2014 IEEE Security and Privacy Workshops

Insider Attack Identification and Prevention Using a Declarative Approach

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

***Abstract*—A process is a collection of steps, carried out using data, by either human or automated agents, to achieve a specific goal. The agents in our process are *insiders*; they have access to different data and annotations on data moving in between the process steps. At various points in a process, they can carry out attacks on privacy and/or security of the process through their interactions with different data and annotations, via the steps which they control. These attacks are sometimes difficult to identify as the rogue steps are hidden among the majority of the usual non-malicious steps of the process. We define process models and attack models as dataflow-based directed graphs. An attack A is successful on a process P if there is a mapping relation from A to P that satisfies a number of conditions. These conditions encode the idea that an attack model needs to have a corresponding similarity match in the process model to be successful. We propose a declarative approach to vulnerability analysis. We encode the match conditions using a set of logic rules that define what a *valid* attack is. Then we implement an approach to generate all possible ways in which agents can carry out a valid attack A on a process P, thus informing the process modeler of vulnerabilities in P. The agents, in addition to acting by themselves, can also collude to carry out an attack. Once A is found to be successful against P, we automatically identify *improvement opportunities* in P and exploit them, eliminating ways in which A can be carried out against it. The identification uses information about which steps in P are most heavily attacked, and try to find improvement opportunities in them first, before moving onto the lesser attacked ones. We then evaluate the improved P to check if our improvement is successful. This cycle of process improvement and evaluation iterates until A is completely thwarted in all possible ways.**

I. Introduction

Real-world processes are often large and complex and determining if an attack can take place on them is quite challenging. Recent works [1], [2] on process vulnerability analysis have focused on the security or privacy aspects of *specific parts* of a process. But a *holistic* vulnerability analysis investigating the interactions among colluding agents, steps, data and annotations on data, has been under-studied.

We propose **D**ata **A**nnotation **S**tep **A**gent **I**nteraction analysis or DASAI to addresses these shortcomings, handling a case in which an attack concerns an interplay among all players in a process, and analyzing if the attack is possible on the whole process. A high level overview of DASAI is presented in Section III. We model the process and the attack as directed graphs having data, annotations, steps, agents and filters (Section IV). We then establish the criteria of a successful attack in the form of a mapping from the nodes in the attack graph to those in the process graph (Section V). The intuition behind this is: an attack model is structurally similar to a process model but with a few different or additional malicious steps. So examining whether this similarity exists reduces to a matching between the components of the two graphs. Another way to look at it is, as if, given a specification of a process model, we try to find out if an attack model's goal can be implemented via the process.

We use logic-based rules to implement both our match conditions and to generate the different possibilities for a valid attack based on those conditions (Section V). These rules generate and test the different possible ways in which the attack graph A is similar to the process graph P according to the match conditions, 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. Note that DASAI also identifies annotation-based attacks where agents pass secret information along the process dataflow in order to achieve some malicious goal. An advantage of using match conditions encoded as logic rules is that, they can be changed easily to alter the semantics of a successful attack.

Once a valid attack is identified, DASAI automatically searches for improvement opportunities in the process, and if found, incorporates them to eliminate the attack (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 robustly improved process model. We then evaluate our improvements and iteratively exploit the improvement opportunities to ensure that the process is indeed made robust against the attack in all possible ways.

Thus, given a set of process and attack models, we can use DASAI to identify 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, using applicable improvement opportunities.

A big advantage of DASAI is that it provides a formal analysis mechanism to identify and remove vulnerabilities from a process statically, without the actual process needing to be carried out. This can help the process domain experts to render their processes robust against a large set of attacks, thereby avoiding lots of time and money associated with after

© 2014, Anandarup Sarkar. Under license to IEEE.
DOI 10.1109/SPW.2014.41

IEEE computer society

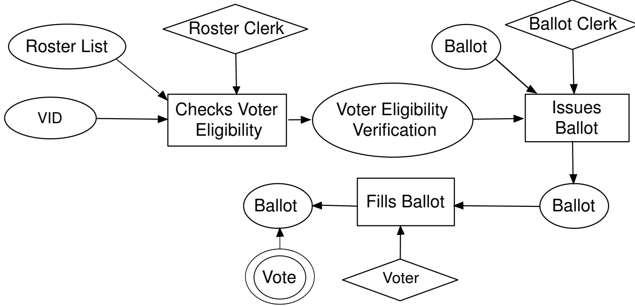

Fig. 1. Voting process model example: A roster clerk checks for a voter's eligibility to vote; if eligible, the voter is given an empty ballot by the ballot clerk on which he fills out his vote.

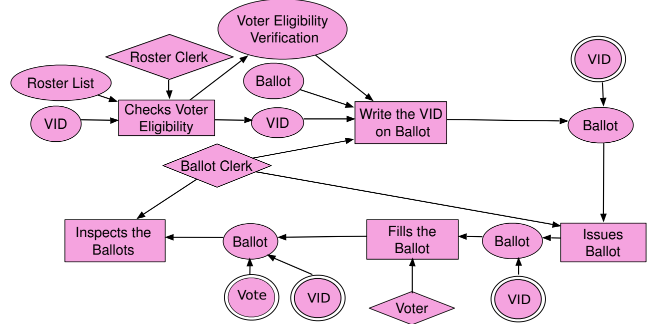

Fig. 2. Voter Confidentiality Attack: A rogue roster clerk passes on the voter's ID to a colluding ballot clerk who puts this secret information on an empty ballot, issued to the unsuspecting voter. Once the voter puts his vote on the ballot, his confidentiality is compromised, since the ballot also contains his ID.

the fact analyses.

To show the generality of our mechanism, we have explained DASAI in terms of abstract process and attack models. To show the practicality of our mechanism, we have used a real world example from the election domain as a motivating use case in Section II; in Section IX, we show the results of our DASAI implementation on this election example. The results show, for each identified successful attack, which agents are involved and whether they need to collude for the attack to succeed. We conclude our paper with related work in Section X, and summary of DASAI and future directions in Section XI.

II. Motivating Example

We have used an election process as a representative use case. Figure 1 shows an "On Election Day" voting process model and Figure 2 shows the model of a *voter confidentiality* attack which can possibly take place on that process. The ovals in these figures represent data or artifacts, rectangles the steps, diamonds the agents, and the double circles denote annotations on the data.

In Figure 1, a voter, modeled as an agent, is checked for his eligibility to vote during the *Check Voter Eligibility* step. The roster clerk agent verifies whether that voter's ID, *VID*, is present in the artifact *Roster list* that is a list of registered voters. If the verification succeeds, the roster clerk tells the ballot clerk to give the voter a ballot of a specific type; *Voter Eligibility Verification* in the figure is an abstract representation of this communication between the roster and ballot clerks. The voter now gets a blank ballot data from the ballot clerk on which he fills out his vote, modeled as an annotation on that ballot.

Now let us consider that insider agents collude to breach voter confidentiality, finding out for whom a voter has voted. Figure 2 shows an example model attack. Once a roster clerk finds out that a voter is eligible to vote, he covertly passes on the *VID* data uniquely identifying that voter to the ballot clerk. The ballot clerk writes that secret information on an empty ballot as an annotation on it and hands it over to the voter. The unsuspecting voter casts his vote on the ballot as usual. Thus we have a ballot data with two annotations on it, the *VID* and the *Vote* thereby breaking the voter's confidentiality.[1] Now given Figure 1 and Figure 2 as a process model and an attack model, can we automatically determine whether this attack can take place on this process? Also, the same attack maybe carried out in several different ways on the process. For example, the attack becomes easier if the roster clerk acts also as the ballot clerk. If the ballot clerk marks the ballot before issuing it to the voter, he would still have to collude with the roster clerk to be sure that the right voter got the ballot with the *VID* associated with that voter. But if the roster clerk is the ballot clerk himself, that problem goes away. So once we determine that an attack can be successful on a process, can we automatically find out in how many ways it can be carried out by which different agents, acting alone or colluding among themselves? These are the principal questions which we solve using DASAI in the following sections.

III. High Level Overview of DASAI

Figure 3 shows a high-level overview of DASAI. A process domain expert first defines a set of process and a set of 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 $A_1, A_2, \dots A_m$ constituting the underlying attack model set. We use Answer Set Programming (ASP) [3], [4], 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 respecting those conditions. We denote this set of enumerated attack scenarios as $M_1, M_2, \dots, M_k$. If this set is empty, then A_i cannot take place on P.

[1] This *VID* can just be a time stamp written on the ballot which later on can be matched up against the roster list as long as the order in which ballots are issued to the voters is also recorded on the roster.

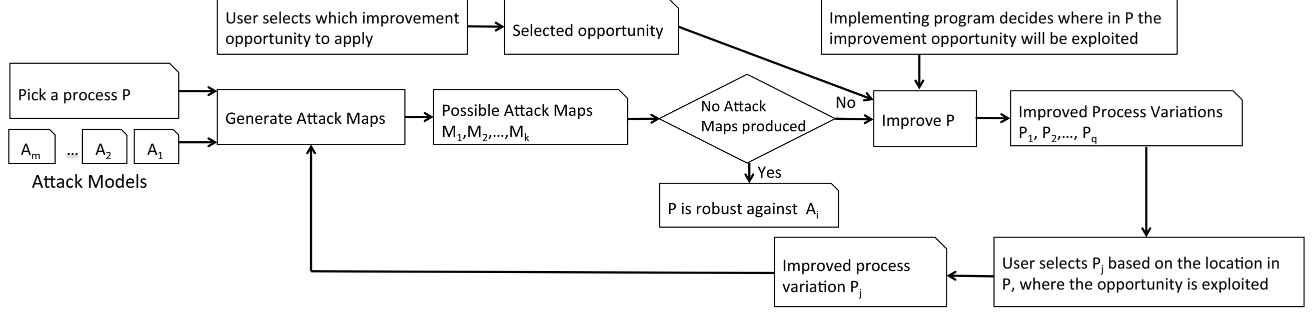

Fig. 3. Approach overview. Rectangles represent activities, rectangles with cut corners denote inputs or resultant outputs, and a diamond denotes a decision box. A user tests a process P for its robustness against an attack A_i through generation of valid attack maps. P is improved, if found to be vulnerable in one or more ways against A_i, and then an iterative process evaluation and improvement cycle ensues, till P becomes completely robust against A_i.

If non-empty, this set of attack scenarios is input to the `Improve` P, which tries to modify P to thwart A_i. The user is given the choice to select 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, then moving on to the lesser attacked ones. Thus it is the implementing program which decides the location in P in which the user selected improvement opportunity will be exploited.

The output from `Improve` P is a set of "improved" process models $P_1, P_2, \ldots, P_q$, produced by exploiting the user selected opportunity, where every set element is a variant of process P with certain avenues of attack 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, thus making it robust against A_i. These variants arise out of the fact that there can be multiple steps in P, 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. In such a scenario, depending on which step in P is actually exploited by `Improve` P, we get a new variant. The user is then again given an option of selecting a variant, say P_j from this improved set, which is then again provided as an input to the `Generate Attack Maps` implementing program. `Generate Attack Maps` again runs the ASP rules for attack determination based out of 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. Once that is ensured, we can reapply DASAI to check the robustness of the next process selected from the set of process models.

IV. Process and Attack Models

In this section, we formally define our process graph, based on which our process model and attack model are defined. A process graph is a directed, acyclic graph $G = (V, E)$ whose nodes $V = \mathsf{S} \cup \mathsf{D} \cup \mathsf{N} \cup \mathsf{Ag} \cup \mathsf{F}$ are *steps* S, *data* D, *annotations* N, *agents* Ag, or *filters* F. A step is the basic unit of task in a process. An *annotation* can be used to encode extra information for data. An *agent*, either human or automated, controls or performs a step in a process. A *filter* is an additional activity associated to a particular step; it restricts a step from producing a certain type of output or a certain type of annotation on the output. We also associate *types* with the nodes in G, i.e., for each kind $K \in \{\mathsf{S}, \mathsf{D}, \mathsf{N}, \mathsf{Ag}, \mathsf{F}\}$ of node in V, the function type: $K \rightarrow \mathbb{T}_K$ associates a type to K-nodes. For example, for a step $s \in \mathsf{S}$, the function type : $\mathsf{S} \rightarrow \mathbb{T}_{\mathsf{S}}$ (i.e., $\mathsf{type}(s)$) will yield a step type, which further describes s. In G, data d_1 with a set of annotations T_1 is a *subtype*($\sqsubseteq$) of data d_2 with a set of annotations T_2, provided $T_2 \subseteq T_1$ and $\mathsf{type}(d_1) = \mathsf{type}(d_2)$.

The edges $E = \mathsf{R} \cup \mathsf{W} \cup \mathsf{C} \cup \mathsf{T} \cup \mathsf{X}$ are as follows: $\mathsf{R} \subseteq \mathsf{D} \times \mathsf{S}$ 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. $\mathsf{W} \subseteq \mathsf{S} \times \mathsf{D}$ is the set of *write* edges signifying that steps produce data. When a step is successfully performed, it produces all the data to which it is connected via write edges. $\mathsf{C} \subseteq \mathsf{Ag} \times \mathsf{S}$ is the set of *control* edges signifying that agents control or perform steps. $\mathsf{T} \subseteq \mathsf{N} \times \mathsf{D}$ is the set of *annotate* edges denoting that annotators add information to data. $\mathsf{X} \subseteq \mathsf{F} \times \mathsf{S}$ is the set of *filter* edges denoting that filters remove certain types of data, thus preventing them from appearing as outputs, or annotations on the outputs from the steps with which the filters are associated. A *path* is a sequence of nodes $v_1, v_2, \ldots, v_n$ such that $(v_i, v_{i+1}) \in E$.

Example. Figure 4 shows an example process graph. The distinct shapes of the nodes uniquely identify their kinds like steps, data etc. and the names of the nodes identify their types, for example ballot is a data type in the process.

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

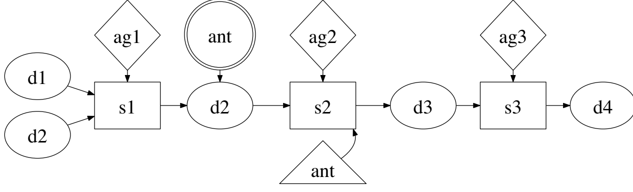

Fig. 4. Process graph, consisting of *steps* (rectangles), *data* (ovals), *annotations* (doubled circles), *agents* (diamonds), and *filters* (triangles). Nodes are labeled with their corresponding *types*. Edge types are: $data \rightarrow step$ (read), $step \rightarrow data$ (write), $agent \rightarrow step$ (control), $annotation \rightarrow data$ (annotate), and $filter \rightarrow step$ (filter).

graph $G_A = (V_A, E_A)$ where $\mathsf{F}_A = \emptyset$ and $\mathsf{X}_A = \emptyset$, since an attack model does not contain any construct as filter, and correspondingly the *filter* edges.

V. Attack Mappings

We now describe what constitutes a *valid* attack, and determine in how many possible ways an attack can take place on a process. With the underlying intuitions behind these concepts, we also present a few selected accompanying formalisms and implementations of the same.

A. Map Conditions

For an attack A to be successful on a process P, we test if the process model P provides us with the right steps and data required 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 structural 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., $M = M_{\mathsf{S}} \cup M_{\mathsf{D}} \cup M_\alpha \cup M_\omega \cup M_{\mathsf{Ag}}$. We only consider well-formed mappings in this paper.

Here, M_{S}: $\mathsf{S}_A \rightarrow \mathsf{S}_P$ maps attack model steps to process model steps, $M_{\mathsf{D}} \subseteq \mathsf{D}_A \times \mathsf{D}_P$ relates attack data with process data, $M_\alpha, M_\omega \subseteq \mathsf{D}_A \times \mathsf{D}_P$ also relate attack data and process data, but are used to identify the beginning and end of a *sequence mapping*, respectively. Finally, $M_{\mathsf{Ag}} \subseteq \mathsf{Ag}_A \times \mathsf{Ag}_P$ relates attack agents with process agents.

A well-formed M is said to be a *valid* M if all the following mapping conditions are satisfied. These conditions define precisely when an attack A is “similar enough” to a process model P to be considered successful against P.

Condition 1: Mapping All Nodes. M relates all attack data D_A, steps S_A, and agents Ag_A nodes to corresponding nodes D_P, S_P, and Ag_P in the process model.

Condition 1a: All Steps Match. In a successful attack, all steps are carried out. Thus all steps in S_A are mapped to some steps in S_P (unless they are part of an attack *sequence map*):

For all $s_a \in \mathsf{S}_A$ there is a $s_p \in \mathsf{S}_P$ such that $M_{\mathsf{S}}(s_a, s_p)$, or else s_a is part of an attack sequence map (Condition 5).

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

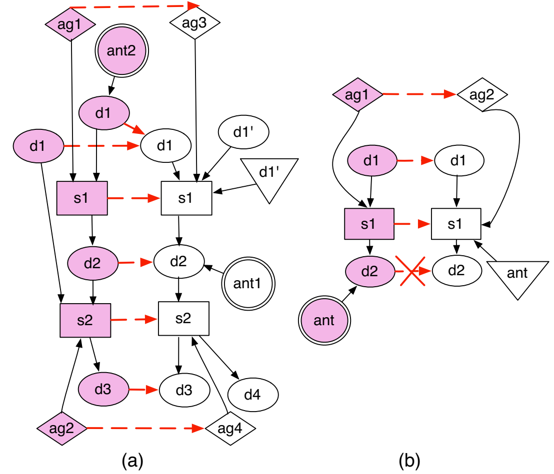

Fig. 5. (a) Attack map: the pink attack model A is mapped to the white process model P. Steps in A are mapped to steps in P with matching types, shown by dashed (red), *attack map* edges. Data nodes in A are mapped to type matching data nodes in P. The attack is *valid* since all steps, data and agents from A are mapped. (b) The filter (triangle) prevents the annotation of type `ant`, to be produced on the output data of type $d2$, thus thwarting an otherwise successful attack.

Condition 1b: All Data Match. In a successful attack, all input data and all output data of attack steps must be mapped to some data in D_P (unless they are part of a sequence map): For all $d_a \in \mathsf{D}_A$ there is a $d_p \in \mathsf{D}_P$ such that $M_{\mathsf{D}}(d_a, d_p)$, or else d_a is part of an attack sequence map (Condition 5).

Condition 1c: All Agents Match. All agents in the attack model must be mapped to agents in the process model (unless they are part of a sequence map): For all $ag_a \in \mathsf{Ag}_A$ there is a $ag_p \in \mathsf{Ag}_P$ such that $M_{\mathsf{Ag}}(ag_a, ag_p)$, or else ag_a is part of an attack sequence map (Condition 5).

Condition 2: Steps Match. A step type represents the kind of action needed at a particular point in a process; only if the process model provides a matching step type, can the attack succeed: For all $M_{\mathsf{S}}(s_a, s_p) : \mathsf{type}(s_p) = \mathsf{type}(s_a)$.

Figure 5(a) shows an example illustrating our map conditions. In this subfigure, all the steps in the attack model A (the pink thread) have their corresponding matching types in the process model P (the white thread).

Condition 3: Inputs Match. An attack model step may need to read certain inputs to be successful. Thus, we require the matching process step to provide all these matching inputs. Thus, for all $M_{\mathsf{D}}(d_a, d_p) : \mathsf{type}(d_p) \sqsubseteq \mathsf{type}(d_a)$. Datatypes match when the types are equal or one datatype is a subtype of another.

Figure 5(a) shows that all of the input data to any of A's step have their corresponding matches in P as shown by the dashed red edges.

Note that M_{D} preserves *read* edges, and thereby dataflow. For example, in Figure 5(a) , in attack model A, given that the step of type $s1$ reads data of type $d1$, and, this (step,data) pair is mapped to their corresponding counterparts in P, for

a valid attack to take place, it must be the case that in P, the step of type $s1$ reads data of type $d1$. The attack model data of type $d1$ cannot be mapped to matching data, which is written by the step of type $s1$ in P, or to any downstream data which appears after the step of type $s1$ in P.

A process model step can meet an attack requirement by reading an indirect input via an upstream step occurring before it in the timeline. This is because we assume that data is never destroyed in our model and any data read by a process step is available indirectly as an implicit input to all downstream steps. Figure 5(a) shows an example where P's step of type $s2$ reads upstream data of type $d1$, which is the target of map from the attack model.

A filter should not block the transitive availability of an input, say d_p, to a process step s_p, for a valid attack. 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. Formally: if $M_{\mathsf{D}}(d_a, d_p)$ and $M_{\mathsf{S}}(s_a, s_p)$ 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., $\mathsf{type}(f) \neq \mathsf{type}(d_p)$. $\mathsf{in}(s_a)$ 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.

Condition 4: Outputs Match. When an attack model step is successfully performed, it produces certain data. So the corresponding process step, should also produce all the matching output data. Thus, if $M_S(s_a, s_p)$, 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 $d'_p \in \mathsf{out}(s_p)$. Figure 5(a) shows this output signature match condition; all of the output data from any of A's step have their corresponding matches in P.

Also, for a valid attack, there should not be any filter, associated with process step s_p, checking for a data of type of d'_a, or of the type of annotation on d'_a. Otherwise due to this filter, process step s_p cannot produce an output or an annotation on the output, as demanded by the attack, thereby failing the attack.

Figure 5(b) shows such an invalid attack mapping denoted by the crossed dashed red edge, in which the filter restricts the output from A's step of type $s1$ from being mapped to its counterpart in P because of the matching annotation of type ant. Formally: If $M_{\mathsf{S}}(s_a, s_p)$ and $M_{\mathsf{D}}(d'_a, d'_p)$ and $d'_a \in \mathsf{out}(s_a)$, then $d'_p \in \mathsf{out}(s_p)$ and for all annotations n'_a of d'_a and all filters f of s_p, f neither matches n'_a nor d'_a, i.e., $\mathsf{type}(f) \neq \mathsf{type}(n'_a)$ and $\mathsf{type}(f) \neq \mathsf{type}(d'_a)$.

Condition 5: Sequence Mapping. Sometimes a sequence of attack steps $s_a^1, \ldots, s_a^n$ might be achieved by a malicious attacker using a single step s_p of the process model. We consider this possible if at least one of the steps s_a^i in the sequence matches s_p. We also require that the inputs and outputs of the attack sequence match those of s_p. 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: $d_a, s_a^1, \ldots, s_a^n, d'_a$ in A there is a matching step $s_a^i \in \pi_a$ with $\mathsf{type}(s_a^i) = \mathsf{type}(s_p)$.

Attack steps in π_a, including their inputs, outputs and controling agents are part of a sequence map and assumed to be carried out via s_p. Thus they are exempt from being explicitly mapped via $M_{\mathsf{S}}, M_{\mathsf{D}}$, and M_{Ag} (Condition 1).

Condition 6: Non-Collusive Agents match trivially. If $M_{\mathsf{Ag}}(ag_a, ag_p)$, i.e., an attack agent ag_a is mapped to a process agent ag_p, and ag_a controls step s_a and agent ag_p controls step s_p, then, we require that, $M_{\mathsf{S}}(s_a, s_p)$. Also, the inputs and outputs of s_a should match the inputs and outputs of s_p respectively.

We assume that any agent in the process can be made rogue; so whenever an attack model step requires an agent to perform it, the process model can always provide one and it may not have the same signature as that of the attack model agent type, but still be capable of carrying out the required attack model 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. 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 [5], [6], a state-of-the-art implementation of ASP [4], [7], [8], [9], to implement our valid attack map conditions (in Section V-A). In the interest of space, we have included only a portion of the entire implementation as a representative. The program source code is available online. [3]

We encode the constructs in the process model and attack model like step, data, agent, filter 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

[3] https://code.google.com/p/agent-artifact-analysis/

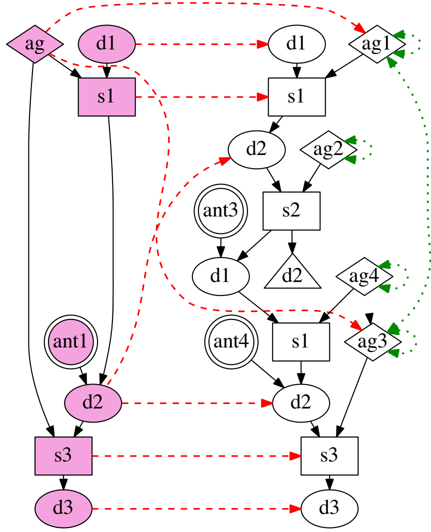

Fig. 6. `Generate Attack Maps` in Figure 3 automatically outputs valid attack map between A and P. Dotted (green) edges show *collusion* between agents.

process model step s *reads* data d i.e., (d,s) $\in$ R_P. Attack model facts are similar, but they are prefixed with am.

Next, we encode, what constitutes a valid attack on a process, implementing mapping Conditions 2, 3 and 4.

```
map(A1,S1,A2,A3,S2,A4):-
                am_steptype(S1,X),
                pm_steptype(S2,X),
                allInMap(A1,S1,A3,S2),
                allOutMap(S1,A2,S2,A4),
                not filter_restricts(S1,S2).
```

The above DLV rule implements the criteria for an (input data, step, output data) triple in an attack model i.e., (A1,S1,A2) to be validly mapped to its counterpart, in this case, (A3,S2,A4) in the process model. If an attack model step S1 has the same step type X as that of a process model step S2, and the datatypes of all the data input to and output from S1 match the datatypes of at least some data input to, and output from S2 (modeled by predicates allInMap and allOutMap, respectively), then we can claim that S1, along with its input and output data can be mapped to S2 and its input and output data respectively, signifying that an attack step can be successfully realized via a process step (modeled by the map predicate). The last conjunct in the above rule ensures that there is no restrictive filter on the process model step S2 which can prevent its output A4 from being the target of the map (referring to the filter restriction requirement in Condition 4). Thus we can claim that an attack is successful on a process if all steps in an attack model along with their input and output data show up as a member of the map predicate in our program's output.

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 semantics, based on stable model semantics [10], which is amenable to computationally difficult (e.g., NP-hard) search problems.

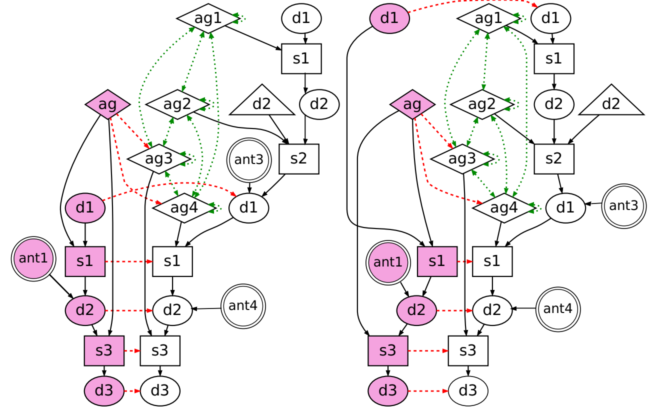

Fig. 7. `Generate Attack Maps` in Figure 3 outputs two other ways (left and right) via which attack A can succeed on process P.

Let us consider an abstract attack model A (pink thread) and an abstract process model P (white), as shown in Figure 6, where we try to determine if A is a valid attack on P, and if so, in which ways.

```
inmap(A1,S1,A3,A2,S2,A4) v outmap(A1,S1,A3,A2,S2,A4)
     :- map(A1,S1,A3,A2,S2,A4).
```

Using the above rule, DLV generates all attack maps or stable models in which valid attack conditions are satisfied. Each attack map corresponds to a way in which an attack can be carried out on a process. Each such map contains inmap atoms denoting that certain entities in the attack model are mapped to those in the process model, or outmap atoms denoting that a pair of entities is not mapped.

To illustrate the point, let us consider Figure 6, in which we input the attack and process models, along with our valid attack conditions and rules used for visualization, to the DLV answer set solver implementing `Generate Attack Maps`. The solver generates a set of stable models or attack maps, each showing a way in which all the attack model steps, data 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 by the visualization logic. Using DLVWrapper [11], a Java interface for the DLV system, we have implemented a Java-based method to collect these node and edge atoms and construct a graph out of it in *dot* format, as shown in Figure 6 and Figure 7.

Figure 6 and each of the subfigures in Figure 7 shows an attack map by which the attack A can be carried out on the process P. In the attack map in Figure 6, DLV chooses to map

the step of type $s1$ in A to the first occurrence of the step of type $s1$ in P as shown by the dashed red edges. These mapped steps are part of the `inmap` predicate generated in this attack map by DLV. But note that the attack model step of type $s1$ could also have been mapped to the second occurrence of the step of type $s1$ in P. DLV chooses *not* to map this pair and makes it part of the `outmap` predicate in this attack map. The reverse scenario is true in a different model, as shown in the subfigures in Figure 7. `Generate Attack Maps` implementation outputs 15 possible models out of which we have shown 3 of them in Figure 6 and Figure 7. In 5 attack maps, the first occurrence of the step of type $s1$ in P is the target of map, whereas in the remaining 10 attack maps the second occurrence is the target of map.

Note that Figure 7 shows multiple possible mapping scenarios for the data too. Thus, in the attack map on the left, DLV maps the attack model data of type $d1$, an input to the step of type $s1$, to the process model data of type $d1$, which is a *direct* input to the step of type $s1$ (second occurrence). The right subfigure shows the alternative mapping, in which attack model step $s1$ with its direct input $d1$ is mapped to the second occurrence of the step $s1$ in the process model which reads data $d1$ *indirectly* via some upstream steps.

Thus, utilizing the power of the answer set solver DLV, we can implement the generation of all possible ways in which an attack can take place on a process.

VII. Agent Collusion Scenarios

The mapping of the agents as described in Condition 6 in Section V-A becomes non-trivial when agent collusion scenarios are considered. Consider Figure 6, in which the same agent of type ag in the attack model A needs to carry out two distinct steps of types $s1$ and $s3$. This can be because the agent needs to utilize the information from one step, in another step. The process model P can meet this requirement in two ways: either the same agent carries out both of the corresponding steps of types $s1$ and $s3$, or two distinct agents carry out this pair of steps separately and are allowed to communicate or collude. `Generate Attack Maps` implementation first enumerates all possible combinations of pairs of agents participating in a process model, and consider this, as the pairs that *can collude* in the process. Then an integrity constraint is used to eliminate the attack maps in which the pairs of agents who are required to collude in the process to meet the attack model requirement, *can not collude*, as modeled by the lack of green edges between the corresponding agent pair.

Thus, the 15 possible ways of attack as produced as an output from `Generate Attack Maps` implementation in Section VI, can be grouped into 5 distinct categories. Each category is identified by a different combination of pairs of agents in P who can collude among themselves. In category one, all agents in P can pairwise collude among each other; in category two, agents of types $ag1$, $ag3$ and $ag4$ can pairwise collude among them; in category three, agents of types $ag1$, $ag2$ and $ag3$ can pairwise collude among them; in category four, agents of types ($ag1$ and $ag3$) and ($ag2$ and $ag4$) can pairwise collude among them and in category five, agents of types $ag1$ and $ag3$ can collude between them. [4] The other possible combinations of agent pair collusions do not become a part of our output of 15 attack maps, since they do not satisfy the criterion of meeting the attack model requirement and hence, eliminated by the integrity constraint as mentioned in the previous paragraph. The attack model A demands that the same agent controls the steps of types $s1$ and $s3$, hence in all of these 15 attack maps, the pair of agents controlling either the first occurrence of the step of type $s1$ and $s3$, or the second occurrence of the step of type $s1$ and $s3$ in P should be able to pairwise collude between them. For example, Figure 6 shows an attack map in which A can be carried out via P respecting the integrity constraint. The dashed red edges between the agent of type ag in A and each of the pair of P's agents of types $ag1$ and $ag3$ indicate that, either this pair of agents controlling the first occurrence of the step of type $s1$ and $s3$ respectively in P *needs to collude*, or this pair should be the same individual for the attack to be successful. The dotted green edges between the $ag1/ag3$ agent pair in P denote that they *can collude*; hence, the attack is successful. In both attack maps in Figure 7, each agent in P can pairwise collude with all other agents. This means that an attack is possible, since collusion is only required between the agents controlling steps of types $s1$ (second occurrence) and $s3$ in P ($ag3/ag4$) for the attack to be successful. Note that the collusion relation is commutative on the elements of Ag_P and Ag_A, so the green edges run in both directions between any colluding agent pair.

VIII. Process Model Improvement

Once `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. 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 steps, agents, input or output data from the steps, and annotations on the output data from the steps in the process model are deleted or updated in their types while the process is being improved.

Once improved, the resulting process model is again provided as an input to `Generate Attack Map`, as shown in Figure 3, to confirm that the process has been indeed made robust against the concerned attack in all possible ways. It may take multiple improvement iterations before this goal is achieved.

We have described how to optimize our `Improve` P implementation to ensure that the iterations eliminating a

[4]Note that each of these categories consists of three members; in the first one, the first occurrence of the step of type $s1$ in P is the target of attack map, in the second and third one, the second occurrence of the step of type $s1$ is the target of attack map. The second and the third member are distinct from each other by the different possible data mapping scenarios, as discussed in Section VI.

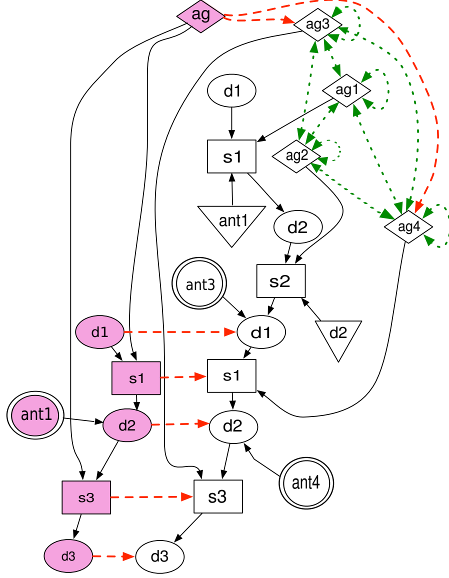

Fig. 8. The intermediate output from `Improve` P run, shows how in P, a filter of type $ant1$ (denoted by the triangle) is added to the first occurrence of the step of type $s1$, thereby preventing its output from being the destination of map from its counterpart in A. Thus attack shown in Figure 6 is now, no longer possible.

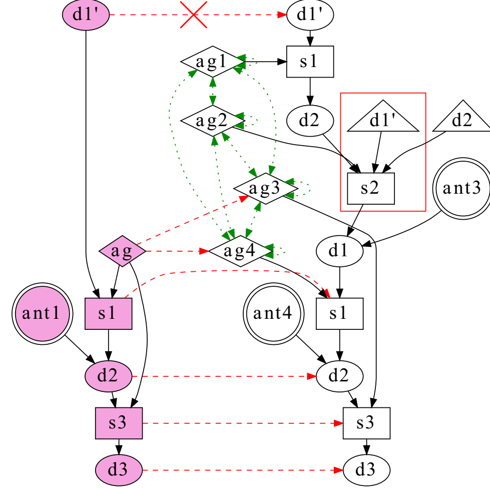

Fig. 9. Another process improvement opportunity where an additional filter of type $d1'$ on the step of type $s2$ in P removes any data of type $d1'$ from the datastream; thus the step of type $s1$ in A can no longer utilize any upstream data of type $d1'$ in P as its input data map target, thereby failing the attack (indicated by the red cross between the data of type $d1'$ in A and P).

larger number of ways of attack, are carried out before the iterations which prevent fewer violations.

There can be many different ways to improve a process, out of which we have implemented two ways, including optimized improvement for one of the ways.

A. Process Improvement (Filter influencing map on the output)

Sometimes an attack model contains a step that writes annotated data, whereas its counterpart in the process model does not contain any matching annotation on the corresponding output data. For example, in in the attack model A of Figure 6, the step of type $s1$ writes data of type $d2$ with an annotation of type $ant1$, whereas the output data of type $d2$ from the first occurring step of type $s1$ in the process model P does not contain an annotation of type $ant1$.

In such a scenario, our `Improve` P implementation prevents the attack by automatically adding a filter of type $ant1$ to the step of type $s1$ in P, as shown in Figure 8. This is because Condition 4 in Section V-A requires that, if we can validly map an output data from an attack model step to that from a process model step, then the attack step's output must not contain an annotation whose type matches the process model step's filter's type. Thus, the addition of the filter of type $ant1$ on P's step of type $s1$ will ensure that Condition 4 does not hold. The output data from the attack step of type $s1$ can no longer be mapped to the output data from the process step of type $s1$, thereby rendering the attack a failure.

But as Figure 8 shows, the step of type $s1$ in A can still be validly mapped to the second occurrence of the step of type $s1$ in P. Once `Generate Attack Maps` iteratively identifies this remaining attack avenue, `Improve` P automatically adds an additional filter to the second occurrence of the step of type $s1$ in P, thereby preventing A from occurring in any possible way. The outputs of step $s1$ in A can no longer be mapped to either step of type $s1$ in P and thus P is made fully robust.

B. Process Improvement (Filter influencing map on the input)

The addition of a filter can prevent the map from an input data of an attack step, to an input data of a process step. Let us consider a slightly modified scenario from the right subfigure of Figure 7, where both the first occurrence of step $s1$ in P and $s1$ in A read data of type $d1'$ directly. In this scenario as shown in Figure 9, the input to step $s1$ in A does not have the same immediate input to the second occurrence of step $s1$ as its map target, but it can utilize one of the upstream inputs of type $d1'$, wich is read indirectly by the second occurrence of the step of type $s1$ in P via the upstream steps, as its target. Thus, we see that the availability of data of type $d1'$ as an indirect input to the second occurrence of step $s1$ in P is a threat; `Improve` P exploits this improvement opportunity, adding in P a filter of type $d1'$ on the step of type $s2$ (shown in the red box in Figure 9), immediately preceding the second occurrence of the step of type $s1$. This filter prevents data of type $d1'$ from appearing as an output from the step of type $s2$. Thus, the second occurance of step $s1$ in P cannot read the data of type $d1'$, thereby failing the attack.

C. Optimized Process Improvement

The order in which filters are added to different occurrences of steps in P can affect the number of attack maps being eliminated in each iteration. Our goal is to improve P through successive rounds, always choosing the improvement

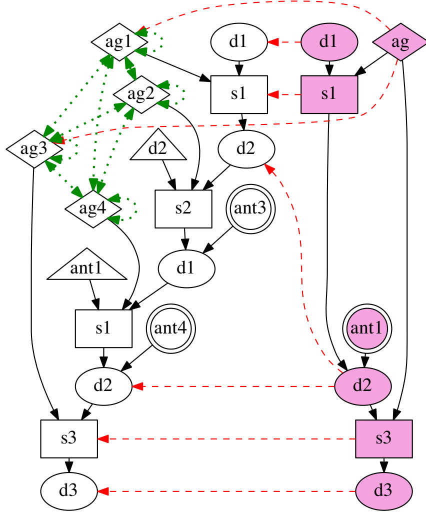

Fig. 10. This is the optimized `Improve` P run's intermediate output; a filter of type $ant1$ is added to the second occurrence (rather than the first) of the step of type $s1$ in P, thereby preventing its output from being the destination of map from its counterpart in A. Thus, attacks shown in Figure 7 are now, no longer possible. The optimized version eliminates 10 possible ways of attack in the first improvement round only, as compared to eliminating 5 ways in the unoptimized version as discussed in Figure 8.

that eliminates a higher number of attack maps first. The motivation behind this is to provide the user of our system with a *more improved* process model in fewer improvement rounds.

In our running example in Figure 10, the second occurrence of the step $s1$ in P is more frequently attacked, occurring as the target of an attack map in 10 possible ways. Second most frequently attacked is the first occurrence, occurring as the target of the attack map in 5 possible ways. `Improve` P looks for improvement opportunities starting from the second occurrence of $s1$, and then proceeds to the first occurance of $s1$. Thus, in the first round of improvement, `Improve` P adds a filter of type $ant1$ to the second occurance of step $s1$, thereby eliminating 10 possible avenues of attack against it. In the next improvement round, the first occurrence of the step $s1$ in P is addressed, eliminating the rest of the possible ways of attack.

`Improve` P is implemented in Java. Separate Java based methods are defined for scanning of process improvement candidates from most heavily to least heavily attacked, determining if the improvement opportunities exist in them, and exploiting them iteratively.

There are scenarios where a combination of different improvement opportunities are applied to the same process model to eliminate various attack ways. Also there may be scenarios, where a process model cannot be improved completely, since the remaining attack maps cannot be eliminated by any identified improvement opportunity. Alternatively, a user might be satisfied with a partially improved process and terminate the improvemnt process prematurely.

IX. Evaluation

Running the `Generate Attack Maps` (in Figure 3) implementation on the motivating example of Section II generates 80 attack maps, each showing a way in which voter confidentiality on election day can be breached. In the interest of space, we have shown only two of the interesting ones among them, along with the identification of the rogue insiders responsible for the attack, in Figure 11(a) and Figure 12 with the abbreviated node names. The 80 attack maps can be classified into 6 categories:

1. isb is the target of the attack map in the process model with the following as the source in the attack model:
   a) The sequence from the steps cve to isb
   b) The sequence from the steps wvb to isb
   c) The sequence from the steps cve to $fillb$
   d) The sequence from the steps wvb to $fillb$
2. $fillb$ is the target of the attack map in the process model with the following as the source in the attack model:
   a) The sequence from the steps cve to $fillb$
   b) The sequence from the steps wvb to $fillb$

Figure 11(a) shows an example for category 1.a) where the sequence in the attack model starting at step cve and ending at step isb is mapped to step isb in the process model thereby satisfying Condition 5 in Section V-A. In our implementation, an attack sequence enclosure is visualized through a mapping (blue edges) of the start (cve) and the end step (isb) of the sequence, in addition to the mapping M_α and M_ω. Note that the regular attack step mappings which are not part of the sequence (M_S) are shown in red, for example the mapping of step $fillb$.

The interpretation of these categories requires knowledge of the process semantics as well as domain knowledge and is left to the user. For example, the first category can be interpreted as if the entire malicious goal of writing the vid on the ballot (wvb) along with the non-rogue steps of checking the voter eligibility (cve) and issuing the ballot to the voter (isb) is carried out by the ballot clerk alone. This scenario is indicated by the pair of blue mapping edges from the cve-isb sequence in the attack model to the step of type isb in the process model. The roster clerk and the ballot clerk then will be the same agents as shown in Figure 11(a) by the mapping edges from rcl and bcl in the attack model to bcl in the process model. After the ballot clerk issues the ballot, the voter fills it out (as shown by the red mapping edges between the corresponding steps of type $fillb$ in the attack and process model). Now, the same ballot contains both the vid and $vote$ thereby breaching voter confidentiality. A self loop (green edge) on a process agent in this case, indicates that the agent does not collude with any other agent.

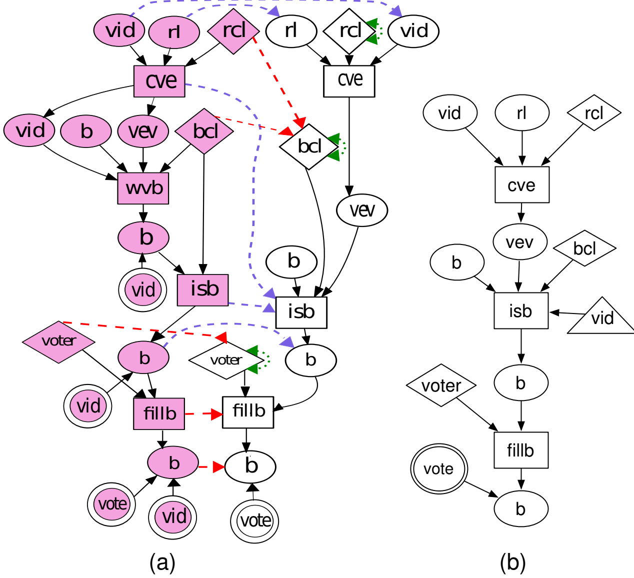

Fig. 11. (a) This shows the output of `Generate Attack Maps` (in Figure 3) run on the motivating example of Section II. It demonstrates a possible way how the voter confidentiality attack can be carried out by the ballot clerk alone. (b) shows the voting process improvement to prevent the attack in (a).

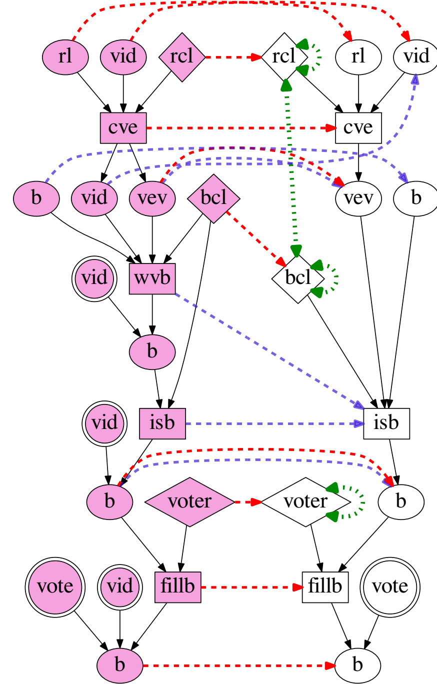

Fig. 12. This shows the output of `Generate Attack Maps` (in Figure 3) run on the motivating example of Section II, demonstrating another possible way how the voter confidentiality can be breached when the roster clerk and the ballot clerk collude.

Similarly, as shown in Figure 12, consider the second category 1.b) which can be interpreted as the roster clerk and the ballot clerk colluding to carry out the attack. The step cve in the attack model is mapped to its counterpart in the process model (shown by the red mapping edge), while the sequence of step types $wvb, ..., isb$ in the attack model is mapped to the single step of type isb in the process model via a pair of blue edges. Also there are mapping edges, from the roster and ballot agents of type rcl and bcl in the attack model to its counterpart in the process model. All these can imply that the roster clerk performs the usual step of checking the voter eligibility (cve) and the ballot clerk performs the malicious step of writing the vid on the ballot (wvb) along with the non-rogue step of issuing the ballot to the voter (isb). The dotted green edges between the agents of type rcl and bcl in the process model signify that the roster and the ballot clerk can collude. The roster clerk can pass on the secret data of type vid (uniquely identifying the voter) to the ballot clerk (modeled by the $write$ and $read$ of data of type vid by the steps cve and wvb respectively in the attack model). In step wvb, the ballot clerk now writes this data as an annotation on the ballot of type b and issues it to the voter. The voter fills out the ballot, thereby breaching the voter confidentiality since now the same ballot contains both the annotations of type vid and $vote$.

Voting Process Improved. Figure 11(b) shows the output of `Improve` P (in Figure 3). The output of the step isb in the attack model has an annotation of type vid which does not exist on the output from the corresponding step isb in the process model. The implementation of `Improve` P automatically adds a filter of type vid on step isb in the process model as shown in Figure 11(b). This filter prevents the ballot output from step isb to contain the secret information of type vid. The ballot clerk is unable to issue any ballot to a voter with the secret information of type vid on it. Thus vid does not get carried downstream and at no point in the process both the annotations of type vid and $vote$ exist on the ballot, thereby preventing this type of voter confidentiality attack. To prevent categories of attacks where the step of type $fillb$ is the target of map in the process model, a similar filter of type vid gets added to the process step $fillb$. Once the filter of type vid is added to both steps isb and $fillb$ in the process model, all 6 categories of possible ways of attacks are prevented, and the process model becomes robust against the voter confidentiality attack in all possible ways described by the attack model. Note that out of 80 possible ways of attack, there are 20 possible ways in which step isb is not the target of attack map in the process model and there are 20 other possible ways where step $fillb$ is not the target of attack map. Since these numbers are equal, there is no clear majority as to which is the most heavily attacked step in the process model; in such a scenario, `Improve` P randomly picks one of the candidate steps for improving the process model.

X. Related work

Process-based security analysis of agents and data in domains like elections is an emerging area of study. Security analyses of elections have focused on the technology used, such as electronic voting machines [12], [13], [14], [15], [16], [17], or on the cryptographic protocols proposed and used for election systems [18], [19], [20]. Red-team tests have examined systems both individually and in the context of an election process [21], [22], [23] the latter being done informally and non-rigorously.

As Barr et al. [24] pointed out, the security of elections and the accuracy of their results depends just as much upon the processes and procedures followed as upon the technology used. Weldemariam et al. have examined the security of business processes, and applied that work to elections procedures [25], [26]. Others have used a formal process modeling language called Little-JIL to represent an election process, used that as a basis for calculating what steps must fail for the process to fail, and introducing compensating steps ("exception handlers") to minimize the chances of failure [27], [28], [29].

The latter work has verified security properties of an election process by considering specific combinations of agent behaviors. Unlike theirs, our work considers detailed scenarios in which agents collude. We also take into account data annotation-based attacks. Huong et al. [30] use the above methods to analyze security properties of an election process under attack using model checking. Unlike our approach, the concept of agents in their analysis is not explicit; they realize it implicitly via steps in the process. Also, our logic-rule based approach for determining the mapping criteria for a successful attack is more flexible. By changing our logic rules, we can 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. Another advantage of DASAI in comparison to their approach is that, it provides a method for automated improvement of the process once the attack has been identified.

XI. Discussion

Our paper presents DASAI, a logic rule-based static analysis approach for determining if an attack can take place on a process. If an attack is found to be possible, DASAI also determines in which ways this attack can be performed on the process and who are the rogue insiders involved. Dataflow-based process and attack models are considered, and a holistic perspective is used that looks at steps, data, annotations on data and controlling agents to determine if a process is vulnerable to an attack. The problem of attack determination is esentially reduced to a graph matching-based search problem. We use a declarative programming paradigm to automatically enumerate the possible ways in which an attack graph can be matched against a process graph according to a concept of a valid mapping encoded as logic rules. Each mapping gives rise to a possible avenue of attack. 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 mostly attacked steps to the lesser attacked ones, to make the process robust against the attack in all possible ways.

Note that DASAI does not automatically generate the attack models from a given process model. Current literature [31] describes the use of model checking to automatically generate attack models as they do not conform to a desired property of a process. Once the model checker identifies an attack model that is successful against a process, we can convert that attack model structure into our format of a data-flow based attack graph. Then, DASAI can be used to check against which other processes this attack will be successful and in how many different ways the attack is possible. Finally, DASAI can be used to improve the set of vulnerable processes, making them robust against this attack.

Currently, we have used DASAI on abstract scenarios and various use cases from the election domain. As part of our future work, we would like to test our rogue insider identification and process improvement program on various use cases from other process domains. For example, we plan to identify an attacker carrying out a real estate fraud, thereby demonstrating the broader applicability base of DASAI.

References

[1] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach, "Analysis of an electronic voting system," in *IEEE Symposium on Security & Privacy*, 2004.

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

[3] V. Lifschitz, "What is answer set programming," in *Proceedings of the AAAI Conference on Artificial Intelligence*, 2008, pp. 1594–1597.

[4] P. Bonatti, F. Calimeri, N. Leone, and F. Ricca, "Answer set programming," in *A 25-year perspective on logic programming*. Springer-Verlag, 2010, pp. 159–182.

[5] 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," in *Proceedings of the 12th Workshop on Logic Programming*, 1997, pp. 128–137.

[6] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello, "The dlv system for knowledge representation and reasoning," *ACM Transactions on Computational Logic (TOCL)*, vol. 7, no. 3, pp. 499–562, 2006.

[7] M. Gelfond and V. Lifschitz, "The stable model semantics for logic programming," in *Proceedings of the 5th International Conference on Logic programming*, vol. 161, 1988.

[8] ——, "Classical negation in logic programs and disjunctive databases," *New Generation Comput.*, vol. 9, no. 3/4, pp. 365–386, 1991.

[9] V. Lifschitz, "Answer set planning," in *Logic Programming and Nonmonotonic Reasoning*. Springer, 1999, pp. 373–374.

[10] M. Nogueira, M. Balduccini, M. Gelfond, R. Watson, and M. Barry, "An a-prolog decision support system for the space shuttle," in *Practical Aspects of Declarative Languages*. Springer, 2001, pp. 169–183.

[11] F. Ricca, "The dlv java wrapper." in *APPIA-GULP-PRODE*. Citeseer, 2003, pp. 263–274.

[12] T. Kohno, A. Stubblefield, A. D. Rubin, and D. S. Wallach, "Analysis of an electronic voting system," in *Proceedings of the 2004 IEEE Symposium on Security and Privacy*, May 2004, pp. 27–40.

[13] A. J. Feldman, J. A. Halderman, and E. W. Felten, “Security analysis of the diebold accuvote-ts voting machine,” in *Proceedings of the 2007 USENIX/ACCURATE Electronic Voting Technology Workshop*. Berkeley, CA, USA: USENIX Association, Aug. 2007.

[14] E. Proebstel, R. Sean, F. Hsu, J. Cummins, F. Oakley, T. Stanionis, and M. Bishop, “An analysis of the hart intercivic dau eslate,” in *Proceedings of the 2007 USENIX/ACCURATE Electronic Voting Technology Workshop*. Berkeley, CA, USA: USENIX Association, Aug. 2007.

[15] C. Sturton, S. Jha, S. A. Seshia, and D. Wagner, “On voting machine design for verification and testability,” in *Proceedings of the 16th ACM Conference on Computer and Communications Securityecurity*, ser. CCS ’09. New York, NY, USA: ACM, Oct. 2009, pp. 463–476.

[16] K. Weldemariam, R. A. Kemmerer, and A. Villafiorita, “Specification and analysis of the electronic voting process for the es&s voting system,” in *Proceedings of the ARES ’10 International Conference on Availability, Reliability, and Security*, feb 2010, pp. 164–171.

[17] D. Balzarotti, G. Banks, M. Cova, V. Felmetsger, R. A. Kemmerer, W. Robertson, F. Valeur, and G. Vigna, “An experience in testing the security of real-world electronic voting systems,” *IEEE Transctions on Software Engineering*, vol. 36, no. 4, pp. 453–473, Jul. 2010.

[18] C. Karlof, N. Sastry, and D. Wagner, “Cryptographic voting protocols: A systems perspective,” in *Proceedings of the 14th USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, Jul. 2005, pp. 33–50.

[19] D. Chaum, A. Essex, R. Carback, A. Sherman, J. Clark, S. Popoveniuc, and P. Vora, “Scantegrity: End-to-end voter-verifiable optical-scan voting,” *IEEE Security & Privacy*, vol. 6, no. 3, pp. 40–46, May 2008.

[20] A. T. Sherman, R. A. Fink, R. Carbeck, and D. Chaum, “Scantegrity III: Automatic trustworthy receipts, highlighting over/under votes, and full voter verifiability,” in *Proceedings of the 2011 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections*. Berkeley, CA, USA: USENIX Association, Aug. 2011.

[21] RABA Innovative Solution Cell, “Trusted agent report Diebold AccuVote-TS voting system,” RABA Technologies LLC, Columbia, MD, Tech. Rep., Jan. 2004.

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

[23] J. L. Brunner, “Project everest: Evaluation and validation of election related equipment, standards and testing,” Ohio Secretary of State, Tech. Rep., Dec. 2007.

[24] E. Barr, M. Bishop, and M. Gondree, “Fixing federal e-voting standards,” *cacm*, vol. 50, no. 3, pp. 19–24, Mar. 2007.

[25] K. Weldemariam and A. Villafiorita, “A methodology for assessing procedural security: A case study in e-voting,” in *Proceedings of the 3rd International Conference on Electronic Voting*, ser. Lecture Notes in Informatics, R. Krimmer and R. Grimms, Eds. Bonn, Germany: Gesellechaft für Informatik e.V., Aug. 2008, pp. 83–94.

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

[27] M. S. Raunak, B. Chen, A. Elssamadisy, L. A. Clarke, and L. J. Osterweil, “Definition and analysis of election processes,” in *Proceedings of the 2006 International Software Process Workshop and International Workshop on Software Process Simulation and Modeling*, ser. Lecture Notes in Computer Science, vol. 3966, 2006, pp. 178–185.

[28] B. I. Simidchieva, M. S. Marzilli, L. A. Clarke, and L. J. Osterweil, “Specifying and verifying requirements for election processes,” in *Proceedings of the 2008 International Conference on Digital Government Research*, 2008, pp. 63–72.

[29] 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,” in *Proceedings of the 2010 Electronic Voting Technology/Workshop on Trustworthy Elections*. Berkeley, CA, USA: USENIX Association, Aug. 2010.

[30] 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,” in *Proceedings of the 2012 USENIX/ACCURATE Electronic Voting Technology Workshop*. Berkeley, CA, USA: USENIX Association, Aug. 2012.

[31] R. W. Ritchey and P. Ammann, “Using model checking to analyze network vulnerabilities,” in *Security and Privacy, 2000. S&P 2000. Proceedings. 2000 IEEE Symposium on*, 2000.