A Systematic Process-Model-Based Approach for Synthesizing Attacks and Evaluating Them

Huong Phan* *hphan@cs.umass.edu*
George Avrunin* *avrunin@cs.umass.edu*
Matt Bishop† *mabishop@ucdavis.edu*
Lori A. Clarke* *clarke@cs.umass.edu*
Leon J. Osterweil* *ljo@cs.umass.edu*

Abstract

This paper describes a systematic approach for incrementally improving the security of election processes by using a model of the process to develop attack plans and then incorporating each plan into the process model to determine if it can complete successfully. More specifically, our approach first applies fault tree analysis to a detailed election process model to find process vulnerabilities that an adversary might be able to exploit, thus identifying potential attacks. Based on such a vulnerability, we then model an attack plan and formally evaluate the process's robustness against such a plan. If appropriate, we also propose modifications to the process and then reapply the approach to ensure that the attack will not succeed. Although the approach is described in the context of the election domain, it would also seem to be effective in analyzing process vulnerability in other domains.

1 Introduction

An election is a complex process composed of many steps, some executed by computers, and others by election officials, voters, and other people. Improving election security has been an important research area. Past work has focused largely on the security and accuracy of the computers (e.g., [1, 3, 10, 28, 31, 37]) or on the cryptographic protocols (e.g., [15, 7]) used in election processes. Our focus is slightly different: we examine the overall process itself, attempting to improve its security. The complexity of the election process, coupled with the need for formal, rigorous analysis, makes examining such a process and determining how to modify it to improve its security a daunting task.

Our earlier work on evaluating election processes focused on identifying potential points of failure [33]. This paper extends that work by modeling potential attacks, then integrating each attack with a model of the election process that is to be attacked and determining the changes to the process model that will suffice to thwart that attack. Specifically, we apply fault tree analysis (FTA) to a precisely-specified, detailed election process model to automatically locate vulnerabilities that an adversary might exploit. Next, we work with election officials to develop models of specific plans that might be used to exploit these vulnerabilities. We then use model checking to formally evaluate the election process's robustness against each such plan. Experts can propose modification to the process, if it is deemed insufficiently robust. After these modifications are reflected in the process model, the process model can then be analyzed again to confirm that it indeed is more robust. This improved process model can then be further improved by considering additional attacks, and repeating the improvement procedure just described.

Thus, the contribution of this paper is a systematic and semi-automated approach that uses rigorously defined process models and rigorous analyses to support continuously improving the security of election processes. This approach, while requiring significant human participation and insight, also exploits automated computer analyses (FTA and model checking) to effect much faster and more thorough analyses than could be achieved solely through human effort. By looking for vulnerabilities *before* the occurrence of attacks that seek to exploit them, our work complements existing work that focuses on detecting attacks such as intrusions that occur during or after the fact [32, 6, 25]. Penetration testing also tries to identify vulnerabilities beforehand, but typically does so by informal, non-rigorous means (e.g., [36, 19]). Other work either does not consider processes (such as work on analyzing computer network vulnerabilities [13, 30]) or does not explicitly or formally define them [24]. The

*Laboratory for Advanced Software Engineering Research (LASER), Department of Computer Science, University of Massachusetts Amherst.

†Computer Security Laboratory, Department of Computer Science, University of California, Davis

formal analysis of rigorously-defined processes allows us to proactively identify and remove vulnerabilities before attacks that attempt to exploit them actually occur. It also enables us to analyze modifications to determine their effectiveness in thwarting the attacks without actually implementing the modifications, thereby saving time and money, and enabling determination of possible adverse impacts of the changes. Election officials can directly use the results of this work to make elections more secure and safer to change as new technologies, laws, and regulations need to be enforced.

The remainder of the paper is organized as follows. Section 2 presents an overview of the approach. Section 3 introduces the process modeling language Little-JIL. Section 4 reviews FTA and presents as examples two attack scenarios developed to exploit the vulnerability of a specific election process allowing an unqualified voter to get a regular ballot. Section 5 describes how we model the attack using Little-JIL and then compose the resulting attack model with the original process model before using model checking to analyze their composition. Section 6 describes related work and Section 7 describes limitations of this approach along with some of our plans to address these limitations. Section 8 summarizes the benefits.

2 Approach overview

Figure 1 shows an overview of our incremental process security improvement approach. It consists of two phases: identifying potential attacks on a process and then, for each selected credible attack, analyzing the process's robustness in the presence of that specific attack.

In the first phase, we use a process modeling language with rigorous semantics to model a real-world process so that we can reason about it. We hypothesize that an adversary might attack the process, creating an undesired state, called a *hazard*. Using a specification of the hazard, we apply FTA to the process model to find out how that hazard might occur. FTA produces a set of *minimal cut sets*, each of which contains a set of events that causes the hazard. An earlier paper [33] described in detail the application of FTA to election process models to determine the possibility of the occurrence of hazards. Here, we build upon these understandings to propose *attack plans*—models of specific ways in which an attack might cause the specified hazard to occur, and an *"attack-always-fails"* property that defines precisely what it means for the modeled attack to never succeed.

In the second phase, we compose the model of the attack with the model of the process being attacked to produce a *composed process model*. This composed model represents the execution of both the election process and the attack, represented as concurrently executing subprocesses. We perform model checking on the composed model to check if the *"attack-always-fails"* property is satisfied by the composed model, thus determining whether the modeled attack can succeed. If the property does not hold, it is possible for the attack to succeed, and the model checker will produce a trace of how this could happen. Election security experts can then suggest modifications to the election process that should succeed in thwarting the attack. We can then reapply our analyses to evaluate the effectiveness of the suggested modifications as countermeasures. Once an improvement has been shown to be effective, we can continue our process improvement by examining another minimal cut set produced by FTA, or proposing and reasoning about additional hazards and attacks.

As the analysis of the process relies on the process model, that model must reflect reality as closely as possible. The process model in this paper describes the election process in Yolo County, California (http://www.yoloelections.org/). It was developed by interviewing Freddie Oakley, the Yolo County Clerk-Recorder (and head election official) and her chief deputy, Tom Stanionis, both of whom developed the election policies and procedures and supervise their implementation. In addition to interviewing experts to validate the model, we also used model checking as a tool for uncovering process modeling defects. A *property* is a requirement that the process must adhere to. If model checking determines that a property is violated, we then work with domain experts to review the process model to determine whether the process itself does not conform to the property, or whether the property is stated incorrectly, or whether the process model has a defect. In the latter case, the process model must be corrected with the assistance of the domain experts. We iteratively improved the model until the domain experts felt that it was an accurate description, as described in Simidchieva et al. [33], and is the model upon which this current work is based.

It should be noted that the process model is a living document—if at some point in the future the process changes or some details need to be added, it is necessary that the process model be modified and re-evaluated with the existing hazard and property specifications. Thus, this process improvement loop may need to continue indefinitely as election processes evolve to respond to new laws and technology or as new hazards or attacks are envisioned.

3 Process modeling with Little-JIL

Our approach relies on models of real-world processes that are represented in a modeling language with expressive and well-defined semantics so that the resulting models can closely capture the processes and can

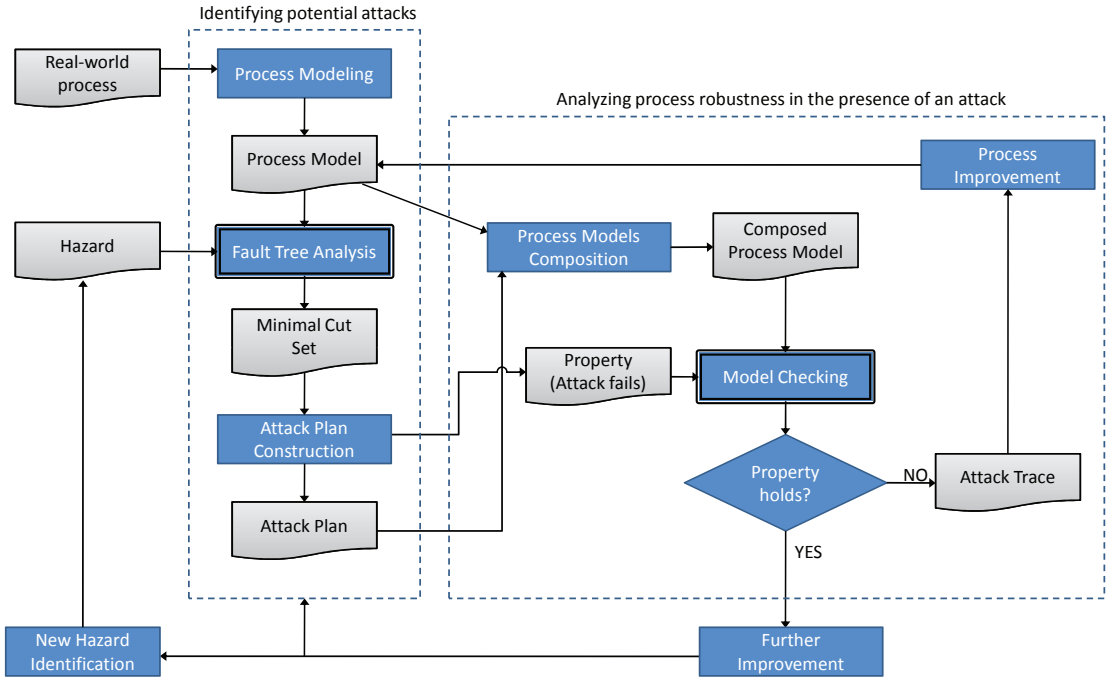

Figure 1: The systematic process-model-based approach. Fully automated steps are in double-edged boxes.

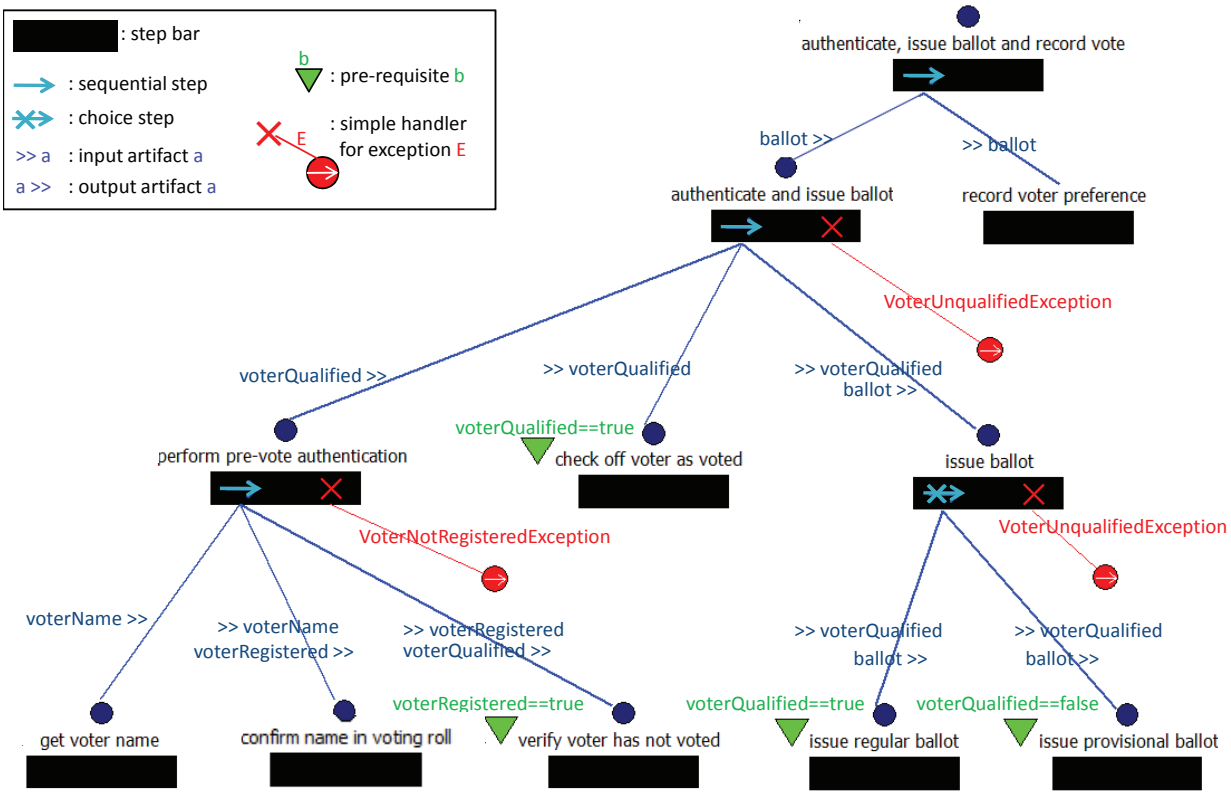

Figure 2: Little-JIL process model: Elaboration of step “authenticate, issue ballot, and record vote”

be analyzed rigorously. It also relies upon the use of a notation that makes process models readily understandable by election officials. This section provides an introduction to Little-JIL, a visual process modeling language with such formally defined semantics [39].

Little-JIL represents a process as a hierarchy of steps carried out by agents that may be humans, hardware devices, or software systems. A Little-JIL model consists of activity diagrams showing the hierarchical decomposition of steps, a specification of the artifacts manipulated by the steps, and a specification of the agents and resources needed to perform the steps.

A Little-JIL step is a specification of a unit of work assigned to an agent in the process. A step may be decomposed into sub-steps. A leaf step has no sub-steps and its behavior depends entirely on its assigned agent. A non-leaf step's behavior consists of the behaviors of its sub-steps and their order of execution. For example, the model reflecting the election process used by Yolo County, California consists of the root step "conduct election", which is decomposed into sub-steps representing activities such as the preparations made before election day, the conduct of the election at a single precinct, the counting of ballots, and the post-election canvass. These sub-steps in turn are decomposed into steps that specify lower levels of detail. Figure 2 shows the part of the elaboration of "authenticate, issue ballot, and record vote"—one of the steps in the Yolo County election process. This step has two sub-steps, "authenticate and issue ballot" and "record voter preference", which are executed sequentially (denoted by the arrow on the step bar).

Each step may contain a specification of pre-requisites that must be satisfied before an agent can begin the work (e.g., `voterQualified==true` is the pre-requisite of the step "issue regular ballot"), and post-requisites to check that the work was completed correctly (not present in this example).

A step may also specify how to handle exceptions that occur during the execution of its descendant steps. An exception handler can be a Little-JIL step, capable of defining an arbitrarily complex response to an exception, or can be a *simple handler* that specifies only how execution should continue after the occurrence of the exception. In the example in Figure 2, the X on the "authenticate and issue ballot" step bar connects to a simple handler that handles `VoterUnqualifiedException` that might be thrown by the sub-step "check off voter as voted". In this example, the simple handler specifies that the sub-step throwing the exception will be terminated, and process execution moves on to the next step in line ("issue ballot"—which in turns invokes "issue regular ballot" or "issue provisional ballot" depending on the value of the artifact `voterQualified`).

Each Little-JIL step has an artifact declaration defining the artifacts it will be accessing and/or providing. Artifacts are generally passed through the coordination hierarchy between steps and their sub-steps. For example, the `ballot` artifact is output from the step "authenticate and issue ballot" to its parent step "authenticate, issue ballot and record vote", which then passes it down as the input into the sub-step "record voter preference".

A Little-JIL process model also includes agent specifications. Each step specifies the kind of agent that is to be assigned to be responsible for the execution of that step. *Voter* and *Election Official* are two specifications of agents in the election process model (not shown in the diagram).

More details about Figure 2 are discussed later in section 4.2 where we explain the application of FTA to the process model.

4 Identifying process vulnerabilities using FTA

This section presents FTA and describes the application of FTA to the Yolo County election process model to identify process vulnerabilities and create attack scenarios that might exploit the vulnerabilities to cause the hazard that an unqualified voter receives a regular ballot.

4.1 FTA overview

FTA is a deductive, top-down analytical technique used in a variety of industries [9, 35, 12, 4] to study hazards. In an election process, an example of a hazard is that "an unqualified voter gets a regular ballot". With FTA, one first postulates the possibility of a hazard, and then attempts to find out which events in the process could combine to cause the actual occurrence of the hazard. Given the hazard, FTA produces a fault tree, a graphical model of all the various combinations of events that can lead to the hazard.

A fault tree consists of two basic elements: *events* and *gates*. At the top (root) of the fault tree is the hazard. In the fault tree, *intermediate events* are elaborated further, and *primary events* are not. Events are connected to each other by Boolean-logic gates. A gate connects one or more lower-level input events to a single higher-level output event. There are three types of gates:

- AND gates: the output event occurs if and only if all the input events occur, implying that the occurrence of all of the input events causes the output event;
- OR gates: the output event occurs if and only if at least one of the input events occurs, implying that the occurrence of any of the input events causes the output event; and

- NOT gates: the output event occurs if and only if the (only) input event does not occur.

Figure 3 shows a fault tree example with the top event, or hazard, *Artifact "ballot" to "record voter preference" is wrong*[1]. An OR gate connects this event with another lower-level event, *Artifact "regularBallot" is wrong when step "issue regular ballot" is completed*, meaning that the higher-level event occurs if the lower-level events occurs. With further elaboration, the event *Artifact "regularBallot" is wrong when step "issue regular ballot" is completed*, which occurs if and only if both of the two lower-level events occur, is connected to the lower-level events through an AND gate. The event *Step "get voter name" produces wrong "voterName"* is a primary event so it is not elaborated further in the fault tree. This fault tree is an abstract view of the larger fault tree resulting from the application of FTA to the Yolo County process model with the hazard "an unqualified voter gets a regular ballot", which is discussed further in the next section 4.2.

A *cut set* is a set of *event literals* such that the occurrence of all the event literals is a sufficient condition for the hazard to occur. An event literal is either a primary event or the negation of a primary event. A cut set is considered *minimal* if, when any of its event literals is removed, the resulting set is no longer a cut set.

A minimal cut set (MCS) indicates one potential process vulnerability, which might be a flaw or weakness in the process's design, implementation, or operation and management, that could be exploited to allow a hazard to occur. An MCS with one element represents a *single point of failure*. The probability of a hazard occurring can be calculated if sufficient information about the probabilities of event literals in the MCSes is available.

Many software tools, commercial as well as open-source, facilitate the manual construction of fault trees. When fault trees become large, which they typically do, manual construction, even with such tool support, may be error-prone and time-consuming. There have been attempts to generate fault trees automatically, for example from source code written in Ada [18]. We developed a process-driven FTA tool to automate fault tree construction and MCS calculation from process models written in sufficiently precisely-defined languages [4]. Thus, for example, given a process model written in the Little-JIL language, and a hazard specification, this tool constructs a fault tree and then calculates its MCSes.

[1] Section 4.2.1 will explain in detail how this top event corresponds to "an unqualified voter gets a regular ballot".

4.2 Example: can an unqualified voter get a regular ballot?

In this section we present an example of applying FTA to the Yolo County election process model, including the specification of a hazard of interest, the derived fault tree and its MCSes, and the multiple interpretations of one MCS that resulted in developing multiple attack scenarios.

4.2.1 The hazard and its specification

One of the requirements of an election process is that only qualified voters may vote. By definition, a "qualified voter" at the time the voting takes place is one who has registered and not cast a ballot. By applying FTA to the election process model with a representation of the hazard "an unqualified voter gets to vote" we expect to expose vulnerabilities in the process that attackers might exploit to allow unqualified voters to cast ballots.

The process-driven FTA tool allows a fault tree hazard to be modeled as an artifact being either incorrect input to, or incorrect output from, a step[2]. Therefore, we model the hazard of interest as an incorrect artifact provided or accessed by a step in the process. With respect to the hazard "an unqualified voter gets to vote", the *ballot* is the artifact of interest in the Yolo County's election process model. The step "record voter preference" is where a voter casts the ballot. This step takes the artifact *ballot* as input (see Figure 2). Thus the hazard specified with this tool is: *Artifact "ballot" to "record voter preference" is wrong*.

It is important to note that this fault tree hazard models two different cases, namely *"an unqualified voter getting a regular ballot"* and *"a qualified voter not getting a regular ballot"*. Thus some analysis skill is required to model a fault tree hazard, and to interpret the MCSes resulting from FTA. We address this issue further in the next subsection, and in Section 8.

4.2.2 The derived fault tree and MCSes

With the hazard defined as above, the FTA tool produces a fault tree with more than 100 nodes. At the top of the tree is the hazard *Artifact "ballot" to "record voter preference" is wrong*.

Based on that fault tree, the FTA tool computes 11 MCSes having cardinalities ranging from 2 to 4; there are no single points of failure. Although an MCS contains event literals that lead to the occurrence of the hazard, understanding how the MCS events could occur in the process and making sense out of the MCS are non-trivial. It is worth noting that only 5 of those 11 MCSes consist of events that cause the process hazard of interest,

[2] An artifact may be incorrect for a number of reasons such as being of wrong type or containing incorrect information. At this point we do not distinguish between these cases.

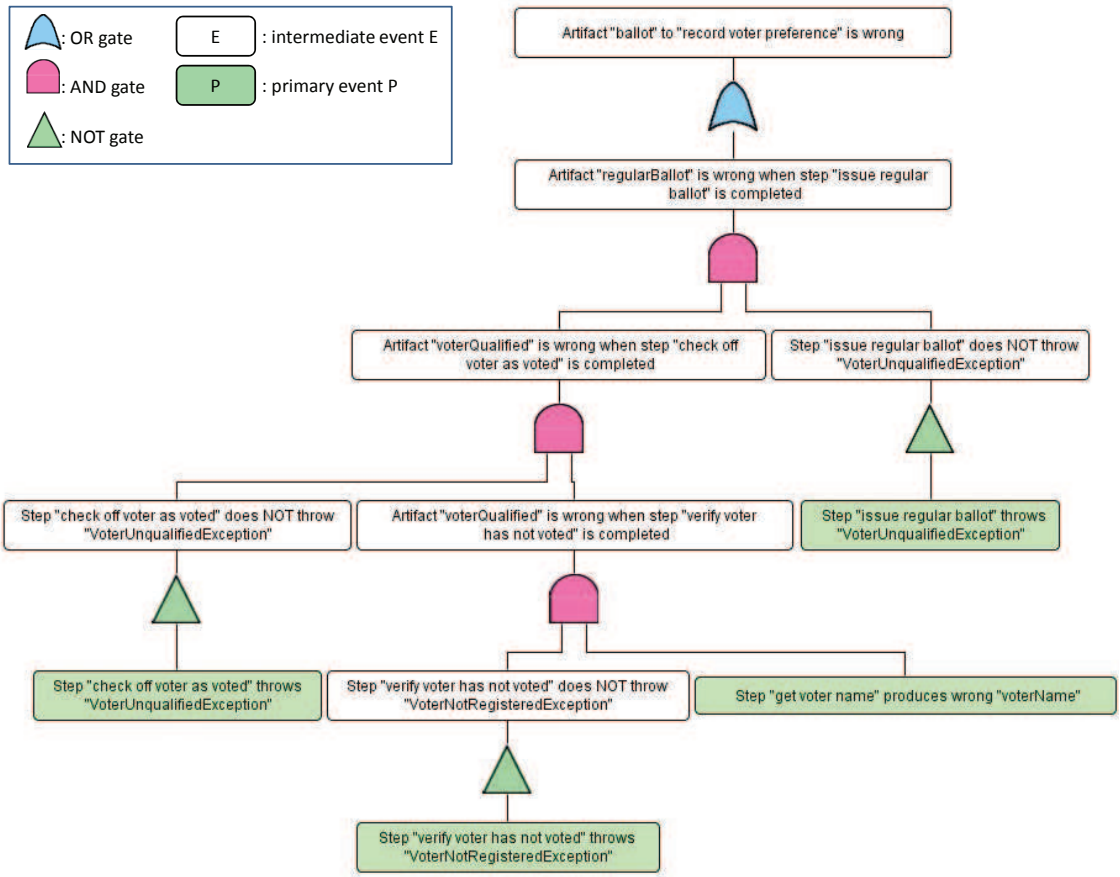

Figure 3: Abstract view of the Yolo fault tree containing only the events relevant to one example MCS

"an unqualified voter gets a regular ballot". Another 5 MCSes consist of events that cause the *"a qualified voter does not get a regular ballot"* process hazard. The remaining MCS indicates a wrong ballot being issued due to an incorrect voting roll.

To illustrate, here we describe in more detail the elaboration of the step "authenticate, issue ballot and record vote" shown in Figure 2. The step "authenticate, issue ballot and record vote" is decomposed into two sequential sub-steps, "authenticate and issue ballot" and "record voter preference". The latter sub-step is where the hazard is defined. Its details are elaborated in another diagram but they are not important for the discussion here. The former sub-step, "authenticate and issue ballot", itself has three sub-steps, performed in the following order: "perform pre-vote authentication", "check off voter as voted", and "issue ballot". The step "issue ballot" itself is a *try* step (denoted by an arrow with an X on its step bar); this means its first sub-step "issue regular ballot" will be attempted first. If no exception is thrown when "issue regular ballot" is completed, the step "issue ballot" is also completed. If an exception is thrown while performing "issue regular ballot", the next alternative "issue provisional ballot" will be executed. In this case, the process is modeled so that `VoterUnqualifiedException` will be thrown while performing "issue regular ballot" if its prerequisite `voterQualified==true` fails.

To understand the difficulties of interpreting an MCS, consider the 4 event literals that comprise one of the computed MCSes:

1. Step "get voter name" produces wrong "voterName"
2. Step "verify voter has not voted" does not throw "VoterNotRegisteredException" (while checking prerequisite)
3. Step "check off voter as voted" does not throw "VoterUnqualifiedException" (while checking prerequisite)
4. Step "issue regular ballot" does not throw "VoterUnqualifiedException"

Figure 3 displays an abstract view of the derived fault tree. This view contains only the nodes related to the

events in the MCS being discussed. It shows how the combination of the primary events lead to the hazard. We provide two different scenarios in which all of the event literals in this MCS can happen but differ depending how one event literal is interpreted.

4.2.3 Scenario 1

Event literal 1 *Step "get voter name" produces wrong "voterName"* can be interpreted as an impostor gives the election official the name of a legitimate voter but the name given is not the impostor's name. This is the step "get voter name". The next step is "confirm voter name in in voting roll". In this case, the interpretation is that the name really *is* in the voting roll. Thus the artifact `voterRegistered` output from this step evaluates to `true`, and is then passed to the next step "verify voter has not voted" (literal 2). As the prerequisite for this step is `voterRegistered==true`, no exception is thrown. The real registered voter has not voted, so the step "verify voter has not voted" produces `voterQualified` with value `true`. This artifact is passed to the subsequent steps "check off voter as voted" and "issue regular ballot" (literals 3 and 4 respectively). Neither of these steps would throw an exception while checking prerequisites, satisfying event literals 3 and 4 in the MCS respectively.

Thus, in this scenario, an impostor has provided the name of a registered voter who has not voted, and so the impostor can vote with a regular ballot.

4.2.4 Scenario 2

The previous scenario assumed that the real voter had not yet voted. Changing this assumption to one in which the real voter has *already* voted changes the interpretation of the MCS. More precisely, in this case the artifact `voterQualified`, output from the step "verify voter has not voted" (literal 2) evaluates to `false`.

Now the step "check off voter as voted" (literal 3) should throw `VoterUnqualifiedException`, because the prerequisite condition `voterQualified==true` is not satisfied. But that literal in the MCS says the step "check off voter has voted" does not throw the `VoterUnqualifiedException`—perhaps the agent performing the step is not doing the job right. It could be a mistake, but it could also be malicious behavior. One interpretation is that the agent maliciously ignores the fact that `voterQualified`'s value is `false` (that is, that the named voter has already voted). Similarly, suppose the agent performing the step "issue regular ballot" (literal 4) maliciously ignores that `voterQualified`'s value is `false` and does not throw the `VoterUnqualifiedException`.

In this particular scenario, the impostor is using the name of a registered voter who has already voted. The hazard occurs because of collusion with the election official(s), who ignores the fact that the voter is marked as having voted in the voting roll, and issues a regular ballot to the impostor voter. A slightly different scenario, having the same result, is that the impostor uses the name of a registered voter who has already voted, but the election officials simply make mistakes performing the steps (marking the voter as voted and issuing the regular ballot) and let the impostor vote with a regular ballot—there need not be any malicious intent on the part of the election officials.

This example shows how interpreting one event in an MCS in different ways can result in different scenarios causing a hazard. The challenge of interpreting MCSes will be discussed further in section 8. Each interpreted MCS scenario exposes a potential vulnerability in the process. One question, then, is whether all of the event literals in the MCS can occur in one process execution, and, in particular, whether attacker agents could cause all of those events to occur. In short, "can attackers exploit the potential vulnerability to create a real vulnerability and undermine the integrity of the process?" Note that an MCS shows what events must or must not occur for the hazard to occur. It also shows which agents are involved in those events but not what specific actions each must take, how practical these actions are, and what the associated costs are. Domain expert knowledge is required to devise a credible *attack plan*—a scheme to exploit a vulnerability to create a hazardous situation. A well-defined attack plan enables further analysis, for example, to formally prove that a process model is resilient against such an attack, or to derive a process execution in which the attack succeeds so that countermeasures can be proposed for modifying the process. The next section presents the analysis of a process model when such an attack plan is available.

5 Analyzing a process model in the presence of an attack plan using model checking

The work described in the previous section produces informal attack scenarios based on the MCSes. Domain experts can select which potential attacks are worth investigating. These can be crafted into formal attack plans, allowing analysis to evaluate whether the attacks can succeed. This section describes an approach for using model checking to analyze a process model in the presence of an attack plan to determine if the attack would succeed. The idea is to consider both the process of interest and the attack. We illustrate this approach using an example from the Yolo County election process model with an impostor attack plan.

Model checking exhaustively explores all possible ex-

ecution paths in a finite model of a process, determines whether a particular property holds in the model, and produces a counterexample if the property does not hold. If we can combine a detailed attack plan with the model of the attacked process, we can apply model checking to the combined model to determine whether the attack could succeed. To do this, we need to have appropriate formal representations of the process model, the attack plan, and the property representing the failure of the attack. The model checking tool FLAVERS [8] can take as input a process model represented in Little-JIL and a property represented as a finite-state automaton (FSA). We therefore model the attack plan as a process definition in Little-JIL and compose it with the model of the process being attacked.

We proceed as follows:

1. Model the attack plan in Little-JIL to obtain *the attacking process model*.
2. Compose the attacking process model with the attacked process model, yielding the *composed process model*.
3. Apply FLAVERS to determine if the composed process model satisfies the property "the attacking process fails". If the property is not satisfied, we examine a counterexample trace produced by FLAVERS to propose process improvements via process modifications.

The following sub-sections describe each step.

5.1 Modeling an attack plan in Little-JIL

An attack plan needs to be more than a set of attacking events; it needs to specify where in *the attacked process* these events are to occur, and which artifacts the attacking events need to corrupt. As noted in the previous section, an MCS derived from the process model's fault tree contains information about the process steps that an adversary might exploit. In order for domain experts to create a Little-JIL attack model, the experts will have to augment the MCS information with coordination diagrams, artifact flows, and agent specifications. The level of detail in the resulting model will have to match that of the attacked process model so that the attacking and attacked process models can be composed[3].

The first interpretation of the MCS example in the previous section 4.2.3 suggests that if an impostor has provided the name of a registered voter who has not voted, the impostor can vote with a regular ballot. Based on this insight, together with the attacked process model's steps that are involved in the MCS, an attack plan might be crafted into a Little-JIL process model as shown in Figure 4. The attacking process model requires the step "gather name of unlikely voter" to provide the artifact "voterName", which is passed into the step "give name" (a sub-step of the step "impostor pass the authentication check"), and then from there it will be passed to the attacked process model's step "get voter name" as we will explain later when we compose the process models. Also, to match the level of detail of the attacked process model, we make sure that the attacking process model is elaborated to contain the steps "get regular ballot" and "impostor cast ballot" corresponding to the steps "issue regular ballot" and "record voter preference" in the attacked process model respectively.

5.2 Composing the attacking process model with the attacked process model

The next step is to compose the attacking process model with the attacked process model. To do this, we create a new process model in which the two sub-processes, the attacking and the attacked process, execute in parallel with appropriate synchronization. Currently, this is done manually based on the intuitions and experiences of domain experts. For this example, we create a new root step "composed process" with a parallel sequencing badge and make the two sub-process models ("conduct election" and "attack by impersonation") sub-steps of the new step, as shown in Figure 5.

To synchronize the two sub-processes, we carefully examine common artifacts and common activities shared between the sub-processes. In this example, the artifact representing a voter's name named `voterName` is common to both sub-processes. A quick traversal through the steps providing or accessing that artifact suggests where to set up the synchronization: the step "give name" in the attacking sub-process model has `voterName` as output, and that artifact can be passed to the step "get voter name" in the attacked sub-process model, which takes `voterName` as input. Also, the step "issue regular ballot" in the attacked model produces the artifact "ballot", which can be passed into the step "get regular ballot" in the attacking model.

When devising the attack plan from the fault tree's MCSes, domain experts may already have some idea about where the synchronization points should be, and which attack steps aim at which parts of the attacked process. In the future, we plan to implement support for identifying these points when devising attack plans from MCSes to assist the process model composition.

Steps representing the same activities must also be examined carefully. For example, the step "impostor cast ballot" in the attacking sub-process and the step "record

[3] If an attack plan already exists in another format it could be converted to a Little-JIL attacking process model. The method of conversion will vary depending on the original format and the information contained in the attack plan. Note that conversion to Little-JIL might not be straightforward if the original attack plan lacks details or is too abstract, requiring domain experts to be consulted. In some cases, it might be easier to model an attack plan in Little-JIL from scratch rather than convert it from an existing format.

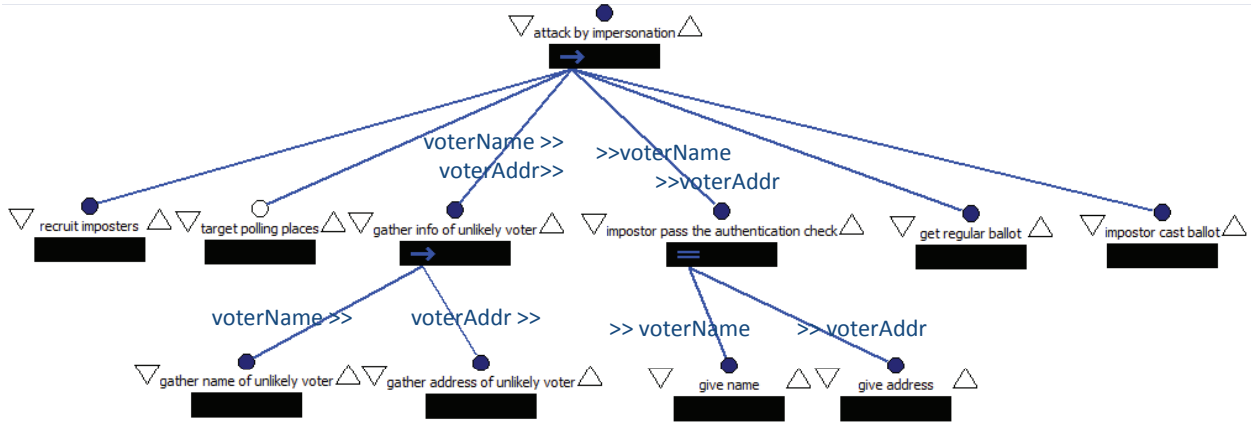

Figure 4: Impostor attack plan as a Little-JIL process model

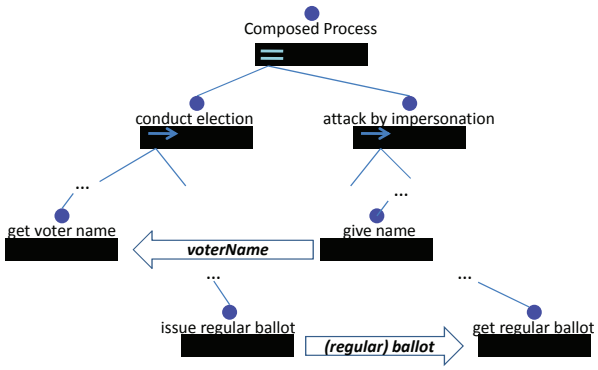

Figure 5: Composed process model: The root step "composed process" is a parallel step composing the two sub-processes: the attacked and the attacking. The Little-JIL channel feature is used to define how artifacts such as "voterName" and "ballot" must be passed between the steps of the attacking and attacked sub-processes in order to support the defined attack.

voter preference" in the attacked sub-process both represent the activity of the voter casting the ballot, as shown in Figure 6. These pairs of steps can be synchronized using message passing, so that the steps of the attacking and attacked sub-processes occur in the proper temporal order for the attack.

Little-JIL channels provide support for synchronous and asynchronous message passing. A channel declared at a Little-JIL step is accessible by all of its descendant steps. If a step *writes* an artifact to a channel, that artifact is available in the channel for access as soon as the writing step completes without an exception being raised, but the artifact is not available if the step terminates because of an exception. If a step is declared to *take* an artifact from a channel and the channel is empty, the step is blocked from execution until the artifact becomes available. For example (see Figure 5), the channel `VoterName` is declared at the root step of the composed process model to allow synchronization: the step "give name" writes the artifact `voterName` to the channel; and the step "get voter name" takes the artifact from that channel. Thus, the step "get voter name" can only proceed if the "give name" step writes the artifact to the channel. Similarly, the step "issue regular ballot" in the attacked sub-process writes the "ballot" artifact into a predefined channel when it completes its execution. Only then can the step "get regular ballot" in the attacking sub-process proceed and complete.

This same approach can be used for synchronizing the step representing the same activity. For example, both steps "record voter preference" (part of the attacked sub-process) and "impostor cast ballot" (part of the attacking sub-process) represent the activity of the voter casting the ballot (Figure 6). Through a predefined channel, a token is passed upon the completion of the step "record voter preference" to the step "impostor cast ballot". By doing this, we make sure that the step "impostor cast ballot" cannot complete unless the step "record voter preference" completes.

5.3 Performing verification

Having the composed process model, we now want to perform the verification of the property representing the absence of a successful attack. In other words, the attack must never complete. We specify this property as an FSA as shown in Figure 7. The event "attack succeeds" is the only event specified in the FSA, which starts in an accepting state. If the event "attack succeeds" occurs, the FSA will progress to the violation state. Note that the property is specified somewhat independently of the process model, thus before the verification can be done, the association must be specified between the property events and the process events.

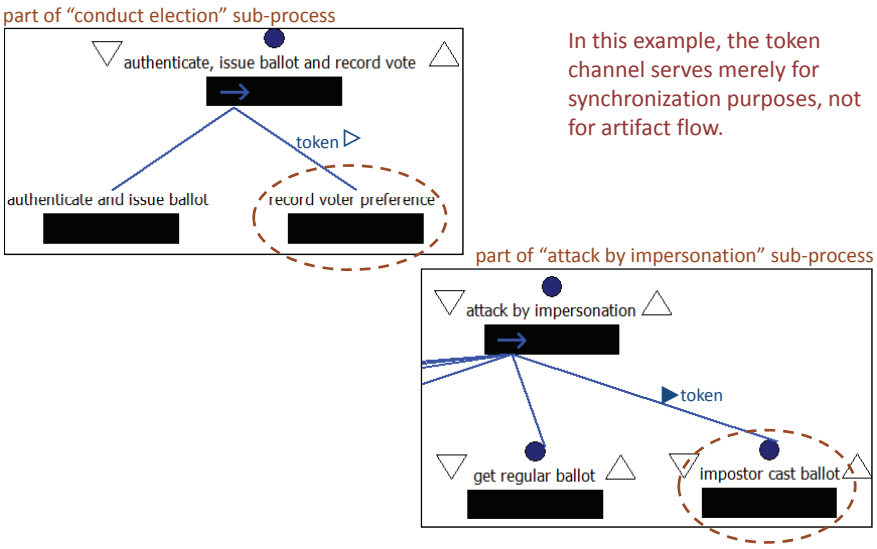

Figure 6: Example of synchronization using steps representing the same activities.

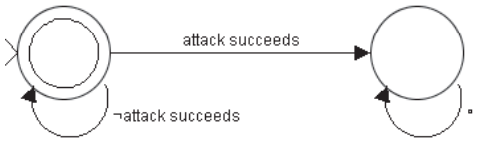

Figure 7: Property as an FSA representing attack never succeeding

When verifying a Little-JIL process model using FLAVERS, each event in the property has to be *bound* (correspond) to one or more events in the process model. Events in a Little-JIL process model include a specific exception being thrown by a step or a step being in a specific state. The execution of a Little-JIL step is modeled as progress through several states. Step execution begins in the *posted* state during which execution of the step is assigned to an agent. Execution then proceeds to the *started* state, when the agent begins performing the step. Eventually the step enters either the *completed* state (normal execution) or the *terminated* state (execution ends with an exception). In this example, the event “attack succeeds” in the property FSA is bound to the event *the step “attack by impersonation” is completed* in the process model.

Running FLAVERS on the composed model shows that the property is violated—the step “attack by impersonation” can succeed. The produced counterexample reveals an attack trace in which the impostor provides the election official with a voter’s name, the step “confirm name in voting roll” does not throw an exception, and all the subsequent steps are carried out normally without any exception being thrown: the election official verifies that the voter has not voted, checks off the voter’s name in the voting roll, issues a regular ballot, the impostor gets the regular ballot and casts the ballot. The counterexample suggests that the impersonation attack could succeed when an attacker has access to the name of a registered voter who has not voted.

The attack trace may also provide domain experts with some insights about how to modify the process to thwart the attack. In this case, an additional authentication step could be added to the attacked sub-process model, so that the impostor will fail to authenticate (for example, a secret PIN that only the voter knows, or some biometric information). The step “additional authentication” is declared to take an authentication artifact from a predefined channel in order to proceed. Since there is no step in the attacking sub-process that provides such an artifact to the channel, the attacking sub-process will not be able to reach the *completed* state. This is verified by running the verification on the new composed model.

In the second scenario (Section 4.2.4), the impostor has the name of a registered voter, but in this case the registered voter has already voted. An election official, however, ignores that the voting roll shows the voter has voted. The election official issues a regular ballot to the impostor voter, allowing the hazard to occur. The attack plan devised from this scenario is similar to the one discussed above, except that the agent performing the step “verify voter has not voted” and the agent performing the step “check off voter as voted” must never raise any exception while performing these steps. Composing this attack plan with the attacked election process model is similar to the previous case; that is, the parallel root step is created and channels for synchronization are

declared. But now we have to model the behavior of the *inside attackers*, the agents performing the steps "verify voter has not voted" and "check off voter as voted", such that no exception will ever be raised at these steps. For this to work, the step "verify voter has not voted" only throws `VoterUnqualifiedException` if its prerequisite `voterQualified==true` fails. The same is true for the step "check off voter as voted". Thus, we can model these inside attack behaviors by either setting the value of the Boolean artifact `voterQualified` to `true`, or removing the prerequisites from these steps.

Applying FLAVERS to this new composed process model shows that the attack might succeed. The countermeasure for this scenario is more complex than that of the first scenario. Perhaps the most appropriate is to have election officials work in pairs, in the hope that errors by one election official would be caught by the second. Thus, the improvement here would be to add an additional agent and extra checking steps.

These examples show that the model can validate that an attack will be successful. For more complex attacks, one could send each artifact to a number of different places in the process and look for combinations of destinations that would cause the attack to succeed. In this way, one could also examine variants that might attack the process in unanticipated ways.

6 Related Work

6.1 Attack modeling

Several attack models have been proposed, each providing a different representation of attacks. Moore et al. modeled attacks in the form of attack trees and attack patterns for the purpose of documentation [23]. Lazarus created a catalog of election attacks in the form of a single attack tree [17], attempting to provide a threat model and a quantitative threat evaluation that are reusable across different jurisdictions.

These attack trees lack specification of artifact flows and temporal ordering of events that are necessary for supporting formal analyses of the sort described in this paper. In fact, Lazarus's catalog of informally specified attacks includes many attacks that could be represented with our approach, including the impostor attack that we use as an example in this paper. To study the vulnerability of our modeled election processes to this attack, however, would require the addition of such temporal ordering and artifact flow. Several researchers have proposed approaches to overcome the limitations with attack trees. Jürgenson and Willemson introduced temporal order to the attacker's decision-making process [14]. Helmer et al. used augmented Software Fault Trees (SFTs), attack trees with temporal order, to model intrusions; in their representation, the root node represents the intrusion and an MCS contains events to be monitored to detect intrusions [11]. Their SFTs are then automatically converted to colored Petri nets for intrusion detection systems. Simple Petri nets have also been used to model attacks [21,40]. McDermott [21] suggested using labeled tokens in Petri nets to indicate different attackers (an approach to agent specification). The modeled attacks are then used to discover and analyze attack scenarios in penetration testing.

Templeton and Levitt proposed a requires/provides model to represent attacks. A concept (sub-attack) is specified by the capabilities it requires and provides [34]. This model features the composition of sub-attacks to form more sophisticated attacks. Using this model, Peisert et al. described multi-stage attacks for use in forensic analysis [25] and forensic analysis of elections in particular [2]. This technique sought to use the natural constraint of the limited number of entry points into a system and the limited, enumerable number of assets that one might wish to defend to provide insight as to where and how to monitor an otherwise complex and unwieldy system. However, like attack trees, this model lacks agent specifications and artifact flows.

In addition to these graph-based approaches, Cuppens and Ortalo proposed a declarative language (LAMBDA) for specifying attacks in terms of pre- and post-conditions [6]. The language is modular and hierarchical. Higher-level attacks can be described using lower-level attacks as components. Attack specifications contain information for specification-based intrusion detection systems, which detect malicious activity whenever specifications are violated [16].

Attacks are modeled differently for different purposes as discussed above. We refer to a process modeling language with rich semantic features as a process definition language. Little-JIL is an example of such a language, which we have found serves our purposes well.

6.2 Formal reasoning in security

Applying model checking in the security field is not new. Ritchey and Ammann used it to analyze network vulnerabilities by encoding the vulnerabilities in a state machine description suitable for a model checker and then asserting that an attacker cannot acquire a given privilege on a given host [30]. Similar to the second phase of our approach, using model checking to formally assess an attack's success, Ritchey and Ammann's approach requires knowing the vulnerabilities and their exploits in advance. The first phase of our approach, using FTA, complements their approach by helping to identify vulnerabilities and to devise attack plans that can then be used for model checking. Another difference is that their vulnerabilities are simply system attributes (e.g., a host running Apache

version 1.04), and exploits are described by lists of prerequisite vulnerabilities, host computer access levels, and the resulting access levels of the connected hosts. Thus, the attack plans that are developed using our methods are more detailed and have more structure.

A large body of work has applied model checking to security protocol verification. For example, Lowe used the FDR model checker to find a subtle attack on the Needham-Schroeder authentication protocol [20]. Meadows created the NRL Protocol Analyzer, a tool based on a combination of state exploration and theorem proving techniques, and analyzed the Internet Key Exchange protocol [22]. Compagna verified security protocols using a SAT-based model checking approach [5]. Additionally, Powell and Gilliam proposed a compositional model checking approach. In their approach, security property verification results of individual components of a large system are extrapolated for the overall system, which they claim would otherwise be beyond the capabilities of current state of the art model checkers due to the state space explosion problem [26]. They successfully applied their approach in verifying the Secure Socket Layer protocol for NASA systems [27].

Another line of formal reasoning in security studies attack generation. Sheyner et al. began with rules capturing atomic attacks such as buffer overflows [32]and modeled a computer network system as a finite state machine where state transitions correspond to the atomic attacks. They then used model checking to generate an attack graph in which any path from a root node (an initial system state) to a leaf node (an unsafe system state) shows a sequence of atomic attacks an intruder can employ to attack the system. Based on the generated attack graph, they developed a minimal set of atomic attacks that must be removed to thwart the intruder. Our approach uses FTA, rather than model checking, to generate possible attacks. The attacks generated by their approach are more like what we have called *attack scenarios*. In our approach, given a hazard we devise (a class of) attack plans that are more detailed than attack scenarios. Also, our focus is on overall processes, rather than on the components (such as computer networks and sub-systems) that these processes integrate.

6.3 Process-based security analysis in the election domain

Raunak et al. [29] used model checking to show that a property about an election process holds if all agents perform the steps correctly, but that the same property may be violated if some agents are dishonest. Our work develops the attacks in a more systematic manner.

Closest to our approach, and also applied in the election domain, is the work of Weldemariam and Villafiorita [38] that attempts to discover attacks. They model *procedures* that are best practices, defining how critical assets are to be managed, elaborated, and transformed. They then inject *threats*—actions that alter some features of an asset or allow some actors privileges (e.g. a *read* privilege) on some assets—into the original model to get an extended model that is encoded as the input for the NuSMV model checker. They then specify a property (e.g., "It is never the case that poll officers receive an altered version of the election software that can be run on the machines") and use the model checker to verify it. A counterexample produced when the property is violated is an example of an attack. Our approach differs from theirs. First, we use FTA instead of model checking to devise more structured and more detailed attack plans given a hazard. Their attack generation method is very similar to Sheyner et al's mentioned above. Second, their paper does not describe how the feasibility of the generated attack is determined whereas we describe how our approach does so by applying model checking to the composition of a process model and an attack plan (Section 5). In addition, they claim that injecting all possible threat-actions at all possible steps of the procedure is "the best and most general strategy". However, our view is that this approach is likely to produce many false-positives that obscure the true positives, with feasible attacks potentially being obscured by an excessive number of attacks that will prove to be infeasible. Thus our approach emphasizes the importance of determining the feasibility of a candidate attack by finding a verification counterexample. Finally, since the example procedure shown in their paper contains only sequential steps with asset flows, it is not clear whether their process models are able to represent concurrency and exception handling, key parts of real-world election processes.

7 Limitations and Future Work

Our approach has some limitations. First, it is based on process models, which inevitably capture real-world processes incompletely. Any model will omit some details, possibly because they are irrelevant or possibly because they cannot be modeled conveniently in a given modeling notation. But we view the process model as a living document to be updated on an ongoing basis, for example when the process is modified, when more process details are needed for analysis, or when superior modeling capabilities become available. In general we expect that when changes or elaborations are necessary, they will be of the sort that can be modeled relatively easily and then re-analyses can be done relatively quickly on the modified model using existing specifications of hazards and properties. Radical process changes can be expected to entail considerably more modeling and analysis work, but are expected to be far more infrequent.

Second, since our approach starts with the identification of specific hazards that are then used to identify specific process vulnerabilities, we must acknowledge that there will always be hazards that were not initially considered or that might be particularly difficult to represent. Given that election process vulnerabilities will probably be recognized incrementally over time (perhaps as previously recognized vulnerabilities are better defended against) it will be necessary to perform our analyses incrementally as these vulnerabilities are identified. Thus, the work described here should be taken as a specification of a single iteration in what we believe must be a continuing iterative procedure of identification of hazards and removal of consequent vulnerabilities.

Despite these limitations, our initial evaluations and experiences with this process-model-based analysis approach are promising and suggest several avenues of research to extend this work. One such extension is to increase the level of automation as much as possible so that human participation can be restricted to activities that make the best possible use of human intuition and judgment. Currently, we can automatically generate fault trees and therefore the MCSes used to identify process vulnerabilities; and we can automate use of the formal verification to determine the possibility of the success of an attack plan. However, using insights derived from the MCSes, domain experts still need to manually devise attack plans. In future work, we hope to be able to automate the construction of at least initial attack plans from an MCS. The resulting plans might be coarse but it would give domain experts a better starting point for elaborating these plans into ones that are more detailed. Moreover, process model composition (integrating an attack plan and the attacked process model) was done manually in this work by having humans study the two models and identify synchronization point. We believe that this model composition can also be at least partially automated.

We also want to look into ways to simplify the interpretation of MCSes. One MCS can be interpreted in multiple ways, complicating the analysis. For example, the MCS event "failure to throw an exception" could be interpreted to mean that the exception is not thrown at all or that it is not thrown when it is supposed to be thrown. These interpretations lead to different scenarios, as shown in the example in Section 4.2. An OR gate in the fault tree can be used to distinguish between these interpretations, but this increases the fault tree size, and probably the number of MCSes as well. Thus the value of such automated help needs to be determined through further research. Other research is needed to address such additional issues as determining whether resulting scenarios are equivalent, or whether we should define one formal attack plan for each scenario or one formal attack plan that covers multiple scenarios.

Another way in which our approach could be extended to suggest process vulnerabilities is by considering the agents that perform the steps involved in an MCS. It might be the case that an MCS does not comprise a single point of failure, but that all the steps in the MCS are performed by only one agent. In that case, that agent might be viewed as a single point of threat. Or when two or more agents are involved, collusion or coercion might be a possibility worth considering. Therefore analyses of the agents that execute MCS steps can provide domain experts with additional insights into process vulnerabilities.

Our approach could also be extended to provide better insights into vulnerabilities to insider attacks. In section 5.2, we used channels as a mechanism for studying how well outsider attacking behaviors could be defended by insiders executing the election process. But other approaches are likely to be more effective in representing an attack that involves collusion between outside attackers and privileged agents inside the process. We intend to explore the use of detailed specifications of agent behaviors, including specifications of malicious and collusive behaviors of agents inside the attacked process (inside attackers) as well as the behaviors of those attacking the process (outside attackers), to explore vulnerabilities to insider attacks.

8 Conclusion

This paper describes a systematic and semi-automated approach to continuous process improvement by automatically identifying process vulnerabilities by applying FTA to a detailed model of the election process. The derived MCSes provide insights about potential attacks that are then used to create attack plans. After composing each attack plan with the detailed process model, model checking is used to determine if the attack can succeed. The generated counterexample(s) can be used by domain experts to improve the process so that it can thwart such attacks. We envision this approach being applied incrementally as election processes evolve with the introduction of new laws and technology.

To evaluate this approach, we applied it to a portion of the Yolo County election process. Our preliminary results seem encouraging. Some results were what we expected, such as identifying the vulnerabilities that an impostor can exploit to attack the process, formally verifying that the attack might succeed, and verifying that the same attack will fail after appropriate process modification. What was unexpected was the variety of possible attack scenarios that could often be derived from a single MCS. We believe that reducing this variability so as to improve the focus on more feasible

and worrisome attacks is one of the more important directions for future work that we should explore. We also look forward to conducting a more extensive evaluation using more comprehensive models of real-world election processes. We hope that such an extensive evaluation will identify unrecognized or overlooked potential attacks and demonstrate the effectiveness of process modeling and analysis to detect and successfully defend those attacks.

Acknowledgments: This work was supported by the National Science Foundation (NSF) under Awards CCF-0905530, CNS-1049738, CCF-0820198, and IIS-0705772. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF. We would like to thank Freddie Oakley and Tom Stanionis for describing the Yolo County election process and validating the process model. We also would like to thank Sean Peisert, Michael Clifford, Heather Conboy, and Bobby Simidchieva for their feedback and help with defining the process model.

References

[1] APPEL, A. A., GINSBURG, M., HURSTI, H., KERNIGHAN, B. W., RICHARDS, C. R., TAN, G., AND VENETIS, P. The New Jersey Voting-Machine Lawsuit and the AVC Advantage DRE Voting Machine. In *Proceedings of the 2009 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections* (Berkeley, CA, USA, aug 2009), USENIX Association.

[2] BISHOP, M., PEISERT, S., HOKE, C., GRAFF, M., AND JEFFERSON, D. E-Voting and Forensics: Prying Open the Black Box. In *Proceedings of the 2009 Electronic Voting Technology Workshop/Workshop on Trustworthy Computing (EVT/WOTE '09)* (Montreal, Canada, August 10–11, 2009), USENIX Association.

[3] CALIFORNIA SECRETARY OF STATE. Top-to-bottom review. http://www.sos.ca.gov/voting-systems/oversight/top-to-bottom-review.htm, July 2007.

[4] CHEN, B. *Improving Processes Using Static Analysis Techniques*. PhD thesis, Universiry of Massachusetts Amherst, 2010.

[5] COMPAGNA, L. *SAT-based Model-Checking of Security Protocols*. PhD thesis, Universit'a degli Studi di Genova and the University of Edinburgh, 2005.

[6] CUPPENS, F., AND ORTALO, R. LAMBDA: A Language to Model a Database for Detection of Attacks. In *Proceedings of the Third International Workshop on Recent Advances in Intrusion Detection* (London, UK, 2000), RAID '00, Springer-Verlag, pp. 197–216.

[7] DELAUNE, S., KREMER, S., AND RYAN, M. Coercion-Resistance and Receipt-Freeness in Electronic Voting. In *Proceedings of the 19th IEEE Workshop on Computer Security Foundations* (Washington, DC, USA, jul 2006), IEEE Computer Society, pp. 28–42.

[8] DWYER, M. B., CLARKE, L. A., COBLEIGH, J. M., AND NAUMOVICH, G. Flow analysis for verifying properties of concurrent software systems. *ACM Transactions on Software Engineering and Methodology (TOSEM) 13* (October 2004), 359–430.

[9] ERICSON II, C. A. Fault Tree Analysis - A History. In *17th International System Safety Conference* (1999).

[10] ESTEHGHARI, S., AND DESMEDT, Y. Exploiting the Client Vulnerabilities in Internet E-voting Systems: Hacking Helios 2.0 as an Example. In *Proceedings of the 2010 Electronic Voting Technology/Workshop on Trustworthy Elections Electronic Voting Technology Workshop/Workshop on Trustworthy Elections* (Berkeley, CA, USA, aug 2010), USENIX Association.

[11] HELMER, G., WONG, J., SLAGELL, M., HONAVAR, V., MILLER, L., WANG, Y., WANG, X., AND STAKHANOVA, N. Software Fault Tree and Colored Petri Net Based Specification, Design and Implementation of Agent-Based Intrusion Detection Systems. *IEEE Transactions of Software Engineering 7* (2002), 2002.

[12] HYMAN, WILLIAM A.; JOHNSON, E. Fault tree analysis of clinical alarms. *Journal of Clinical Engineering* (2008), 85–94.

[13] JHA, S., SHEYNER, O., AND WING, J. Two Formal Analyses of Attack Graphs. In *CSFW '02 Proceedings of the 15th IEEE workshop on Computer Security Foundations* (2002).

[14] JÜRGENSON, A., AND WILLEMSON, J. Serial model for attack tree computations. In *Proceedings of the 12th International Conference on Information Security and Cryptology* (Berlin, Heidelberg, 2010), ICISC'09, Springer-Verlag, pp. 118–128.

[15] KARLOF, C., SASTRY, N., AND WAGNER, D. Cryptographic Voting Protocols: A Systems Perspective. In *Proceedings of the 14th USENIX Security Symposium* (Berkeley, CA, USA, jul 2005), USENIX Association, pp. 33–50.

[16] KO, C., RUSCHITZKA, M., AND LEVITT, K. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-Based Approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy* (Los Alamitos, CA, USA, May 1997), IEEE Computer Society, pp. 175–187.

[17] LAZARUS, E. L., DILL, D. L., EPSTEIN, J., AND HALL, J. L. Applying a Reusable Election Threat Model at the County Level. In *Proceedings of the 2011 Conference on Electronic Voting Technology/Workshop on Trustworthy Elections (EVT/WOTE)* (2011), USENIX Association, pp. 1–14.

[18] LEVESON, N., CHA, S., AND SHIMEALL, T. Safety verification of Ada programs using software fault trees. *Software, IEEE 8*, 4 (jul 1991), 48 –59.

[19] LINDE, R. R. Operating System Penetration. In *Proceedings of the AFIPS National Computer Conference* (May 19–22 1975), pp. 361–368.

[20] LOWE, G. *Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 1996, ch. Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR, pp. 147–166.

[21] MCDERMOTT, J. P. Attack net penetration testing. In *Proceedings of the 2000 workshop on New Security Paradigms* (New York, NY, USA, 2000), NSPW '00, ACM, pp. 15–21.

[22] MEADOWS, C. Analysis of the Internet Key Exchange Protocol Using the NRL Protocol Analyzer. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy* (Oakland, CA, May 1999), pp. 216–231.

[23] MOORE, A. P., ELLISON, R. J., AND LINGER, R. C. Attack Modeling for Information Security and Survivability. Tech. rep., Carnegie Mellon University, Software Engineering Institute, 2001.

[24] NORDEN, L. D., AND LAZARUS, E. The machinery of democracy: Protecting elections in an electronic world. Tech. rep., Brennan Center for Justice at NYU School of Law, 2006.

[25] PEISERT, S., BISHOP, M., KARIN, S., AND MARZULLO, K. Toward Models for Forensic Analysis. In *Proceedings of the Second International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE)* (Seattle, WA, April 2007), pp. 3–15.

[26] POWELL, J. D., AND GILLIAM, D. P. Model checking for network security requirements via a flexible modeling framework. In *Proceedings of the 5th IEEE International Symposium on Requirements Engineering* (2001).

[27] POWELL, J. D., AND GILLIAM, D. P. Model based verification of the Secure Socket Layer (SSL) Protocol for NASA systems. In *Ground System Architectures Workshop 2004* (2004), Pasadena, CA : Jet Propulsion Laboratory, National Aeronautics and Space Administration, 2004.

[28] RAMILLI, M., AND PRANDINI, M. An Integrated Application of Security Testing Methodologies to E-Voting Systems. In *Proceedings of the Second IFIP WG 8.5 International Conference on Electronic Participation* (Berlin, Germany, Aug. 2010), E. Tambouris, A. Macintosh, and O. Glassey, Eds., vol. 6229 of *Lecture Notes in Computer Science*, Springer, pp. 225–236.

[29] RAUNAK, M. S., CHEN, B., ELSSAMADISY, A., CLARKE, L. A., AND OSTERWEIL, L. Definition and Analysis of Election Processes. In *Proceedings of the 2006 Software Process Workshop (SPW'06) and Process Simulation Workshop (PROSIM'06)* (2006).

[30] RITCHEY, R. W., AND AMMANN, P. Using model checking to analyze network vulnerabilities. In *Proceedings of the 2000 IEEE Symposium on Security and Privacy* (2000), pp. 156–165.

[31] SHERMAN, A. T., FINK, R. A., CARBECK, R., AND CHAUM, D. Scantegrity III: Automatic Trustworthy Receipts, Highlighting Over/Under Votes, and Full Voter Verifiability. In *Proceedings of the 2011 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections* (Berkeley, CA, USA, aug 2011), USENIX Association.

[32] SHEYNER, O., HAINES, J., JHA, S., LIPPMANN, R., AND WING, J. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy* (2002), pp. 273 – 284.

[33] SIMIDCHIEVA, B. I., ENGLE, S. J., CLIFFORD, M., JONES, A. C., PEISERT, S., BISHOP, M.,

CLARKE, L. A., AND OSTERWEIL, L. J. Modeling and analyzing faults to improve election process robustness. In *Proceedings of the 2010 Electronic Voting Technology Workshop/Workshop on Trustworthy Elections (EVT/WOTE '10)* (2010).

[34] TEMPLETON, S. J., AND LEVITT, K. A requires/provides model for computer attacks. In *Proceedings of the 2000 Workshop on New Security Paradigms* (New York, NY, USA, 2000), NSPW '00, ACM, pp. 31–38.

[35] WARD, J., LYONS, M., BARCLAY, S., ANDERSON, J., BUCKLE, P., AND CLARKSON, P. Using fault tree analysis (FTA) in healthcare: a case study of repeat prescribing in primary care. In *Patient Safety Research: Shaping the European Agenda* (2007).

[36] WEISSMAN, C. *Information Security: An Integrated Collection of Essays*. IEEE Computer Society Press, Los Alamitos, CA USA, 1995, ch. Essay 11: Penetration Testing.

[37] WELDEMARIAM, K., KEMMERER, R. A., AND VILLAFIORITA, A. Formal Analysis of an Electronic Voting System: An Experience Report. *Journal of Systems and Software 84*, 10 (oct 2011), 1618–1637.

[38] WELDEMARIAM, K., AND VILLAFIORITA, A. Modeling and Analysis of Procedural Security in (e)Voting: the Trentino's Approach and Experiences. *EVT'08 Proceedings of the Conference on Electronic Voting Technology* (2008).

[39] WISE, A., CASS, A. G., LERNER, B. S., MCCALL, E. K., OSTERWEIL, L. J., AND SUTTON, JR, S. M. Using Little-JIL to Coordinate Agents in Software Engineering. In *Proceedings of the 15th IEEE International Conference on Automated Software Engineering* (Washington, DC, USA, 2000), ASE '00, IEEE Computer Society, pp. 155–164.

[40] WU, R., LI, W., AND HUANG, H. An Attack Modeling Based on Hierarchical Colored Petri Nets. In *Proceedings of the 2008 International Conference on Computer and Electrical Engineering(ICCEE 2008)* (dec. 2008), pp. 918 –921.