $\twoheadrightarrow$ or the change in the marking of events through $\rightarrow+$, $\rightarrow\%$, $\bullet\rightarrow$ and $\twoheadrightarrow$. Consider e' to be a new event or an event whose marking or enabledness has changed, and δ the security level of e'. There are two cases to consider: (1) $\delta \not\sqsubseteq \delta_o$: the event is not observable at δ_o. (2) $\delta \sqsubseteq \delta_o$: the event is observable at δ_o; Let:

$\Pi_{\delta_o}(G_f) = (E, F, D, L, M, \rightarrow*, w, \mathrm{pc})$

$\Pi_{\delta_o}(G_f') = (E', F', D', L', M', \rightarrow*', w', \mathrm{pc}')$

Case 1: One of the conditions required to apply low one-step non-interference is that $\Pi_{\delta_o}(G_i) = \Pi_{\delta_o}(G_i')$, and the event is not observable, by Definition 5.1, $e' \notin E \wedge e' \notin F$ and $e' \notin E' \wedge e' \notin F'$, therefore, $\Pi_{\delta_o}(G_f) = \Pi_{\delta_o}(G_f')$. By Definition 5.2, we have $G_f \sim_{\delta_o} G_f'$. **Case 2**: By Definition 4.3 the effects of executing the same event e on G_i and G_i' are identical since the data used in the event is necessarily visible ($[\![D(e)]\!]_M = v_{\delta'}$ and $\delta' \sqsubseteq \mathrm{pc}(e)$) and, by Definion 5.1 equal in G_i and G_i'. Together with the condition $\Pi_{\delta_o}(G_i) = \Pi_{\delta_o}(G_i')$ required to apply the lemma, we have that $E = E'$ and $F = F'$. By Definition 5.1, we have that $\Pi_{\delta_o}(G_f) = \Pi_{\delta_o}(G_f')$ and by Definition 5.2 we conclude $G_f \sim_{\delta_o} G_f'$. □

We now observe that Lemma 5.5 and Lemma 5.6 lemmas are transitive, *i.e.*, if there is a transition from one graph to another through multiple high (low) transitions then it is possible to apply Lemma 5.5 (Lemma 5.6) multiple times to

conclude that the initial and final graphs are indistinguishable. *Lemma 5.5 transitive closure.* Considering a graph G_0 that transits through n high transitions to G_n, it is possible to establish $G_0 \sim_{\delta_o} G_n$ through n applications of Lemma 5.5 and the transitivity property of Definition 5.2 (Indistinguishability). *Lemma 5.6 transitive closure.* Two indistinguishable graphs G_0 and G_0' that execute the same sequence of n low transitions will result in two indistinguishable graphs G_n and G_n'. The lemma's conclusion is the necessary premise for its application. Together with the transitivity property of Definition 5.2 (Indistinguishability), it is possible to arrive at the result that $G_n \sim_{\delta_o} G_n'$.

Definition 5.7 ($\longrightarrow_{\delta_o}$ - Mixed transitions)**.** Mixed transitions, $\xrightarrow[L]{H}_{\delta_o}$, intertwine H high and L low transitions. Mixed transitions are composed of a sequence of l low transitions and a sequence of h high transitions followed by a smaller sequence of mixed transitions:

$$G \xrightarrow[L]{H\ \overline{e}}_{\delta_o} G_f \triangleq G \xrightarrow[l]{\overline{e}'}_{\delta_o} G' \xrightarrow{h\ \overline{e}''}_{\delta_o} G''' \xrightarrow[L-l]{H-h\ \overline{e}'''}_{\delta_o} G_f,$$

with $\overline{e} = \overline{e}' \cdot \overline{e}'' \cdot \overline{e}'''$

Theorem 5.8 (Non-interference)**.** *Let G_i and G_i' be two indistinguishable DCRSec graphs. If G_i and G_i' make a transition to graphs G_f and G_f' through a sequence of mixed transitions with the same sequence of (observable) low-transitions, then G_f and G_f' are indistinguishable for any observation level δ_o:*

$$\frac{G_i \xrightarrow[L]{H\ \overline{e}}_{\delta_o} G_f \qquad G_i' \xrightarrow[L]{H'\ \overline{e}'}_{\delta_o} G_f' \qquad G_i \sim_{\delta_o} G_i'}{G_f \sim_{\delta_o} G_f'},$$

with $\Pi_{\delta_o}(\overline{e}) = \Pi_{\delta_o}(\overline{e}')$, where $\Pi_{\delta_o}(\overline{e}) = \overline{\Pi_{\delta_o}(e)}$

Notice that the sequences of low events must agree; the observable events in one sequence must be in the other and in the same order. To obtain the low part of a sequence of events, we map the projection over each element of the sequence, and the projection yields the event if it is observable (its security level is lower or equal to δ_o) or the empty sequence otherwise.

Proof. We prove non-interference by inductive reasoning on the sequence size. Consider the breakdown of the sequence according to Definition 5.7 (mixed transitions), applying the transitive closure of Lemma 5.6 to the first part of the sequence containing only low transitions, and applying the transitive closure of Lemma 5.5 to the next sequence, of high transitions. By transitivity of Definition 5.2 (Indistinguishability), we reach the conditions necessary to apply the induction hypothesis to the remaining (smaller) sequence, thus proving non-interference. □

Theorem 5.8 supports the correctness of our monitor, hence, processes preserve the confidentiality of data.

5.2 Monitor Transparency

A monitor is transparent, or *true* transparent according to Bielova *et al.* [9], if all executions that do not violate the security policies are preserved. Consider a *DCRSec* graph G executing event e and reaching a target graph G' through a non-monitored transition. We want to prove that if G executes event e in the monitored semantics it is either stopped by the monitor or it reaches the same target graph G'.

Theorem 5.9 (Transparency)**.** *Let G and G' be two DCRSec graphs, an event e and an observation level δ_o. When G makes a monitored transition to G' through an event e, $G \xrightarrow{e}_{\delta_o} G'$, then it also has the same non-monitored effect $G \cdot e = G'$.*

Proof. The proof of the monitor's transparency is straightforward. According to Definition 4.4, the effect of the monitored execution of e is only defined if e is enabled and its execution does not result in information leaks. Hence, if G makes a monitored transition to G' through e, the execution of e resuts in no information leaks. Moreover, by Definition 4.5, we have that $G' = G \hat{\cdot} e$, and by Definition 4.4 we know that $G \hat{\cdot} e = G \cdot e$. From this, we conclude that $G \cdot e = G'$. □

Models for information flow control. There is a plethora of extensions of Denning's seminal model [17], some notably increasing its expressiveness. For instance, Lourenço and Caires [38] use data-dependent types, enabling parametrised labels and precise information flow policies [22]. Polikarpova *et al.* [40] use liquid types to achieve richer mechanisms for policy definition. Myers and Liskov [39] propose a decentralised label model with dynamic labels, conferring more flexibility. Our approach combines dependent labels into sets to implement sharing and achieve even higher flexibility.

Information flow control for programming languages Austin and Flanagan [7] enforce information flow control in a dynamically typed language using sparse labelling to reduce the overhead incurred. Volpano *et al.* [52] produce a type-system following Denning's model [17]. Disney and Flanagan [19] present an extension to simply-typed lambda calculus, supporting the gradual labelling of programs. Similarly, Fennel and Thiemann [21] develop an ML-based language with references which supports gradual labelling. Siek and Taha [45] develop a gradual type-system for information flow control in an object-oriented language. Finally, Toro *et al.* [49] present a gradual security typed higher-order language. Information flow also has applications in dynamically typed languages like JavaScript [30, 44] and Python [29]. In software verification tools, JOANA [47] enforces information flow control through static bytecode analysis. Snitch [25, 27] performs hybrid bytecode analysis, to enforce value-dependent information flow control. In mobile applications, FlowDroid [5] statically enforces information flow control in Android applications and TaintDroid [20]

instruments the Android virtual machine to monitor information flows. We apply these ideas to business processes to dynamically prevent information leaks.

Process calculi. Honda *et al.* present a seminal approach for an asynchronous π-calculus that associates security labels with channels and is capable of embedding known fragments with type-based security [34]. Kobayashi proposes a typed π-calculus based on channel types extended with channel usages and a type inference algorithm with information flow capabilities [35]. Although static, these ideas lay a foundation for our work on business processes and data.

Privacy in business processes. A considerable amount of work relating information privacy to business processes consists of strategies to encode privacy requirements in business processes [43] Agostinelli *et al.* [4] dissect the privacy constraints imposed in GDPR and propose a set of design patterns to integrate them in business processes. Pleak [48] is a tool capturing and analysing the extent of data leaks in privacy-enhanced business process models. Pullonen *et al.* [41, 42] extend the BPMN language to capture information leaks and analyse information disclosure. Barati and Rana [8] employ smart contracts to verify business processes' compliance with GDPR. Anica [3, 37] enforces place-based noninterference [11] in Petri nets representing business processes through reachability tests.

Our approach, unlike some of the existing solutions, allows for both the specification and verification of confidentiality policies in business processes. Moreover, it follows a language-based approach to prevent data confidentiality breaches at run-time thanks to the embedded termination-insensitive information flow monitor. We do not translate business processes to different representations, avoiding implicit and explicit leaks resulting through covert channels introduced with representation translation operations.

6 Final Remarks

We present *DCRSec*, a confidentiality-aware extension to DCR Graphs with data. It features security mechanisms for both access control and information flow control, *DCRSec* allows for the definition of secure processes where sensitive data will not leak to unauthorized parties. On one hand, the access control mechanism stops principals from executing events they should not be able to trigger, and the embedded information flow monitor ensures that all the effects of events are compliant with the defined security policies.

To ensure a process exhibits non-interference, the information flow monitor prevents the execution of processes whose result is more sensitive than what is defined in the process or events whose execution could reveal information about previous conditions and events. For instance, the inclusion/exclusion of an event may leak information about data used in guards of relations, and result in implicit leaks just by changing the state of an event. We present the semantics of the information flow monitor, and most importantly, we prove its correctness by ensuring the non-interference and transparency properties.

Future Work We identify several directions from the current work. We plan to explore and implement this model in the prototype for the ReGraDa language [24]. ReGraDa has an underlying graph database engine, that represents data and control-flow relations and can now also represent the dependent security lattice. The implementation of the reference monitor relies on the querying of the lattice made explicit in the graph database. Next, we plan to extend the analysis of *DCRSec* to a static and hybrid approach, *c.f.* [27]. Static analysis in the realm of less structured code like *DCR* graphs is challenging, but we believe that a type system can be developed to ensure non-interference. The expansion of the access control mechanism for a full-blown integrity analysis is also worth mentioning as a future research topic.

Acknowledgments

Funded by FCT/MCTES SFRH/BD/149043/2019, by EU Horizon Europe under Grant, Agreement no. 101093006 (TaRDIS), by NOVA LINCS (FCT UIDB/04516/2020), and by Independent Research Fund Denmark, Grant no. 9131-00077B (PA-PRiCaS).

References

[1] Martín Abadi, Anindya Banerjee, Nevin Heintze, and Jon G. Riecke. 1999. A Core Calculus of Dependency. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, San Antonio, TX, USA, January 20-22, 1999*, Andrew W. Appel and Alex Aiken (Eds.). ACM, 147–160. https://doi.org/10.1145/292540.292555

[2] Martin Abadi, Michael Burrows, Butler Lampson, and Gordon Plotkin. 1993. A calculus for access control in distributed systems. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 15, 4 (1993), 706–734.

[3] Rafael Accorsi and Andreas Lehmann. 2012. Automatic Information Flow Analysis of Business Process Models. In *Business Process Management*, Alistair Barros, Avigdor Gal, and Ekkart Kindler (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 172–187.

[4] 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 (Lecture Notes in Business Information Processing, Vol. 350)*, Cinzia Cappiello and Marcela Ruiz (Eds.). Springer, 10–22. https://doi.org/10.1007/978-3-030-21297-1_2

[5] Steven Arzt, Siegfried Rasthofer, Christian Fritz, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves Le Traon, Damien Octeau, and Patrick McDaniel. 2014. FlowDroid: Precise Context, Flow, Field, Object-Sensitive and Lifecycle-Aware Taint Analysis for Android Apps. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Edinburgh, United Kingdom) *(PLDI '14)*. Association for Computing Machinery, New York, NY, USA, 259–269. https://doi.org/10.1145/2594291.2594299

[6] Aslan Askarov, Sebastian Hunt, Andrei Sabelfeld, and David Sands. 2008. Termination-Insensitive Noninterference Leaks More Than Just a Bit. In *Proceedings of the 13th European Symposium on Research in Computer Security: Computer Security* (Málaga, Spain) *(ESORICS '08)*.

Springer-Verlag, Berlin, Heidelberg, 333–348. https://doi.org/10.1007/978-3-540-88313-5_22

[7] Thomas H. Austin and Cormac Flanagan. 2009. Efficient Purely-Dynamic Information Flow Analysis. In *Proceedings of the ACM SIGPLAN Fourth Workshop on Programming Languages and Analysis for Security* (Dublin, Ireland) *(PLAS '09)*. Association for Computing Machinery, New York, NY, USA, 113–124. https://doi.org/10.1145/1554339.1554353

[8] Masoud Barati and Omer Rana. 2021. *Design and Verification of Privacy Patterns for Business Process Models*. Springer Singapore, Singapore, 125–139. https://doi.org/10.1007/978-981-33-6470-7_8

[9] Nataliia Bielova and Tamara Rezk. 2016. A Taxonomy of Information Flow Monitors. In *Proceedings of the 5th International Conference on Principles of Security and Trust - Volume 9635*. Springer-Verlag, Berlin, Heidelberg, 46–67.

[10] Niklas Broberg and David Sands. 2010. Paralocks: Role-Based Information Flow Control and Beyond. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Madrid, Spain) *(POPL '10)*. Association for Computing Machinery, New York, NY, USA, 431–444. https://doi.org/10.1145/1706299.1706349

[11] Nadia Busi and Roberto Gorrieri. 2009. Structural Non-Interference in Elementary and Trace Nets. *Mathematical. Structures in Comp. Sci.* 19, 6 (dec 2009), 1065–1090. https://doi.org/10.1017/S0960129509990120

[12] Luís Caires, Jorge A. Pérez, João Costa Seco, Hugo Torres Vieira, and Lúcio Ferrão. 2011. Type-Based Access Control in Data-Centric Systems. In *Programming Languages and Systems - 20th European Symposium on Programming, ESOP 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings (Lecture Notes in Computer Science, Vol. 6602)*, Gilles Barthe (Ed.). Springer, 136–155. https://doi.org/10.1007/978-3-642-19718-5_8

[13] João Costa Seco, Søren Debois, Thomas T. Hildebrandt, and Tijs Slaats. 2018. RESEDA: Declaring Live Event-Driven Computations as REactive SEmi-Structured DAta. In *22nd IEEE International Enterprise Distributed Object Computing Conference, EDOC 2018, Stockholm, Sweden, October 16-19, 2018*. IEEE Computer Society, 75–84. https://doi.org/10.1109/EDOC.2018.00020

[14] Søren Debois, Thomas Hildebrandt, and Tijs Slaats. 2014. Hierarchical Declarative Modelling with Refinement and Sub-processes. In *Business Process Management*, Shazia Sadiq, Pnina Soffer, and Hagen Völzer (Eds.). Springer International Publishing, Cham, 18–33.

[15] Søren Debois, Thomas Hildebrandt, and Tijs Slaats. 2015. Safety, Liveness and Run-Time Refinement for Modular Process-Aware Information Systems with Dynamic Sub Processes. In *FM 2015: Formal Methods*, Nikolaj Bjørner and Frank de Boer (Eds.). Springer International Publishing, Cham, 143–160.

[16] Søren Debois, Thomas T. Hildebrandt, and Tijs Slaats. 2014. Hierarchical Declarative Modelling with Refinement and Sub-processes. In *Business Process Management - 12th International Conference, BPM 2014, Haifa, Israel, September 7-11, 2014. Proceedings (Lecture Notes in Computer Science, Vol. 8659)*, Shazia Wasim Sadiq, Pnina Soffer, and Hagen Völzer (Eds.). Springer, 18–33. https://doi.org/10.1007/978-3-319-10172-9_2

[17] Dorothy E. Denning. 1976. A Lattice Model of Secure Information Flow. *Commun. ACM* 19, 5 (may 1976), 236–243. https://doi.org/10.1145/360051.360056

[18] Dorothy E Denning, Peter J Denning, and G Scott Graham. 1976. *On the derivation of lattice structured information flow policies*. Technical Report 76-180. Purdue University.

[19] Tim Disney and Cormac Flanagan. 2011. rmation Flow Typing. In *Proceedings of the 2nd International Workshop on Scripts to Programs Evolution (STOP '11)*.

[20] William Enck, Peter Gilbert, Byung-Gon Chun, Landon P. Cox, Jaeyeon Jung, Patrick McDaniel, and Anmol N. Sheth. 2014. TaintDroid: An Information Flow Tracking System for Real-Time Privacy Monitoring on Smartphones. *Commun. ACM* 57, 3 (mar 2014), 99–106. https://doi.org/10.1145/2494522

[21] Luminous Fennell and Peter Thiemann. 2013. Gradual Security Typing with References. In *2013 IEEE 26th Computer Security Foundations Symposium*. 224–239. https://doi.org/10.1109/CSF.2013.22

[22] Paulo Jorge Abreu Duarte Ferreira. 2012. MSc Dissertation. Information flow analysis using data-dependent logical propositions. NOVA School do Science and Technology, NOVA University Lisbon.

[23] Lúcio Ferrão, João Costa Seco, Luís Caires, Gonçalo Borrega, and António Melo. 2013. SYSTEMS, METHODS, AND APPARATUS FOR MODEL-BASED SECURITY CONTROL. Patent US20130246995A1. https://patents.google.com/patent/US20130246995A1/en.

[24] Leandro Galrinho, João Costa Seco, Søren Debois, Thomas T. Hildebrandt, Håkon Norman, and Tijs Slaats. 2021. ReGraDa: Reactive Graph Data. In *Coordination Models and Languages - 23rd IFIP WG 6.1 International Conference, COORDINATION 2021, Held as Part of the 16th International Federated Conference on Distributed Computing Techniques, DisCoTec 2021, Valletta, Malta, June 14-18, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12717)*, Ferruccio Damiani and Ornela Dardha (Eds.). Springer, 188–205. https://doi.org/10.1007/978-3-030-78142-2_12

[25] Eduardo Geraldo. 2022. SNITCH: A Platform for Information Flow Control. In *Integrated Formal Methods: 17th International Conference, IFM 2022, Lugano, Switzerland, June 7–10, 2022, Proceedings* (Lugano, Switzerland). Springer-Verlag, Berlin, Heidelberg, 365–368. https://doi.org/10.1007/978-3-031-07727-2_24

[26] Eduardo Geraldo and João Costa Seco. 2018. SNITCH: Dynamic Dependent Information Flow Analysis for Independent Java Bytecode. In *Proceedings of the Second Workshop on Verification of Objects at RunTime EXecution, VORTEX@ECOOP/ISSTA 2018, Amsterdam, Netherlands, 17th July 2018 (EPTCS, Vol. 302)*, Davide Ancona and Gordon Pace (Eds.). 16–31. https://doi.org/10.4204/EPTCS.302.2

[27] Eduardo Geraldo, José Fragoso Santos, and João Costa Seco. 2021. Hybrid Information Flow Control for Low-Level Code. In *Software Engineering and Formal Methods - 19th International Conference, SEFM 2021, Virtual Event, December 6-10, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 13085)*, Radu Calinescu and Corina S. Pasareanu (Eds.). Springer, 141–159. https://doi.org/10.1007/978-3-030-92124-8_9

[28] Eduardo Miguel Pereira Cano Rico Geraldo. 2018. *SNITCH: Dependent Dynamic Information Flow Analysis on Intermediate Java Code*. Master's thesis. NOVA School of Science & Technology.

[29] Sandip Ghosal and R. K. Shyamasundar. 2021. Pifthon: A Compile-Time Information Flow Analyzer For An Imperative Language. *CoRR* abs/2103.06039 (2021). arXiv:2103.06039 https://arxiv.org/abs/2103.06039

[30] Daniel Hedin and Andrei Sabelfeld. 2012. Information-Flow Security for a Core of JavaScript. In *2012 IEEE 25th Computer Security Foundations Symposium*. 3–18. https://doi.org/10.1109/CSF.2012.19

[31] Thomas T. Hildebrandt, Amine Abbad Andaloussi, Lars R. Christensen, Søren Debois, Nicklas Pape Healy, Hugo A. López, Morten Marquard, Naja L. H. Møller, Anette C. M. Petersen, Tijs Slaats, and Barbara Weber. 2020. EcoKnow: Engineering Effective, Co-Created and Compliant Adaptive Case Management Systems for Knowledge Workers. In *Proceedings of the International Conference on Software and System Processes* (Seoul, Republic of Korea) *(ICSSP '20)*. Association for Computing Machinery, New York, NY, USA, 155–164. https://doi.org/10.1145/3379177.3388908

[32] Thomas T. Hildebrandt and Raghava Rao Mukkamala. 2010. Declarative Event-Based Workflow as Distributed Dynamic Condition Response Graphs. In *Proceedings Third Workshop on Programming Language Approaches to Concurrency and communication-cEntric Software, PLACES 2010, Paphos, Cyprus, 21st March 2010 (EPTCS, Vol. 69)*, Kohei Honda and Alan Mycroft (Eds.). 59–73. https://doi.org/10.4204/EPTCS.69.5

[33] Thomas T. Hildebrandt, Håkon Normann, Morten Marquard, Søren Debois, and Tijs Slaats. 2022. Decision Modelling in Timed Dynamic Condition Response Graphs with Data. In *Business Process Management Workshops*, Andrea Marrella and Barbara Weber (Eds.). Springer International Publishing, Cham, 362–374.

[34] Kohei Honda, Vasco Thudichum Vasconcelos, and Nobuko Yoshida. 2000. Secure Information Flow as Typed Process Behaviour. In *Programming Languages and Systems, 9th European Symposium on Programming, ESOP 2000, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS 2000, Berlin, Germany, March 25 - April 2, 2000, Proceedings (Lecture Notes in Computer Science, Vol. 1782)*, Gert Smolka (Ed.). Springer, 180–199. https://doi.org/10.1007/3-540-46425-5_12

[35] Naoki Kobayashi. 2005. Type-based information flow analysis for the pi-calculus. *Acta Informatica* 42, 4-5 (2005), 291–347. https://doi.org/10.1007/s00236-005-0179-x

[36] Maxwell Krohn, Alexander Yip, Micah Brodsky, Natan Cliffer, M. Frans Kaashoek, Eddie Kohler, and Robert Morris. 2007. Information Flow Control for Standard OS Abstractions. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles* (Stevenson, Washington, USA) *(SOSP '07)*. Association for Computing Machinery, New York, NY, USA, 321–334. https://doi.org/10.1145/1294261.1294293

[37] A. Lehmann and D. Fahland. 2012. Information Flow Security for Business Process Models - just one click away. In *Proceedings of the 10th International Conference on Business Process Management - Demo Track (BPM 2012)*.

[38] Luísa Lourenço and Luís Caires. 2015. Dependent Information Flow Types. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Mumbai, India) *(POPL '15)*. Association for Computing Machinery, New York, NY, USA, 317–328. https://doi.org/10.1145/2676726.2676994

[39] Andrew C. Myers and Barbara Liskov. 2000. Protecting Privacy Using the Decentralized Label Model. *ACM Trans. Softw. Eng. Methodol.* 9, 4 (oct 2000), 410–442. https://doi.org/10.1145/363516.363526

[40] Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. 2020. Liquid Information Flow Control. *Proceedings of the ACM Programming Languages* 4, ICFP, Article 105 (aug 2020), 30 pages. https://doi.org/10.1145/3408987

[41] Pille Pullonen, Raimundas Matulevičius, and Dan Bogdanov. 2017. PE-BPMN: Privacy-Enhanced Business Process Model and Notation. In *Business Process Management*, Josep Carmona, Gregor Engels, and Akhil Kumar (Eds.). Springer International Publishing, Cham, 40–56.

[42] Pille Pullonen, Jake Tom, Raimundas Matulevičius, and Aivo Toots. 2019. Privacy-enhanced BPMN: enabling data privacy analysis in business processes models. *Software & Systems Modeling* 18 (12 2019). https://doi.org/10.1007/s10270-019-00718-z

[43] Sasha Romanosky, Alessandro Acquisti, Jason Hong, Lorrie Faith Cranor, and Batya Friedman. 2006. Privacy Patterns for Online Interactions. In *Proceedings of the 2006 Conference on Pattern Languages of Programs* (Portland, Oregon, USA) *(PLoP '06)*. Association for Computing Machinery, New York, NY, USA, Article 12, 9 pages. https://doi.org/10.1145/1415472.1415486

[44] José Fragoso Santos, Petar Maksimovic, Daiva Naudziuniene, Thomas Wood, and Philippa Gardner. 2018. JaVerT: JavaScript verification toolchain. *Proceedings of the ACM Programming Languages* 2, POPL (2018), 50:1–50:33. https://doi.org/10.1145/3158138

[45] Jeremy Siek and Walid Taha. 2007. Gradual Typing for Objects. In *ECOOP 2007 – Object-Oriented Programming*, Erik Ernst (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 2–27.

[46] Vincent Simonet. 2003. Flow Caml in a Nutshell. In *Proceedings of the first APPSEM-II workshop*, Graham Hutton (Ed.). Nottingham, United Kingdom, 152–165.

[47] Gregor Snelting, Dennis Giffhorn, Jürgen Graf, Christian Hammer, Martin Hecker, Martin Mohr, and Daniel Wasserrab. 2014. Checking probabilistic noninterference using JOANA. *it Inf. Technol.* 56, 6 (2014), 280–287. http://www.degruyter.com/view/j/itit.2014.56.issue-6/itit-2014-1051/itit-2014-1051.xml

[48] 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. *CoRR* abs/1902.05052 (2019). arXiv:1902.05052 http://arxiv.org/abs/1902.05052

[49] Matías Toro, Ronald Garcia, and Éric Tanter. 2018. Type-Driven Gradual Security with References. *ACM Trans. Program. Lang. Syst.* 40, 4 (2018), 16:1–16:55. https://doi.org/10.1145/3229061

[50] Wil M. P. van der Aalst. 2009. *Process-Aware Information Systems: Lessons to Be Learned from Process Mining.* Springer Berlin Heidelberg, Berlin, Heidelberg, 1–26. https://doi.org/10.1007/978-3-642-00899-3_1

[51] Wil M. P. van der Aalst, Alessandro Artale, Marco Montali, and Simone Tritini. 2017. Object-Centric Behavioral Constraints: Integrating Data and Declarative Process Modelling. In *Proceedings of the 30th International Workshop on Description Logics, Montpellier, France, July 18-21, 2017 (CEUR Workshop Proceedings, Vol. 1879)*, Alessandro Artale, Birte Glimm, and Roman Kontchakov (Eds.). CEUR-WS.org. http://ceur-ws.org/Vol-1879/paper51.pdf

[52] Dennis M. Volpano, Cynthia E. Irvine, and Geoffrey Smith. 1996. A Sound Type System for Secure Flow Analysis. *J. Comput. Secur.* 4, 2/3 (1996), 167–188. https://doi.org/10.3233/JCS-1996-42-304