The Journal of Systems and Software 84 (2011) 1114–1129

Contents lists available at ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Procedural security analysis: A methodological approach

Komminist Weldemariam*, Adolfo Villafiorita

Center for Scientific and Technological Research, Fondazione Bruno Kessler, via Sommarive 18, Trento 38123, Italy

ARTICLE INFO

Article history:
Received 22 May 2010
Received in revised form
15 December 2010
Accepted 31 January 2011
Available online 1 March 2011

Keywords:
Security assessment
Formal specification and verification
Electronic voting

ABSTRACT

This article introduces what we call procedural security analysis, an approach that allows for a systematic security assessment of (business) processes. The approach is based on explicit reasoning on asset flows and is implemented by building formal models to describe the nominal procedures under analysis, by injecting possible threat-actions of such models, and by assuming that any combination of threats can be possible in all steps into such models. We use the NuSMV input language to encode the asset flows, which are amenable for formal analysis. This allows us to understand how the switch to a new technological solution changes the requirements of an organization, with the ultimate goal of defining the new processes that ensure a sufficient level of security.

We have applied the technique to a real-world electronic voting system named ProVotE to analyze the procedures used during and after elections. Such analyses are essential to identify the limits of the current procedures (i.e., conditions under which attacks are undetectable) and to identify the hypotheses that can guarantee reasonably secure electronic elections. Additionally, the results of the analyses can be a step forward to devise a set of requirements, to be applied both at the organizational level and on the (software) systems to make them more secure.

1. Introduction

Often risks and attacks not only depend upon the security levels software and hardware systems offer, but also on the procedures and controls regulating the way in which they are operated. In such cases, introducing technical security mechanisms—such as Adida (2006), Sastry et al. (2006) and Yee (2007)—is not sufficient. To fill this gap, we focus on the definition of methodologies and techniques to model and formally verify procedures and systems processes. This requires not only the definition of adequate modeling convention, but also the definition of general techniques for the injection of attacks, and for the transformation of business processes into representations which can be given as input to model checkers. For this purpose, we have defined a methodology that allows one to perform security assessment on the procedures. The underlying approach that we follow lies on the intersection of three areas, namely, Business Process Engineering and Re-engineering (BPR), security, and formal methods.

1.1. Motivation

We all take actions to avoid security risks in our daily life. They may be as simple as locking the office door when leaving for a day. For our home computer, maybe it is sufficient to activate the firewall and keep updated on relevant security patches. The situation becomes more complex and difficult if the system we need to protect is a major information system that performs critical steps of complex business processes, whose execution might also require several manual actions to be carried out.[1] This is exactly the case of (voting and) e-voting: even in those countries that have adopted a high level of automation, procedures and controls performed by people on physical assets (e.g., printouts of the digital votes) remain an integral and unavoidable part.

In order to ensure a sufficient level of security, therefore, there is a need for a thorough security risk analysis methodology that considers procedures as part of the modeling and analysis process. The approaches discussed so far, e.g., Fovino and Masera (2006), Basin et al. (2003) and Hogganvik (2007), say little or are otherwise ineffective on these *procedurally rich* scenarios. With "procedurally rich" we denote situations in which software systems are just part of a complex organizational setting, in which procedures have to be executed on security-critical assets that belong both to the digital and physical realms.

We address (some of) the issues above by dealing with the identification, modeling, establishment, and enforcement of security policies about the procedures that regulate access and manipula-

* Corresponding author.
E-mail addresses: sisai@fbk.eu, komminist@gmail.com (K. Weldemariam).

[1] In the rest of the paper we use the terms business processes, processes, procedures, and workflows as synonyms.

0164-1212/$ – see front matter © 2011 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2011.01.064

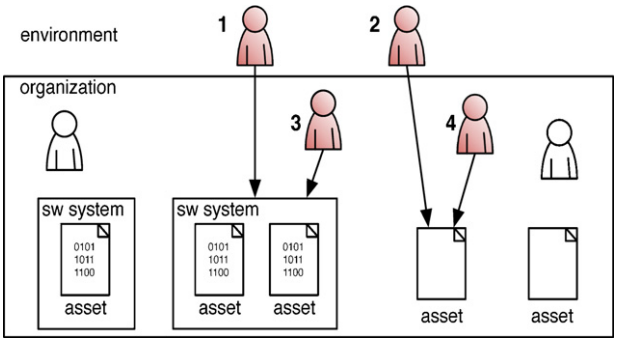

Fig. 1. A reference scenario for procedural security analysis.

tion of critical assets. The breach of security objectives during the execution of the procedures or usage of systems is known as *threat to the procedures* (or *procedural threats*). We call *procedural security analysis* the process of understanding the impact and effects of procedural threats, namely courses of actions that can take place during the execution of the procedures, and which are meant to alter, in an unlawful way, the assets manipulated by procedures.

The reference scenario is shown in Fig. 1. Our target of evaluation, a term borrowed from Common Criteria (Common Criteria, 2007), is a (complex) organizational setting in which procedures transform and elaborate assets. Assets might be made of bits (such as, for instance, an electronic vote stored in a voting machine) or live in the physical world (like, for instance, the paper trail of a voting machine). The procedures and organization are meant to add value to the assets and protect valuable assets from attacks. Think, for instance, of paper ballots: before an election a ballot is a replaceable piece of paper.[2] After a vote has been cast, the same ballot represents the decision of a voter, that needs to be appropriately accounted for and protected.

Attacks in our model might either come from external sources or from insiders, as shown in the figure, where the shaded actors represent a set of adversaries, whereas the non-shaded represent trusted actors.

We distinguish, in particular, the following kinds of attacks:

1. *Attacks on digital assets* (item 1 and item 3 in Fig. 1). These attacks are meant to alter one or more digital assets of an organization. Attacks can either be carried out by external sources (the environment), by internal sources, or a combination of both. Opportunities for attacks are determined by the organizational setting and by the security provided by the digital systems. At attacker, in fact, might need to get access to a digital resource (possibly by circumventing existing control procedures) and overcome the software/hardware protections put in place to protect the digital assets.
2. *Attacks on other kind of assets* (item 2 and item 4 in Fig. 1). These attacks are meant to alter one or more non-digital assets of an organization. Attacks can either be carried out by external sources (the environment), by internal sources, or by a combination of both. Opportunities for attacks are determined by the organizational settings only. The attacks may lead to compromise digital assets as well (e.g., stealing a password provides access to digital assets).

Existing security assessment methodologies, like Fovino and Masera (2006) and Hogganvik (2007), usually focus on understanding items 1 and 3, namely, types and effects of attacks on (software) systems. We propose a tool-supported methodology to tackle also points 2 and 4, namely attacks on assets that are not necessarily digital and that derive from the way in which procedures are implemented and carried out.

1.2. Technical elements of the approach

In order to achieve the goal stated above, we approach the problem by reasoning about the procedures and controls that regulate the usage of systems:

- *Provide models of the procedures under evaluation.* During which we provide models that describe the procedure or procedures to be analyzed. In order to ease the task of translating the models into executable assets flows, we stick to a subset of the Unified Modeling Language (UML) notations (Booch et al., 2005).
- *Extend model.* During which we generate an *extended model* from the models defined in the previous step. The extended model is generated by injecting[3] threat actions into the nominal flow of the procedures. Thus, in the extended model, not only assets are modified according to what the procedures define, but they can also be transformed by the (random) execution of one or more threat actions.
- *Encode the asset flows.* During which we transform the model obtained at the previous step into asset-flows—namely executable specification written in the NuSMV input language (Cimatti et al., 2002), that describe the evolution of assets. The NuSMV model of the asset flows is based on the definition of program counters that ensure that procedures are executed according to the specifications, and by defining one module per asset with one state variable per asset feature. The state variables encode how features change during the execution of the procedures. Accessory information, such as actors responsible for the different activities can be used, e.g., to enrich the language used to express security properties. The necessity of modeling actors roles in NuSMV depends upon the target of the security analysis.
- *Specify security properties to model check.* During which we specify the (un-)desired (procedural) security properties—namely, the security goals that have to be satisfied or violated are then encoded using Linear Temporal Logic (LTL) or Computational Tree Logic (CTL) (Pnueli, 1977) formulas. These, together with the model, are given as input to NuSMV.
- *Perform analysis and results analysis.* During which we run the model checker to perform analyses. If a property is proved to be false, NuSMV generates a counterexample which opens up further discussion. Counterexamples of security properties encode the sequence of actions that have to be executed in order to carry out an attack on an asset.

With this approach analyzing attacks is reduced to a model checking problem in which the required final state of some key assets is expressed using LTL/CTL and the counterexample generated by executing the *extended model* contains the sequence of threat-actions causing the final state not to be reached. The model checker takes care of pruning useless threats, namely threats which do not lead to any successful attack. Analogously to what happens in safety analysis when analyzing, e.g., the loss of critical functions, enhancing the procedures results in reducing the probability of an attack or making the attack more complex, rather than eliminating

[2] Simplifying quite a bit. In fact, by stealing a blank ballot it is possible to implement an attack that allows to control voters.

[3] Note that by fault injection we mean the extension of the assets-flow model with a specification of the possible threat-actions. We adopt this terminology, which is standard in safety analysis, even though it may not be fully appropriate in our domain. See, e.g., Bozzano and Villafiorita (2007).

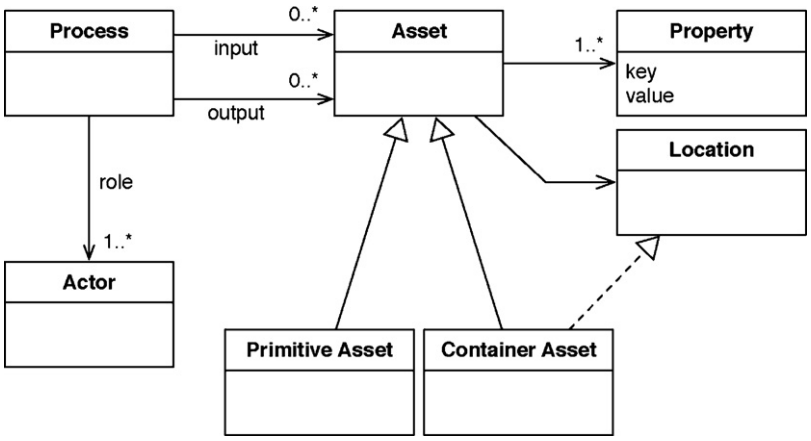

Fig. 2. A class diagram depicting the characteristics of an asset.

it Bozzano and Villafiorita (2007), Manian et al. (1998) and Hsiung et al. (2007).

1.3. Benefits of the approach

The benefits that the proposed methodology brings are twofold. First of all, it helps identifying the *security boundaries*, that is, the conditions under which procedures can be carried out securely. More specifically it is possible to understand what are the hypotheses and conditions under which a given security goal is achieved or breached. Second, it helps devising a set of requirements to be applied both at the organizational level and on the (software) systems used to make systems and system processes secure. This can be achieved by analyzing the counterexamples generated by NuSMV, since they provide information useful to modify the existing procedures and take care of the security breaches.

Thus, this work build upon the motivations for procedural security discussed in Xenakis and Macintosh (2004) and Xenakis and Macintosh (2005). Additionally, we believe that the outputs of the analyses could also be used to familiarize actors (e.g., election officials or polling officers) with the possible procedural threats and attacks that could happen during an election; thus, it complements works like Volkamer and McGaley (2007) and Volkamer (2009).

1.4. Paper organization

The rest of the article is organized as follows. In the next section, we present the conceptual framework for the approach we propose. Section 3 presents the technical elements of the methodology in detail. In Section 4, we demonstrate the approach using examples taken from ProVotE e-voting system (Villafiorita et al., 2009). Finally, we discuss related work in Section 5 and we outline our conclusions in Section 6.

2. Conceptual framework: understanding an asset under analysis

Fig. 2 represents the main concepts we model in our framework. The starting point is a set of processes, that are performed by actors with different roles (left hand side of the figure). The explicit representation of actors is an important feature of the model. In fact, it allows us to represent security breaches related to opportunities for insiders (e.g., an actor participating in different activities might have access to all the different resources needed to carry out an attack) and the security boundaries of the procedures (e.g., what actors do I need to carry out a specific attack?).

In the figure, a process can have zero or more inputs, that is, the assets required for the execution of the process. It can have zero or more outputs, that is, the assets that the process produces or transforms. We call *features* all the structural characteristics of an asset. Processes read and transform assets' features. For the analyses we wish to carry out, in particular, we characterize assets with the following features: nature, value, location, content, a set of domain and security specific properties, and number of instances. In our experience, this characterization is sufficient to understand possible weaknesses of the procedures and the impact of attacks.

The *nature* of an asset can be *primitive* (this is the case of assets such as names, symbols, keywords, passwords, electronic ballots, and electronic data in general) or *container*, when the asset can also contain other assets (e.g., a memory support that can contain electronic ballots). The nature of an asset is immutable, namely it cannot be changed during the execution of the procedures. The *value* of an asset (which we represent qualitatively, e.g., *"no value"*, *"high value"*) allows one to reason about the impact of threats (e.g., an attack to an asset with *no value* does not cause any harm). The initial value of assets is assigned in the model by a domain analyst, and it may change upon the execution of an activity. The value of container assets is determined by their intrinsic value (determined by the analyst), together with the value of the assets they contain.

In our methodology, assets are situated in a *location* (e.g., an electoral office, a safe, a container asset). Breaking into a location or being able to access a container is a mean to lead an attack against a (contained) asset (e.g., stealing a memory support containing electronic ballots allows to attack the electronic ballots and can cause a discrepancy among the election outcomes). The initial locations are determined by the analyst and can be changed by the execution of an activity. The location of primitive assets corresponds to the location of the containers in which they reside. Furthermore, a container asset can be a relative location for another container asset. This allows us to speak about which kind of (sensitive) information an asset has at a certain (discrete) point in time, which we call *content* of an asset. The content of an asset can be known in advance or determined by the execution of a workflow activity—e.g., an *envelop* that is used to ship the voting kits to polling station may contain all relevant information needed to run the election, such as the election software, PIN, etc.

We allow to characterize an asset also by means of a set of *properties* that describe the current situation of the asset such as the security measures that are enacted (e.g., a safe can be *closed*, a file can be *plain* or *encrypted*). While for any domain there can be domain-specific states of the asset, we are particularly interested in security states such as *open, closed, unsigned, signed, encrypted, plain,* etc.

In addition to the mentioned qualities, a number of copies (called *number of instances*) is particularly relevant in the electoral domain because the same asset can be replicated many times. When printing the ballots, for instance, copies go through different responsible people. This is also important to estimate the effects of a threat on an asset, for instance before or after a duplicating operation.

Finally, we call *state* of an asset the values of the features of an asset (i.e., value, location, number of instances, and properties) at a given instance, whereas *asset-flow* is the sequence of states through which an asset goes during the execution of a process. Even though an asset can have multiple instances, we focus on how a single asset instance can evolve from some initial state through other states as the result of the execution of workflows. An example: the abstract concept of *ballot box* (class) is associated with several instances which describe its evolution during processes execution, like a *state* change from "open" (e.g., during the voting phase) to "closed & sealed" (e.g., right after the election is over), and also its *value* changes accordingly, which is "low" when the *ballot box* is empty (before the election) but which becomes "very high" after voter deposits a marked ballot.

2.1. Asset threats and attacks

The information reported previously is sufficient to describe non-trivial flows of the assets in terms of state transition systems in the *nominal* case—i.e., when all assets "flow" as expected. However, this is not always the case, because attacks can often be carried out "bypassing" the procedures. The key question is how flexible and complete our methodology is in describing such attacks (see, e.g., Bishop (2002) and Xu and Nygard (2005)).

Typically, an attack model describes the intents a malicious actor might have (e.g., modify the election software before the election, sabotage election result), the types of malicious actors (e.g., election officials, poll workers, messengers, or voters), the privilege to each type (of malicious) actors (e.g., who has read/write access to a critical asset), and the location where malicious actors can possibly implement the attack.

For our purpose, it is enough to assume Fig. 1 as our reference attack model, since it distills the essential characteristics of a procedural rich scenario. As noted in the figure, as a minimum, we distinguish two types of assets (digital and physical assets), the attackers' nature (insiders and outsiders), and the environment in which the procedure transforms the assets under analysis. Domain-specific formalizations typically provide additional details on top of the ones we just mentioned.

In principle, an attacker might wish to pursue any kind of goal, such as financial gain or political interest (e.g., see Jones (2003), Lambrinoudakis et al. (2003) and Sastry (2007)). We consider the following goals: compromising data integrity (e.g., produce incorrect vote counts), compromising the availability of a system causing a denial of service attack (e.g., block some or all voters from voting in key polling site, delay the announcement of election results), compromising the confidentiality of the system (e.g., allow voters to cast doubt on the legitimacy of the election results). The dimensions of attacks we considered are those described in Kohno et al. (2004) and Balzarotti et al. (2008).

2.1.1. Asset threats and attacks

By assigning each asset a value and a location we can highlight in a model where a threat may be implemented and how much harm it can cause. A denial of service, for instance, could be caused by deleting a valuable asset (an asset whose *content* value is not null) when the asset is in a given location. The notation used previously, however, does not allow to represent the how, e.g., the flow of actions causing termination of the procedures in some undesired

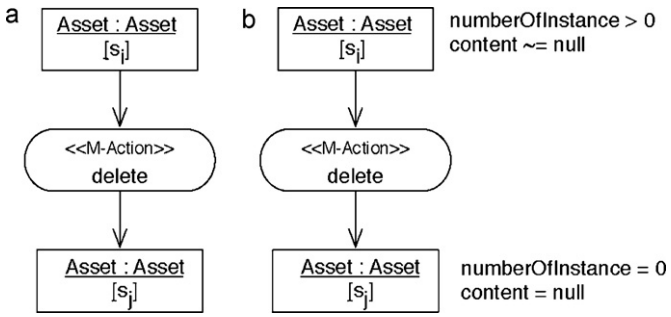

Fig. 3. The *Delete* threat Action and an Example. (a) The *Delete* Threat-action. It changes the state of the asset from s_i to s_j. (b) Instance of *Delete* threat action.

state nor to reason about composition of threats—that is, what happens if multiple assets are attacked simultaneously. To tackle this, we propose the concepts of *asset threats* and *attacks*.

We define a threat (or threat-action) to an asset as an action that alters feature(s) of the assets or allows some actors privileges (e.g., a "read" privilege) on some assets. That is, it is what an adversary might try to do to an asset in a system and it is described by a sentence like *"adversary does something to the asset"*. Like workflow activities, threat-actions can transform assets or their state or both. Unlike the workflow activities, however, they transform assets to non-nominal situation, that is, the assets are in undesired states due to the effect of the threat-action.

To differentiate between the *nominal* and *non-nominal* behavior of assets, we enlarge the set of processes and assets by distinguishing between:

- *malicious processes*, we identify some of the possible operations that an adversary may conduct against an asset, and
- *malicious assets*, we describe all the instruments and data built or obtained by an adversary to achieve his/her goal(s).

Malicious processes can require as input some particular assets, produce new assets to complete other wicked activities, and modify the original assets to alter the asset flow. To give an example, a sniffer could be considered as a malicious asset needed by the adversary for capturing the password and the result is a new malicious asset: the unauthorized password copy in plain to use for other purposes. This further led us to distinguish between threat-actions into:

- *basic* threat-actions, are usually considered to be elementary if no decomposition will reveal any further information of interest.
- *composed* threat-actions, are obtained by composing actions to produce more complex behaviors.

Both basic and composed threat-actions can alter, e.g., the content of an asset, its state or both. Some actions might be executed just to access a resource, which in turn could trigger other more risky actions that can directly modify the asset. Examples of *basic* threat-actions include: *delete* (see also Fig. 3), *read*, *write* (or *update*), and *create*. Notice that we always distinguish threat-actions from intended actions by using the *threat* or *threat-action* along with the action name (e.g., "delete" and "delete threat-action" refer two different intents.)

Replace is an example of a composed threat-action. It substitutes an asset with another (malicious) asset. It takes as input two assets, that is, the original asset and the malicious asset and produces one output, namely the malicious asset. In its basic form, *Replace* is composed of *delete* and *write*; the action, however, can also include a *copy* operation, to simplify the alteration of some parameters of the original asset.

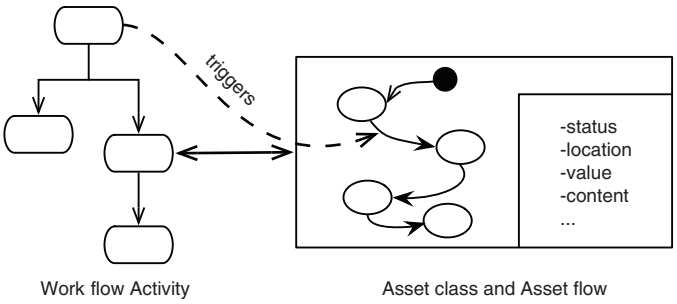

Fig. 4. An asset-flow view of a business process model.

Notice that in our approach we use the NuSMV input language to encode the threat-actions while extending the nominal assets-flow models. An alternative approach is using first order logic and first order reasoning to formalize coordination of attacks; see Koubarakis and Plexousakis (2000) and Braynov and Jadiwala (2003) for more details.

Finally, to carry out an asset threat the adversary may need to execute one or more (more elementary) threat-actions or a sequence of asset-threats against other asset—e.g., adversary reads "password" and signs "data". Therefore, we define an attack as a sequence of asset-threats that lead to an undesired state, that is, one or more assets are in an undesired asset-state.

3. A methodology for procedural security

As mentioned earlier, we developed a precise methodology to perform formal procedural security analysis. In the following, we discuss each element of the methodology in detail.

3.1. Formal model of asset-flows

The first activity in our security analysis of a procedure consists of providing models of the assets-flow, which are represented using transition systems. Such models describe the assets to be analyzed, and the way in which their features are transformed by procedures. The key elements in the model include assets, (workflow) activities, and transition semantics.

Fig. 4 shows high-level representation of the lifecycle of assets. Unlike the classical UML Activity Diagrams (UML ADs) (Booch et al., 2005), the perspective shown in the figure offers three views: workflow, assets class, and state machine. In the workflow diagram view, workflow activity sequences are defined. The state machine view describes the behavior of an asset in terms of a transition system in which transitions are enabled due to explicit execution of workflow activities. The activities in the workflow are transformation functions that influence the behaviors of the assets. A finite state transition diagram for each feature of an asset constitutes the global state machine for that asset.

3.1.1. Formalization of the models

The formalization allows the model to be more amenable to formal analysis, since it removes the strategic flavor of the (business) process models and shifts the focus to dynamic aspects of the assets, and hence procedures under analysis. Our formalism borrows and extends the approaches proposed in Gerede and Su (2007),Bhattacharya et al. (2007) and Hull (2008). However, we differ on the interpretation of some concepts and on the analysis goals we wish to perform. In fact, our approach moves forward the cited works (in which the authors provide formal model for business artifacts and analyze them for better construction of business operations and processes) by complementing the their approach with security analysis.

To begin with, we assume the following notations and their definitions.

- $\mathcal{T}_p$ be a set of primitive types, such as bounded integer and boolean;
- $\mathcal{C}$ be a set of asset classes (*names*);
- $\mathcal{A}$ be a set of attributes (*names*);
- $\mathcal{ID}_{\mathcal{C}}$ be a set of identifiers that describes the identifiers for each asset class $C \in \mathcal{C}$;
- $\mathcal{S}$ is a set of assets states, where each $s \in \mathcal{S}$ is a truth assignment over the variables values.

Note that all the above sets are finite, which is essential for the model checking process.

A *type* $\mathcal{T}$ is an element of the primitive types $\mathcal{T}_p$ and the class identifiers C; namely, $\mathcal{T} = \mathcal{T}_p \cup \mathcal{C}$ (we assume that $\mathcal{T}_p$ and $\mathcal{C}$ are disjoint).

Definition 3.1. An asset class signature is a triple $\langle C, A, \psi \rangle$ where $C \in \mathcal{C}$, $A \in \mathcal{A}$ is a a set of attributes for the asset class C, and $\psi : \mathcal{A} \rightarrow \mathcal{T}$ is a total function that maps each attribute of an asset into its corresponding type.

Definition 3.1 represents the asset class signature that specifies assets that are present in the domain. In the definition, besides the features of an asset discussed previously, the specification of the attributes can comprise of domain specific or assets specific attributes.

Notation 3.1. We write Σ to denote the set of all possible assets classes that exist in the domain, including malicious assets that can be introduced by an adversary.

Without loss of generality, we assume a fixed interpretation domain associated to each $\mathcal{T}$ type. That is the *domain* of each type $t \in \mathcal{T}$, denoted $\mathcal{D}^t$, is defined in the following way: if $t \in \mathcal{T}_p$ is a primitive type, then the domain $\mathcal{D}^t$ is some known set of *values* of type (e.g., integer or boolean); if $t \in \mathcal{C}$ an identifier type, then $\mathcal{D}^t$ defines existing instances of an asset class identifier for t (i.e., $\mathcal{D}^t = \mathcal{ID}_t$). We require all variables must have their corresponding *values* all along their life. For undefined location and unassigned content of an asset, we use an *undefined* and a *null* constant values, respectively. The interpretation is that the location is not known and the content value is not either assigned yet or reset to contain null. For instance, the *content* of a memory support (such as USBKey) can be *null* prior to loading the election software by a responsible actor (e.g., by a technician or an election official).

Definition 3.2. An asset instance is a triple $\langle ID_C, C, \phi \rangle$, where $ID_C \in \mathcal{ID}$ is a class identifier and ϕ a partial function, given an instance of a class C, that assigns each variable $a \in A_C$ of type $t \in \mathcal{T}$ a value in D^t (i.e., $\phi(a) = D^t(\psi(a))$).

Notation 3.2. We use $S_C \subseteq \mathcal{S}$ to denote all the possible states (i.e., both nominal and non-nominal states) of an asset class $C \in \mathcal{C}$.

An asset can have multiple instances. We denote the set of asset instances by $\vec{\mathcal{O}}_{C, C \in \mathcal{C}}$ and $\vec{\mathcal{O}}$ for all instances over Σ. Duplicating the election software for each polling station, for example, creates as the number of polling stations instances where each of this instance can behave differently after duplicating operation in place. As we noted previously, in this work, we mainly focus on how a single asset instance $I \in \vec{\mathcal{O}}_C$ can evolve from some initial state through to other states. The set of *variable-value* pairs for I defines the *state* of a given asset. The *state* of an asset is, therefore, the current situation called *"snapshot"* of an asset instance I and its value is the truth assignment over the variables. An asset is *initial*, if all the variables are in their initial state and ϕ is undefined for some attributes and *final*, if all the variables values do not change anymore.

F ormal model of workflow specification. The initial values for the variables of an asset can be assigned at the time of the instance creation or otherwise assigned by an analyst. However, only due to the execution of a workflow activity over these variables can change the initial configuration of the asset. Roughly speaking, a workflow activity is described by input assets, preconditions, and effects of the activity over the assets (a similar interpretation can be found in Koubarakis and Plexousakis (2000), Gerede and Su (2007) and Bhattacharya et al. (2007)). The effect of a workflow activity is regarded as a change in state of the input assets. Not all assets change their states thought, since it is not always the case that an execution of a workflow activity enables state transition to all the input assets (e.g., reading the content of the election software does not change its state).

For each executable workflow activity, moreover, we specify which actors participate in the workflow with predefined privileges or responsibilities or both. These information – we call accessory information – not only allows to describe who does what during the execution of an activity, but, more importantly in the context of procedural security analysis, who manages what data and with what privileges. Such information are static, namely they are known before executing a workflow and are encoded in our model to describe a workflow scenario. We, therefore, use these information along with the activities to describe a workflow model as a deterministic finite state machine in which the states are constructed by a set of activities and the transitions are described by the current state and a matching condition over the accessory information. Formally, we define the workflow model as follows.

Definition 3.3 *(W).* A workflow model is a quadruple $\langle P, s_0, s_f, C, \Delta\rangle$ where P is a set of activities or processes (*names*); $s_0, s_f \in P$ are initial and final activities of the workflow respectively; C is guard expression over accessory information, and $\Delta \subseteq P \times C \times P$ is a transition relation between a current activity and its successor activities in which a transition is labelled with a condition over accessory information.

The above definition is meant to express the fact that there exist a set of activities within a particular workflow, that describes a procedure under analysis, thereby by knowing the current state of the workflow, and if a condition is met, it should be obvious to determine the next state of the workflow. We call an instance of a workflow model a program counter ' 'pc", which contains the value of the current state (i.e., the active activity) in the workflow. There is one program counter ' 'pc" for each workflow model at run time. In actual business process or workflow specification, in fact, it is possible to have multiple activities that can run in synchronous or asynchronous mode. We focus on sequential execution of a workflow in this work.

Formal model of asset-flow specification. The state of an asset is specified by the assignments of values to variables—or simply valuations. This, in turn allows to describe the evolution of an asset, and is expressed by the sequence of states through which an asset undergoes during the execution of a process. It makes sense to encode the state of each variable as a finite state machine, since the state of an asset is described by the *valuations* over its variables. The workflow instances, along with some matching conditions, define transitions for modeling the lifecycle of the assets. Thus, an asset-flow can easily be modeled using a transition system, thereby facilitating the sformal analysis.

We now define our notion of assets-flow models more formally as,

Definition 3.4. An asset-flow model (AFM) is a 5-tuple $\langle AS, I, \mathcal{W}_\pi, C_\pi, \Delta_\pi\rangle$ where

- $AS \in \mathcal{S}$ is a finite set of assets' (instances) states;
- $I \subseteq AS$ is an initial states of the assets;
- $\mathcal{W}_\pi$ is a set of workflow instances;
- C_π is a set of conditions constructed over the attributes representing the matching construct as a guard, that specify the condition must meet for the state to be changed, along with the current activity;
- $\Delta_\pi \subseteq AS \times \mathcal{W}_\pi \times C_\pi \times AS$ is a transition relation between a current state of an asset and its successor states in which a transition is labelled with an activity and a condition.

A collection of individual *AFM* constitutes *assets-flow* models, and is represented by $\mathcal{M}_1$. Therefore, $\mathcal{M}_1$ is regarded as the global configuration of the domain of interest, namely the procedures under analysis. The semantic of the global configuration $\mathcal{M}_1$ can be interpreted in the following way. Each $m \in \mathcal{M}_1$ is regarded as an abstract state machine, which has three major components: a workflow activity sequence (possibly maintained in a queue), a workflow activity dispatcher, and an activity processor. Workflow activities are added to the end of the activity queue. The activity dispatcher chooses, dequeues, and provides the next ' 'pc" (i.e., an activity) to the activity processor. Each ' 'pc" is then used as a *transformation* function that can possibly change the state of an asset by modifying or changing one or more variables values of the asset. One state machine per feature variable encodes the lifecycle of that state variable (e.g., see Fig. 7). A set of such state machines constitutes the global state machine for the corresponding asset instance. By defining a semantic for the state machines corresponding to each feature of an asset and linking it with $m \in \mathcal{M}_1$, therefore, we have implicitly defined how $\mathcal{M}_1$ behaves.

3.2. Model extension

The formal model we presented earlier represent the *nominal* behavior of the assets ($\mathcal{M}_1$)—that is, the model describing the standard procedures, e.g., what is prescribed by the law. In order to analyze what are the possible attacks of a given (set of) procedures, we need to encode asset threats in the nominal model and generate the extended model $\mathcal{M}_2$. Structurally, in fact, there is no difference between $\mathcal{M}_1$ and the extended model $\mathcal{M}_2$. However, the main difference lies on the assets state *set* and on the transitions specification. This means that, the extended model possibly will have more states than the other due to the execution of threat-actions that can change the state of an asset but into undesired state.

On the transitions side, on the other hand, the definition did specify the fact that transitions are triggered only by nominal workflow activities. We need to incorporate in $\mathcal{M}_2$ the fact that an asset could be in any possible states and that such states can also be changed by the execution of malicious processes. However, it is pretty straightforward from the definition we gave and by extending the definition of the workflow model (e.g., Definition 3.3) to include all the malicious processes that an adversary might execute. Thus, in $\mathcal{M}_2$, assets are not only manipulated according to what should happen in the nominal case (i.e., according to the electoral laws), but can also be transformed by the execution of one or more assets threat-actions.

3.3. Encoding the assets-flow models in NuSMV

Assets-flow models $\mathcal{M}$ can become executable specification to allow formal analysis through verification tools on their evolution, including their malicious evolution due to threat-actions. Our aim here is to represent the model $\mathcal{M}$ into executable specification using NuSMV input language. The NuSMV semantic (see in Cimatti et al. (2002)) is based on a state-based formalism in which the behavior is defined by Kripke transition systems. However, the above definition for $\mathcal{M}$ is an action-based formalism in which the behavior

is defined by (a sort of) labelled transition systems. Thus, we need to rearrange the previous definition to align with the semantic of Kripke structure so that the encoding of NuSMV specifications can be tackled.

Definition 3.5 *(AFM, $\mathcal{M}_K$).* Let APs are set of atomic propositions ranged over some boolean expressions on the valuations of the variables. An asset flow model (AFM) is a Kripke structure over a set of atomic propositions *AP* defined by a quadruple $\langle AS_K, I_K, \Delta_K, \mathcal{L}_K \rangle$ where

- AS_K is a finite set of assets (instances) states;
- $I_K \subseteq AS_K$ is set of initial states;
- $\Delta_K \subseteq AS_K \times AS_K$ is a transition relation between a current state of an asset and its successor states;
- $\mathcal{L}_K : AS_K \rightarrow 2^{AP}$ is the labeling function which returns the set of atomic propositions which hold in a state.

Therefore, the encoding of $\mathcal{M}$ in the NuSMV input language can be treated as a problem of defining a mapping between the two structures, i.e., between the structure specifying the model $\mathcal{M}$ and the Kripke structure.[4]

In general, the translation is performed as follows:

1. We define a module that works as the program counter, which encodes the sequence of activities defined by the workflow. The transition from one activity state to the next is determined by the current state of the activity and some accessory information such as role(s), which encodes the relationship between the workflow activity and actors who are assigned to perform the execution of the workflow.
2. A module is defined for each asset that it is specified in the assets-flow models (or specified in the process diagram). Each feature of the asset is defined as a state variable within the module. The transition from one asset state to the next is determined by the program counter (which represents the execution of an action of the workflow) and some boolean expressions over the current state of the asset;
3. We define some boolean appendage variables to capture malicious flows of assets and the execution of threat actions to form the extended model. That is to say, these variables help for the encoding of both malicious assets and threat actions to derive the extended NuSMV model. Additionally, we introduce a state variable in workflow module to capture malicious processes.

The above general strategies are spilt into a number of rules. More specifically, the following encoding rules can be used to map $\mathcal{M}$ into the NuSMV counterpart.

Rule 1 The workflow model is encoded in NuSMV as a special module, and each workflow activity $p_i \in P$ *for* $i = 1, \ldots, n$ representing the domain activities (i.e., *processes*) in $\mathcal{W}$ are encoded in the NuSMV input language as a scalar variable program counter (`pc`) in which p_i are its symbolic values.

```
MODULE Workflow (...)
    VAR
        pc : {p_1, p_2, ..., p_n};
```

Rule 1 defines the special module that works as the program counter. In particular, it specifies the workflow model ($\mathcal{W}$) and the declaration of state variable `pc` under the module, where all domain activities in the workflow are the scalar values of `pc`. The `pc` ensures that the order in which activities are executed is the one defined by the workflows.

[4] We should be clear that this kind of rearrangement is not new, e.g., a similar work can be found (Lam and Padget, 2004).

Table 1
Examples of accessary information as predicates.

Predicate	Meaning	NuSMV variable
AssignR(a,r)	assignment of actor $a \in Actor$ to role $r \in Role$	assign_a_r
AssignA(a,p)	assignment of actor $a \in Actor$ to an activity $p \in P$	assign_a_p
r_Active_for_a	role $r \in Role$ is active for actor $a \in A$	activefor_a_r
ExecA(a,p)	actor $a \in A$ executes an activity $p \in P$	exec_a_p

In order to determine the state transition of the program counter, we introduce some predicates (see Table 1). Notice that these information can easily be inferred by looking at the process diagrams. They are mainly associated with the accessory information, such as actor-role and actor-activity (i.e., R_{AP}) assignments. The table also shows the corresponding state variables in NuSMV input language.

Rule 2 The accessory information are encoded in the NuSMV input language within the Workflow module in the following way (see also Table 1):

- For each actor-role assignment, we introduce a variable assign_a_r. assign_a_r is true iff the predicate AssignR(a,r) is true for an actor $a \in Actor$ and a role $r \in Role$;
- For each actor-process assignment, we introduce a variable assign_a_p. assign_a_p is true iff the predicate AssignA(a,r) is true for an actor $a \in Actor$ and an activity $p \in P$;
- For each role activation r_Active_for_a, we define a state variable Activefor_a_r;
- Similarly, we define a variable Exec_a_p for every actor performing an activity, i.e., iff ExecA(a,p) is true.

Rule 2 defines accessory information for the transition relation of `pc` state variable. Notice that activities can only be executed if the activity instance in question is assigned to an actor—i.e., ExecA(a,p) $\Rightarrow$ AssignA(a,p). Moreover, a group of actors can perform the same activity.

Rule 3 For each asset instance in $\vec{\mathcal{O}}$, a NuSMV module is defined:

```
MODULE ASSET_NAME  (...)
```

Rule 4 An asset with no content in $\mathcal{M}_1$ is mapped to a symbolic value *"null"* in NuSMV. Similarly, an asset whose current location is not known or unspecified in $\mathcal{M}_1$ is mapped to a symbolic value *"unspecified"* in NuSMV.

Rule 5 The *location*, which represents all the possible places of an asset, is encoded in the NuSMV input language as scalar variables `loc` in which loc_i for $i = 1, \ldots, n$ and "*undefined*" are its symbolic values. The *content*, representing all the contents of an asset at a particular point of time, is encoded in the NuSMV input language as `content` in which $content_i$ for $i = 1, \ldots, n$ and "*null*" are its symbolic values. The *value*, representing all security risk values for an asset, is encoded in NuSMV input language as `value` in which *noValue*, *low*, *high* and *critical* are its symbolic values. Finally, each domain specific `property` of an asset in an asset-flow model is encoded as a boolean value in NuSMV.

```
MODULE ASSET_NAME  (...)
 VAR
  loc :  {loc_1, loc_2, ..., loc_n, unknown};
  content :  {content_1,content_2, ...,
                        content_n, null};
  value :  {noValue, low, high, critical};
  property_1, property_2, ... : boolean;
 ...
...
```

Rule 3 states that a module is defined for each asset (instance) in $\mathcal{M}_1$. In Rule 5, whereas each feature of the asset is defined as a state variable within the asset module specification. An *unknown*l

ocation and a *null*value are both encoded by symbolic values as defined by Rule 4.

Rule 6 The transition specification for each state variable is encoded by the current value of the program counter and some boolean expressions over the current state of the asset. The below code shows a template for a transition specification for each state variable.

```
MODULE ASSET_NAME (pc, ...)
 [...]
 /*template for content transition encoding.*/
  next(content) :=
   case
pc.pc = activity & bool_expr: content_j;
[...]
1 :  content
esac;
[...]
```

The above rule (i.e., Rule 6) encodes the transition specifications. The transition from one asset state to the next is determined by the current value of the pc and some condition over the current state of the asset instance. That is, the current state of the pc – which is passed as a parameter to each asset module – along with boolean expression, *bool_expr*, is encoded as a boolean expression like pc.pc = *activity*&*bool_expr* from $\mathcal{W}_\pi \times C_\pi$.

Model extension. Since all the above rules are related to the encoding of $\mathcal{M}_1$, we need to provide additional rule for encoding $\mathcal{M}_2$. The model extension corresponds to proving an extension in the NuSMV model with one or more applicable attack-actions. That is, a specification of how the assets can be in undesired states. This can be done by associating threat-actions with variables defined inside the module per asset instance. Moreover, the Workflow module should also need to be extended in order to include the malicious process executions.

Note that attacks depend on what threat-actions are carried out, the effectiveness of the analysis depends upon the injection strategy that is chosen. It turns out that the best injection strategy consists of injecting all possible threat-actions at all possible steps of the nominal procedures and let the model checker to find the possible combination of the sequences that lead to undesired state for an asset flow. Therefore, the problem of encoding of asset threats corresponds to extending the nominal assets-flow specification with threat actions.

In particular, the model extension can be done by using the following strategies:

- by defining a scalar state variable to encode all the possible malicious process within the Workflow module. Therefore, the program counter does not only have values from nominal workflow activities but also from possible set of malicious workflow activities;
- by defining a transition specification for each activation of a threat-action on asset instance under the corresponding asset module in NuSMV, where the malicious activity (i.e., the current value of the pc) is in place for enabling the transition;
- by defining boolean variable to monitor the execution of the corresponding threat-action. This variable will be true iff when the corresponding threat-action takes place;
- by introducing a scalar value "garbage" for the content state variable related to the introduction of malicious asset and a boolean variable. This variable will be true iff a predicate associated with the action (e.g., MAsset(t)) is true by a threat-action, say *t*.

The above strategies facilitate the task of model extension, by adding a number of boolean appendage variables that are needed to capture the malicious asset flows and the execution of threat actions to form the extended model specification in NuSMV input language. That is to say, these variables help for the encoding of malicious processes, malicious assets, and threat-actions to derive the extended NuSMV model. In this way, therefore, the model extension is performed for each applicable threat-action against the normal flow of assets. Notice that the model extension can be done in two ways with different abstraction levels, namely either at higher level (i.e., at UML diagrams level) or at lower level (i.e., at the NuSMV specification level). The list of activities executed to carry out, e.g., an attack, we can derive the list of actors involved, simply by looking at the UML activity diagrams.

3.4. Property capturing and model checking

During this phase, an analyst defines (procedural) security properties that will be used at a later stage to assess the behavior of the procedure under analysis. More specifically, assets-flow model definition, asset attacks definition, and model extension are just a part of the verification and security analysis process. Formal verification is carried out by defining properties in the form of temporal specifications.

We use LTL and CTL to encode security properties. LTL allows to specify properties related to each possible state of a system along a path—namely, to reason on the computational path scenarios of an asset (e.g., *"what can happen as asset travels along different locations"*). By contrast, CTL is used to specify properties related to how the state of a system can evolve overtime along all the possible computational paths—namely, to reason about the existence of specific states (e.g., *"is there any particular state in which an asset can be altered in an undesired way"*).

In fact, the types of properties to specify depend on the goals of analysis we wish to perform on the models. We are interested, in particular, in the following classes of properties:

1. The **Actor-Play-Role**, namely the roles actors have in the execution of the attack and the privileges they get on assets.
2. **Undetected attacks**, namely sequence of actions that succeed in altering one or more assets and for which the procedures provide no check to highlight the alteration.
3. **Denial of services**, namely attacks which are meant to alter one or more assets in such a way that procedures have to be stopped. In the optimistic case, a denial of service in an election represents a cost and a "nuisance" for the community (as, e.g., results are delayed; the administration needs to re-run the election). In the pessimistic case, e.g., repeated attacks, it may represent a serious threat to democracy.
4. **Reachability analysis**, namely the sequence of actions leading to the violation of a security goal, with particular respect to the execution of asset-threats.

Once all the properties of interest are specified with respect to the analysis goals described above, the next activities are formal verification and analysis of the results. That is to say that as long as a nominal model ($\mathcal{M}_1$) or extended model ($\mathcal{M}_2$) is available, it is possible to verify its behavior with respect to the desired CTL/LTL properties.

The model under analysis is checked (i.e., model checking) against the security properties using NuSMV. And, the results of the analysis can be used for further discussions.

In the case of a system property, the model checking engine can test validity of the property, and generate a counterexample in case the system property is proved to be false. For instance, if we consider a property that is required to hold for every possible path of the asset-flow (CTL property), the model checking engine will generate a counterexample showing one particular path along which the property has failed. In standard situations, the counterexample will contain the execution of one (or more) asset threat. Notice that

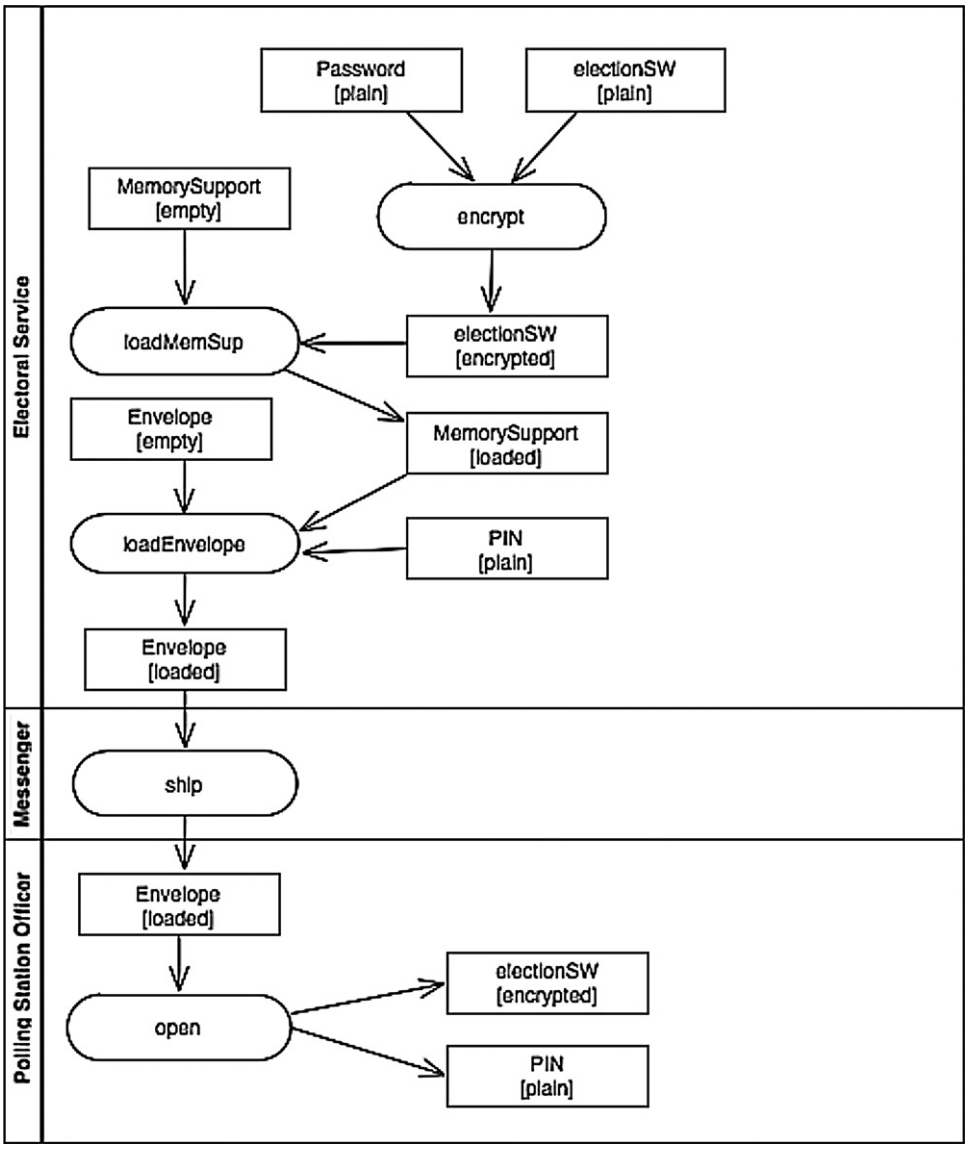

Fig. 5. An example of asset flows taken from real procedure as used in the ProVotE system.

a counterexample in which no asset threats are executed would show an inherent weakness in the *nominal* workflows or otherwise a result of poor specification. The counterexamples of security properties encode sequences of actions that, if executed, pose a threat to security of one or more assets. Furthermore, before calling the verification engine, it is possible to perform constrained or random simulation and several kinds of formal verification analyses using the facilities provided by the NuSMV engine.

4. Using the methodology: a case study

We have applied our approach in scenario in which part of a complex procedure followed during election in Italy to experiment an e-voting system in the polling stations.

The e-voting system, a DRE with printed trail, is described in Villafiorita et al. (2009). Here suffice saying that the experimentation, that involved more than twenty thousand citizens, were meant to test also the procedures in place for managing all the logistics of the elections. One such procedures is the delivery of the voting software to the polling stations. This is an interesting procedure, as the delivery of electronic data to polling stations (be it the software to run the machines or just the data with the specification of the election) is common to a wide set of e-voting systems and a source of concern.

This is described in Fig. 5, where we show an excerpt of the procedure.

The diagram shows how, before the election, the Electoral Office encrypts the e-voting software and creates a memory support which contains the final software release. The responsible person at the Electoral Office prepares an envelope with the PIN code (that it is used to activate the voting functions) and the memory support. We classify Electoral Office's into different level of technicians, for example, technician for generating encryption key and another technician for preparing the envelope. A messenger (e.g., a police officer) takes the envelope and delivers it to the polling station, where the polling officers, once verified that the enveloped is sealed, opens it, insert the memory support in the voting machine, insert the PIN and start the voting operations.

We then injected threats into the model of the procedures to build the extended model. Fig. 6 depicts the extended model resulting from the injection of some *delete* and *replace* threat-actions in the example of Fig. 5. Note that the semantics of delete and replace actions may slightly vary when applying them to different kinds of assets. In the extended model, we marked threat actions with the

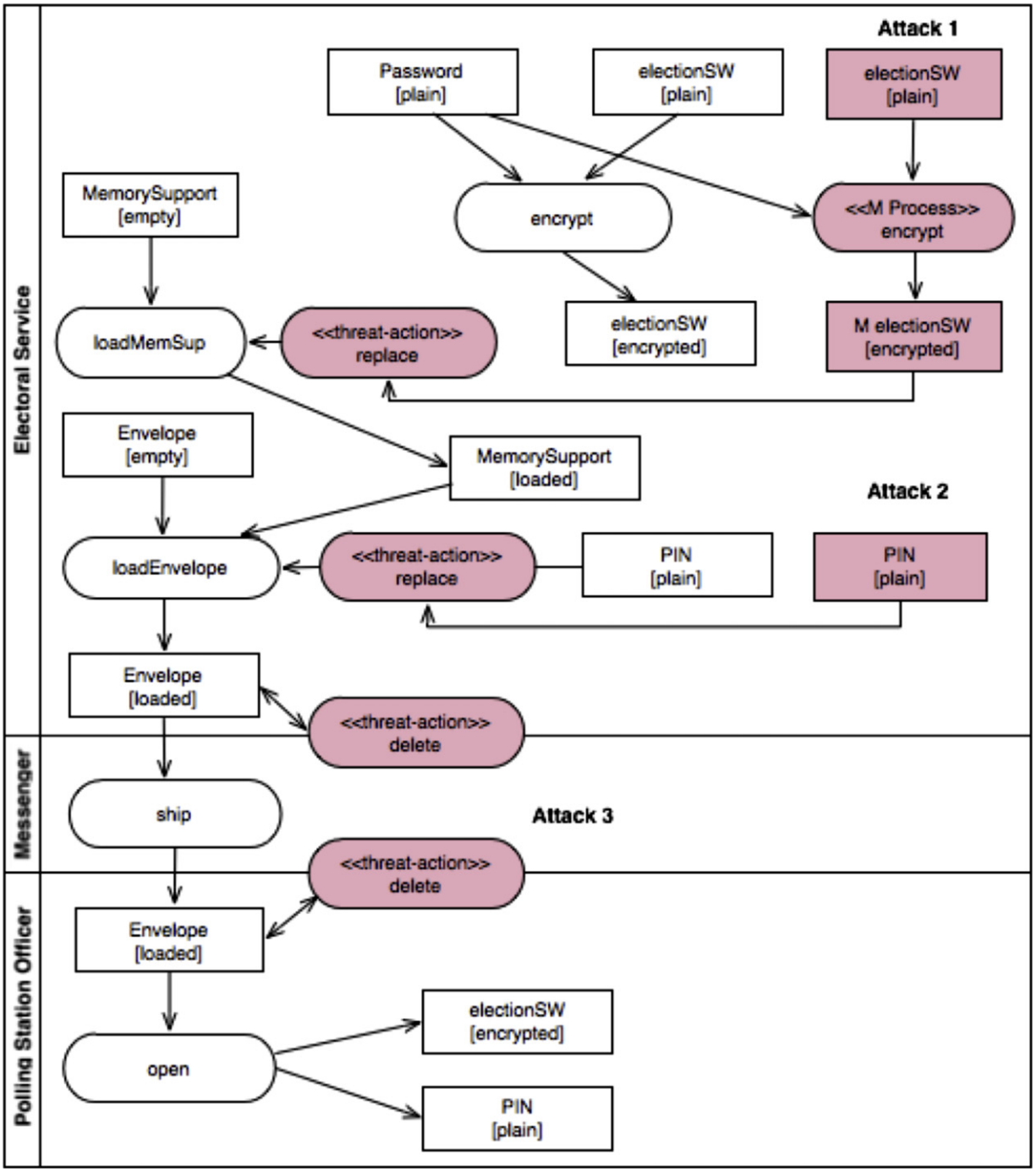

Fig. 6. An example of extended model for Fig. 5, where the introduction of the attacks are colored. It shows delete and replace threat-actions change the flow of the procedure under evaluation.

`threat-action` stereotype. As noted before, the model extension (or threat injection) step is not necessarily done after the process modeling step; it can also be done after encoding the asset flows.

We then modeled the asset-flows into executable specification using the NuSMV input language using the translation rules described previously. We declared four modules corresponding to each assets in the diagram (Fig. 5): *Password*, *electionSW*, *MemorySupport*, and *PIN*. The following snippet of code defines the asset type `electionSW` and some of its features, named `status` (security protections applied to the software), `value` (the criticality of the asset, as assigned by analysts), and `content` (that is, the qualitative value of the `electionSW` can be), etc.

```
MODULE electionSW ( ... )
 VAR
  value   : {NoValue, Low, High, Critical};
  content : {PlainSW, EncryptedSW, SignedSW,
             EncryptedEnvSW, Null},
  loc : {ElectoralOffice, PollingPlace,
         PoliceOffice, Messenger, Undefined},
```

Similarly, other modules with their corresponding feature variables are declared (e.g., `MODULE Password(...)`, `MODULE MemorySupport (...)`).

The nominal workflow is composed of five activities (see Fig. 5), namely, `encrypt`, `loadMemSupport`, `loadEnvelope`, `shipEnvelope`, and `openEnvelope`. We declare a module called `Workflow` and specify state variable `pc` that assumes these five scalar values exactly in the order allowed by the execution of the workflow.

```
MODULE Workflow ( ... )
 VAR
pc : {encrypt, loadMemSupport, loadEnvelope,
     shipEnvelope, openEnvelope};
execute_actor2_encrypt : boolean;
execute_actor1_openEnvelope : boolean;
[...]
```

Accessory information, such as the actors responsible for each activity, is encoded in the model through `DEFINE` in the main module, such as in the following snippet:

```
DEFINE
 ElectoralServiceActive_for_actor3 := pc  =
      loadMemSup || pc = loadEnvelope || [...]
 POfficeActive_for_actor1 := pc  = openEnvelope
      || [...]
```

Evolution of assets' properties are encoded in NuSMV with the `next` construct (which specifies the value of a variable at step $n+1$, given the value at step n). Notice that asset flows are defined both in terms of the program counter (e.g., the current step of the workflow) and the value of the asset features. Fig. 7 shows a model of feature `content` variable of `electionSW` where its state changes according to the program counters and according to the current values of some state variables; its corresponding snippet NuSMV code is also shown.

Note that in the code shown below, we have left some detail specification for the matter of presentation purpose.

```
[...]
next(sw.content) := case
 (pc.pc = encrypt &&  content =  PlainSW
  && loc = ElectoralOfffice)
  || (content = EncryptedEnvSW
    && pc.pc =  openEnvelope && POfficersActive
    && loc = PollingPlace)
 : EncryptedSW;
 (pc.pc = loadEnvelope && (TechnicianTwoActive
     || ElectoralServiceActive)
     && content = EncryptedSW)
  : EncryptedEnvSW;
 pc.pc = decrypt && POfficersActive
      && (status = Encrypted
       || content = EncryptedSW) && !fakeKey
    : plainSW;
  [...]
```

Model extension. Next, we show the extension of the model according to the diagram depicted in Fig. 6. Threat injection (model extension) corresponds to augmenting the state machine of the asset flow with new transitions corresponding to the execution of threat-actions. Fig. 8, for instance, shows an asset flow with some threat-actions that may alter a feature of an asset (e.g., `content`), in some undesired way.

The triggering of a threat-action is monitored through boolean variables that are set to true when the action takes place, as illustrated by the following pieces of code. We first declare one boolean variable per threat:

```
can_mesw, can_meesw: boolean;
can_garbageSW: boolean;
can_mPPin, can_mePPin: boolean
```

The above variables are initially set to false. When a variable is set to true (either because constrained to do so by the model or, more often, at random), a transition in the state machine encoding the asset flow is triggered and the value of the asset flow changed according to the threat-action (rather than to the nominal flow), as illustrated by the following piece of code:

```
next(can_mesw) := case
 (malSW && pc.replaceSW
    && next (pc.pc) = loadMemSup
    && loc = ElectoralOffice)
   || (can_meesw && pc.pc =  openEnvelope
    && loc = PollingPlace)
   :1;
   1: can_mesw;
esac;
```

Beyond the above variables, we introduced control variables which model the execution flow and correspond to the introduction of malicious assets. For example, we have introduced a boolean variable `malSW` indicating that we deal with the introduction of malicious `electionSW`. Due to the fact that we do not want to restrict this variable in advance and that on the other hand the variable should be constant during the whole execution, we use the following trick of specifying `next (malSW):= malSW` without initialization. This means that we can choose the value of `malSW` for the introduction of malicious `electionSW` at random, but once chosen, the value does not change anymore.

After encoding the relevant information of asset-flows in NuSMV, the next activity consists of capturing and specifying the security properties using LTL/CTL. The security properties formalize the analysis goals discussed in Section 3.4. The e-voting domain is particularly challenging when it comes to the formalization of security properties. The higher level principles, in fact, derive directly from the laws and are typically meant to protect fundamental rights of voters, such as secrecy and anonymity. We do not have a formal machinery for the refinement of such principles into properties we can model check. In Weldemariam et al. (2009a) we show the approach we followed to refine and allocate such principles to systems and procedures for the e-voting machine we built.

Here suffice saying that an analysis of the requirements at different levels of abstraction allows domain experts to come out with a set of properties that have to be guaranteed or the attacks violating such properties. For instance, in the example we have shown (in the extended model) it is possible to implement at least three different attacks:

- the first attack consists of replacing the software which is sent to the polling stations. By reading the `password` with which the `electionSW` is encrypted and substituting a modified version of the software in the `MemorySupport`, it is possible for a malicious actor eventually to deliver a modified copy of the software to the polling station.
- The second attack consists of replacing the `PIN`. A malicious actor with access to the `PIN` code may substitute the `PIN` which is loaded in the `envelope`. Thus, a wrong `PIN` is delivered to the polling station which eventually causing a denial of service—namely, the voting functions cannot be activated by the polling officers.
- The third attack consists in deleting (or destroying) the `envelope` during transportation, possibly causing another denial of service.

We show two examples of properties that allow us to highlight such attacks.

i) Verifying a property about the delivery of election software. In this example, we want to check a generic property about the delivery of software to the polling station. We are interested in checking that: *"It is never the case that poll officers receive an altered election software"*. In other words, this property says that election software always remains "useful". The property is specified in CTL as:

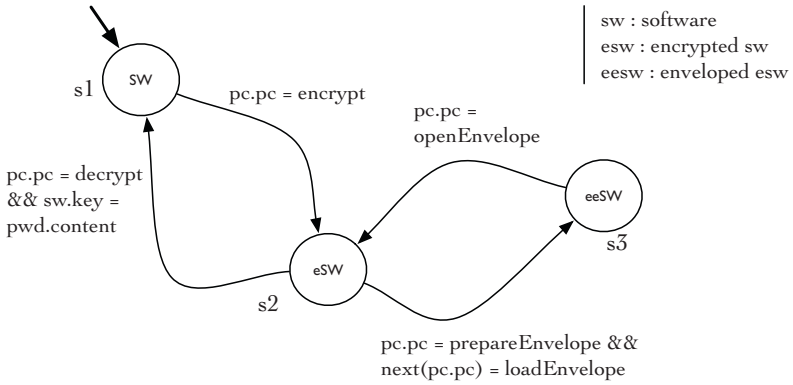

Fig. 7. A simple example of state transition model for `content` feature of `electionSW`.

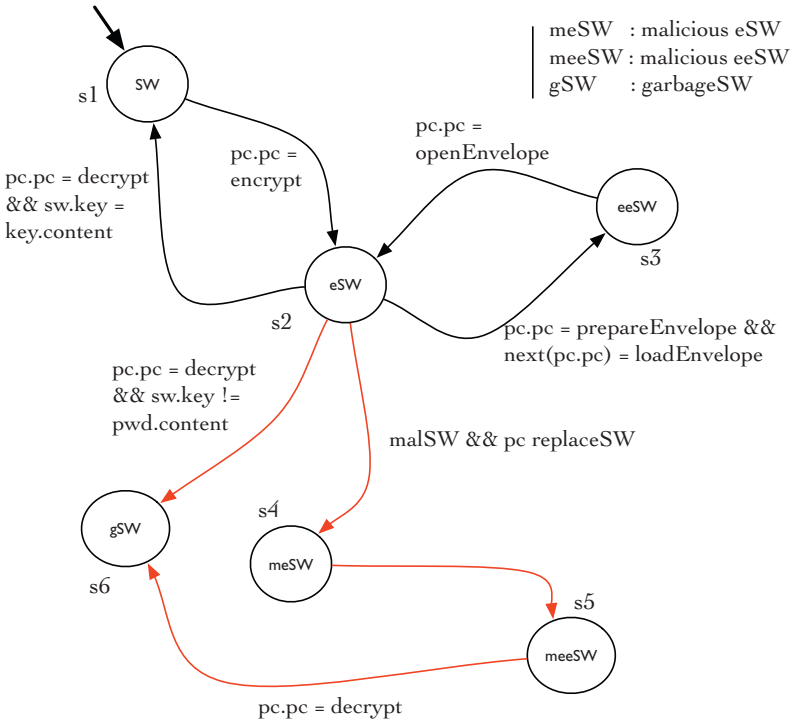

Fig. 8. An extension of Fig. 7 due to the injection of threat-actions.

```
AG ! (sw.content = garbageSW && sw.location =
      pollStation && PollOfficersActive )
```

When checking the above formula in NuSMV, it proves to be false, which is indicated by the counterexample shown in Table 2. Namely, it is possible to deliver a wrong software to the poll station if, for instance, at time t_2 a malicious encrypted software (`malSW`) is used to replace the nominal encrypted election software into the memory support (`replaceSW` action). Then, at time t_3`can_mesw` becomes true—that is, the attacks has succeeded.

It is easy to derive what are the possible scenarios in which an adversary can invalidate the election software, given the above sequence of actions in place. From the property failure above (at time t_4), we can say that after the replace attack action takes place, the content of `electionSW` is replaced by "`garbage`" right after loading it into the memory support. The replacement is not

Table 2
Example 1: A counterexample showing the alternation of election software at poll station.

	t_0	t_1	t_2	t_3	t_4	...	t_9	t_{10}	...
pc.pc	–	–	encrypt	loadMem	preEnv	...	openEnv	decrypt	
sw.conten	sw				garbage	...			
malSW	$\bot$		$\top$	$\bot$		...			
replaceSW	$\bot$		$\top$	$\bot$		...			
sw.can_mesw	$\bot$			$\top$		...			
sw.can_garbage	$\bot$				$\top$	...			
sw.is_sw	$\top$				$\bot$	...			

Table 3
Example 2: Denial of service attack counterexample.

	t_0	...	t_4	t_5	...	t_7	...	t_{10}	...
pc.pc		...	preEnv	loadEnv	...	openEnv	...		
malPiN	$\bot$	...	$\top$	$\bot$	...		...		
replacePiN	$\bot$	...	$\top$	$\bot$	...		...		
pinReady	$\bot$	...		$\top$	...		...		
can_mPPiN	$\bot$	...			...	$\top$	...		
pin.can_garbage	$\bot$	...			...		...	$\top$	

detected before step t_9, that is, when poll officers try to use it at the polling stations. In this way, the counterexample is used to highlight sequence of actions in which the software has been altered during the delivery process.

ii) Verifying a property about the Denial of Service attack. In this example, we are interested in checking a denial of service attack that could happen in a poll station. Note that our aim here is also deriving the possible sequence of actions that an adversary can take to cause this attack. The property of interest is that: *"It is never the case that poll officers get denial of service due to PIN code"*. Notice that the PINs are used by poll officers to activate the voting machines to activate voting functions. This property is expressed in CTL formula as:

```
AG ! (PIN.can_garbage
        &&  PIN.location = pollStation)
```

We give the above property to NuSMV to check that the property holds. However, the tool generates the counterexample depicted in Table 3, in which the PIN is replaced with a fake one, before being inserted in the package delivered to the polling stations, which in turns causes denial of service attack—namely, the poll officers unable to activate the voting machine due to wrong PIN.

5. Related work

Several strategies have been proposed in the literature to understand, model, and analyze business process models. Three aspects are central in these approaches. The first is the tools that are used for creating (business) process models. Second, notations used to represent the modeling elements and concepts. Third, techniques used for formally specifying and verifying how such models respect the intended goals.

With respect to the application of formal methods, there are a number of approaches focused on specifying and verifying business process models. These include automata, process algebra, and Petri Nets. Each equips with manual and/or automated analysis techniques. Automata based approaches are common, which comprise of a set of states, actions, transitions between states, and an initial state. Labels denote the transition from one state to another. The NuSMV and SPIN/Promela[5] (Mauw et al., 1998) input languages, for instance, are derived from automata to express system behavior in terms of transition systems.

A formal model for business process modeling and design is discussed in Koubarakis and Plexousakis (1999) and Koubarakis and Plexousakis (2000), which builds upon Loucopoulos and Kavakli (1995). Their formalization is based on the use of situation calculus (Levesque et al., 1998) and the concurrent logic programming language ConGolog (Giacomo et al., 2000) for representing knowledge about organizations and their processes. More specifically, their approach allows to develop the so-called enterprise model. It comprises five interconnected sub-models to formally describe different aspects of an organization. The core elements that constitute the enterprise model include actors with their roles, goals, process (distinguished between primitive and complex actions), enterprise entities, and constraints. A goal-oriented methodology for business process design is also outlined for developing a new business process. The ConGolog formal specification is developed as a set of sub-models to capture the new process from various viewpoints. Formal verification can be carried-out to check, for instance, whether each role responsibility is fulfilled and each constraint is maintained by the ConGolog procedures defined for each individual role.

The usage of model checking for verifying functional requirements on workflow specifications is discussed in Eshuis (2002) and Eshuis (2006). To specify workflows, the author used UML-ADs by defining a formal semantic for them in order to meet the semantics of the target specification language that is suitable for formal analysis. Two kinds of semantics are specifically introduced for the ADs, namely a requirements-level and an implementation-level semantic (Ch. 2 and 3, Eshuis, 2002). A number of translation rules have been defined to convert AD nodes (such as activities, objects, data and control flows) into NuSMV input language semantics, such that functional requirements about the business process models can be model checked. The authors present a tool in Eshuis and Wieringa (2004), so that the models specified in the ADs are translated into the NuSMV input language for model checking.

An approach for the specification and verification of artifact behaviors in business process models is presented in Bhattacharya et al. (2007), Gerede and Su (2007) and Gerede et al. (2007). The center of the approach is the artifact-centric operational modeling for constructing models with an appropriate formalism. The approach consists of three key constructs for the artifact-centric models: the business artifacts, business work descriptions, and repositories. These constructs define the operational modes of the modeling, where each of the construct is represented formally. Differently from the verification goal discussed in Koubarakis and Plexousakis (2000), which focuses on verifying properties of business processes, with this approach the verification goal is checking whether models satisfy certain artifact properties. Examples of properties they claim to verify are reachability and arrival of artifacts into a repository in bounded domains. The authors also present a logic language based on CTL named Artifact Behavior Specification Language (ABSL). ABSL allows to specify the lifecycle properties of artifacts, where the lifecycle of an artifact type specifies the possible sequencing of services that can be applied to an artifact of this type as it goes through the business process. A similar concept is presented in Deutsch et al. (2009) and Fritz et al. (2009). However, the authors do not show how to perform automated analysis or verification nor hint the integration of their approach with existing formal verification tools. Mostly their focus is in constructing business process models with correct creation and termination of artifacts during their lifecycle, as opposed to the paradigm of organizing and modeling workflow or business processes around relatively flat process-centric models. Additionally, they hardly speak about security analysis.

We have discussed various works demonstrating their usage scenarios insofar as they can used for modeling, specifying, and analyzing business processes and workflows. Unfortunately, the attempt to model procedures, as well as to perform formal analysis in favor of the public administration (PA) is not satisfactory. The

[5] http://spinroot.com/spin/whatispin.html.

advantages and difficulties related to the (re-)engineering of PA can be found in Wastell et al. (1994) and Alpar and Olbrich (2005).

Our work builds upon and complements existing approaches by widening the scope of the analysis and by taking into account aspects related to threats, procedures, and interaction of the system with its environment. Namely, we developed a generic assets-centered methodology for the analysis. Using the approach, we have shown how to encode executable models describing the nominal procedures using NuSMV input language – mainly from the dynamic view of process models – and to extend such models with possible attacks, by assuming all possible combination of attacks can be made at each execution step. We have tested the applicability of the proposed approach using core use cases taken from the ProVotE e-voting system. Among the advantages of the approach the possibility of reasoning about threat composition (e.g., coordinated attacks; complex and unforeseen attacks resulting from the composition of elementary threats), and the possibility of reasoning about evolution of assets over time.

To our knowledge, the usage of formal methods in e-voting systems is relatively new. Various approaches are aimed at providing a higher level of assurance to the secure development of e-voting systems through formal techniques. Existing works in this area present formal specification and verification of an e-voting system at different level of abstractions. In this area the work closest in sprit to ours can be grouped in two closely related directions: verifying cryptographic protocols (e.g., Kremer and Ryan (2005), Delaune et al. (2009), Cansell et al. (2007), Sampigethaya and Poovendran (2006) and Backes et al. (2008)) and verifying system behavior (e.g., Tiella et al. (2006), Weldemariam et al. (2009b), Sturton et al. (2009) and Weldemariam et al. (2010)). Some of these works selectively apply formal modeling techniques where these techniques add rigor to the development or help assessing an e-voting systems. By contrast, other works use e-voting as a case study to improve verification techniques or to come out with general requirements to help build a new generation of better e-voting systems. However, none of these works focus on the aspects related to procedures in their modeling and analysis. In that regard, this work presented in this article complements the above-mentioned works by widening the scope of the analysis to processes.

6. Summary and conclusion

In this article, we have demonstrated the importance of procedural security to tackle the security risks associated with the e-voting systems and eventually strengthen the level of security. Since asset mobility, state, evolution, and the context in which asset instances are used in e-voting are an inherent challenge, we have developed assets-centered methodology for procedural security analysis. The methodology can be used to analyze and evaluate the impact of threats, and consequently to come out with a set of (security) procedural requirements that guarantee the desired level of protection. We presented the approach by introducing the guidelines we follow for modeling, and encoding the electoral procedures and hinted its usage through an example.

The outputs of the analysis conducted using model checking helped us in understanding possible threats and composition of threats on critical assets. Certain patterns can be combined to understand the level of coordination in order to understand undetectable attacks. For instance, whenever the assets necessary to encrypt and sign the software are put together (either by the process or by malicious actors), it is possible to deliver an arbitrary software to the polling stations. Such results are also foundations to familiarize actors with the possible procedural threats and attacks that can happen in elections. Additionally, this kind of analysis and reasoning has the potential to serve as a trust building measure in the new e-voting processes.

It must be clear that the construction of the extended model, whose generation can be automated, is currently performed by hand using the methodology and the translation strategies we described. The analysis approach we took, however, is very similar to that of FSAP/NuSMV-SA (Bozzano and Villafiorita, 2007), for safety analysis, whereby a system specification is "enriched" with information about faults and analyzes are carried out to understand the effect and impact of faults on safety requirements expressed in the form of LTL/CTL formulae. Analogously to what happens in safety analysis when analyzing, e.g., the loss of a critical functions, enhancing the procedures results in reducing the probability of an attack or making the attack more complex, rather than eliminating it.

The examples, although trivial, show how – by reasoning on the extended model – it is possible to explicitly represent the attacks that can be carried out, determine what assets are needed, when they are needed, and who can carry the attacks. Similarly to what happens in model checking, we do not provide any quantitative information about the likelihood of the attacks. However, even in this simple case, we believe that the output of the attacks can provide experts the information and the requirements to enhance the current procedures, to eliminate certain attacks or, at least, to make them more difficult to implement.

Some research issues remain open. In our future work, we will consider multiple instance of an asset in the analysis. More threat-actions should be considered in the case study. In fact, we also need to precisely isolate generic threat-actions from domain-specific ones, so that we can come out with generic (and domain specific) libraries. Furthermore, we are currently investigating the possibility of automating the threat injection on top of the FSAP/NuSMV-SA platform. Specifically, we plan to develop a tool in order to integrate and/or extend the current usage scenario of the platform, with emphasis on procedural security analysis. Additionally, we will conduct further research on the possibilities of integrating our approach in the Common Criteria (Common Criteria, 2007) methodology and on the evaluation procedures discussed in Volkamer and McGaley (2007) and Volkamer (2009).

Acknowledgments

The research reported in this paper was done while K. Weldemariam was with the University of Trento, supported by Fondazione Bruno Kessler. An earlier version of this paper has appeared in Weldemariam and Villafiorita (2008a,b). This paper extends and presents the scope of procedural security analysis.

References

Adida, B., 2006. Advances in Cryptographic Voting Systems, Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Alpar, P., Olbrich, S., 2005. Legal Requirements and Modelling of Processes in e-Government, Electronic Journal of e-Government 3.

Basin, D.A., Doser, J., Lodderstedt, T., 2003. Model driven security for process-oriented systems. In: SACMAT, 100–109.

Booch, G., Rumbaugh, J., Jacobson, I., 2005. Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series), Addison-Wesley Professional.

Bozzano, M., Villafiorita, A., 2007. The FSAP/NuSMV-SA safety analysis platform. International Journal Software Tools Technology Transfer 9 (1), 5–24.

Bishop, M., 2002. Computer Security Art and Science. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Balzarotti, D., Banks, G., Cova, M., Felmetsger, V., Kemmerer, R., Robertson, W., Valeur, F., Vigna, G., 2008. Are your votes really counted?: Testing the Security of real-world electronic voting systems. In: Proceedings of the 2008 International Symposium on Software Testing and Analysis , ACM, New York, NY, USA, pp. 237–248.

Braynov, S., Jadiwala, M., 2003. Representation and analysis of coordinated attacks. In: Proceedings of the 2003 ACM workshop on Formal methods in security engineering , ACM Press, New York, NY, USA, pp. 43–51.

Bhattacharya, K., Gerede, C.E., Hull, R., Liu, R., Su, J., 2007. Towards formal analysis of artifact-centric business process models. In: Gustavo Alonso, Peter Dadam, Michael Rosemann (Eds.), BPM, vol. 4714 of Lecture Notes in Computer Science, Springer, 288–304.

Backes, M., Hritcu, C., Maffei, M., 2008. Automated verification of remote electronic voting protocols in the applied pi-calculus. In: Proceedings of the 2008 21st IEEE Computer Security Foundations Symposium , IEEE Computer Society, Washington, DC, USA, pp. 195–209.

Common Criteria, 2007. Common Criteria for Information Technology Security Evaluation, http://www.commoncriteriaportal.org/.

Cimatti, A., Clarke, E., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., Tacchella, A., 2002. NuSMV 2: an open source tool for symbolic model checking. In: Computer Aided Verification , Lecture Notes in Computer Science, Springer, 241–268.

Cansell, D., Gibson, J.P., Mery, D., 2007. Formal verification of tamper-evident storage for e-voting. In: Proceedings of the Fifth IEEE International Conference on Software Engineering and Formal Methods , IEEE Computer Society, Washington, DC, USA, pp. 329–338.

Deutsch, A., Hull, R., Patrizi, F., Vianu, V., 2009. Automatic verification of data-centric business processes. In: Proceedings of the 12th International Conference on Database Theory , ACM, New York, NY, USA, pp. 252–267.

Delaune, S., Kremer, S., Ryan, M., 2009. Verifying Privacy-Type Properties of Electronic Voting Protocols, Journal of Computer Security 17 (4), 435–487, ISSN 0926-227X.

Eshuis, R., 2006. Symbolic model checking of UML activity diagrams. ACM Transactions on Software Engineering and Methodology 15 (1), 1–38.

Eshuis, R., Wieringa, R., 2004. Tool support for verifying UML activity diagrams, IEEE Transaction on Software Engineering 30 (7).

Eshuis, R., 2002. Semantics and Verification of UML Activity Diagrams for Workflow Modelling, Ph.D. thesis, Centre for Telematics and Information Technology (CTIT) University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands.

Fovino, I.N., Masera, M., 2006. Through the description of attacks: a multidimensional view. In: SAFECOMP, Lecture Notes in Computer Science , Springer-Verlag, 15–28.

Fritz, C., Hull, R., Su, J., 2009. Automatic construction of simple artifact-based business processes. In: ICDT '09: Proceedings of the 12th International Conference on Database Theory , ACM, New York, NY, USA, pp. 225–238.

Gerede, C.E., Su, J., 2007. Specification and verification of artifact behaviors in business process models. In: Bernd J. Krämer, Kwei-Jay Lin, Priya Narasimhan (Eds.), ICSOC, vol. 4749 of Lecture Notes in Computer Science, Springer, 181–192.

Giacomo, G.D., Lespérance, Y., Levesque, H.J., 2000. ConGolog, a concurrent programming language based on the situation calculus.

Gerede, C.E., Bhattacharya, K., Su, J., 2007. Static analysis of business artifact-centric operational models. In: Proceedings of the IEEE International Conference on Service-Oriented Computing and Applications , IEEE Computer Society, Washington, DC, USA, pp. 133–140.

Hogganvik, I., 2007. A Graphical Approach to Security Risk Analysis, Ph.D. thesis, Faculty of Mathematics and Natural Sciences, University of Oslo.

Hsiung, P.-A., Chen, Y.-R., Lin, Y.-H., 2007. Model checking safety-critical systems using safecharts. IEEE Transactions on Computers 56 (5), 692–705.

Hull, R., 2008. Artifact-centric business process models: brief survey of research results and challenges. In: Proceedings of the OTM 2008 Confederated International Conferences, CoopIS, DOA, GADA, IS, and ODBASE 2008. Part II on On the Move to Meaningful Internet Systems , Springer-Verlag, Berlin, Heidelberg, pp. 1152–1163.

Jones, D.W., 2003. The Evaluation of Voting Technology, Chap. 1, Advances in Information Security. Kluwer Academic, 3–16.

Kohno, T., Stubblefield, A., Rubin, A.D., Wallach, D.S., 2004. Analysis of an electronic voting system. In: IEEE Symposium on Security and Privacy 0 , p. 27.

Koubarakis, M., Plexousakis, D., 2000. A formal model for business process modeling and design. In: Wangler, B., Bergman, L. (Eds.), CAiSE, Lecture Notes in Computer Science. Springer, pp. 142–156.

Koubarakis, M., Plexousakis, D., 1999. Business process modelling and design—a formal model and methodology. BT Technology Journal 17 (4), 23–35.

Kremer, S., Ryan, M., 2005. Analysis of an electronic voting protocol in the applied pi calculus. In: ESOP , pp. 186–200.

Lambrinoudakis, C., Kokolakis, S., Karyda, M., Tsoumas, V., Gritzalis, D., Katsikas, S., 2003. Electronic voting systems: security implications of the administrative workflow. In: DEXA '03: Proceedings of the 14th International Workshop on Database and Expert Systems Applications , IEEE Computer Society, Washington, DC, USA, p. 467.

Lam, V.S., Padget, J.A., 2004. Symbolic model checking of UML statechart diagrams with an integrated approach. In: Proceedings of the 11th IEEE International Conference and Workshop on Engineering of Computer-Based Systems , IEEE Computer Society, pp. 337–347.

Loucopoulos, P., Kavakli, E., 1995. Enterprise modelling and the teleological approach to requirements engineering. International Journal of Cooperative Information System 4 (1), 45–79.

Levesque, H.J., Pirri, F., Reiter, R., 1998. Foundations for the situation calculus. Electronic Transaction in Artificial Intelligence 2, 159–178.

Manian, R., Bechta, J.D., Coppit, D., Sullivan, K.J., 1998. Combining various solution techniques for dynamic fault tree analysis of computer systems. In: IEEE International Symposium on High-Assurance Systems Engineering , IEEE Computer Society, Washington, DC, USA, pp. 21–28.

Mauw, S., Mateescu, R., Janssen, W., 1998. Verifying business processes using spin. In: Proceedings of the International SPIN Workshop , pp. 21–36.

Pnueli, A., 1977. The temporal logic of programs. In: FOCS , pp. 46–57.

Sastry, N., Kohno, T., Wagner, D., 2006. Designing voting machines for verification. In: Proceedings of the 15th conference on USENIX Security Symposium , vol. Volume 15, USENIX Association, Berkeley, CA, USA.

Sastry, N.K., 2007. Verifying Security Properties in Electronic Voting Machines, Ph.D. thesis, EECS Department, University of California, Berkeley, URL http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007–61.html.

Sampigethaya, K., Poovendran, R., 2006. A framework and taxonomy for comparison of electronic voting schemes. Computers & Security 25 (2), 137–153.

Sturton, C., Jha, S., Seshia, S.A., Wagner, D., 2009. On voting machine design for verification and testability. In: ACM Conference on Computer and Communications Security , pp. 463–476.

Tiella, R., Villafiorita, A., Tomasi, S., 2006. Specification of the Control Logic of an eVoting System in UML: the ProVotE experience. In: Proceedings of the 5th International Workshop on Critical Systems Development Using Modeling Languages, 84–94ISSN 0809–1021.

Volkamer, M., McGaley, M., 2007. Requirements and evaluation procedures for evoting. In: Proceedings of the The Second International Conference on Availability , Reliability and Security, IEEE Computer Society, Washington, DC, USA, pp. 895–902.

Volkamer, M., 2009. Evaluation of electronic voting: requirements and evaluation procedures to support responsible election authorities. In: Springer Publishing Company, Incorporated.

Villafiorita, A., Weldemariam, K., Tiella, R., 2009. Development, Formal Verification, and Evaluation of an E-Voting System With VVPAT, IEEE Transactions on Information Forensics and Security 4 (4).

Weldemariam, K., Villafiorita, A., Mattioli, A., 2009a. Managing requirements for e-voting systems: issues and approaches motivated by a case study. In: Proceedings of the first International Workshop on Requirements Engineering for E-voting Systems In conjunction with the 17th IEEE International Requirements Engineering Conference (RE'09) , IEEE.

Wastell, D., White, P., Kawalek, P., 1994. A methodology for business process redesign: experiences and issues. Journal of Strategic Information Systems 3, 23–40.

Weldemariam, K., Kemmerer, R.A., Villafiorita, A., 2009b. Formal analysis of attacks for e-voting system. In: Forth International Conference on Risks and Security of Internet and Systems, IEEE.

Weldemariam, K., Kemmerer, R.A., Villafiorita, A., 2010. Formal Specification and Analysis of an e-Voting System. In: The 5th International Conference on Availability Reliability and Security, IEEE Computer Society.

Weldemariam, K., Villafiorita, A., 2008a. Formal procedural security modeling and analysis. In: Proceedings of 3rd International Conference on Risks and Security of Internet and Systems , IEEE, pp. 249–254.

Weldemariam, K., Villafiorita, A., 2008b. Modeling and Analysis of Procedural Security in (e)Voting: The Trentino's Approach and Experiences, In: EVT/USENIX, USENIX Association, Berkeley, CA, USA.

Xenakis, A., Macintosh, A., 2004. Procedural security analysis of electronic voting. In: Proceedings of the 6th international conference on Electronic commerce , ACM Press, New York, NY, USA, pp. 541–546.

Xenakis, A., Macintosh, A., 2005. Procedural security and social acceptance in e-voting. In: Proceedings of the Proceedings of the 38th Annual Hawaii International Conference on System Sciences - Track 5 , IEEE Computer Society, Washington, DC, USA, 118.1.

Xu, D., Nygard, K., 2005. A threat-driven approach to modeling and verifying secure software. In: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering , ACM Press, New York, NY, USA, pp. 342–346.

Yee, K.-P., 2007. Extending prerendered-interface voting software to support accessibility and other ballot features. In: EVT'07: Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology, USENIX Association , Berkeley, CA, USA, 5–5.

Komminist Weldemariam received the BS degree in Computer Science from Addis Abeba University, Ethiopia, in 2003, the MS degree in Computer Science and Engineering from Indian Institute of Technology, Bombay, India, in 2006, and PhD degree in Computer Science from the University of Trento, Italy, in 2010. He is currently a post-doctorate follow at the Fondazinoe Bruno Kessler, Italy. He has been visiting scholar at the Computer Security Group of the University of California in Santa Barbara. His research interests include Software Engineering, Security, Electronic Voting Systems, and ICT4G. He is a member of the IEEE.

Adolfo Villafiorita received the MS degree from the University of Genoa, Italy, in 1993, and the PhD degree from the University of Ancona, in 1997, both in Computer Science. He is a senior researcher in the Center of Information Technology at the Fondazione Bruno Kessler, Italy. His current interests include ICT4G, software and system engineering, security, formal methods, and safety analysis. He has participated and led several industrial projects related to the development of safety critical applications in the railway, aerospace, and e-Government sector.