Insider Attack Identification and Prevention in Collection Oriented Dataflow-based Processes

Anandarup Sarkar*, Sven Köhler*, Bertram Ludäscher*, Matt Bishop*
*University of California, Davis
{asarkar, svkoehler, ludaesch, mabishop}@ucdavis.edu

***Abstract*—We introduce an approach of automatically identifying attacks by insider agents on data-flow based processes having a collection-oriented data model, and then improving the processes to prevent the attacks against them. Some process data, if used by some agents via steps at certain points of timeline, will lead to a privacy attack. A manual identification of these vulnerable data and rogue agents is quite tedious; thus our approach automatically performs these identifications. We model a process and an attack based on a directed, acyclic graph, with steps, reading and writing data, and controlled by agents. Then we perform a declarative implementation to find out if this attack model can be mapped onto the process model based on some similarity criteria. If these criteria are met, we conclude that the attack model is "similar enough" to the process model, to be successfully realized through it. Each possible way of mapping shows an avenue of attack on the process. Agent collusion scenarios are also identified. Finally, our approach automatically identifies process improvement opportunities and iteratively exploits them, thereby eliminating ways in which the attack can be carried out against the process.**

I. Introduction

Determining if an attack can take place on large real-world processes, is not trivial. In elections, vulnerability analyses [1], [2] have focused on security or privacy aspects of specific parts of a process. But few have conducted a holistic vulnerability analysis, investigating the interactions among the steps, data and agents in a process [3], [4].

We present such a holistic vulnerability analysis approach- DIAS (**D**ata **I**nteraction with **A**gents and **S**teps). Section II introduces a a motivating election process example. Section III presents a high level overview of DIAS. There are agents, controlling steps, which do not read some data flowing along the datastream, but have the capability (*can read*) to do so. If they actually read these data at certain points of the process timeline, in combination with other data, via the steps, a privacy breach scenario will arise. In large processes, it is difficult to manually identify which agents, controlling which steps, can read which data, to lead to a privacy attack. Thus, we model a process and a possible attack as directed acyclic graphs with data, steps, agents, filters and restrictors (Section IV). The agents in the graphs are all *insiders* who have different access levels to the collection-oriented data moving in between the steps. Then DIAS identifies automatically, if there are any process steps which can read such data, which in combination with other data, can give rise to the attack. This is achieved by mapping the attack model onto the process model based on some similarity matching criteria. A successful attack identification is defined in the form of satisfaction of these matching criteria (Section V). The intuition behind this is, these criteria examine whether the process model can act as the supportive medium, with agents, being capable of reading certain "normally unutilized" data at certain points of timeline, so that the attack can be realized through it.

DIAS can also identify attacks with apparent dissimilarity between the attack and the process model. For example, if an attack model step requires reading a vote in an election, and a process model has a step which allows a manual reading of the vote, then the attack step can be mapped onto the process step, and hence be successful. This is because the attack model step just needs to read a vote, and whether it is a manual reading of the vote or a computer software reading the vote, as provided for by the process model, the attack step can be realized through this corresponding process step in either case.

DIAS also identifies agent collusion scenarios. We use logic rules to implement (Section VI) the attack model-process model similarity matching criteria. These rules generate and test the different possible ways in which an attack model is similar to a process model according to the matching criteria, each corresponding to a way in which the attack can be carried out on the process, thereby identifying the rogue, "responsible for attack" agents too. Section VII presents the results of applying DIAS on election process examples.

DIAS automatically searches for improvement opportunities to eliminate the identified attacks on a process (Section VIII). The steps in the process are scanned for improvement opportunities in a descending order of the number of times they are attacked across the different possible ways of attack. This scanning order ensures that a larger number of attack ways are eliminated in the initial rounds of improvements, thereby quickly presenting the user with a more robust process model. We then evaluate our improvements and iteratively exploit the improvement opportunities to ensure that the process is made robust against the attack in all possible ways or in as many possible ways as the improvement opportunities can eliminate.

We conclude our paper with related work in Section IX, and summary of DIAS and future direction in Section X.

Thus, given a set of process and attack models, DIAS identifies which attacks may be carried out successfully on which of these processes by which agents in which ways, and then make the processes robust against these attacks.

A major advantage of DIAS is that it provides a formal analysis mechanism to identify and remove vulnerabilities from a process statically, without the actual process needing to

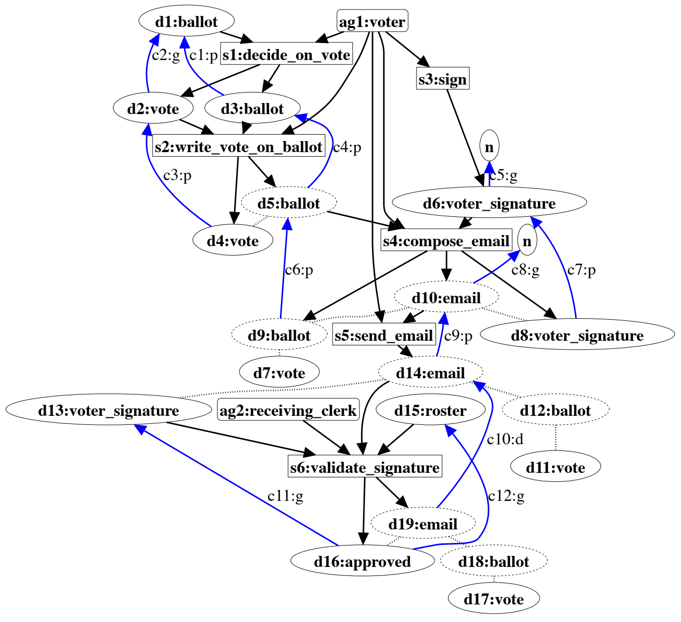

Fig. 1. Vote by Email process example: A voter decides and writes his vote on a ballot file. He also scans his signature on a file. The voter then composes an email with his vote-annotated-ballot and signature files as attachments, sending it out to the Election Office. The receiving clerk approves the legitimacy of his vote, after verifying the voter's signature against the roster registration record.

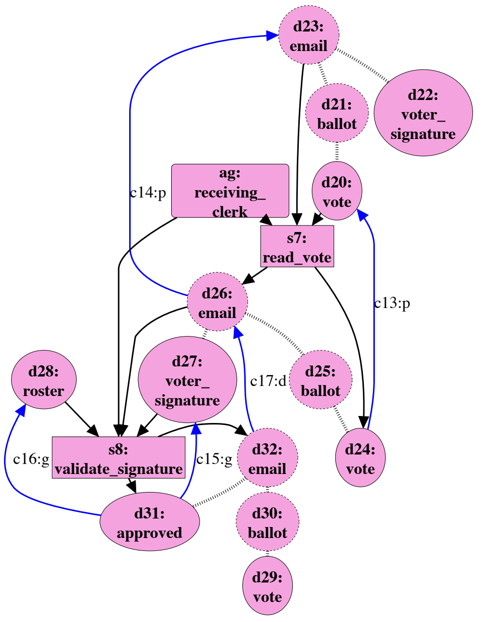

Fig. 2. A privacy attack example: A receiving clerk at the Election Office *Reads Vote* by opening the voted ballot attachment in the email, sent by the voter. He then checks the attached voter's signature as well, during the Validate Signature step, thereby breaching the voter's privacy, since now he knows for whom a voter has voted.

be carried out, thereby saving lots of time and money wasted for after the fact analyses.

II. Motivating example

Figure 1 shows a vote-by-email process example to motivate DIAS. The figure nodes are either oval, rectangle or rounded rectangle. The ovals represent data, rectangles the steps, and rounded rectangles the agents. Nodes are labeled as $id{:}t$ where id is a unique node identifier and t is a node type. For example, in Figure 1, the first step node has the label as $s1$:*decide_on_vote*, where $s1$ uniquely identifies the step, which has a type of *decide_on_vote*. Refer to Section IV for explanations on node types. We consider a collection oriented hierarchy where a datum may possess subdata. A step can contain multiple functions. For function semantics, refer to Section IV. In a vote-by-email process, a voter *decide_on_vote*, and *write_vote_on_ballot*. The ballot is a file, and vote is a marking on that file. He also *signs*, scanning his signature into a file. *Write_vote_on_ballot* and *signing* can be parallel activities. The voter then composes an email, attaching his voted ballot file, and signature file to the email, which are two subdata of different types. He then sends his email via the Internet to the Election Office, where a receiving clerk, compares the signature on roster (a pre-recorded registered voters' list) with the email attached signature. A match signifies that the voter is registered, in which case the clerk 'approve-marks' the email via the *validate_signature* step, indicating that the attached vote must be counted as a valid one. Note that during the *validate_signature* step, the clerk is not authorized to open the voted ballot attachment. He also removes the voter signature file from the email, so that the secret of "who has voted for whom" cannot be inferred by a downstream agent, handling the voted ballot attached email.

Let us now consider how a privacy breach attack may occur, as shown in Figure 2. The initial steps from *decide_on_vote* till *send_email* remain the same as in the vote-by-email process and thus have not been shown. The receiving clerk after getting the email as an output from the *send_email* step, performs the *read_vote* step. During this step, he reads the voter's vote, by opening the voted ballot attachment in the email. Then he reads the attached voter signature file during the *validate_signature* step, thereby accomplishing the voter confidentiality breach attack, since from the voter's signature and the ballot's vote he can know for whom a voter has voted.

Given this example process and attack, can DIAS identify automatically, if the agents can successfully carry out this attack on this process? Can it find out how to improve the process to prevent the attack? These are the principal questions which we solve in the following sections.

III. High Level approach Overview

Figure 3 shows a high-level overview of DIAS. A domain expert defines a set of process and attack models. He picks a process model P and an attack model A_i from the respective sets and provides them as inputs to the `Generate Attack Maps` activity to find out if A_i can be carried out against P. A_i represents any attack in the stream of attacks $A_1, A_2, \ldots A_m$ making up the attack model set. We use Answer Set Programming (ASP) [5], [6], a rule-based declarative programming paradigm, to implement the `Generate Attack Maps` activity. The implementing program encodes the valid conditions under which an attack is successful and also enumerates all possible ways in which A_i can be carried out against P based

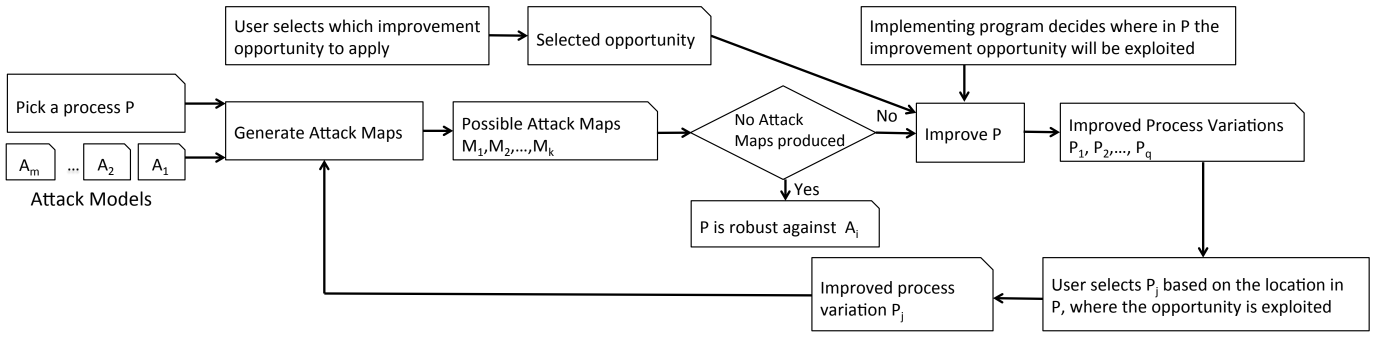
User selects which improvement opportunity to apply
Selected opportunity
Implementing program decides where in P the improvement opportunity will be exploited
Pick a process P
Generate Attack Maps
Possible Attack Maps M1,M2,…,Mk
No Attack Maps produced
No
Yes
Improve P
Improved Process Variations P1, P2,…, Pq
Am … A2 A1
Attack Models
P is robust against Ai
Improved process variation Pj
User selects Pj based on the location in P, where the opportunity is exploited

Fig. 3. Approach overview. Rectangles:Activities, Cut cornered rectangles:Inputs and Outputs, Diamond:Decision box. DIAS presents an iterative process improvement and evaluative approach to test a process P's vulnerabilities against an attack A_i, and eliminating them.

on those conditions. We denote this set of enumerated attack scenarios as $M_1, M_2, \ldots, M_k$. If this set is empty, then A_i cannot take place on P.

If non-empty, this set of attack scenarios is input to `Improve` P, which modifies P to thwart A_i. The user selects an improvement method which can be exploited to prevent A_i. `Improve` P then scans and applies (if the opportunity exists) the user selected improvement method on the steps in P in a descending order, addressing the most attacked step first, followed by the lesser attacked ones. Thus it is the implementing program which decides the location in P in which the selected improvement opportunity will be exploited.

$P_1, P_2, \ldots, P_q$ is a set of "improved" process models, produced as an output from `Improve` P, where every set element is a variant of process P with certain attack avenues for A_i against it, being thwarted. Each variant represents a way by which the same process improvement method can be used in different locations in P to eliminate attack ways. P can contain multiple steps, which are attacked the same number of times across the different possible ways of attack and all or some of them can be exploited for the same improvement method. Thus a new variant is produced depending on the location of P's step actually exploited by `Improve` P. The user then selects a variant, say P_j from this improved set, which is then again provided as an input to the `Generate Attack Maps` program. `Generate Attack Maps` again runs the ASP rules for attack determination based on the attack conditions to test out if P_j is indeed improved against A_i. In this way, the process improvement and evaluation continues iteratively till P becomes robust against A_i in all possible ways, or in as many possible ways as the improvement methods can eliminate. Then, we can reapply DIAS to check the robustness of the next process selected from the set of process models. Thus, DIAS can be applied to all possible combinations of attack model-process model pairs from the input sets, to identify and improve the vulnerable processes.

IV. Process and Attack Models

In this section, we define the syntax and semantics of our graph-based language, modeling processes and attacks. We define process and attack models based on a directed acyclic process graph $G = (V, E)$ whose nodes $V = \mathsf{V_S} \cup \mathsf{V_D} \cup \mathsf{V_T} \cup \mathsf{V_F} \cup \mathsf{V_Y}$ are *steps* $\mathsf{V_S}$, *data* $\mathsf{V_D}$, *agents* $\mathsf{V_T}$, *filters* $\mathsf{V_F}$ and *restrictors* $\mathsf{V_Y}$. A *step* represents a task; a *datum* is a knowledge; an *agent* controls a step; a *filter* is an add-on activity to a step, capable of removing certain data from the datastream and finally, a *restrictor* is also an add-on activity to a step, preventing it from reading certain datum. With each kind $K \in \{\mathsf{V_S}, \mathsf{V_D}, \mathsf{V_T}, \mathsf{V_F}, \mathsf{V_Y}\}$ of node in V, we associate a set of *types* $\mathbb{T}_K$. For example, for $K = \mathsf{V_D}$ (datum), we might have the set of *data types* $\mathbb{T}_D = \{$*email,vote*$\}$. Types of the same kind for data types $\mathbb{T}_D$ and step types $\mathbb{T}_S$, can be arranged into a type hierarchy. Conceptually, a type may be further structured to contain various properties like a *class* in object oriented paradigm [7]. For example, *email* can have properties as *emailId, sender* etc. Type hierarchy follows the inheritance concept of object oriented paradigm [8].

With nodes, for example, for a data node $d \in \mathsf{V_D}$, we associate a single type $t = \mathsf{type}(d)$ from $\mathbb{T}_D$. Formally, we have a family as type_K: $V_K \rightarrow \mathbb{T}_K$. Thus, we have type_D: $\mathsf{V_D} \rightarrow \mathbb{T}_D$, which associates data types to data nodes. Similarly types can be associated with members from other node kinds. If it's clear from the context what kind of node we're working on, we can just say $\mathsf{type}(n) = t$. For example in Figure 1, $\mathsf{type}(d1) = \mathit{ballot}$. Nodes have labels of the form nodeId:type as already discussed in Section II.

Each datum in G maybe composed of one or more similar or differently typed, children subdata. Thus in Figure 1, the parent *email* datum $d10$ has a pair of children, *ballot* datum $d9$, and *voter_signature* datum $d8$. A subdatum is also a datum, which may itself contain other subdata. For example, $d9$ has a child as $d7$. Thus, there can be multiple hierarchical levels in our data model. A datum with zero subdata ($d4$ in Figure 1) is an atomic entity. A *transparent* datum (dashed node) allows read access by a step to all of its subdata. An *opaque* datum (solid node) restricts the read access to all of its subdata by a step. For example in Figure 1, the *send_email* step $s5$ reading the transparent *email* datum $d10$, *canRead* all of its children, i.e., *ballot* datum $d9$ and *voter_signature* datum $d8$. But if $d10$ were solid, then $s5$ cannot read either $d9$ or $d8$.

The edges $E = \mathsf{E_R} \cup \mathsf{E_{CR}} \cup \mathsf{E_W} \cup \mathsf{E_C} \cup \mathsf{E_{DU}} \cup \mathsf{E_{CO}} \cup \mathsf{E_X} \cup \mathsf{E_H}$ are as follows: $\mathsf{E_R} \subseteq \mathsf{V_D} \times \mathsf{V_S}$ is the set of *read* edges signifying that steps *consume* data. A step requires access to all data which are connected to it via *read* edges. For example, in Figure 1, step $s1$ of type *decide_on_vote*, *reads* datum $d1$ of type *ballot*. $\mathsf{E_{CR}} \subseteq \mathsf{V_D} \times \mathsf{V_S}$ is the set of *canRead* edges signifying that steps have the capability to read data, but do not actually read or use them as their inputs, while carrying out the process. For example, in Figure 1 a *canRead* edge can be inferred between the step $s6$ of type *validate_signature* and the datum $d12$ of type *ballot*. For assumptions governing the inference of a *canRead* edge, refer to *Access assumptions* paragraph later in this section. $\mathsf{E_W} \subseteq \mathsf{V_S} \times \mathsf{V_D}$ is the set of *write* edges signifying that steps produce data. A step produces all the data to which it is connected via write edges. For example, in Figure 1, step $s1$ of type *decide_on_vote*, *writes* datum $d2$ of type *vote*. $\mathsf{E_C} \subseteq \mathsf{V_T} \times \mathsf{V_S}$ is the set of *control* edges signifying that agents perform steps. For example in Figure 1, agent $ag1$ of type *voter*, *controls* step $s1$ of type *decide_on_vote*. $\mathsf{E_{DU}} \subseteq \mathsf{V_D} \times \mathsf{V_D}$ is the set of *dependsUpon* edges showing which specific output data are functions of which specific input data to a step. The *dependsUpon* edges (blue) travel opposite to the usual dataflow direction determined by the *read* and the *write* edges. Thus, in Figure 1, datum $d2$ of type *vote* depends on datum $d1$ of type *ballot*. $\mathsf{E_{CO}} \subseteq \mathsf{V_D} \times \mathsf{V_D}$ is the set of *isChildOf* edges denoting the parent-child relationship between data. An *ischildOf* edge in G is modeled as a dotted one between the parent and the child data, with the parent on top of the child in the space layout. Thus, the parent datum $d10$ of type *email* is connected to its pair of children of types *ballot* and *voter_signature*, via the isChildOf edges. The edges are undirected, since they do not model any dataflow or dependency relationship. $\mathsf{E_X} \subseteq \mathsf{V_F} \times \mathsf{V_S}$ is the set of *filter* edges denoting that filters remove certain types of data from the datastream, thus preventing them from appearing as outputs, or children on the outputs from the steps with which they are associated. $\mathsf{E_H} \subseteq \mathsf{V_Y} \times \mathsf{V_S}$ is the set of *restrict* edges denoting that restrictors prevent the associated steps to read a datum or a subdatum of a particular type. Unlike filters, restrictors do not remove any data from the process datastream. A *path* is a sequence of nodes $v_1, v_2, \ldots, v_n$ such that $(v_i, v_{i+1}) \in E$.

The data dependency function declarations determine the semantics of G. I denotes the set of all step function identifiers which relate the output data with the input data in G. O=$\{p, g, d, u\}$ defines the set of types of operations being performed by a data dependency function, indicating the meanings of the data dependencies. They are as follows:

- *Pass through* (p) denotes that a step function, without modifying the input datum, writes it as is, as an output.
- *Generate* (g) denotes that a step function produces a new output datum, previously non-existent on the datastream.
- *Delete* (d) signifies that a step function removes an input datum from the datastream.
- *Update* (u) signifies that a step function, changes one or more property values of an input's data type (but not the type itself) to produce an output datum.

We associate labels to *dependsUpon* edges which are of the form $i : o$ where $i \in I, o \in O$.

For example, the step $s1$ of type *decide_on_vote* in Figure 1 has two data dependency functions, $d3 = c1(d1)$, and $d2 = c2(d1)$. The function $c1$ has a pass through semantics, reading datum $d1$ of type *ballot*, and passing it along the datastream, as an output datum $d3$ of type *ballot*. Another function $c2$ has a generate semantics, reading datum $d1$ of type *ballot*, and writing datum $d2$ of type *vote*. Thus the label on the *dependsUpon* edge between $d3$ and $d1$ is $c1$:p and that on the edge between $d2$ and $d1$ is $c2$:g. [1]

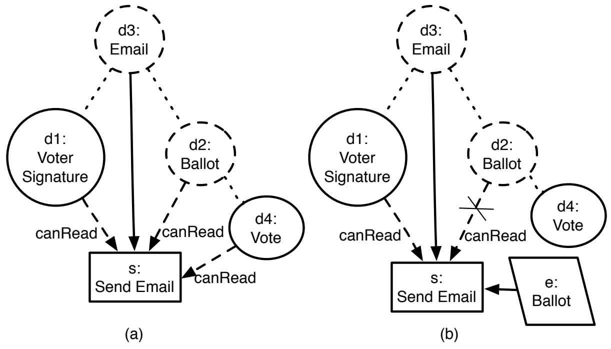

Fig. 4. (a) Step s of type *Send Email* reads transparent datum $d3$ of type *Ballot*- hence s *canRead* all the children of $d3$, i.e., datum $d1$ of type *Voter Signature* and datum $d2$ of type *Ballot*. (b) Step s can no longer read the datum $d2$ because of the matching restrictor.

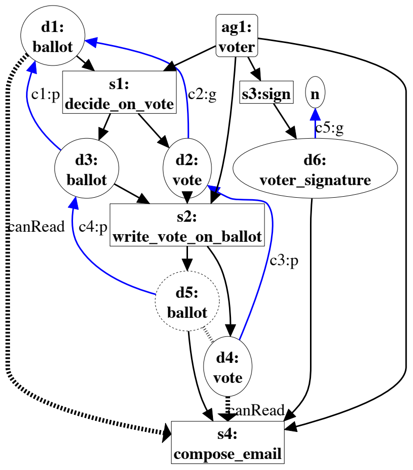

Fig. 5. Step $s4$ of type *compose_email* *canRead* the upstream *ballot* datum $d1$; the intermediate steps $s1$ and $s2$ have functions $c1$ and $c4$ respectively, which passes the ballot unmodified onto $s4$.

A *pass through* of a parent datum by a step implies a pass through of all the subdata of that parent datum by that step.

Next we describe the assumptions, governing the inference of a *canRead* edge in G.

Access assumptions: A *canRead* edge is an inferred edge which follows from the below assumptions:

1) If a step reads a transparent parent datum, then it *canRead* all children subdata of that parent. However, a restrictor may prevent read access by a step to a child subdatum which matches its type. Figure 4 (a) shows an example where the step s of type *Send Email* reads the transparent parent datum $d3$ of type *Email*, implying that it *canRead* all the children data of $d3$, i.e., $d1$ of type *Voter Signature* and $d2$ of type *Ballot*. In Figure 4 (b), a restrictor of type *Ballot* prevents the step s from reading $d2$.

[1] *Generate* function dependency may not have any input, in which case we label the input as n (null). Function $c5$ in Figure 1 shows such an example.

2) If a step reads a subdatum in a transparent parent datum, then it *can read* any other subdatum of that parent datum. However, just like in *Access assumption* 1), a restrictor may prevent read access by a step to a child subdatum which matches its type.
3) If a step reads a subdatum of a parent datum, then it *can read* the enclosing parent as well. The intuition follows from the scoping concept in a programming language, where a read access to an inner entity automatically grants a read access to the enclosing outer entity.
4) The *isChildOf* relation is defined transitively over the set of all data $\mathsf{V_D}$ in G, thus facilitating the inference of additional *canRead* edges between a step and the descendants of a transparent datum. Since a step *canRead* a child only if its parent is transparent, as soon as we encounter an opaque datum, in the path, made up of data connected by *isChildOf* edges, cutting across the parent-child hierarchical levels of G, we can infer no more *canRead* edges between the descendants of that opaque node and the step reading the opaque node, or any of its ancestors. For example in Figure 4 (a), an additional *canRead* edge can be inferred between the step s of type *Send Email* and datum $d4$ of type *Vote*. The step s reading datum $d3$ implies that s *canRead* its descendant $d4$ as well, and the parent of $d4$ being transparent, allows for this possibility.
5) A step s *canRead* a datum d indirectly from the upstream, if all the functions, acting on d as input, belonging to all the steps on any path between s and d, only perform a *pass through* operation on d, or on parent of d. For example in Figure 5, the *compose_email* step $s4$ *canRead* the upstream *ballot* datum $d1$ indirectly, since the functions $c1$ and $c4$ of steps $s1$ and $s2$ respectively, pass along the ballot datum unmodified onto step $s4$.

We define a process model and an attack model as two distinct types of process graphs. The process model is a specification, drawn out by the domain experts to achieve a useful goal, while the attack model represents a malicious plan, sought by the rogue agents via the process model.

Formally, a process model P is a process graph $G^P = (V^P, E^P)$. An attack model A is similarly, simply a process graph $G^A = (V^A, E^A)$ where $\mathsf{V_F}^A = \emptyset$, $\mathsf{V_Y}^A = \emptyset$, $\mathsf{E_X}^A = \emptyset$ and $\mathsf{E_H}^A = \emptyset$, since an attack model does not contain any filter and restrictor constructs, and correspondingly no *filter* and *restrict* edges. Henceforth, in all the following figures we have represented process model as white colored graphs, and attack models as pink colored graphs.

V. Attack Semantics

We now describe a *valid* attack semantics, and determine in how many possible ways an attack can take place on a process.

A. Map Conditions

For an attack A to be successful on a process P, we test if the process model steps have the capability to read and write the data required by the steps of A in order to carry out the attack. Also, the process agents need to collude if A requires that. All these requirements, intuitively reduce to a similarity matching between the corresponding nodes of A and P.

Thus, we define an attack as a mapping relation M between an attack model A and a process model P, i.e., relating nodes in A with nodes in P: $M \subseteq V^A \times V^P$. An attack map[2] M is said to be *well-formed* if it relates A nodes and P nodes of the same kind, i.e., $M = M_{\mathsf{V_S}} \cup M_{\mathsf{V_D}} \cup M_\alpha \cup M_\omega \cup M_{\mathsf{V_T}}$. We only consider well-formed mappings in this paper.

$M_{\mathsf{V_S}}$: $\mathsf{V_S}^A \rightarrow \mathsf{V_S}^P$ maps attack model steps to process model steps where $\mathsf{V_S}^A$ and $\mathsf{V_S}^P$ denote the set of all steps in attack model and process model respectively. $M_{\mathsf{V_D}} \subseteq \mathsf{V_D}^A \times \mathsf{V_D}^P$ relates attack data with process data, where $\mathsf{V_D}^A$ and $\mathsf{V_D}^P$ denote the set of all data in attack model and process model respectively. $M_\alpha, M_\omega \subseteq \mathsf{V_D}^A \times \mathsf{V_D}^P$ also relate attack data and process data, but are used to identify the beginning and end of a *sequence mapping*, respectively. Finally, $M_{\mathsf{V_T}} \subseteq \mathsf{V_T}^A \times \mathsf{V_T}^P$ relates attack agents with process agents, where $\mathsf{V_T}^A$ and $\mathsf{V_T}^P$ represent the set of all agents in attack model and process model respectively.

A well-formed M is *valid* M if all the following mapping conditions are satisfied. These conditions define when a process model allows an attack to be successfully realized through it, because of the similar nature of the pair of models.

Condition 1: Steps Match. An attack model step represents an activity, which can be carried out successfully only if the process model provides a corresponding supportive step of matching type. Thus for $M_{\mathsf{V_S}}$ to be valid, for all $M_{\mathsf{V_S}}(s^a, s^p)$, the types of $s^a \in \mathsf{V_S}^A$ and $s^p \in \mathsf{V_S}^P$ must match. Types match if the process model and attack model step types are equal, or if the process model step type is a subtype of the attack model steptype. Following the inheritance concept, a step s^p's type is a subtype of step s^a's type if the former possesses all properties of the later, along with some additional. The intuition here is that, for successful realization of an attack step, a process model step should provide at least what an attack model step demands, but can provide for more. For example, if an attack model consists of a step s^a with type as *Read Vote*, and a process model consists of a step s^p with type as *Read Vote with a software*, then we can claim that $M_{\mathsf{V_S}}(s^a, s^p)$ is valid, the process step's type being a subtype of the attack step's type. The attack step in this case can be successfully carried out via the process step (similarly reasoned as in Section I).

However if an attack model step's type is a subtype of the process model step's type, then the mapping between them is valid conditionally, i.e., the attack can succeed only if the process step type allows for the additional condition to be met. The intuition is, as long as the supportive process model step type does not explicitly prohibit the characteristics demanded by an attack model step type, a valid mapping can exist between them, with the condition that attack step's additional requirements must be met. For example, if, s^a's type is *Read Vote with a software* and and s^p's type is *Read Vote*, then a valid mapping can exist between them, with the proviso that the process model allows, reading the vote with a software.

For an attack to be successful, all of its steps need to be

[2]short for: attack mapping relation (i.e., M is not a function but a relation)

carried out. Thus all steps in ${\mathsf{V_S}}^A$ are mapped to some steps in ${\mathsf{V_S}}^P$ (unless they are part of an attack *sequence map* as explained in Condition 4). For all $s^a \in {\mathsf{V_S}}^A$ there is a $s^p \in {\mathsf{V_S}}^P$ such that $M_{\mathsf{V_S}}(s^a, s^p)$, or else s^a is part of an attack sequence map (Condition 4).

Condition 2: Inputs Match. An attack model step may need to read certain input data to be successful. Thus the corresponding process model step must provide for these matching input data. The process model can meet this requirement in two ways: either the process model step actually reads the data (where a *read* edge exists between the process data and the step) needed by the attack model step to be carried out successfully, or the process model step possesses the capability to read the data (where a *canRead* edge can be inferred between the process data and the step) needed by the attack model step. The capability of a process step to read a data is governed by the *Access assumptions* (Section IV). Thus for $M_{\mathsf{V_D}}$ to be valid, for all $M_{\mathsf{V_D}}(d^a, d^p)$, the types of $d^a \in {\mathsf{V_D}}^A$ and $d^p \in {\mathsf{V_D}}^P$ must match, indicating that the process model do indeed support the attack model's requirement. Types match if the process and the attack model data types are equal, or if the process model data type is a subtype of the attack datatype. A datum d^p's type can be a subtype of a datum d^a's type according to the inheritance concept, or if, d^a possesses only a subset of the children, possessed by d^p.

A filter must not block the transitive availability of an input, say d^p, to a process step s^p, when s^p satisfies an attack model requirement through a *canRead* edge. If a filter, checking for type of d^p, is present on any step on the path, $d^p, \ldots, s^p$, then it removes d^p from the datastream, thereby disallowing it to act as the target of data map from the attack model. Hence there should exist at least one path $d^p, \ldots, s^p$ in the process model, such that there is no filter for d^p in this path. Similarly there should be no restrictor associated with step s^p, which prevents the access to d^p by s^p, for a valid attack to take place. Formally: if $M_{\mathsf{V_D}}(d^a, d^p)$ and $M_{\mathsf{V_S}}(s^a, s^p)$ and $d^a \in \mathsf{in}(s^a)$, then $d^p \in \mathsf{in}^+(s^p)$ and there exists a path $\pi_p : d^p, \ldots, s^p$ in P such that for all steps $s \in \pi_p$ and for all filters f of s, f does not match d^p, i.e., $\mathsf{type}(f) \neq \mathsf{type}(d^p)$ and for all restrictors e of s^p, e does not prevent *read* of d^p by s^p, i.e., $\mathsf{type}(e) \neq \mathsf{type}(d^p)$. $\mathsf{in}(s^a)$ is the set of all direct data inputs to s^a. $\mathsf{in}^+(s^p)$ is the set of all data d that are direct or indirect inputs to s^p, i.e., there is a path from d to s^p in P.

$M_{\mathsf{V_D}}$ preserves *read* edges, and thereby the dataflow. For example, if a step of type s_1 reads a datum of type d_1 in an attack model, and this (step,datum) pair is mapped to its counterpart in the process model, for a valid attack to take place, it must be the case that in the process model, the step of type s_1 (or subtype of s_1) reads (or *canRead*) datum of type d_1 (or subtype of d_1). The attack model datum of type d_1 cannot be mapped to a matching datum, which is written by the step of type s_1 (or subtype of s_1) in the process, or to any downstream datum which appears after the step of type s_1 (or subtype of s_1) in the process timeline.

An attack model step needs to read all the inputs in order to be successful. Thus all of the inputs read by any attack model step, must have their corresponding matches in the process model, for a successful attack. For all $d^a \in {\mathsf{V_D}}^A$ there is a $d^p \in {\mathsf{V_D}}^P$ such that $M_{\mathsf{V_D}}(d^a, d^p)$, or else d^a is part of an attack sequence map (Condition 4).

Condition 3: Outputs Match. When an attack model step is successfully performed, it produces certain data. Thus the corresponding process model step must provide the supportive output data of matching types, to which the attack model data can be mapped to, for a successful attack. Thus, if $M_{\mathsf{V_S}}(s^a, s^p)$, i.e., an attack step s^a is mapped to a process step s^p, then we require that any output $d'^a \in \mathsf{out}(s^a)$ also matches an output $d'^p \in \mathsf{out}(s^p)$ for a successful attack. Here $\mathsf{out}(s^a)$ represents the set of all data d' that are written by step s^a. $\mathsf{out}(s^p)$ is defined similarly.

Also, for a valid attack, there should not be any filter, associated with process step s^p, checking for a datum of type of d'^a or checking for a datum of type of a descendant of d'^a. Otherwise due to this filter, process step s^p cannot produce an output or descendant of the output, as demanded by the attack, thereby failing the attack. Formally: If $M_{\mathsf{V_S}}(s^a, s^p)$ and $M_{\mathsf{V_D}}(d'^a, d'^p)$ and $d'^a \in \mathsf{out}(s^a)$, then $d'^p \in \mathsf{out}(s^p)$ and for all descendants $des^a \in {\mathsf{V_D}}^A$ of d'^a and all filters f of s^p, f neither matches des^a nor d'^a, i.e., $\mathsf{type}(f) \neq \mathsf{type}(des^a)$ and $\mathsf{type}(f) \neq \mathsf{type}(d'^a)$.

An attack model step needs to produce all the data it is connected to, via the write edges, in order to be declared completed. Thus all of the outputs written by any attack model step, must have their corresponding matches in the process model, for a successful attack.

Condition 4: Sequence Match. Sometimes a sequence of attack steps $s_1^a, \ldots, s_n^a$ can be realized by a malicious agent using a single step s^p of the process model. We consider this possible if at least one of the steps s_i^a in the sequence matches s^p in type. We also require that the inputs and outputs of the attack sequence match those of s^p, similar to Conditions 2 and 3 before. To this end, the mapping M "encloses" the attack sequence via special edges M_α and M_ω, relating the data inputs (start of the sequence) and outputs (end of the sequence) to those of s^p. Formally: if $M_\alpha(d^a, d^p)$ and $M_\omega(d'^a, d'^p)$, then there exists a path π_p: $d^p, \ldots, s^p, d'^p$ in P such that for all paths π_a: $d^a, s_1^a, \ldots, s_n^a, d'^a$ in A there is a matching step $s_i^a \in \pi_a$ with $\mathsf{type}(s_a^i) = \mathsf{type}(s^p)$.

Attack steps in the sequence, including their inputs, outputs and controlling agents are assumed to be carried out via a single process step. Thus they are exempt from being explicitly mapped via $M_{\mathsf{V_S}}, M_{\mathsf{V_D}}$, and $M_{\mathsf{V_T}}$.

Condition 5: Non-Collusive Agents match trivially. If $M_{\mathsf{V_T}}(t^a, t^p)$, i.e., an attack agent t^a is mapped to a process agent t^p, and t^a controls step s^a and agent t^p controls step s^p, then, we require that, $M_{\mathsf{V_S}}(s^a, s^p)$. Also, the inputs and outputs of s^a match the inputs and outputs of s^p respectively.

We assume that any process agent can be made rogue; so whenever an attack step requires an agent to perform it, the process model can always provide one, and it may not have the same type as that of the attack model agent type, but still be capable of carrying out the required attack step, as long as the corresponding step types in the attack and process match.

When agents *collude*, agent mapping scenarios becomes non-trivial, as discussed in Section VII-B. Note that for a valid attack, mapped steps are either part of an attack sequence or are mapped individually.

B. Multiple Ways of Valid Attack

For a given attack model A, there can be many mapping relations M, that relate A to a given process model P. We try to find out all such Ms, each corresponding to a way, in which A can be carried out against P. If there is no M, we infer that P is robust against A in all possible ways. The problem of determining all such Ms is in essence a search problem: each possible way of mapping attack steps/data to process steps/data must be examined. Each combination is generated, and then tested against the requirements of a valid mapping as explained in Section V-A. Thus we use a *generate and test* paradigm to generate all attack mapping possibilities and test the validity of the mapping.

VI. Implementation

We use DLV [9], [10], a state-of-the-art implementation of ASP [6], [11], to implement our valid attack map conditions (Section V-A). We have included a selected portion of our entire implementation as a representative.

We encode the constructs in the process model and attack model like step, datum, agent etc., their types and the interactions between them, as a set of DLV *facts*. For example `pm_read(d,s)` is a process model fact encoding that process model step `s` *reads* data `d` i.e., `(d,s)` $\in \mathsf{E_R}^{P}$. Attack model facts are similar, but they are prefixed with `am`.

Next, we encode, a valid attack on a process, implementing mapping Condition 4 in Section V-A.

```
mapsequence(A1,S1,S3,A3,A2,SP,A4):-          1
                 am_sequence(S1,S3),         2
                 am_connected(S1,S2),        3
                 am_connected(S2,S3),        4
                 am_steptype(S2,X),          5
                 pm_steptype(SP,X),          6
                 allInMap(A1,S1,A2,SP),      7
                 allOutMap(S3,A3,SP,A4),     8
                 not filter_restricts(S2,SP).  9
```

The above DLV rule implements the criteria for an (input data,first step in sequence,last step in sequence,output data) in an attack model i.e., `(A1,S1,S3,A3)` to be validly mapped to an (input data,step,output data), i.e., `(A2,SP,A4)` in the process model (modeled by `mapsequence`). If an attack model step `S2` in a sequence, has the same type `X` as that of a process model step `SP`, and the datatypes of all the data input to the first step `S1` in the attack sequence and output from the last step `S3` in the attack sequence match the datatypes of at least some data input to, and output from the process step `SP` (modeled by the predicates `allInMap` and `allOutMap`, respectively), then we can claim that `S1` and `S3`, along with its input to `S1` and output from `S3` data, can be mapped to `SP` and its input and output data respectively, signifying that a sequence of attack steps can be successfully realized via a process step. The last conjunct in the above rule body on line 9 ensures that there is no restrictive filter on the process model step `SP` which can prevent its output `A4` from being the target of the map (referring to the filter restriction requirement in Condition 3). A pair of steps is *connected* (modeled by `am_connected`) if the former writes a datum which is read by the latter, directly or indirectly (A step can indirectly read a datum as discussed in *Access assumption 5* in Section IV). A pair of *connected* steps where the steps are different from each other, form a *sequence*, as modeled by `am_sequence` in line 2 of the rule. The `allInMap` atom implements the condition, where a process model step can support an input read requirement by an attack step, either by actually reading the input data, or by having the capability to read the input data (Condition 2 in Section V-A). Similarly, `allOutMap` supports the output condition (Condition 3 in Section V-A).

Given an attack and a process model, `Generate Attack Maps` (in Figure 3) implementation finds out in how many different ways this attack is validly possible on this process based on the attack mapping conditions. This implementation is realized using ASP paradigm, based on stable model semantics [12], amenable to computationally difficult (e.g., NP-hard) search problems. This implementation is as follows:

```
inmapsequence(A1,S1,S3,A3,A2,SP,A4) v
outmapsequence(A1,S1,S3,A3,A2,SP,A4):-
mapsequence(A1,S1,S3,A3,A2,SP,A4).
```

Using the above rule, DLV generates all attack maps or stable models in which all valid attack conditions (as explained in Section V-A) are satisfied. Each attack map corresponds to a way in which an attack can be carried out on a process. Each such map contains `inmapsequence` atoms denoting that certain entities in the attack model are mapped to those in the process model, or `outmapsequence` atoms denoting that those attack model entities are not mapped.

We feed as input, the attack and process models, along with our valid attack conditions (of Section V-A) and rules used for resultant attack map visualization, to the DLV answer set solver implementing `Generate Attack Maps`. The solver generates a set of attack maps, each showing a way in which all the attack model steps, data (or attack sequence) and agents can be mapped to some process model steps, data and agents, respectively, as per our valid attack conditions. The model constructs in each of these attack maps get projected onto *node* atoms, and the relations and corresponding mappings among the constructs onto *edge* atoms of a graph, by the visualization logic. Using DLVWrapper [13], a Java interface for the DLV system, we implement a Java-based method to collect these node and edge atoms and construct a graph in *dot* format (Section VII shows such program-generated output graphs).

Thus, utilizing the power of the answer set solver DLV, we can implement successful attack semantics, and generate all possible ways of attack based on this semantics.

VII. Results

This section shows our results of running `Generate Attack Maps` implementation (Section III) on input process and attack models. Section VII-A shows an automatic privacy attack identification, arising due to subtle read capabilities of process steps, in motivating example of Section II. Section VII-B deals with privacy attack identification on a *vote-in-person* process, involving agent collusions.

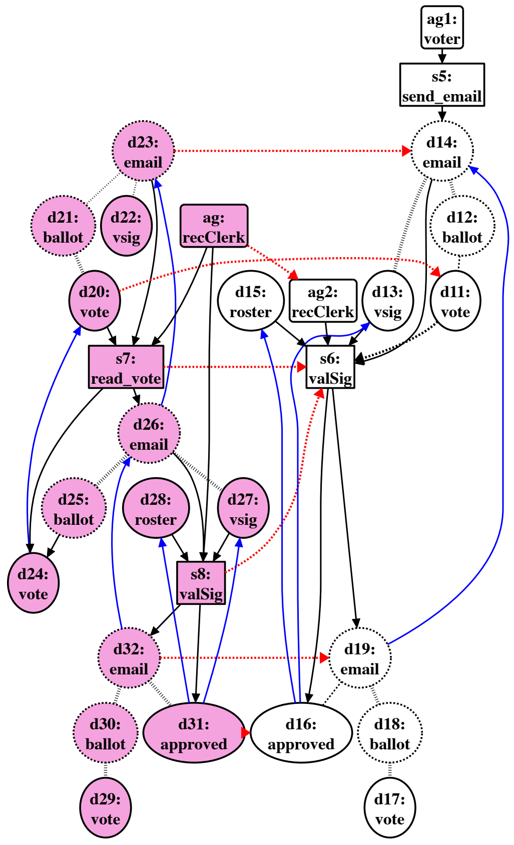

Fig. 6. Output of `Generate Attack Maps` (in Figure 3) run on our motivating example, demonstrating a voter confidentiality attack, where a clerk reads the voted ballot, before validating the voter signature, thereby knowing for whom a voter has voted.

A. *Can Read Capability Attack*

Figure 6 shows the result, automatically produced, when we run `Generate Attack Maps` implementation on our motivating example. The interpretation of the output requires knowledge of the process semantics as well as domain knowledge and is left to the user. For example, this output can be interpreted as the malicious goal of *read_vote* along with the non-rogue step of *validate_signature* in the attack model, being realized through the single step of *validate_signature* in the process model. This is visualized by the pair of red, dashed mapping edges from step $s7$ of type *read_vote* and step $s8$ of type *valSig* (denoting *Validate Signature*) in the attack model, both terminating in the step $s6$ of type *valSig* in the process model. [3] The mapping edge between ag and $ag2$ in the attack and process model respectively, identifies the *recClerk* (receiving clerk) as the rogue agent. The *receiving clerk* reads the *vote* on the *ballot*, attached in the *email* during the *read_vote step* ($s7$), and then carries out the *valSig* step ($s8$), reading the *vsig* (denoting voter signature) attachment in the email as well. Thus a privacy attack has been now realized since, the clerk knows for whom the voter has voted. Note that satisfying Condition 4 in Section V-A, all the inputs to the first step $s7$, and all the outputs from the last step $s8$ of the attack

[3]Some node label types in figures starting from Figure 6 are abbreviated. Also, only a relevant portion of the process model in Figure 6 is shown.

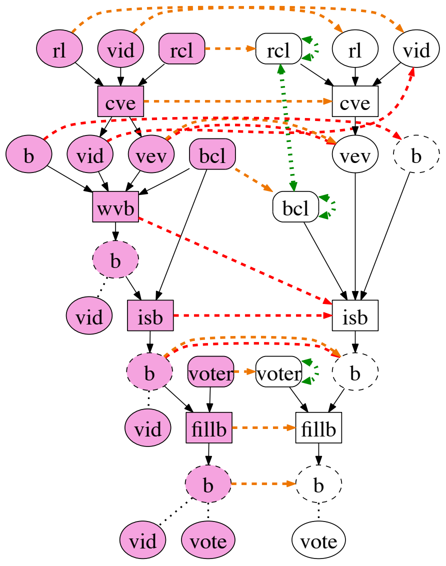

Fig. 7. Output of `Generate Attack Maps` run on a vote-in-person process, demonstrating how, the voter confidentiality can be breached when the roster clerk and the ballot clerk collude.

sequence, are mapped to some inputs and outputs respectively, of the step $s6$ in the process model. In the attack model, the step $s7$ requires to read the *vote* datum ($d20$) as one of its input. The process model provides for this requirement via the *canRead* edge (dashed black edge in Figure 6) between $d11$ and $s6$ (satisfying Condition 2 in Section V-A). Step $s6$ in the process model reads the *email* datum $d14$, which is a transparent one. As per *Access assumption* 4, $s6$ *canRead* any descendant of the transparent datum $d14$, as long as there is no opaque datum on the path between $d14$ and that "read-by" descendant, along the *isChildOf* edges. The datum $d11$ is a descendant of $d14$. Also the intermediate datum $d12$ between $d14$ and $d11$ is transparent. Hence a *canRead* edge can be inferred between $d11$ and $s6$, thereby realizing the attack.

Thus DIAS automatically identifies attack scenarios where normally unused process data are used by agents in a malicious context, leading to a privacy breach. In a non-malicious scenario, the *receiving clerk* does not read the *vote* datum, while he is validating the voter signature. But if he does, a privacy breach occurs.

B. *Agent Collusion Attack*

The mapping of the agents as described in Condition 5 in Section V-A becomes non-trivial when agents collude. Consider a vote-in-person process (white graph of Figure 7) where a roster clerk *rcl* [4] checks a voter's eligibility to vote during the *cve* (Check Voter Eligibility) step when the voter reaches the polling booth on the election day. The clerk verifies whether *vid* (voter's ID), is present in *rl* (roster list). If the verification succeeds, *rcl* tells *bcl* (the ballot clerk) to give the voter a ballot of a specific type; *vev* (*Voter Eligibility Verification*) is an abstract representation of this communication. The voter

[4]In some figures starting from Figure 7 and in text body, we refer to nodes with only types; For example, step *isb* means step of 'type' *isb*.

now gets a blank ballot from *bcl* via the *isb* (Issue Ballot) step, on which he fills out his vote, as a child of the ballot, via the *fillb* (Fills Ballot) step.

Now let us consider that insider agents want to collude, to find out for whom a voter has voted. The pink graph in Figure 7 shows such a scenario. Once *rcl* finds out that a voter is eligible to vote, he covertly passes on the *vid* to *bcl*. The *bcl* writes that secret on an empty ballot as a child of it via the *wvb* (*Write Vote on Ballot*) step, and hands it over to the voter. The unsuspecting voter casts his vote on the ballot, as a child of it, as usual. Thus we have a ballot with two children subdata, the *vid* and the *vote* thereby breaking the voter's confidentiality.

Figure 7 shows the result of running the `Generate Attack Maps` implementation on these vote-in-person process and possible privacy breach attack models, automatically identifying how, the roster and the ballot clerk can collude to actually realize the attack. The step *cve* in the attack model along with its inputs and outputs is mapped to its counterpart in the process model (satisfying Conditions 1, 2 and 3 in V-A and shown by the orange mapping edge), while the sequence of steps *wvb,...,isb* in the attack model is mapped to the single step of *isb* in the process model via a pair of red edges (satisfying Condition 4 in V-A). Also there are mapping edges, from *rcl* and *bcl* in the attack model to their counterparts in the process model. All these can be interpreted as that, the *rcl* performs the usual step of checking the voter eligibility and the *bcl* performs the malicious step of writing the *vid* on the ballot along with the non-rogue step of issuing the ballot to the voter. The attack model requires that the secret *vid*, gained by the *rcl* via the *cve* step, needs to be passed onto the *bcl*. The process model supports this requirement, via the dotted green edges between the agents rcl and bcl, which signify that the *rcl* and the *bcl* can collude. Thus through the process step *isb*, the *bcl* can perform two activities: it writes the datum *vid* (as a child on the ballot *b*) which it *canRead* from the upstream, passed onto him by the *rcl* (assuming that the step *cve* does a *pass through* on *vid*), and then issuing the ballot to the voter. The voter fills out the ballot, (as shown by the orange mapping edges between the corresponding steps of $fillb$ in the attack and process model) thereby breaching the voter confidentiality since now the same ballot contains the children, vid and $vote$.

VIII. Process Improvement

After `Generate Attack Maps` identifies the possible ways in which an attack can take place on a process, `Improve` *P* (in Figure 3) automatically searches for, and applies improvement *opportunities* in the original process model to prevent the attack from succeeding in any possible way, or as many possible ways as it can eliminate. However, in the course of these improvements, `Improve` *P* does not modify the process model in such a way that the original process goal is inhibited. Thus, none of the process steps, agents, or data are deleted or updated in their types during the improvement.

Once improved, the resulting process model is again provided as an input to `Generate Attack Map`, to confirm that the process has been indeed made robust against the concerned attack in different possible ways. It may take multiple improvement iterations before this goal is achieved.

A. Restrictor addition

In this section we show how restrictors can be added as add-on activities to process steps, to prevent attacks. We have used this improvement method to prevent the voter confidentiality attack on vote-by-email process (Section VII-A).

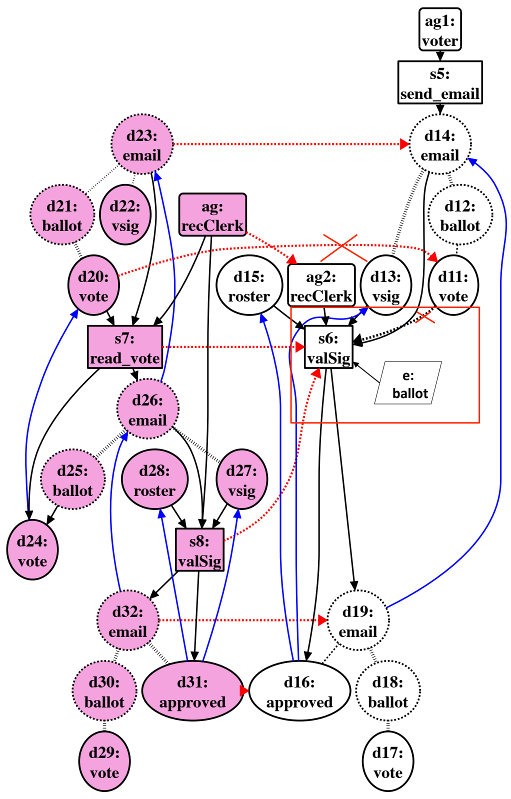

Fig. 8. A process improvement example where a restrictor e of type *ballot* (in the red box) prevents the capability of step *validate_signature* to read *ballot* ($d12$), thereby preventing its capability to read the child *vote* ($d11$) as well. Without the *canRead* edge between $d11$ and $s6$, the data mapping from $d20$ to $d11$ fails, thereby failing the voter confidentiality attack.

Figure 8 shows how a vote-by-email process can be improved to prevent the voter confidentiality breach attack. The attack model (as shown in Figure 6), requires the agent ag of type *recClerk*, carrying out the step $s7$ of type *read_vote*, to read datum $d20$ of type *vote*. The process model satisfies this requirement via the step $s6$, having the capability to read $d11$ using the *canRead* edge. `Improve P` (in Figure 3) finds an improvement opportunity here, where a restrictor e of type *ballot* is added as an add-on activity to the step $s6$ of type *valSig* as shown in Figure 8. The restrictor e prevents $s6$ from having the capability to read datum $d12$ of type *ballot* or any of its descendants. As per the *Access assumptions* (Section IV), a step *canRead* a child subdatum only if its parent is transparent. The restrictor e, by preventing the read access of step $s6$ to $d12$, has the effect of making $d12$ as opaque. Thus $s6$ cannot read $d11$, which is a child of $d12$ (shown by the red cross on the *canRead* edge between $d11$ and $s6$ in Figure 8), thereby preventing the map from $d20$ to $d11$ (shown by the red cross on the mapping edge), hence failing the attack.

A restrictor addition of type *vote* to process step $s6$, would

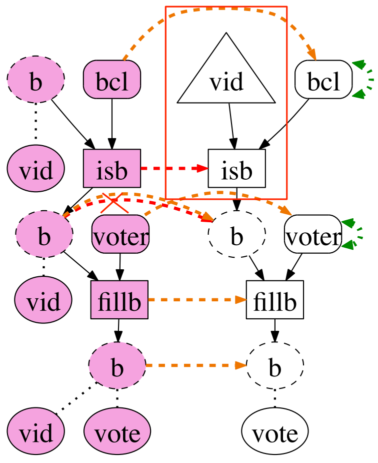

Fig. 9. `Improve P` run on a vote-in-person process, showing how an automatically added filter to the process step *isb* (in the red box), prevents the ballot clerk from issuing a *vid* containing ballot to the voter, thereby failing the voter confidentiality breach attack.

have the same effect as adding a restrictor of type *ballot* to step $s6$. The restrictor of type *vote* would have prevented $s6$ from having the capability to read any datum of type *vote*, thereby disallowing the map from $d20$ to $d11$, hence failing the attack. This alternative process improvement way through more fine-grained data access control, is desirable in scenarios where a process step reads siblings of the secret ($d11$), we are trying to prevent the read of. For example if $s6$ reads $d11$'s siblings (if present), then adding a restrictor of type *ballot* to $s6$ will prevent $s6$ from reading those siblings as well, hindering the original goal of the process in the first place.

Thus generalizing we can say that, if we need to prevent a step s from having the capability to read a datum d (s *canRead* d) in a process, then either we can add a restrictor of type equal to d's type to s, or we can add a restrictor of type equal to the type of any ancestor of d, to s, provided there is no read edge between that ancestor or any other descendant of that ancestor and s. Note that if a restrictor of type, equal to d's type is added to s, then it must be the case that there exists no read edge between d or any descendant of d, and s in the process model. Otherwise, the improvement opportunity can not be applied because it hinders the original goal of the process.

Currently this 'restrictor-addition' improvement method is only conceptualized; in future we will implement it.

B. Filter addition

We introduce a second improvement method with filters as add-on activities to process steps, to prevent attacks. This method prevents the voter confidentiality attack on vote-in-person process (Section VII-B).

Sometimes an attack model step writes a datum with a child, whereas its counterpart in the process model does not contain any matching child on the corresponding output datum. For example, in the attack model in Figure 7, the step isb writes a datum b with a child vid, whereas the output datum b from the corresponding isb step in the process model does not contain any child as vid. Thus a process improvement opportunity exists which is utilized by `Improve` P implementation. Figure 9 shows the output of `Improve` P run on the vote-in-person process, (Section VII-B). The implementation of `Improve` P automatically adds a filter, vid, on the process step, isb as shown within the red box of Figure 9. Condition 3 in Section V-A requires that, if we can validly map an output datum from an attack model step to that from a process model step, then the attack step's output must not contain a descendant whose type matches the process model step's filter's type. Thus, the addition of the filter of type vid on the step of type isb in the process model will ensure that Condition 3 does not hold. The output datum from the attack step isb can no longer be mapped to the output datum from the process step isb (red cross in Figure 9) because of this prohibitive filter, thereby failing the attack. The filter prevents the ballot output from step isb to contain the secret vid. Thus vid does not get carried downstream, where both the children subdata vid and $vote$ can exist on the ballot, thereby preventing the voter confidentiality attack.

There maybe scenarios where a combination of different improvement opportunities are applied to the same process model to eliminate various attack ways.

IX. Related Work

First we compare DIAS to rest of the literature. Then we explain, how DIAS, with respect to our previous work on which it is based upon, advances our research contributions.

A. Comparison with other literature

1) Holistic perspective: Most security analyses focus on specific parts of a process, rather than taking a holistic view. Kohno et al. [1] have analyzed the source code of voting machines having a significant share in US market, to realize that they do not meet sufficient security standards. Not only are these voting machines vulnerable to insider threats, like vote manipulation, but they are susceptible to simple outsider attacks as well, for example, against attacks over the computer network. Feldman et al. [14] have demonstrated that malicious code can be easily injected in Diebold AccuVote-TS voting machine, to steal votes undetectably, and manipulate machine records and logs. Similarly, [15], [16] have focused on the voting machine technology, which is a specific part of the process. In contrast to all these approaches, we examine the interactions among agents, steps, and data, i.e., all players in a process, rather than analyzing a specific aspect of a process, to identify attacks and improve the process.

2) Process-based attack analysis: Osterweil [17] first applied a formal method of process analysis to software development in order to improve the way in which software was developed. Several other models were developed for various uses such as co-ordination in the workplace, and several languages were developed to enable formal analysis of processes [18], [19], and applied to domains beyond software development, such as health care [20]–[22] and elections. But none of these approaches focus on identification and prevention of malicious attacks in processes. Curtis et al. [18] have studied the human aspect, attributable to a software process. Cass et al. [19] have shown the benefits of using Little-JIL as a graphical, yet formal, process programming language for defining and analyzing processes. Clarke et al. [20] have used Little-JIL to define a blood transfusion medical process, then applied PROPEL [23] system to elucidate desirable properties which must be adhered to by the process, and then used verifiers (SPIN [24], FLAVERS [25], LTSA) to check if the properties are actually respected. The verifier algorithmically checks all possible execution traces through the process model to determine if any execution violates any desired property to be conformed to, by the process. A counterexample trace is generated by the verifier if a property is violated, thereby detecting errors in the medical process. Christov et al. [21] takes a similar approach in identifying errors in a chemotherapy medical process. Simidchieva et al. [3] have used Little-JIL to define a detailed election process model, and then automatically derive a fault-tree to identify combination of failures that may allow a selected potential error to occur. All the errors identified by these aforementioned works in this paragraph, are inadvertent mistakes, committed by the process agents, whether human or automated. On the other hand, DIAS identifies intentional attacks on a process by rogue agents. Thus we see that, process-based attack analysis (as against error analysis) of agents, steps and data in any domain, including elections, is an emerging area of study, where a lot of work still needs to be done. Red-team tests have performed some work in this area, examining systems both individually and in the context of an election process [26]–[28], but the latter is being done informally and non-rigorously. Barr et al. [29] pointed out, the security of elections and the accuracy of their results depend just as much upon the processes and procedures followed as upon the technology used. But they have focused on pointing out the weaknesses of the standards, upon which US election processes and voting machine systems are designed, and process and system certifications are performed, rather than providing a formal attack analysis mechanism, as we do. Weldemariam et al. have examined the security of business processes, and applied that work to elections procedures [30]–[32]. They have discussed a formal methodology for assessing the procedural security of an organization. However unlike, our logic-rule based, attack identification approach, they have first encoded the process asset-flows in terms of executable specifications using a formal language, where there is an allowance for malicious transformation of assets by random execution of one or more threat actions in the model. Then they have specified, the desirable security properties using mathematical formulae, which the process needs to satisfy, and then have performed a security verification, using a model checker, to test if the security properties are fulfilled with respect to the threat scenarios in the model. Similarly Huong et al. [4] have analyzed the security properties of an election process under attack, using model checking. But unlike DIAS, the concept of agents in their analysis is not explicit; they realize it implicitly via the process steps. Also, our logic-rule based approach for determining the criteria for a successful attack is more flexible. By changing our logic rules, we can easily change the definition of a "successful attack", whereas in Huong's approach, the definition of a successful attack via a process not satisfying an "attack-always-fails" property is somewhat rigid. Also, their approach, unlike ours, does not provide a method for automated improvement of the process once the attack has been identified.

B. Advancing our previous work

DIAS is based on, and extends our previous work of insider attack identification and prevention using a declarative approach [33]. The improvements made by DIAS, and their advantages, in relation to our previous work, are as follows:

1) Our extended model supports multi-level, hierarchical, parent-child decomposition of data (missing in our previous work), thereby being able to model and analyze a much wider range of real-world processes.
2) Hierarchical data decomposition enables differential data access modeling capability, which helps to identify more subtle privacy attacks. We can now model and identify scenarios, where rogue process agents utilize their not-so-apparent capabilities to read data across the parent-child hierarchy via the steps, leading to privacy attacks.
3) DIAS is simpler and concise, but, subsumes all the concepts expressed by our previous model. For example, *annotation* was a construct in our previous model, encoding extra information in a datum. Hierarchical data modeling has replaced its need. A child of a datum is now, equivalent to an annotation on a datum. Also with annotations, we could not model multi-level data hierarchy. This is because an 'annotates' edge existed between an annotation and a datum, and annotation on an annotation on a datum, concept, could not be encoded.
4) Flexible step type matching in our extended model allows for more attack identification opportunities. A process model step should allow for at least what an attack model step demands, but can provide for more. Thus if a process step type is a subtype of the attack step type, then the attack step can be realized through the process step. This was disallowed by our previous approach, which needed the attack and process step types to be strictly equal, for a successful attack.
5) DIAS has an enriched semantics, expressed by the fine-grained data dependency declarations. In our previous model, a process step was a "black-box"- it abstracted away the type of operations it performs on the inputs to produce the outputs. Also, our model did not specify which output from a step depends on which particular input. Our current work expresses all those fine-grained data dependencies and their types.
6) DIAS introduces additional process improvement methods like utilizing restrictors, missing in our previous work.

X. Discussion

We present DIAS, a novel, logic rule-based, static analysis approach for automatically determining if an attack can take place on a process and if so, in how many ways this attack can be performed and who are the rogue insider agents involved. Interesting agent collusion, attack scenarios are also demonstrated. Dataflow-based process and attack models are considered, and a holistic perspective is used that looks at steps, data and agents to determine if a process is vulnerable. DIAS also automatically identifies subtle attacks on processes having collection-oriented data models, showing how rogue insiders, can make use of the not-so-apparent, read capabilities of the process steps, to carry out privacy attacks. The problem of attack determination is essentially reduced to a graph matching-based search problem. We have introduced a graph-based language for modeling the processes and attacks. The language has a natural visual syntax which provides a simple understanding of how a process and a possible attack on that process may look like. Then we use a declarative programming paradigm to automatically enumerate the possible ways in which an attack model graph can be matched against a process model graph according to a concept of a valid mapping, encoded as logic rules. Each mapping gives rise to a possible avenue of attack, showing how a process model can act as the supporting framework, via which the attack can be carried out.

Apart from being intuitive in expressing a valid attack mapping concept and being useful in automatically enumerating attack possibilities, our logic rule-based approach is also very amenable to addition of new constraints to change the definition of an attack mapping and hence the meaning of a successful attack. Once attack possibilities are determined, our Java-based implementation automatically and opportunistically searches and exploits improvement opportunities in the process, starting from the highest attacked steps to the lesser attacked ones, to make the process robust against the attack.

DIAS does not automatically generate attacks from a given process model, as achieved by a model checker in current literature [34]. DIAS is complementary to this model checking approach. Once the model checker generates an attack trace that is successful against a process, we can convert that attack trace structure into our format of a data-flow based attack graph. Then, DIAS can be used to check against which other processes, this attack will be successful, and in how many different ways the attack is possible.

As a future work, we want to apply DIAS on process domains, beyond elections, like real estate and medical domains, thereby demonstrating its broader applicability base.

Acknowledgements

We thank Professor Leon J. Osterweil, in the Department of Computer Science, University of Massachusetts, Amherst, for his feedback on improving the semantics of our graph-based process and attack models.

Our work is supported by the National Science Foundation under Grant Number CCF-1219993 and by the US. Department of Commerce, National Institute of Standards and Technology under Grant Number O-60NANB13D165. Any opinions, findings, conclusions and recommendations expressed here are solely of the authors, and do not necessarily reflect the views of the supporting organizations.

References

[1] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach, "Analysis of an electronic voting system," May 2004.

[2] *Security analysis of the Diebold AccuVote-TS voting machine*. Center for Information Technology Policy, Princeton University, 2007.

[3] B. I. Simidchieva, S. J. Engle, M. Clifford, A. C. Jones, S. Peisert, M. Bishop, L. A. Clarke, and L. J. Osterweil, "Modeling and analyzing faults to improve election process robustness," Berkeley, CA, aug 2010.

[4] H. Phan, G. Avrunin, M. Bishop, L. A. Clarke, and L. J. Osterweil, "A systematic process-model-based approach for synthesizing attacks and evaluating them," Berkeley, CA, aug 2012.

[5] V. Lifschitz, "What is answer set programming," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2008.

[6] P. Bonatti, F. Calimeri, N. Leone, and F. Ricca, "Answer set programming," in *A 25-year perspective on logic programming*, 2010.

[7] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. E. Lorensen *et al.*, *Object-oriented modeling and design*, 1991.

[8] M. Stefik and D. G. Bobrow, "Object-oriented programming: Themes and variations," *AI magazine*, 1985.

[9] S. Citrigno, T. Eiter, W. Faber, G. Gottlob, C. Koch, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello, "The dlv system: Model generator and application frontends," 1997.

[10] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The dlv system for knowledge representation and reasoning," 2006.

[11] V. Lifschitz, "Answer set planning," in *Logic Programming and Non-monotonic Reasoning*, 1999.

[12] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry, "An a-prolog decision support system for the space shuttle," 2001.

[13] F. Ricca, "The dlv java wrapper." in *APPIA-GULP-PRODE*. Citeseer, 2003.

[14] A. J. Feldman, J. A. Halderman, and E. W. Felten, "Security analysis of the diebold accuvote-ts voting machine," Berkeley, CA, aug 2007.

[15] E. Proebstel, R. Sean, F. Hsu, J. Cummins, F. Oakley, T. Stanionis, and M. Bishop, "An analysis of the hart intercivic dau eslate," Berkeley, CA, aug 2007.

[16] C. Sturton, S. Jha, S. A. Seshia, and D. Wagner, "On voting machine design for verification and testability," New York, NY, Oct. 2009.

[17] L. Osterweil, "Software processes are software too," in *Proceedings of the 9th International Conference on Software Engineering*, Los Alamitos, CA, 1987.

[18] B. Curtis, M. I. Kellner, and J. Over, "Process modeling," in *cacm*, New York, NY, Sep. 1992.

[19] A. G. Cass, B. S. Lerner, E. K. McCall, L. J. Osterweil, S. M. Sutton Jr., and A. Wise, "Little-jil/juliette: A process definition language and interpreter," Los Alamitos, CA, Jun. 2000.

[20] L. A. Clarke, Y. Chen, G. S. Avrunin, B. Chen, R. Cobleigh, K. Frederick, E. A. Henneman, and L. J. Osterweil, "Process programming to support medical safety: A case study on blood transfusion," May 2005.

[21] S. Christov, B. Chen, G. S. Avrunin, L. A. Clarke, L. J. Osterweil, D. Brown, L. Cassells, and W. Mertens, "Rigorously defining and analyzing medical processes: An experience report," Oct. 2007.

[22] C. Bertolini, Schäf, and V. Stolz, "Towards a formal integrated model of collaborative healthcare workflows," Aug. 2011.

[23] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil, "Propel: an approach supporting property elucidation," 2002.

[24] G. J. Holzmann, "The model checker spin," *IEEE Transactions on software engineering*, 1997.

[25] J. M. Cobleigh, L. A. Clarke, and L. J. Osterweil, "Flavers: A finite state verification technique for software systems," 2002.

[26] RABA Innovative Solution Cell, "Trusted agent report Diebold AccuVote-TS voting system," Columbia, MD, Tech. Rep., Jan 2004.

[27] M. Bishop, "Overview of red team reports," Office of the California Secretary of State, Sacramento, CA, USA, Jul. 2007.

[28] J. L. Brunner, "Project everest: Evaluation and validation of election related equipment, standards and testing," Ohio Secretary of State, Tech. Rep., dec 2007.

[29] E. Barr, M. Bishop, and M. Gondree, "Fixing federal e-voting standards," *cacm*, Mar. 2007.

[30] K. Weldemariam and A. Villafiorita, "A methodology for assessing procedural security: A case study in e-voting," Bonn, Germany, Aug. 2008.

[31] ——, "Procedural security analysis: A methodological approach," in *Journal of Systems and Software*, 2011.

[32] ——, "A formal methodology for procedural security assessment," in *ICDS 2011, The Fifth International Conference on Digital Society*, 2011.

[33] A. Sarkar, S. Köhler, S. Riddle, B. Ludäscher, and M. Bishop, "Insider attack identification and prevention using a declarative approach," 2014.

[34] R. W. Ritchey and P. Ammann, "Using model checking to analyze network vulnerabilities," in *Security and Privacy, 2000.*