Iterative Analysis to Improve Key Properties of Critical Human-Intensive Processes: An Election Security Example

Leon J. Osterweil, University of Massachusetts Amherst
Matt Bishop, University of California at Davis
Heather M. Conboy, University of Massachusetts Amherst
Huong Phan, University of Massachusetts Amherst
Borislava I. Simidchieva, University of Massachusetts Amherst and Raytheon BBN Technologies
George S. Avrunin, University of Massachusetts Amherst
Lori A. Clarke, University of Massachusetts Amherst
Sean Peisert, University of California at Davis and Lawrence Berkeley National Laboratory

In this paper, we present an approach for systematically improving complex processes, especially those involving human agents, hardware devices, and software systems. We illustrate the utility of this approach by applying it to part of an election process and show how it can improve the security and correctness of that subprocess. We use the Little-JIL process definition language to create a precise and detailed definition of the process. Given this process definition, we use two forms of automated analysis to explore whether specified key properties, such as security and safety policies, could be undermined. First, we use model checking to identify process execution sequences that fail to conform to event-sequence properties. After these are addressed, we apply fault-tree analysis to identify when the misperformance of steps might allow undesirable outcomes, such as security breaches. The results of these analyses can provide assurance about the process, suggest areas for improvement, and, when applied to a modified process definition, evaluate proposed changes.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Languages*; *Methodologies*; *Tools*; K.4.3 [**Computers and Society**]: Reengineering; K.6.m [**Management of Computing and Information Systems**]: Security

General Terms: Languages, Management, Reliability, Security, Verification

Additional Key Words and Phrases: Process modeling, iterative analysis, model checking, fault-tree analysis, elections

1. INTRODUCTION

This paper presents an approach for systematically and iteratively evaluating and improving processes by identifying ways in which their performance might result in the violation of key policies, such as security and correctness policies. We use the word process[1] in the colloquial sense that refers to a real-world system that coordinates activities to specify the way in which people, hardware, and software collaborate to create specified products or achieve desired goals. Our approach to studying such processes requires that both the process and the policies be specified precisely to support automated and rigorous analyses. We present a particular approach for rigorously specifying and analyzing such processes, and illustrate the benefits of this approach by focusing on a critical part of a larger election process. Our purpose is to demonstrate the approach rather than to amass a comprehensive set of specific analytic results. The detailed explanation of the approach is intended to enable others to apply it to other processes and properties, either in the election domain or in other domains where human participation is critically important.

Our approach complements that of formal verification and proofs of protocol correctness. Rather than addressing the details of the protocols, as others have done, we are concerned with the *coordination* of the mixed human and automated procedures

[1]This use of the word "process" is not to be confused with its use in the operating systems literature where it refers more narrowly to the execution of a specific program as part of a larger computer system.

within which the protocols are embedded. Those protocols may be mathematical, cryptographic, or involve human activities. Protocol verification, in addressing the specific details of one protocol, might be able to analyze the protocol completely and concisely. But, because an entire election from voter registration to canvass is very large and diverse, the analysis should consider all the actors and their activities and interactions, and how specific protocols are used appropriately in the overall process. Here we focus on a part of an election process to illustrate our approach, and the details of the techniques it incorporates, so that our work can be used to obtain analogous results about other parts of this process or about processes in other domains.

Our approach exploits the rigor of both a precise process model that describes the various usage contexts, as well as precise policy specifications that describe either desirable or undesirable process behaviors. We apply two analysis techniques to determine how well the process model satisfies the policies. We use *model checking* to determine if any possible execution of the modeled process could fail to satisfy specified *properties* and, if so, to identify usage scenarios that illustrate such failures. *Fault-tree analysis* (FTA) is then used to identify different ways in which undesirable execution states, referred to as *hazards*, could be reached due to the *incorrect performance* of some process activities.

To emphasize the rigor of our process models, we refer to them as *process definitions*. In our work we use the Little-JIL [Cass et al. 2000] process definition language, which has rigorously defined semantics capable of supporting the analyses we perform. Moreover, it provides rich semantics such as concurrency, nondeterminism, and exception management, needed for specifying key process details. Responses to exceptions, or the lack of specified responses, often lead to vulnerabilities in processes that otherwise might be difficult to detect. Thus our work incorporates careful analyses of how processes handle exceptions, even in the presence of concurrency and non-determinism.

Model checking determines if a process definition is consistent with a set of requirements, specified formally as properties, by considering every relevant path through a representation of the process. We use this to analyze these process definitions by comparing them to rigorously defined properties that specify desirable and undesirable sequences of events. Such specifications can be effective in defining an extensive range of safety and security properties. For example, a property that might seem prudent or required by law is that at least two people are present whenever ballots are counted. When model checking determines that a property does not hold, it can provide one or more scenarios, called *counterexamples*, each of which illustrates a possible process execution that causes the property violation. These scenarios can be used to suggest process modifications aimed at eliminating the causes of the violation.

To complement the underlying model checking assumption that every process activity is performed correctly, we use FTA to study the effects of incorrect performance of process activities. Given a specification of a hazard, such as the reporting of an incorrect election result, FTA determines the conditions or events that might allow that hazard to occur. We do this by using our process definition to automatically derive a fault tree that can then be used to compute the combinations of incorrectly performed activities that would cause the hazard. This analysis is agnostic about whether these incorrect performances were intentional or unintentional. Indeed, human performers could perform incorrectly either intentionally or unintentionally; and non-human performers could be set up to perform incorrectly either deliberately or accidentally, or suffer a random mechanical failure. Thus, our analysis is capable of addressing both accidental misperformance (e.g. if a confused poll worker makes a mistake) and intentional misperformance (e.g. if tabulation software is programmed to skim votes).

In practice, we expect analysts to work with election officials to specify key properties and hazards, and then when problems are found to identify process modifications, and to reapply the analyses to assure that proposed changes eliminate the detected flaws without adding new ones. Regardless of whether the proposed process changes are because of flaws detected by our analyses, actual observed security violations, modifications to the laws, or proposed efficiency improvements, our approach is the same, providing systematic support for *continuous process improvement*.

Our approach can be applied to a broad range of processes, especially those where humans are participants. Process definitions, and consequently the associated analysis of these definitions, are usually more complicated when human activities are incorporated, since humans often desire a high level of autonomy and often display wider variability and greater fallibility than is typical of non-human components. Thus our approach seems particularly useful in specifying and analyzing *human-intensive systems*, namely those where both humans and automated entities are active participants. And indeed the approach has been applied in a number of other domains, such as health care and software development [Chen et al. 2008; Avrunin et al. 2006; Osterweil et al. 2007; Henneman et al. 2007; Avrunin et al. 2010; Wise et al. 2000].

This paper makes several contributions. It shows that broad ranges of hazards can be expressed by specifying specific critical steps to which incorrect inputs can arrive or from which incorrect outputs can be produced. This enables important defects and vulnerabilities to be inferred automatically from sufficiently detailed and rigorously-defined definitions. Further, it shows how two previously presented analytic approaches, both of which have been shown to be useful and effective, can be made even more so by their integration around a single process definition. While some previous work has shown the value of model checking and other work showed the value of Fault Tree Analysis our work demonstrates the complementarity of these two forms of analysis, suggesting that the integration of still more diverse and comprehensive suites of analyzers should be pursued. Finally, this work shows that not just protocols and laws, but also the specific details of the actual processes through which these protocols and laws are implemented and assured, must be considered.

1.1. Election Processes

An election is the "formal choosing of a person for an office, dignity, or position of any kind; usually by the votes of a constituent body" [Simpson and Weiner 1991]. An election process may be as simple as counting raised hands in a room (e.g., a caucus) or as complex as tallying votes across a multiplicity of jurisdictions, each of which uses its own rules to control the casting, reporting, and tallying of votes.

The process is important because an election's results can affect the course of history. Imagine how different United States history would have been had George McClellan, rather than Abraham Lincoln, become president in 1864. It is critical to verify that an election has been carried out consistent with criteria that assure correctness, fairness, and privacy properties. Ideally the verification should satisfy all parties that have stakes in the election, especially key stakeholders such as the voters and candidates.

Currently election officials typically use *ad hoc* approaches to address problems as they arise and to anticipate problems before they arise. Some *ad hoc* approaches have resulted in election process improvements. But given the frequent changes to election law over time, current *ad hoc* procedures are often a patchwork of responses to legislation at varying levels of government. Using formal analyses of process definitions to identify problems that might occur systematizes the search for problems before they arise. Once problems have been identified, either through such analyses or through experience in using the processes, the same analyses can then demonstrate that proposed solutions do indeed solve problems without creating new problems.

Verification of a real election process entails performing a rigorous comparison of a process definition to characteristics (such as those pertaining to security) stated as rigorous criteria. Specifying both the process and the criteria accurately and precisely is difficult because elections are very large and complex, and the criteria are numerous and diverse. Some examples of criteria are “all qualified voters must be allowed to vote,” “no voter may vote more than once,” and “no one other than the voter may know how that voter voted.” To support rigorous analysis, these natural language statements of criteria must be refined into precisely specified requirements. Thus, “no voter may vote more than once” could be represented by “suppose that v is a voter, and C is the set of all voters who have already cast their ballots. If $v \in C$, then voter v must not be issued a ballot”. We express these statements as specifications using formal logic and automata theory.

Issues concerned with the consistency of these requirements with each other and with the entire body of election criteria arise as the number of requirements grows. For example, to prevent voters from voting more than once, the U.S. state of Ohio kept a list of the names of voters who had voted in the order of their arrival. Expecting to have to verify electronic ballots, they also kept another list of the ballots in the order they were cast. Each list satisfied an important requirement. But the simultaneous existence of both lists enabled people to associate a specific voter with a specific ballot, thereby violating the voter’s expectation of privacy [McCullagh 2007].

Major problems arise from the size, complexity, and diversity of election processes. These processes may need to define how to handle a single ballot that includes races from multiple jurisdictions, each of which may have its own set of election requirements. In the United States, there are over 3,000 jurisdictions, each with the legal right to carry out its own election process, which may be quite different from the processes in other jurisdictions. A good example is a ballot for an election for federal, state, and local candidates in San Francisco, California. San Francisco uses ranked-choice voting for some local races, and majority voting for state and federal races as required by state law. Moreover, some elections for a single official span two or more legal jurisdictions, each with its own set of procedures. Which jurisdiction’s procedures should be used — or should both be used, each in its own jurisdiction? Thus, election requirements may vary even for the elections on a single ballot, and consequently election process specifications must vary accordingly.

Election processes must also specify how to deal with problems arising during balloting. For example, a ballot box might not be submitted for tabulation by a specified deadline, or a set of ballots might not be tabulated, or might be tabulated more than once. If the procedures for handling such contingencies are developed *ad hoc*, how can it be assured that all affected parties will have the same, correct understanding of the *ad hoc* procedure? And if procedures for handling contingencies are only informally specified and understood, what happens when the only person who understands these procedures is sick on election day? Moreover, humans have widely varying degrees of education, training, age, and cultural backgrounds. In some jurisdictions, the average age of poll workers is over 80. These poll workers may still be required to set up heavy voting equipment, understand the intricacies of the operation of the equipment, and fully grasp all of the details of the voting procedures in the jurisdiction. Because unexpected or unforeseen problems may arise, election processes must make appropriate provision for detecting and correcting problems in ways that are known to be consistent with election process requirements. Thus election process definitions will need to be constantly improved and analyzed to assure compliance.

2. ITERATIVE PROCESS IMPROVEMENT

To develop a process definition that precisely and rigorously represents the real-world process, several important aspects of the process must be understood and captured. These include issues that are often overlooked, such as exception handling, different scenarios for different contexts, specification of who is responsible for what activities, and the integration of the efforts of both humans and machines. Developing an appropriately detailed and precise process definition requires substantial effort and consultation with domain experts. But once a suitable process definition has been constructed, it can be leveraged to significantly improve the understanding, security, performance, or automation of the real-world process, as well as to train future cohorts of process performers. It can also be used to evaluate the effect of potential changes to the actual conduct of the process. Because human-intensive processes often require the communication, coordination, and synchronization of many people, machines, and other entities, it is not surprising that such a multi-faceted model may illuminate issues that the domain experts previously overlooked.

We use an iterative approach to identify potential areas for improvement. Shewhart [Shewhart 1931] introduced the basic tenets of continuous process improvement, and they were applied with perhaps the greatest effect by Deming [Deming 1982]. The essence of this approach is to capture the process to be improved in a model, compare the characteristics of the model to those that are desired, identify shortcomings in the model, propose and evaluate improvements to the model, and, once these improvements have been shown to be effective and efficient without introducing additional problems or defects, deploy the improvements in the real-world process to complete the improvement cycle and form the basis for a subsequent improvement cycle. This cycle has been referred to in various ways (e.g., the Plan-Do-Check-Act, or PDCA, Cycle; Define-Measure-Analyze-Improve-Control, or DMAIC; Observe, Orient, Decide, and Act, or OODA) over the past decades. In all of its names and manifestations, it has relied primarily on the ability to understand the process and its desired criteria and to analyze the ways in which the process does or does not adhere to those criteria.

These understandings and analyses have usually been pursued informally, with processes and requirements described in natural language, and analyses done by informal discussion and argumentation. More recently, research has shown that processes and requirements can be defined using precise and rigorous notations that render the evaluation of their consistency amenable to powerful technological support. Our approach moves that approach towards a disciplined engineering practice supported by scientific rigor, and has been used in other domains, including science [Altintas et al. 2004; Ellison et al. 2006], medicine [Clarke et al. 2008; Henneman et al. 2007], and business [Georgakopoulos et al. 1995; Wiegert 1998].

To demonstrate our approach, in this paper we define parts of an election process using Little-JIL, and then analyze the definition using different approaches grounded in mathematical reasoning. This results in the automatic derivation of important assertions about the process definition, and suggestions for an improved process that could then be the basis for further analysis and improvement. Figure 1 illustrates our framework for continuous process improvement. It shows how a single process definition can be leveraged to attain a multi-faceted understanding of the process. Our formal process definition is created using the Visual-JIL environment[2], which provides a visual representation that helps domain experts understand the definition. This formal definition then serves as the input to a variety of reasoning approaches, such as automatic derivation of a hyperlinked textual representation of the process, or discrete event simulations to evaluate the performance or efficiency of different scenarios. Each reasoning approach creates a specific output (illustrated in the last column of data components in Figure 1), and these outputs are used as inputs for the next iteration in the continuous process improvement loop by informing changes to the process definition, the properties representing precise requirement specifications, or both. Applying this

[2]Distributed as a plugin for the Eclipse IDE.

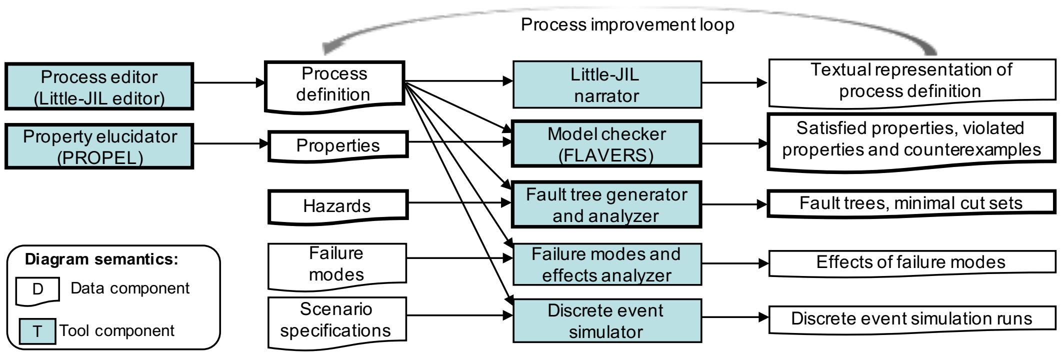

Process improvement loop
Process editor (Little-JIL editor)
Property elucidator (PROPEL)
Process definition
Properties
Hazards
Failure modes
Scenario specifications
Little-JIL narrator
Model checker (FLAVERS)
Fault tree generator and analyzer
Failure modes and effects analyzer
Discrete event simulator
Textual representation of process definition
Satisfied properties, violated properties and counterexamples
Fault trees, minimal cut sets
Effects of failure modes
Discrete event simulation runs
Diagram semantics:
D Data component
T Tool component

Fig. 1. A framework for iterative process improvement

framework iteratively allows us to identify and test improvements to ensure they do not introduce undesirable side effects before deploying them in the real-world process. Here, we focus on a subset of this framework that highlights the tools and components shown in the boxes with thick outlines in Figure 1, showing two analysis approaches, namely model checking and FTA.

2.1. Modeling the Process

The election process described throughout this paper is used in Yolo County, California, USA. We elicited it from laws, procedure documents, and extensive interviews with election officials. We initially used a basic understanding to construct a high-level process definition of the generic election process. We then reviewed this definition with the election officials of Yolo County. Using their feedback, we refined the process definition to match the process they used. We then focused on specific parts of the process, notably the subprocess by which votes are counted. We developed a graphical model of the process (see the next section). We then went back to the election officials, showed them our model, and walked them through what we had done. Sometimes they realized details had been omitted; indeed, one of the benefits of the elicitation process was that their understanding of the process improved by their having to recall and discuss these details. Other times, they clarified parts of the process. They described in detail the tallying of the votes, the California mandatory 1% manual audit, and the canvass, during which the totals are completed and the counts certified. The resulting definition models a wide range of exceptional situations and how they are handled, and specifies what agents perform what activities using what artifacts.

We then began to “drill down” into specific areas of interest. One of the areas, which we examine in this paper, is the subprocess for describing the counting of votes. For that subprocess, we repeated the elicitation process, but confined our focus to that area. We interacted regularly with the election officials to ensure our model reflected their practice. Also, one of the authors observed the counting process over the course of many elections, and participated as a deputy clerk in some. Thus, in addition to the information the election officials provided, we benefitted from firsthand observations.

Note that even though the presented part of the election process may appear relatively modest, its detailed model was substantial in size, comprising several dozens of Little-JIL steps. Thus, its analyses were hardly a toy example. This part of the overall process affords us the opportunity to describe our modeling and analyses in sufficient detail to indicate how others might employ our approach to other domains, other processes, or other parts of this process.

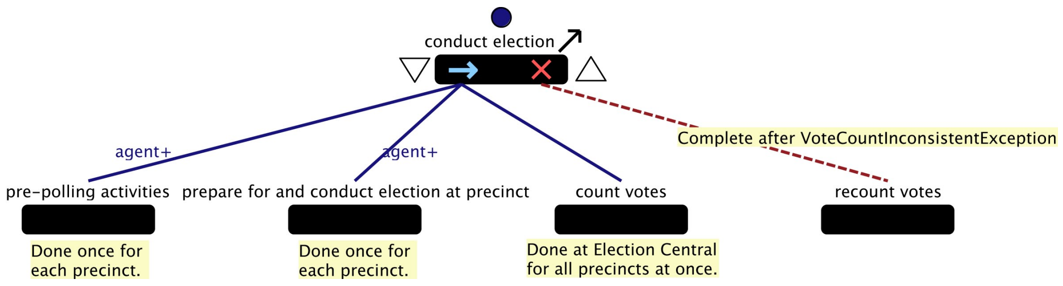

conduct election
agent+
agent+
Complete after VoteCountInconsistentException
pre-polling activities
prepare for and conduct election at precinct
count votes
recount votes
Done once for each precinct.
Done once for each precinct.
Done at Election Central for all precincts at once.

Fig. 2. Little-JIL process definition: Top level of the "conduct election" process

2.1.1. Little-JIL: A Process Definition Language. Little-JIL proved to be an effective vehicle for defining election processes. Its rich semantics support the precise definition of many aspects of processes, such as concurrency, communication, and coordination among human performers and software and hardware components; the specification of human choice and flexibility; the creation and modification of artifacts; and the specification of complex exceptional situations and their remediation. The diagrams presented here necessarily omit some of these details in the interests of clarity and readability.

A Little-JIL process coordination diagram, such as the one shown in Figure 2, specifies a hierarchical decomposition of steps. A step in the process is shown as a black rounded rectangle, with the step name above it. Each step is assigned an agent that is responsible for its execution; this agent may be a human performer, such as an election official or a voter, or a hardware or software component, such as a direct-recording electronic voting machine (DRE)[3]. Agents can also be composites, combinations of other component agents, such as polling places that are defined to consist of various devices, space, and people. A step in turn may be decomposed into *substeps* or children (the steps that connect to the lower left side of the parent step rectangle bar via edges), each with its own agent responsible for its execution. Each step that has children also has a *sequence badge*, which appears in the left half of the step bar and specifies the order in which its children will be carried out. For example, in Figure 2, the root step `conduct election` is a sequential step, indicated by a right arrow, specifying that its children will be executed in left to right order, so `pre-polling activities` will be followed by `prepare for and conduct election at precinct`, which in turn will be followed by `count votes`. Each of these activities is further decomposed in the complete definition of the process, but as noted above, here we focus on the `count votes` activity. A step without children is called a *leaf step*. Responsibility for the execution of leaf steps is left entirely to the step's agent. A step in a Little-JIL process definition is akin to a procedure or method specification that, once specified, can be invoked from anywhere in the process definition through an appropriate reference.

A Little-JIL process definition also contains complete specifications of the artifact flow and the agents responsible for steps. The artifact specification contains all the artifacts that are created, modified, or consumed in the process, for example a `ballot repository` (a repository of all the cast ballots) and different `tallies` (a report of the number of ballots used at a precinct or votes cast for each candidate). Each step definition declares what artifacts it accesses or provides. Artifacts are generally passed within the hierarchical flow of the coordination structure (i.e., from parents to children and vice versa). If steps are thought of as procedures, this artifact passing is essentially a parameter-passing mechanism. Lateral artifact flow and general message

[3]A DRE records votes directly to electronic media without the additional use of a paper trail.

passing are also supported.

The agent specification allows each process step to request that a specific type of agent be responsible for its execution. For the election process, `Voter`, `Election Official`, `Voting Machine`, and `Polling Place` are some example types of agents. Note that the former two are human agents while the latter two are non-human, and the last, `Polling Place` is a compound agent, consisting of such components as voting booths, election officials, and ballot-marking equipment. Little-JIL definitions only specify the type of agent (e.g., `Voter`) that should execute a specific step, and not a specific agent instance (e.g., Jane Doe). In Figure 2, the `agent+` notation on the edges to the first two substeps of `conduct election` indicates that each agent of the type requested should carry out these activities. Given that both steps request a `Polling Place` agent, this indicates that each Polling Place will provide the specific resources (e.g. tabulating devices) needed to support the execution of the specific election activities mandated by the authorities having cognizance over that site. The `count votes` step will occur once afterward, just as in the real-world Yolo County process where the precincts carry out election activities in parallel with each other, but the counting of all votes is carried out at Election Central.

In real-world processes, exceptional conditions may arise frequently and must be resolved before the process continues along its normative path. To accurately model this, Little-JIL provides comprehensive exception-handling semantics. For example, the `recount votes` step in Figure 2 connects to the $\times$ in the right half of the step bar of its parent, `conduct election`, to indicate that `recount votes` is an exception handler. Exceptions in Little-JIL are typed, which means that different exception handlers must be defined for each exception type. This is especially important in complex human-intensive systems such as elections because different exceptions usually necessitate different protocols. Thus, for example, the `recount votes` step is an exception handler for exceptions of the type `Vote Count Inconsistent Exception`. Finally, Little-JIL's exception-handling mechanism also provides flexible continuation semantics after exception handling takes place. In this case, `recount votes` specifies how to resolve inconsistencies in the counting of the votes and the step that threw this exception is considered completed and is not to be repeated or revisited after the exception has been handled. Other exceptions may require the re-execution of the step that threw the exception, and this continuation behavior can be defined in Little-JIL as well.

Here we focus on the tabulation of ballots and votes after the voting is completed. Figure 3 shows the decomposition of the `count votes` step from Figure 2. In Yolo County, every precinct brings its ballots, along with a summary cover sheet (indicating how many ballots were issued to the precinct, and how many of them are used, spoiled or blank after election day), to Election Central for tabulation. There, election officials first `count votes from all precincts`, then `perform random audit`, and then, finally, if no exceptions are raised, `report final vote totals to Secretary of State`. The `agent+` notation on the edge from the first substep to its child step indicates that the decomposition of this activity is into separate `count votes from precinct` steps, each of which tallies the votes from a different precinct separately before the precinct tally is added to a total tally. Ballot counts are compared to the summary sheets for each precinct, and after reconciling the actual and reported numbers the ballots are scanned to obtain the actual vote counts. Random auditing (or a mandatory manual recount of 1% of precincts to ensure consistency) is a state requirement in California and many other states [VerifiedVoting 2013; National Association of Secretaries of State (NASS) 2007].

It is important to understand how the regular tabulation of votes is performed as well as how reconciliation works should any discrepancies occur. Yolo County uses

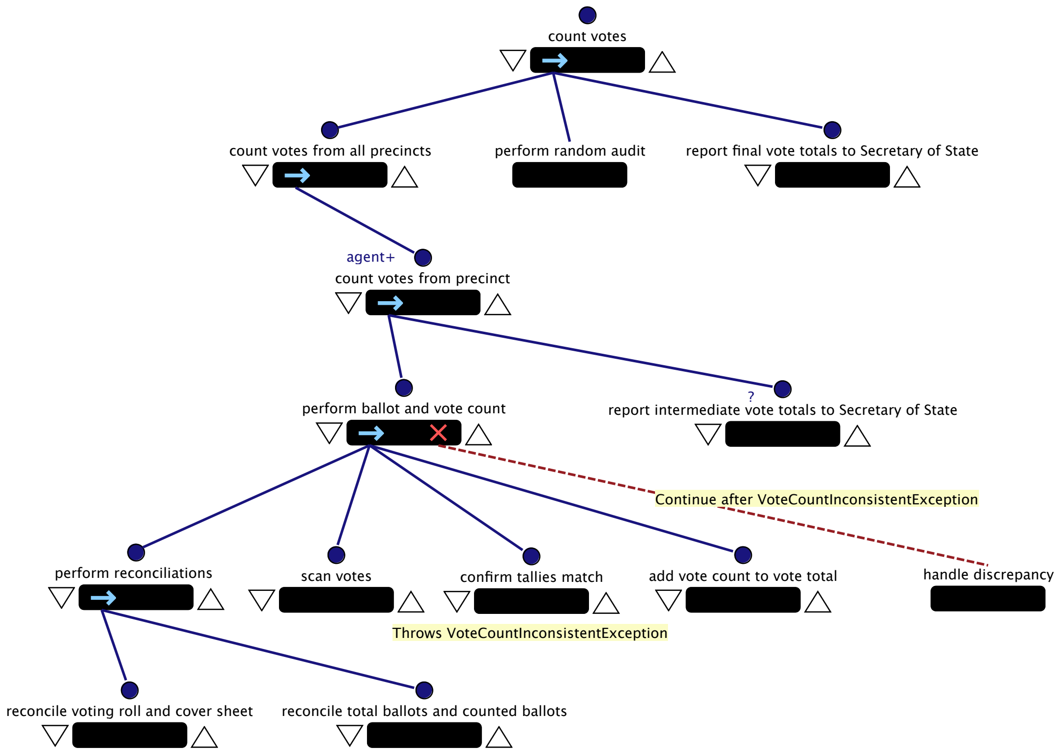

count votes
count votes from all precincts
perform random audit
report final vote totals to Secretary of State
agent+
count votes from precinct
perform ballot and vote count
?
report intermediate vote totals to Secretary of State
Continue after VoteCountInconsistentException
perform reconciliations
scan votes
confirm tallies match
add vote count to vote total
handle discrepancy
Throws VoteCountInconsistentException
reconcile voting roll and cover sheet
reconcile total ballots and counted ballots

Fig. 3. "count votes" sub-process

primarily paper ballots, which are scanned and counted by automated optical scanners. It also has voting machines designed for disabled voters, but that any voter may use. California election law requires all DRE machines to have an attached printer so that a voter-verified paper audit trail (VVPAT) can be maintained at all times. In Yolo County, these paper trails are in fact the artifact used to count votes cast on these machines. A damaged or missing paper trail can therefore lead to many problems in the election process, as fault-tree analysis demonstrates and is discussed in the Results Section.

2.2. Model Checking

Before considering the ways in which an election process might perform in undesired ways due to misperformance of one or more of its steps, we would like to be sure that the process will perform as required when all steps are performed correctly. Thus, we first check that all possible executions of the process, assuming each step is executed correctly with correct inputs and outputs, satisfy the requirements for the process as stated, for example, in election law. But because complex real-world processes such as elections are typically concurrent systems that need to coordinate and synchronize their activities and communications, the number of possible executions of such a system is typically exponential in the number of concurrent activities. This makes it hard to understand all the ways that such processes could be executed, and infeasible to list them and examine each one manually.

Model checking techniques [Clarke et al. 2000; Baier and Katoen 2008] work by constructing a representation of all possible relevant executions of the concurrent system with respect to a given formal specification, usually defined as an automaton or by using a modal logic formalism, and then comparing that representation to the formal specification. We refer to such a precise specification as a *property* to distinguish it from the original requirement or policy that may be informal (e.g., natural language) or even unstated. The model checking technique that we use expects a property to be represented as a finite-state automaton (FSA) that specifies intended (or unintended) sequences of *events* drawn from an *alphabet* of all events of interest.

Model checking techniques try to determine whether every execution represented by the model satisfies a given property. When the property is not satisfied by all executions, the analysis identifies counterexamples, particular executions that violate the property. For most classes of systems, the complexity of model checking techniques is at least NP-hard (and undecidable for some classes), but numerous optimizations have been developed, so that model checking techniques are now sufficiently practical that they are widely used to analyze real-world hardware and software systems.

Our process analysis and improvement framework translates Little-JIL to the Bandera Intermediate Representation (BIR) [Iosif et al. 2005], a guarded command language. From the BIR, we construct models suitable for use with various model checking techniques; for the work described in this paper, we primarily used the FLAVERS [Dwyer et al. 2004] tool. FLAVERS uses qualified data flow analysis [Holley and Rosen 1980] to check whether all executions of a system satisfy a property by propagating tuples of states from the property automaton, as well as various feasibility constraint automata, through a graph describing the possible orderings of events in the process. FLAVERS makes use of symbolic representations of sets of states, such as Zero-suppressed Binary Decision Diagrams [Minato 1996], to handle large processes.

2.2.1. Specifying properties by refining requirements. Requirements for elections are typically given in natural language documents such as laws and regulations. To determine whether a particular election process satisfies such requirements using model checking techniques requires that each requirement be refined to one or more precisely specified properties. This can be tricky and error-prone, especially since natural language is inherently ambiguous and incomplete. We use the PROPEL (PROPerty ELucidator) tool [Smith et al. 2002; Cobleigh et al. 2006] to help address these difficulties.

PROPEL provides templates for commonly occurring property specification patterns [Dwyer et al. 1999], and each template has a set of options to consider in order to specify the property precisely and completely. For instance, the template for properties that require one event to have already occurred before a second event can occur includes options such as whether the first event is required to occur at all, whether it can reoccur, and whether each occurrence of the second event must be preceded by a different occurrence of the first event. PROPEL provides three different views of a property: a hierarchical series of questions (referred to as the question tree view), the answers to which determine the template and the detailed options; a graphical FSA view in which the user selects transitions, transition labels, and accepting states to choose the options; and a Disciplined Natural Language (DNL) view in which the user selects phrases from drop-down boxes. Although the question tree and DNL views assist domain experts, who may not be comfortable with automata, all three views may be visible simultaneously and result in an FSA representation of the property that is then used in model checking.

We again focus on the details of the canvass as an illustrative portion of the election. The canvass is used to validate the results of the election by verifying that the counting is accurate and all applicable laws and regulations have been followed. Figure 4 lists six of the high-level legislative requirements that we verified for the canvass. Each requirement (Ri) has been refined to one or more properties ($Pi.j$). The California

R1. The canvass begins after the polls close.

P1. After the event **close polls** occurs, the event **begin canvass** must occur.

R2. The canvass needs to report the final results to the Secretary of State.

P2. The event **report final results to Secretary of State** must occur.

R3. The canvass must include a reconciliation of the number of voter signatures and the number of recorded ballots.

P3.1. After the event **begin canvass** occurs, the event **reconcile number of voter signatures and number of recorded ballots** must occur.

P3.2. The event **report final results to Secretary of State** is not allowed to occur until after the event **reconcile number of voter signatures and number of recorded ballots** has occurred.

R4. The canvass must include a reconciliation of the number of recorded ballots and the number of tallied ballots.

P4.1. After the event **begin canvass** occurs, the event **reconcile number of recorded ballots and number of tallied ballots** must occur.

P4.2. The event **report final results to Secretary of State** is not allowed to occur until after the event **reconcile number of recorded ballots and number of tallied ballots** has occurred.

R5. The canvass must include a 1% manual audit.

P5.1. After the event **begin canvass** occurs, the event **conduct one percent manual audit** must occur.

P5.2. The event **report final results to Secretary of State** is not allowed to occur until after the event **conduct one percent manual audit** has occurred.

R6. If the 1% manual audit shows a discrepancy, then a recount must be conducted.

P6. After the event **one percent manual audit shows discrepancy** occurs, the event **recount votes** must occur.

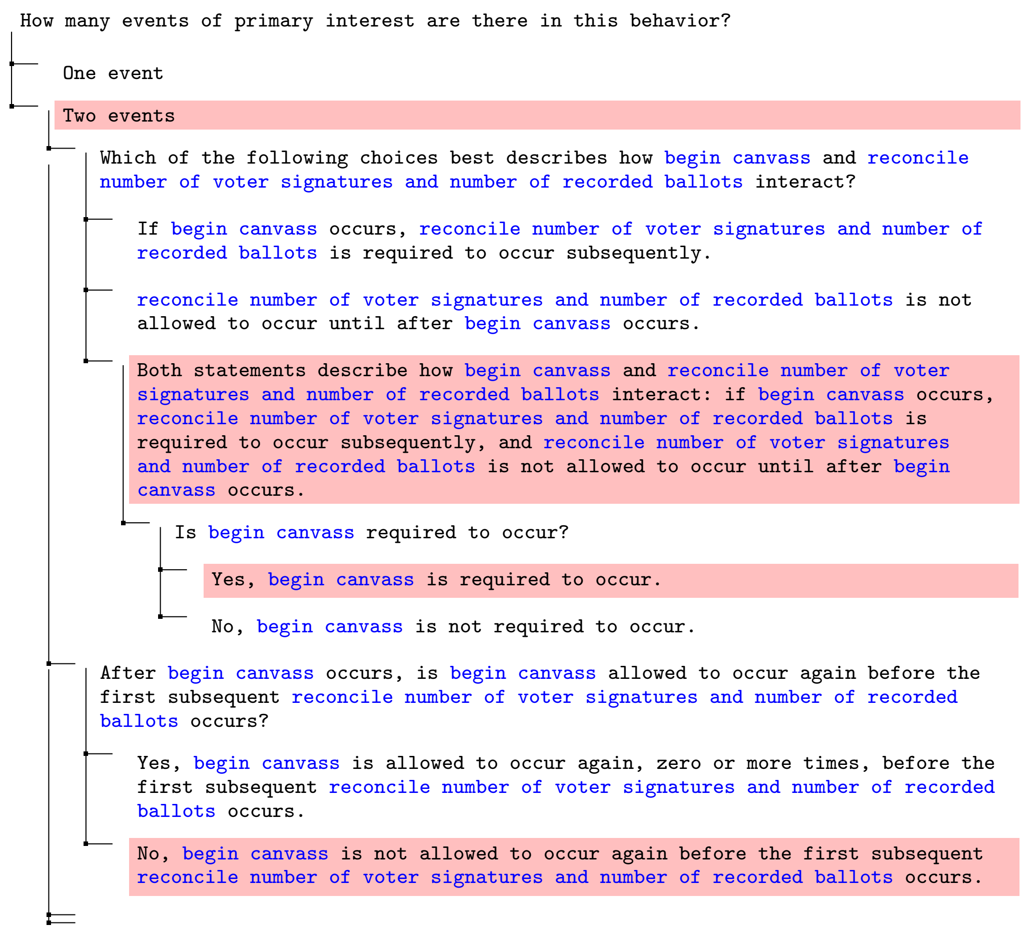
Fig. 4. Refinement of canvass-related requirements to low-level properties

election code[4] requires election officials to conduct a canvass after the close of the polls (R1) and before reporting the results to the Secretary of State (R2). Most of the tasks to be carried out in the canvass are laid out in Section 15302 (R3, R4) and Section 15360 (R5) of the California election code. If electronic voting equipment is used, a manual audit of 1% of the precincts is required (R5). Since Yolo County allows voters to use DREs to mark their ballots and the election officials use scanners to count ballots and votes, the county must always perform this audit. Our formulations of the properties therefore always require the audit. If any audit shows a discrepancy, then a recount must be conducted (R6).

Refining requirements into properties must take into account dependencies between requirements. For instance, requirements R1 and R2 impact requirements R3, R4, and R5. Additionally, PROPEL supports alternative ways to represent a requirement and user choice could affect the number of properties and their complexity. To illustrate, we describe the refinement of requirement R3, which states that the number of signatures on the roster is reconciled with the number of ballots recorded on the ballot statement.

To capture this requirement in PROPEL, we describe the canvass in terms of three events: **begin canvass**, **reconcile number of voter signatures and number of recorded ballots**, and **report final results to the Secretary of State**. We take the first reporting of the final results to signify the end of the canvass (in the case of recounts, there may be more than one report to the Secretary of State). PROPEL provides templates for properties that are intended to hold between two events, and so we could represent this requirement as a single property requiring that the reconciliation

[4]http://www.leginfo.ca.gov/.html/elec_table_of_contents.html



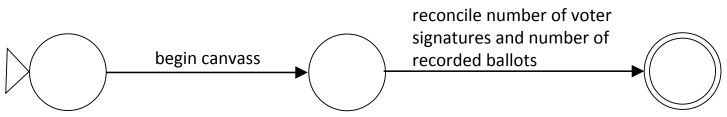
How many events of primary interest are there in this behavior?

- One event
- Two events
  - Which of the following choices best describes how begin canvass and reconcile number of voter signatures and number of recorded ballots interact?
    - If begin canvass occurs, reconcile number of voter signatures and number of recorded ballots is required to occur subsequently.
    - reconcile number of voter signatures and number of recorded ballots is not allowed to occur until after begin canvass occurs.
    - Both statements describe how begin canvass and reconcile number of voter signatures and number of recorded ballots interact: if begin canvass occurs, reconcile number of voter signatures and number of recorded ballots is required to occur subsequently, and reconcile number of voter signatures and number of recorded ballots is not allowed to occur until after begin canvass occurs.
      - Is begin canvass required to occur?
        - Yes, begin canvass is required to occur.
        - No, begin canvass is not required to occur.
  - After begin canvass occurs, is begin canvass allowed to occur again before the first subsequent reconcile number of voter signatures and number of recorded ballots occurs?
    - Yes, begin canvass is allowed to occur again, zero or more times, before the first subsequent reconcile number of voter signatures and number of recorded ballots occurs.
    - No, begin canvass is not allowed to occur again before the first subsequent reconcile number of voter signatures and number of recorded ballots occurs.

Fig. 5. PROPEL question tree for Property 3.1. Red highlighting indicates the selected answer.

occur between the beginning of the canvass and the initial report of the final results to the Secretary of State. We chose, however, to express this requirement using two properties, one saying that the reconciliation occurs after the canvass begins and the other saying that the reconciliation occurs before the final results are reported. We felt that this separation made the choice of options simpler, thereby making it easier for election officials to validate our formalization of this part of the election code.

To illustrate, Figure 5 shows the question tree (some lower-level questions have been omitted for brevity), and Figure 6 shows the FSA and DNL produced by PROPEL for property 3.1. PROPEL is based on patterns, which describe each property using a *scope* that specifies the parts of an execution to which the property applies, and a *behavior* that specifies the restriction on sequences of events within that scope. PROPEL's views give the scope and behavior separately. The "secondary events" refer to other events whose occurrence might need to be restricted; in this case, there are none. The FSA view can show the scope and behavior together, or only the behavior. More formally, the FSAs produced by PROPEL are deterministic and total, so there is exactly one transition from each state labeled by each event in the alphabet of the property. If a particular event should not be allowed to occur in some state, the transition labeled by that event goes to a *violation* state. The violation state is a sink—every transition


begin canvass
reconcile number of voter signatures and number of recorded ballots

Scope:

(1) From the start of any event sequence through to the end of that event sequence, the behavior must hold.

Behavior:

(1) The events of primary interest in this behavior are **begin canvass** and **reconcile number of voter signatures and number of recorded ballots**.
(2) There are no events of secondary interest in this behavior.
(3) If **begin canvass** occurs, **reconcile number of voter signatures and number of recorded ballots** is required to occur subsequently.
(4) Before the first **begin canvass** occurs, **reconcile number of voter signatures and number of recorded ballots** is not allowed to occur.
(5) **begin canvass** is required to occur.
(6) After **begin canvass** occurs, but before the first subsequent **reconcile number of voter signatures and number of recorded ballots** occurs, **begin canvass** is not allowed to occur again.
(7) After **begin canvass** and the first subsequent **reconcile number of voter signatures and number of recorded ballots** occur:
   — Neither **begin canvass** nor **reconcile number of voter signatures and number of recorded ballots** are allowed to occur again.

Fig. 6. PROPEL finite state automaton and disciplined natural language views for Property 3.1.

from the violation state is a loop that goes back to the violation state—and is a non-accepting state. For clarity, the FSAs in the figures do not show the violation state or any transitions to it.

A key part of the requirement partially encoded in Property 3.1 is that, once the event **begin canvass** has occurred, the event **reconcile number of voter signatures and number of recorded ballots** must subsequently occur. But the specification must resolve a number of ambiguities that could lead to the occurrence of events that should be forbidden. Are there any allowed executions of the process in which the **begin canvass** event does not occur? Can **begin canvass** reoccur? Can the reconciliation occur before the canvass begins? Based on discussions with the domain experts, we interpret the legal requirement as meaning that no executions of the election process should be allowed in which the canvass is not begun and the reconciliation of the numbers of signatures and ballots should not occur before the canvass has begun. The canvass may not begin more than once and the reconciliation may not reoccur.

2.2.2. Binding property events to the process definition. The properties discussed in the preceding subsection are formalizations of the requirements for the real-world process. Therefore, any process defined to achieve the same goal should satisfy those properties. But different process definitions may satisfy those properties in different ways and may represent the events in the properties in different ways. So, to check whether our particular Little-JIL process definition satisfies these properties, we must first *bind* each of the events in the properties to all of the Little-JIL process definition activities whose execution causes the event to occur. In some cases it is important to bind a property event to either the beginning or the end of the step execution. Thus, for example, the arrival of an artifact as input to a step is typically considered to take place at the start of the execution of that step. Thus, we bound the event **begin canvass** to the start of the Little-JIL step `count votes` (shown in Figure 3). Conversely the generation of an artifact by the performance of the stop is typically considered to be

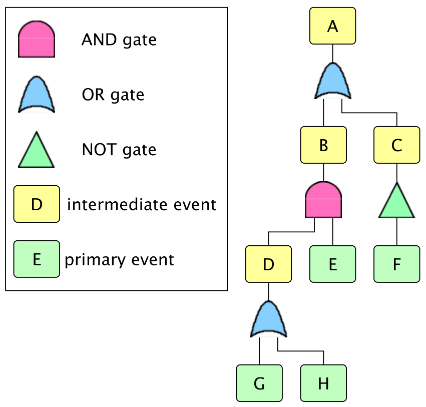


Fig. 7. Simple fault tree

bound to the completion of the execution of the step. For example, we bound the event **reconcile number of voter signatures and number of recorded ballots** to the completion of the Little-JIL step `reconcile voting roll and cover sheet` (shown in the same figure). Our process analysis and improvement framework provides support for indicating which process steps should be bound to each event in a property.

2.2.3. Benefits from model checking. Execution of the FLAVERS model checker succeeded in verifying that our election process definition satisfies the PROPEL representation of all of the properties listed in Figure 4, thereby increasing our assurance that the election process definition adheres to federal and state laws and regulations.

In general, it usually takes many iterations of analysis and refinement of the model (and properties) to convince ourselves and the domain experts that the process model is an accurate representation of the real process. Model checking is an important tool in reaching this consensus. When there is significant concurrency and exceptional behavior, human analysts cannot be sure that they have adequately considered all possible executions. Model checking provides this assurance, at least with respect to the properties that are considered important for the process. Thus, it is not surprising that numerous errors are usually found in the process definition and in the property specifications. After errors in the process definition and property specifications are removed, we begin to find errors that are actual problems in the real process.

Although the initial process modeling and model checking are time consuming, the resulting process models and properties are valuable assets that can be continually modified and improved as the process itself evolves. For election processes that are continually being updated, these are valuable resources that allow important properties such as safety and security to be evaluated before changes are made to the actual process; this is especially important since elections cannot easily be redone. In our framework, the accuracy of these models is vitally important since they become the basis for subsequent analyses such as the one described in the next subsection.

2.3. Fault-Tree Analysis

Model checking evaluates whether the process model adheres to stated properties, assuming that the steps in the model are carried out correctly. As noted earlier, however, process steps may not be done accurately, especially when humans, who may become fatigued or confused or maliciously desire to undermine a process, are involved. Thus we use FTA to evaluate how vulnerable the process, as represented by the process definition, is to incorrectly executed process steps, whether executed by human or non-human agents. There is no difference in our approach whether the incorrect performance is inadvertent or intentional, and thus is equally valid and effective for intentional attacks as well as simple errors or misunderstandings.

FTA is a deductive, top-down analytical technique that is used in a variety of industries [Ericson II 1999; Ward et al. 2007; Hyman and Johnson 2008; Chen 2010] to study conditions under which an accident or hazard that can cause substantial damage or loss might occur. In the case of our election process, a very serious hazard would be for an incorrect count of votes to be delivered to the Secretary of State, creating the condition that the wrong candidate would be declared to have been elected. This hazard might result, for example, if a batch of ballots were not counted. We note that this might happen either mistakenly or maliciously, but that our analysis is the same in either case.

With FTA, one first specifies a hazard and then attempts to determine which process execution events could combine to cause the actual occurrence of that hazard. Given the hazard, FTA produces a *fault tree*, a visualization of all the various combinations of these events that could lead to the hazard. A fault tree consists of *events* and *gates*. At the top (root) of the fault tree lies the hazard. In the fault tree, *intermediate events* are the consequences of previous events, and this dependence is shown by hierarchical elaboration down to *primary events*, which are not further elaborated. 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 only if all the input events occur, implying that the occurrence of all the input events causes the output event;
— OR gates: the output event occurs only if at least one of the input events occurs, implying that the occurrence of any input event causes the output event; and
— NOT gates: the output event occurs only if the (only) input event does not occur.

Figure 7 shows a fault tree with the top event, or hazard, A. An OR gate connects this event with two lower-level events, B and C, so A occurs if B or C or both occur. The event B occurs if and only if both of the two lower-level events connected to it through an AND gate occur. E is a primary event so it is not elaborated further. A *cut set* is a set of *event literals* such that the occurrence of all the events associated with the event literals in the set could allow the hazard to occur. An *event literal* is 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. For example, $\{H, E\}$ is a minimal cut set (MCS) of the fault tree in Figure 7. An MCS indicates a potential process vulnerability, which might be a flaw or weakness in the process 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*. An example of a single point of failure in Figure 7 is the event literal $\{\neg F\}$. The probability of a hazard occurring can be calculated if sufficient information about the probabilities of the events associated with the event literals in the MCSs is available.

Many software tools aid the manual construction of fault trees. However, when fault trees become large, as is typical, manual construction becomes error-prone and time-consuming. We developed a process-driven FTA tool to automate fault-tree construction and MCS calculation from process definitions [Chen 2010]. Given a process definition written in the Little-JIL language and a hazard specification, our tool automatically constructs a fault tree and then calculates its MCSs.

2.3.1. Identifying a hazard.

Once domain experts have validated that the process definition correctly represents the real-world process, the resulting fault trees can lead to the discovery of unforeseen process vulnerabilities and suggest modifications to improve

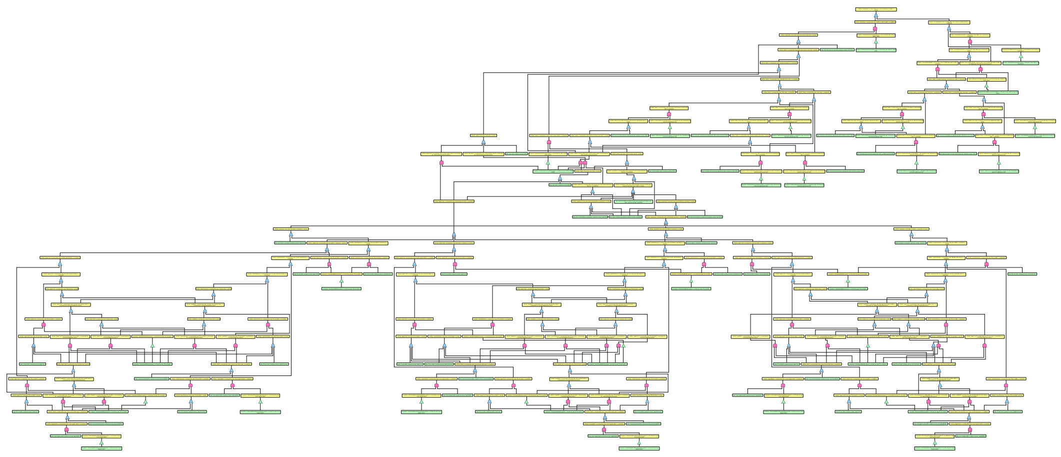


Fig. 8. The fault tree automatically derived for the hazard specification “the final vote totals tally reported to the Secretary of State is wrong.”

the robustness, security, or safety of the real-world process. Typically, domain experts can suggest multiple hazards from their own experiences. Furthermore, they can often evaluate the importance of a hazard, depending on its anticipated impact and the perceived probability of its occurrence. One hazard of particular interest, which we discuss in this paper, is “the final vote totals reported to the Secretary of State is wrong”. As noted above, if realized, this hazard could change the election result.

2.3.2. Tying the events and hazard to the process definition. To automatically generate a fault tree, the FTA tool requires the Little-JIL process definition and the hazard definition. To capture pertinent data and control flow information about process executions, the primary events used by our tool include incorrect artifacts as a step’s input or output, the incorrect execution of a step, and the throwing of any exceptions. The FTA tool requires a hazard to be defined as an artifact being wrong when input to or output from a step. Thus, the hazard “the final vote totals reported to the Secretary of State is wrong” is shown as the root of a tree of executed process steps, and defined in the tool as:

Artifact “finalTallies” to “report vote totals to Secretary of State” is wrong.

2.3.3. Deriving the fault tree. The fault tree is automatically derived from the process definition by tracing process artifact flow back through the steps to determine where artifacts may have been modified or created incorrectly. These incorrect artifacts may have been generated by the erroneous execution of a step or because a step failed to identify an artifact as being incorrect (e.g., when a step should have thrown an exception, but did not). A complete fault tree for the hazard “the final vote totals reported to the Secretary of State is wrong” is shown in Figure 8 for the simplified count votes subprocess shown in Figure 3 to give the reader an intuitive sense of the size and structure of a typical fault tree derived from a complex process. Figure 8 shows an optimized directed acyclic graph (DAG), where repeated nodes have been consolidated to reduce the size of the original tree, in this case by a factor of more than three. Before optimization, there were 1194 events and 1106 gates in the fault tree. After optimization, the DAG contains 735 events and 659 gates. From this example, it is clear why attempting to construct such structures by hand quickly becomes intractable.

2.3.4. Calculating Minimal Cut Sets. Once a fault tree has been automatically derived from the Little-JIL process definition, it could be manually inspected to identify MCSs by tracing paths containing these event literal back to the root. Given that the optimized fault tree generated for this hazard contains hundreds of nodes, however, the ability to automate the calculation of the MCSs becomes particularly valuable, giving the analyst guidance about where to look for vulnerabilities.

MCSs can be automatically calculated from the fault tree by using Boolean algebra techniques to represent and simplify flow equations, where the root node, or hazard, is equal to a disjunction of conjunctive clauses of event literals. Given such equations, the hazard occurs only if one or more of the conjunctive clauses evaluates to $true$, which can only happen if all the terms in a conjunctive clause evaluate to $true$, indicating all event literals in that clause must occur. Therefore each conjunctive clause forms a cut set. The substitution for the simple fault tree in Figure 7 therefore proceeds as follows:

$$\begin{aligned} A &= B + C \\ &= D * E + \neg F \\ &= (G + H) * E + \neg F \\ &= G * E + H * E + \neg F \end{aligned}$$

Thus the fault tree has 3 cut sets: $\{G, E\}$, $\{H, E\}$, and $\{\neg F\}$.

MCSs are then obtained by removing events until non-minimal cut sets become minimal. Applying this to the fault tree shown in Figure 8 results in 125 MCSs: 2 MCSs of size 2, 23 of size 3, 58 of size 4, 30 of size 5, and 12 of size 6.

2.3.5. Leveraging the results to alleviate process vulnerabilities . The fault tree provides a detailed description of the combinations of events that could lead to the hazard occurring. Certain traces through the fault tree structure, however, indicate more likely scenarios than others. By examining the MCSs, analysts can focus on those scenarios that seem more likely to occur or are likely to have a relatively larger impact. In this section, we examine a few of the smaller MCSs.

One MCS for the fault tree is:

MCS-1 (see Figure 9)

(1) Step “increment and announce appropriate tally” produces wrong “tallies”,
(2) Exception “VoteCountInconsistentException” is NOT thrown by step “increment and announce appropriate tally”,
(3) Exception “VoteCountInconsistentException” is NOT thrown by step “perform random audit”

In this case, the tallies produced by “count votes” are incorrect because the steps “increment and announce appropriate tally” and “perform random audit” are not performed correctly since neither step recognized a VoteCountInconsistentException. Figure 9 shows a portion of the fault tree relevant to this MCS. Our tool can automatically generate such a targeted fault tree, which we call a *mini fault tree*, for any selected MCS by extracting partial paths or scenarios from the original high-level fault tree.

MCS-1 indicates how the hazard could occur if all three of these steps are performed incorrectly. Domain expert election officials can then evaluate the likelihood of this happening. They might conclude that accidental misperformance is unlikely, but that if a single election official were assigned to perform all three steps, then that official could successfully undermine the correct results of this election process, either by attacking it intentionally or simply by being addled, confused, or incompetent on election day. This suggests that, in either case, the security and robustness of the process might be improved by putting in place checks to assure that different election officials are always assigned to these steps.

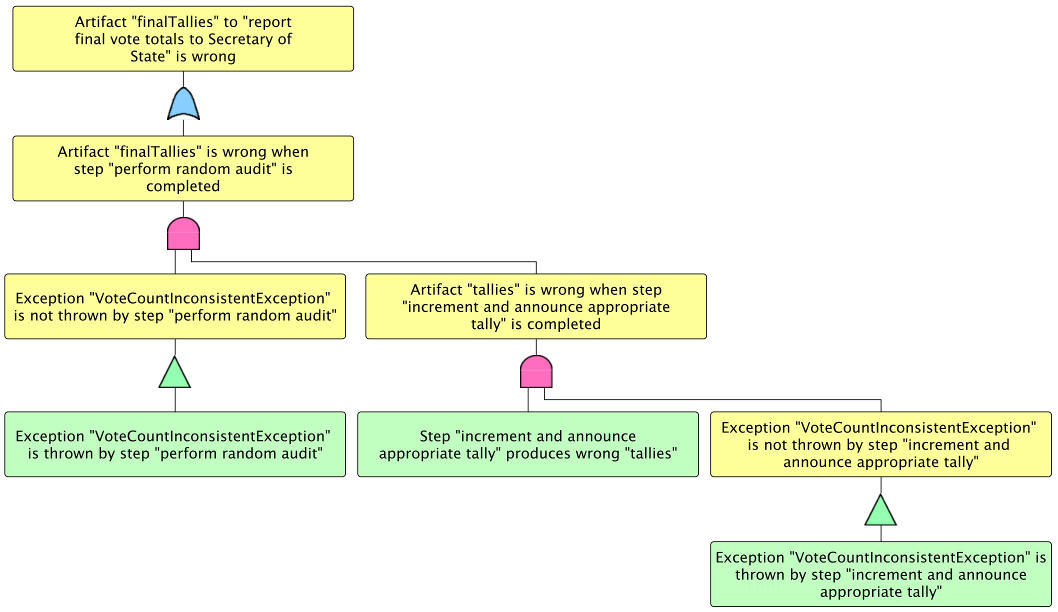

Artifact "finalTallies" to "report final vote totals to Secretary of State" is wrong
Artifact "finalTallies" is wrong when step "perform random audit" is completed
Exception "VoteCountInconsistentException" is not thrown by step "perform random audit"
Artifact "tallies" is wrong when step "increment and announce appropriate tally" is completed
Exception "VoteCountInconsistentException" is thrown by step "perform random audit"
Step "increment and announce appropriate tally" produces wrong "tallies"
Exception "VoteCountInconsistentException" is not thrown by step "increment and announce appropriate tally"
Exception "VoteCountInconsistentException" is thrown by step "increment and announce appropriate tally"

Fig. 9. MCS-1's mini fault tree

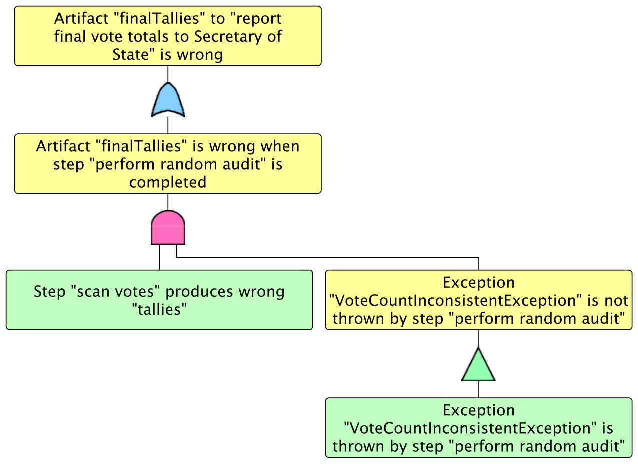

Artifact "finalTallies" to "report final vote totals to Secretary of State" is wrong
Artifact "finalTallies" is wrong when step "perform random audit" is completed
Step "scan votes" produces wrong "tallies"
Exception "VoteCountInconsistentException" is not thrown by step "perform random audit"
Exception "VoteCountInconsistentException" is thrown by step "perform random audit"

Fig. 10. MCS-2's mini fault tree

Another example MCS derivable from this fault tree is:

MCS-2 (see Figure 10)

(1) Step “scan votes” produces wrong “tallies”,
(2) Exception “VoteCountInconsistentException” is NOT thrown by step “perform random audit”

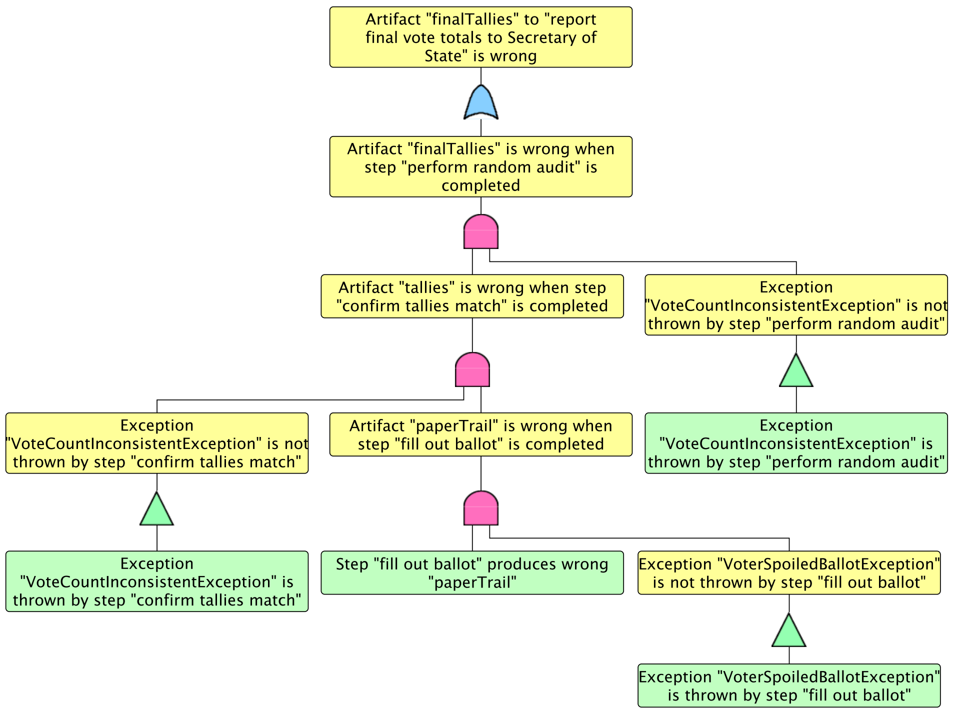

Artifact "finalTallies" to "report final vote totals to Secretary of State" is wrong
Artifact "finalTallies" is wrong when step "perform random audit" is completed
Artifact "tallies" is wrong when step "confirm tallies match" is completed
Exception "VoteCountInconsistentException" is not thrown by step "perform random audit"
Exception "VoteCountInconsistentException" is not thrown by step "confirm tallies match"
Artifact "paperTrail" is wrong when step "fill out ballot" is completed
Exception "VoteCountInconsistentException" is thrown by step "perform random audit"
Exception "VoteCountInconsistentException" is thrown by step "confirm tallies match"
Step "fill out ballot" produces wrong "paperTrail"
Exception "VoterSpoiledBallotException" is not thrown by step "fill out ballot"
Exception "VoterSpoiledBallotException" is thrown by step "fill out ballot"

Fig. 11. MCS-3's mini fault tree

The tallies produced by "count votes" are incorrect as a result of the step "perform random audit" not being carried out correctly. After receiving incorrect tallies from the "scan votes" step, the "perform random audit" step does not recognize a VoteCountInconsistentException. So the audit fails to catch the incorrect result. This MCS suggests another way in which the actions, either intentional or unintentional, of a single election official might cause the hazard to occur, again suggesting that the process be modified to assure that no single official is assigned to perform both of these steps.

The third example MCS demonstrates the potential impact of an optical scanner on election results. Since entire batches of ballots are scanned, incorrect performance of the scanner could have a very large impact. The MCS for this fault tree is:

MCS-3 (See Figure 11)

(1) Step "fill out ballot" produces wrong "paperTrail",
(2) Exception "VoterSpoiledBallotException" is NOT thrown by step "fill out ballot",
(3) Exception "VoteCountInconsistentException" is NOT thrown by step "confirm tallies match",
(4) Exception "VoteCountInconsistentException" is NOT throw by step "perform random audit"

In this case, the voter chooses the electronic voting option, and the step "fill out ballot" fails to produce the correct paper trail. In addition, no exception is thrown at this step nor at the later steps, "confirm tallies match" and "perform random audit". Such a scenario results in the wrong totalTallies being output from the step "count votes," and then input to "report final vote totals to Secretary of State".

This MCS example demonstrates a vulnerability introduced by electronic voting, and suggests the desirability of redundant checking to improve the robustness of this process. Indeed, the event *Step "fill out ballot" produces wrong "paperTrail"* appears in 18 out of 125 MCSs for this fault tree. It seems important to bring the significance of this step's possible failure to the election officials' attention so they can try either to minimize the probability of the event's occurrence, or implement redundant checking.

These examples have shown how MCSs can help a process developer identify areas of the process definition that may be problematic. These areas can then be explored with additional hazard specifications. The MCSs presented here highlight the need for election processes to be more robust with respect to possible incorrect performance, either intentional or unintentional, by the process performers. For example, if an election official unintentionally announces the tally incorrectly once , that might not make a big difference to the final outcome of the election. But if an official maliciously misperforms this step continually, and in collusion with the person doing the random audit, the election results might well be compromised. Similarly, if a defective scanner continually produces results that are not checked redundantly, the final election results could be changed significantly. Lastly, given how important and prevalent the paper trail seems to be for this hazard, the process developer may rightly deduce that it should be protected throughout the process to avoid tampering by malicious entities, or a duplicate trail should be generated on a printer in another location, or both.

3. RESULTS

The process definition for the election process was elicited by interviewing California election officials, studying the California election code, and firsthand observations. The resulting process definition was then validated with manual reviews performed by the election officials along with both model checking and fault tree analysis.

These efforts produced a full process definition with 98 steps: 71 step declarations and 27 references to previously defined steps. The agents include election officials, voters, optical scanners, voting machines, and precincts. The control flow often involves iteration, concurrency, and exceptional situations. Eight exceptions are defined for this process and one third of the twelve leaf steps may throw an exception that then must be handled appropriately. For the subprocess definition discussed in this paper, we ran the model checking and fault-tree analysis tools, both of which are implemented in Java on a laptop with two 2.5 GHz processors and 8 GB of memory using a UNIX-based operating system (OS X Yosemite Version 10.10.5) and Java 7 (Version 1.7.0 80-b15).

For the model checking, we elicited high-level requirements from the California election code. We applied the FLAVERS model checker to the process definition to verify the six high-level requirements represented by the nine lower-level properties shown in Figure 4. Each property had one or two events and three or four states (counting the violation state). Based upon our past experiences, we were not surprised to find that specifying these requirements precisely was a non-trivial task. In general we found that the requirements that involved a larger number of events were more difficult to specify. This was because of the need to specify the correspondingly greater diversity of event sequences needed in order to take into account the complete handling of all possible non-nominal process scenarios.

As is often the case with model checking, the verification of each property typically required several attempts where refinement of the process definition or property was needed before the process definition was shown to be consistent with the property for all possible executions. The simpler properties typically required two attempts while the paired properties required about half a dozen. Initially we often needed to add detail to the process definition to capture important aspects of the process. Later attempts typically required refinements to the properties to capture scope and repeatability aspects accurately. Often we needed to confer with the election officials to determine precisely what the actual requirements should be. In these cases, the property or process definition was often too restrictive, not fully representing all allowed behaviors when unusual special cases are taken into account. After making these improvements to the process definition and to the properties, FLAVERS reported that the process definition satisfies each property. For each property, the space needed was less than 64 MB and the time needed was at most six seconds. Although, we did not discover any problems in the actual process, this exercise had several benefits. The discussion with the election officials about the details of the process and the properties led to a better understanding of the process both by the analysts and by the election officials. Importantly, it led to a more complete and accurate process definition, which was especially important because this definition was then used as the basis for further analyses of properties such as robustness and security. And, indeed, there was at least one change made to a handbook in order to address concern about a situation that was brought to the attention of the authorities by our analyses.

After the model checking was completed, we applied our fault tree generator to the process definition. For the hazard ”the final vote totals reported to the Secretary of State is wrong,” the unoptimized fault tree had 1194 events and 1106 gates, while the optimized structure had 735 events and 659 gates. The generator needed less than 64 MB and took a little under a minute. We then applied the fault tree analyzer to the fault tree to compute its minimal cut sets. The analyzer found 125 minimal cut sets for this hazard, ranging in size from two to six events. The fault tree analyzer needed less than 64 MB and took less than one second. Observe that this hazard demonstrates some of the process vulnerabilities that electronic voting introduces. The event *step 'fill out ballot' produces wrong 'paperTrail'* appears in 18 out of 125 MCSs of the fault tree. Bringing this failure to the election officials' attention enables them to mitigate the risks created by over-reliance on the correctness of the execution of this step.

When used together, model checking and FTA are complementary approaches. Given an MCS with a small number of events having reasonably high probabilities, model checking could be used to identify process execution paths on which all those events occur, providing domain experts with insight into how to modify the process to increase its robustness. Applying model checking to the modified process definition using the original properties could then be used to check whether the modifications had introduced property violations, and FTA could determine whether these modifications had removed the MCS or whether additional paths might need to be considered.

Choosing how to change the process definition requires the input of domain experts. They determine how to respond to errors discovered using model checking. For the election process this involved how to change the process definition or the properties to reflect the real process and actual properties. The domain experts identified the hazards and resulting MCSs that they deem to be of highest priority. This usually begins with the identification of several small candidate MCSs with steps that are often performed incorrectly in the real-world process and so would benefit the most from risk mitigation, such as extra redundancy. Domain experts can also provide insight into steps that, in their experience, have a lower chance of being carried out incorrectly or are performed so infrequently that added redundancy would have little impact.

Once these analyses identify problems, process modifications can be introduced to try to address these problems. Such modifications would typically be proposed by the domain experts through discussions of the process definition to ensure that the proposed modifications are reasonable, would not interfere unacceptably with performance of the real-world process, and would be relatively easy to make. Reanalyzing the modified process definition ensures that it successfully corrects the problems without introducing more vulnerabilities in other parts of the process.

Finally, it is important to acknowledge that the effort required in order to perform the definition and analysis described here is substantial. While we did not record the exact amount of time spent eliciting the process, defining the process, and then analyzing the process, we note that it was substantial. The key people involved for the majority of the modeling and analysis effort included 2 Yolo County election officials, 2 UC Davis computer science professors, 3 UMass Amherst computer science professors, and 3 UMass Amherst computer science graduate students. This team worked to elaborate the process definition over a period of approximately six months and to specify the properties for more than three months.

3.1. Limitations of the approach

Our approach has some limitations. In particular, the rigor and definitiveness of the approach depends on the languages for representing the process definition, the properties, and the hazards having well-defined semantics. Imprecise language semantics cannot be translated unambiguously to the graph structures that are the basis for both kinds of analysis described in this paper. Moreover, even a precisely defined language could be insufficient to support our analyses if that language lacks the semantic features needed to define a complex process precisely or to represent the desired properties or hazards. Although the languages we have chosen have been sufficient to represent most of the semantics we have needed, all have known limitations.

We have already noted the difficulties in rendering imprecisely stated laws, regulations, and process requirements into precisely defined properties. Given the imprecision of these laws and requirements it is difficult to be sure that the associated properties are correct and complete. In our work we have used the PROPEL tool to facilitate the rendering of these properties as finite state automata. But PROPEL is implemented based upon the assumption that properties fall into one or more of a small set of property patterns. Although this is often the case, when a property cannot be represented by PROPEL a deterministic FSA can be used instead. And of course, there are many kinds of important properties that cannot be represented by an FSA. Similarly, it is important to note that our approach assumes that hazards are specified as the arrival of an incorrect artifact as input to a process step, or the generation of an incorrect artifact as output from a step. Other kinds of hazards are possible, and their specification would require a different approach from the one described in this paper.

It is also the case that graph structures can become quite large, and their analysis could become computationally intractable in the case of very large process definitions, especially when they make extensive use of such challenging semantic features as concurrency and intricate exception management.

4. RELATED WORK

Related to our work is prior work in elections and security, process definition and improvement, and model checking and FTA.

4.1. Elections and Security

The widespread introduction of electronic voting machines in the early-to-mid 2000s was originally intended to make the process of casting and counting votes faster and less costly while also eliminating ambiguous markings of ballots. But it introduced a new set of concerns about the accuracy, privacy, and security of elections. This has led to the introduction of Voluntary Voting System Guidelines [Election Assistance Commission 2005; Federal Election Commission 1990; 2002].[5]

Mercuri and Neumann [Mercuri and Neumann 2003] give an overview of how electronic voting systems can be verified and emphasize the importance of a verifiable paper trail. Saltman [Saltman 2003] outlines different techniques for performing audits to improve public confidence for both ballot and non-artifactual systems. The EAC

[5]A new standard [TGDC 2007] has been developed but not yet adopted.

is developing a set of election management Guidelines to complement the technical standards for voting equipment [Election Assistance Commission 2010]. These standards and guidelines, however, focus only on the electronic voting systems. Source code analysis [Kohno et al. 2004; Yasinsac et al. 2007; Office of the California Secretary of State 2007; Brunner 2007] and red team testing [RABA Innovative Solution Cell (RiSC) 2004; Proebstel et al. 2007; Bishop 2007; Brunner 2007; Springall et al. 2014; Wolchok et al. 2012] have shown ways to compromise these systems.

Other work has focused on the requirements an election must meet in dimensions such as privacy, anonymity, accessibility, and ballot design [Brennan Center Task Force on Voting System Security 2006; Lambrinoudakis et al. 2003; Mitrou et al. 2003]. Some have studied the Scantegrity voting system (e.g. [Chaum et al. 2008]), to enable voters to verify that their votes have been counted correctly without being able to prove to others how they voted, thereby preventing vote selling. The Scantegrity work focuses on the cryptographic protocols and system requirements that provide these properties.

Case studies of various e-voting systems describe some of the processes used to support those systems [Weldemariam et al. 2009; Tiella et al. 2006; Heiderich et al. 2011; Adida et al. 2011]. Our work focuses on rigorously modeling and analyzing the actual steps, activities, control flows, exception management, and data flows by which elections are conducted and that are designed to assure the correct performance and ultimate success of elections [Barr et al. 2007; Simidchieva et al. 2008]. This aspect of elections raises critical concerns about such issues as correctness, security, and privacy. For example, consider an election worker misplacing marked but uncounted ballots. The results of the election might be different were those ballots counted. Worse, suppose a malicious election official alters ballots to favor a particular candidate. Such an attack, called an *insider attack*, may well alter the results of the election. Insider attacks are of great concern in other realms as well, and are a topic of active research in the security community [Bishop et al. 2008; Pfleeger et al. 2010; Hunker and Probst 2011; Probst et al. 2010; Bishop et al. 2014; Sarkar et al. 2014]. Unlike work that focuses on usability of systems such as Helios [Karayumak et al. 2011b; Karayumak et al. 2011a; Acemyan et al. 2014; 2015], our work does not examine usability as such, but instead focuses on the consequences of an agent's actions.

Attacks against a *process* such as those identified above, are often more effective than attacks against electronic voting systems because they focus on people. Humans make mistakes, have different competency levels, and often have widely varying notions of security and privacy of elections [Hall et al. 2012]. Similarly, the processes that election officials design to carry out election tasks have vulnerabilities that may cause the tasks not to be completed, or be completed incorrectly. Analyzing the process of how elections are conducted may uncover or potential weaknesses that could compromise the election without compromising any of the electronic systems involved in the election. Further, it may be unclear how system errors or failures impact the results of an election. Thus, studying such processes helps build an understanding of how well they adhere to properties in such areas as security, integrity, and accuracy. Our work undertakes such a study.

4.2. Process Definition and Improvement

In satisfying such requirements, studying the effectiveness of a process falls within the area of process modeling and analysis, which "focuses [on] interacting behaviors among agents, regardless of whether a computer is involved in the transactions" [Curtis et al. 1992]. Raunak et al. apply process definition and analysis to elections to determine if fraudulent behavior can lead to incorrect election results [Raunak et al. 2006]. Simidchieva et al. extend this approach to determine if an election process definition meets selected requirements [Simidchieva et al. 2008], and further improve the robustness of election processes with fault-tree analysis [Simidchieva et al. 2010].

Audit procedures have been a fertile application field for process-oriented techniques. Antonyan et al. use AccuVote Optical Scan systems and a generic election process model to study how additional auditing may improve the integrity of elections [Antonyan et al. 2009]. The authors focus on how different election processes can affect prevention or detection of attacks on the underlying election systems. Our work focuses on how the election processes themselves may fail. Hall et. al. examine audit processes, specifically focusing on post-election audits [Hall 2008; Hall et al. 2009]. Like our work, the authors examine the processes for a specific county and use iterative process improvement before generalizing their approach. Our work, however, focuses not on audit processes, but on automatic analyses that lead to identifying violations of correctness, security, and robustness properties in specific election processes.

4.3. Model Checking

The history of using model checking techniques in security goes back at least to Lowe's application of the FDR model checker to find a subtle attack on the Needham-Schroeder authentication protocol [Lowe 1996]. While much of this work has focused on the analysis of protocols, other work has used model checking to analyze information flow (e.g., [Dimitrova et al. 2012]) or to verify access control policies (e.g., [Wolter et al. 2009]). A number of researchers have used model checking techniques to generate possible attacks. For instance, Sheyner et al. [Sheyner et al. 2002a] constructed atomic attacks, such as buffer overflows, and modeled a computer network as a finite state machine with transitions corresponding to those attacks. They then used model checking to generate an attack graph in which any path from the initial system state to a leaf node represents a sequence of atomic attacks that allow an intruder to violate a specified security property (such as "no intruder can achieve root access on host A").

Other researchers have used model checking techniques to analyze security aspects of business processes (e.g., [Armando and Ponta 2009]. A few papers have applied model checking approaches to election processes (e.g. [Raunak et al. 2006]). Closest to our approach is the work of Weldemariam et al. [Weldemariam and Villafiorita 2008; Villafiorita et al. 2009]. In this work, the authors model processes that incorporated best practice, defining how critical assets are to be managed, elaborated, and transformed, and then inject threats—actions that alter some features of an asset or allow some actors additional privileges. The extended model is then encoded for the NuSMV model checker and a property (such as "poll officers will never receive an altered version of the election software that can be run on the machines") is checked. Any generated counterexample provides an example attack. Our approach differs from theirs in that ours uses FTAs to devise more structured and detailed attacks.

Phan et al. have built on the work described here to use fault-tree analysis to find process vulnerabilities and build attack processes to exploit them [Phan et al. 2012]. Model checking is used to evaluate the robustness of the process to derived attacks.

4.4. Fault-Tree Analysis

Numerous safety-critical industries, including the aerospace, nuclear power, and automotive industries, use FTA. Brooke et al. demonstrate that fault trees may also be used to analyze security-critical systems [Brooke and Paige 2003]. Helmer et al. used augmented Software Fault Trees (SFTs), attack trees with temporal order, to model intrusions [Helmer et al. 2002]. In their models, the root node represents the intrusion and an MCS contains events to be monitored to detect intrusions. Zhang et al. use fault trees for vulnerability evaluation [Zhang et al. 2005]. Rushdi and Ba-Rukab apply fault trees to measure a system's exposure to a vulnerability [Rushdi and Ba-rukab 2005]. Yee discusses how safety cases, a construct similar to fault trees, may be used to increase confidence in voting systems [Yee 2007].

Leveson et al. [Leveson et al. 1991] proposed using fault trees to guide analysts in identifying errors that cause Ada programs to produce incorrect outputs. The incorrect output is represented as the hazard. Templates, one for each kind of Ada statement, are used to elaborate intermediate events to construct the full fault tree. Friedman developed a template-based tool to construct fault trees from a Pascal program and a software-caused hazard [Friedman 1993]. Pai and Bechta Dugan [Pai and Bechta Dugan 2002] showed an algorithm to automatically derive fault trees from UML models, but note the ambiguousness of the resulting fault trees due to the semi-formal nature of UML. Like the programming language based approaches, our approach uses language-based templates for a well-defined language, but we also incorporate a more detailed model of data and control flow to provide a richer representation of the faults that could arise.

Like fault trees, attack trees are hierarchical logic diagrams in which one event is represented as a logical combination of lower-level events [Schneier 1999]. They are used to model the different paths an attacker may take to reach an objective. Moore et al. used attack trees to model attacks and document them [Moore et al. 2001]. Lazarus created a catalog of election attacks in the form of a single attack tree, attempting to provide a threat model and a quantitative threat evaluation approach intended to be reusable across different jurisdictions [Lazarus 2010] . Attack trees have also been used in penetration testing [McDermott 2001], in identifying insider attacks [Ray and Poolsapassit 2005], and for forensics [Bishop et al. 2009; Peisert 2007; Peisert et al. 2007; Poolsapassit and Ray 2007]. Nai Fovino et al. combine fault trees and attack trees for quantitative security risk assessment [Nai Fovino et al. 2009]. Attack graphs [Phillips and Swiler 1998; Sheyner et al. 2002b] are similar to attack trees, but may be cyclical and do not use logic operators between nodes. Attack tree analysis generally assumes that faults arise from malicious intent. Since we do not ascribe an intent to how these faults arise, we focus on FTA in this paper. As fault trees and attack trees are structurally equivalent, the analyses described here for fault trees would apply equally well to attack trees.

5. CONCLUSION

This paper presents a systematic approach for determining the adherence, or lack thereof, of processes to a wide variety of safety and security properties. The approach is especially valuable for processes that involve human performers. We describe and illustrate the application of this approach by defining and analyzing a key part of an election process. With guidance from election officials and election code requirements, we develop a definition of how an election is to be conducted and specifications of properties to be adhered to or hazards to be guarded against. We express our process definition in Little-JIL, and iteratively refine the definition with the help of domain experts. We perform model checking and FTA on this definition and identify errors and vulnerabilities that suggest problems in the election process. The two analysis techniques provide powerful, complementary ways to identify process defects and vulnerabilities.

Taking the analysis results back to the domain experts, we work with them to identify process modifications to remove these errors and reduce vulnerabilities. By proposing a definition of the new process and applying our analysis techniques, we can identify potential problems and address them before the problems occur in practice, when it may be too late to remedy them effectively. This approach can also be used to study alternative hypothetical modifications before make a final decision. Our experience with this analysis-based, iterative improvement suggests that it can be used on a wide range of processes to systematically address many different kinds of concerns, especially for processes that coordinate technologies and human activity.

ACKNOWLEDGMENTS

We thank Freddie Oakley, the former Clerk-Recorder for Yolo County; Tom Stanionis, her chief deputy; and Elaine Ginnold, the former Registrar of Voters for Marin County, for providing details of the election processes, and valuable feedback on our models and work. This material is based upon work supported by the National Science Foundation under Awards CNS-0831002,CCF-0905530, CNS-1049738, CCF-0820198, and IIS-0705772, and by the National Institute for Standards and Technology under award O-60NANB13D165. 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 or NIST.

REFERENCES

Claudia Z. Acemyan, Philip Kortum, Michael D. Byrne, and Dan S. Wallach. 2014. Usability of Voter Verifiable, End-to-end Voting Systems: Baseline Data for Helios, Prêt à Voter, and Scantegrity II. *USENIX Journal of Election Technology and Systems* 2, 3 (July 2014), 26–56.

Claudia Z. Acemyan, Philip Kortum, Michael D. Byrne, and Dan S. Wallach. 2015. From Error to Error: Why Voters Could not Cast a Ballot and Verify Their Vote With Helios, Prêt à Voter, and Scantegrity II. *USENIX Journal of Election Technology and Systems* 3, 2 (July 2015), 1–25.

Ben Adida, Olivier de Marneffe, Olivier Pereira, and Jean-Jacques Quisquater. 2011. Electing a University President Using Open-Audit Voting: Analysis of Real-World use of Helios. In *Proceedings of the 2011 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections*. USENIX Association, Berkeley, CA, USA.

Ilkay Altintas, Chad Berkley, Efrat Jaeger, Matthew Jones, Bertram Ludäscher, and Steve Mock. 2004. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management (SSDBM04)*. IEEE Computer Society, Los Alamitos, CA, USA, 423.

Tigran Antonyan, Seda Davtyan, Sotirios Kentros, Aggelos Kiayias, Laurent Michel, Nicolas Nicolaou, Alexander Russell, and Alexander A. Shvartsman. 2009. State-Wide Elections, Optical Scan Voting Systems, and the Pursuit of Integrity. *IEEE Transactions on Information Forensics and Security* 4, 4 (Dec. 2009), 597–610.

Alessandro Armando and Serena Elisa Ponta. 2009. Model Checking of Security-Sensitive Business Processes. In *6th International Workshop on Formal Aspects of Security and Trust, FAST 2009 (LNCS)*, Vol. 5983. Springer-Verlag, Eindhoven, 66–80.

George S. Avrunin, Lori A. Clarke, Elizabeth A. Henneman, and Leon J. Osterweil. 2006. Complex Medical Processes as Context for Embedded Systems. *ACM SIGBED Review* 3, 4 (October 2006), 9–14.

George S. Avrunin, Lori A. Clarke, Leon J. Osterweil, Stefan C. Christov, Bin Chen, Elizabeth A. Henneman, Philip L. Henneman, Lucinda Cassells, and Wilson Mertens. 2010. Experience Modeling and Analyzing Medical Processes: UMass/Baystate Medical Safety Project Overview. In *Proceedings of the 1st ACM International Health Informatics Symposium*. ACM, New York, NY, USA, 316–325.

Christel Baier and Joost-Pieter Katoen. 2008. *Principles of Model Checking*. MIT Press.

Earl Barr, Matt Bishop, and Mark Gondree. 2007. Fixing Federal E-Voting Standards. *Commun. ACM* 50, 3 (March 2007), 19–24.

Matt Bishop. 2007. *Overview of Red Team Reports*. Technical Report. Office of the Secretary of State of California, Sacramento, CA.

Matt Bishop, Heather M. Conboy, Huong Phan, Borislava I. Simidchieva, George S. Avrunin, Lori A. Clarke, Leon J. Osterweil, and Sean Peisert. 2014. Insider Threat Identification by Process Analysis. In *Proceedings of the 2014 IEEE Workshop on Research in Insider Threats*. IEEE Computer Society, Los Alamitos, CA, USA, 251–264.

Matt Bishop, Sophie Engle, Sean Peisert, Sean Whalen, and Carrie Gates. 2008. We Have Met the Enemy and He Is Us. In *Proceedings of the 2008 New Security Paradigms Workshop (NSPW '08)*. ACM, New York, NY, USA, 1–12.

Matt Bishop, Sean Peisert, Candice Hoke, Mark Graff, and David Jefferson. 2009. E-Voting and Forensics: Prying Open the Black Box. In *Proceedings of the 2009 Electronic Voting Technology Workshop / Workshop on Trustworthy Computing*. USENIX Association, Berkeley, CA, USA, 3:1–3:20.

Brennan Center Task Force on Voting System Security. 2006. *The Machinery of Democracy: Protecting Elections in an Electronic World*. Brennan Center for Justice, New York, NY.

Phillip J. Brooke and Richard F. Paige. 2003. Fault Trees for Security System Design and Analysis. *Computers & Security* 22, 3 (April 2003), 256–264.

Jennifer L. Brunner. 2007. *Project EVEREST: Evaluation and Validation of Election-Related Equipment,* *Standards, and Testing*. Office of the Ohio Secretary of State, Columbus, OH.

Aaron G. Cass, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, Stanley M. Sutton Jr, and Alexander Wise. 2000. Little-JIL/Juliette: A Process Definition Language and Interpreter. In *Proceedings of the 22nd International Conference on Software Engineering*. ACM, New York, NY, USA, 754–757.

David Chaum, Richard Carback, Jeremy Clark, Aleksander Essex, Stefan Popoveniuc, Ronald L Rivest, Peter YA Ryan, Emily Shen, and Alan T Sherman. 2008. Scantegrity II: End-to-End Verifiability for Optical Scan Election Systems Using Invisible Ink Confirmation Codes. In *Proceedings of the 2008 USENIX/ACCURATE Electronic Voting Technology Workshop*. USENIX Association, Berkeley, CA, USA, 14:1–14:13. https://www.usenix.org/legacy/events/evt08/tech/full_papers/chaum/chaum.pdf

Bin Chen. 2010. *Improving Processes Using Static Analysis Techniques*. Ph.D. Dissertation. Universiry of Massachusetts Amherst.

Bin Chen, George S. Avrunin, Elizabeth A. Henneman, Lori A. Clarke, Leon J. Osterweil, and Philip L. Henneman. 2008. Analyzing Medical Processes. In *Proceedings of the 30th International Conference on Software Engineering*. ACM, New York, NY, USA, 623–632.

Edmund M. Clarke, Jr., Orna Grumberg, and Doron A. Peled. 2000. *Model Checking*. MIT Press, Cambridge.

Lori A. Clarke, George A. Avrunin, and Leon J. Osterweil. 2008. Using Software Engineering Technology to Improve the Quality of Medical Processes. In *Companion of the 30th International Conference on Software Engineering*. ACM, New York, NY, USA, 889–898.

Rachel L. Cobleigh, George S. Avrunin, and Lori A. Clarke. 2006. User Guidance for Creating Precise and Accessible Property Specifications. In *Proceedings of the 14th ACM SIGSOFT Symposium on the Foundations of Software Engineering*. ACM, New York, NY, USA, 208–218.

Bill Curtis, Marc I. Kellner, and Jim Over. 1992. Process Modeling. *Commun. ACM* 35, 9 (September 1992), 75–90.

W. Edwards Deming. 1982. *Out of the Crisis*. MIT Press, Cambridge, MA.

Rayna Dimitrova, Bernd Finkbeiner, Máté Kovács, Markus N. Rabe, and Helmut Seidl. 2012. Model Checking Information Flow in Reactive Systems. In *Proceedings of the 13th International Conference on Verification, Model Checking, and Abstract Interpretation (Lecture Notes in Computer Science)*, Vol. 7148. Springer Berlin Heidelberg, Berlin, Germany, 169–185.

Matthew B. Dwyer, George S. Avrunin, and James C. Corbett. 1999. Patterns in Property Specifications for Finite-State Verification. In *Proceedings of the Twenty-First International Conference on Software Engineering*. ACM, New York, NY, USA, 411–420.

Matthew B. Dwyer, Lori A. Clarke, Jamieson M. Cobleigh, and Gleb Naumovich. 2004. Flow Analysis for Verifying Properties of Concurrent Software Systems. *ACM Transactions on Software Engineering and Methodology* 13, 4 (Oct. 2004), 359–430.

Election Assistance Commission 2005. *2005 Voluntary Voting Systems Guidelines*. Election Assistance Commission, Washington, DC.

Election Assistance Commission 2010. *Election Management Guidelines*. Election Assistance Commission, Washington, DC.

Aaron M. Ellison, Leon J. Osterweil, Lori Clarke, Julian L. Hadley, Alexander Wise, Emery Boose, David R. Foster, Allen Hanson, David Jensen, Paul Kuzeja, Edward Riseman, Howard Schultz, and Paula Kuzeja. 2006. Analytic Webs Support the Synthesis of Ecological Data Sets. *Ecology* 87, 6 (2006), 1345–1358.

Clifton A. Ericson II. 1999. Fault Tree Analysis - A History. In *Proceedings of The 17th International System Safety Conference*. System Safety Society Publications, Unionville, VA, USA, 1–9.

Federal Election Commission 1990. *Performance and Test Standards for Punchcards, Marksense, and Direct Recording Electronic Voting Systems*. Federal Election Commission, Washington, DC.

Federal Election Commission 2002. *Voting Systems Standards*. Federal Election Commission, Washington, DC.

M.A. Friedman. 1993. Automated software fault-tree analysis of Pascal programs. In *Proceedings of the 1993 Annual Symposium on Reliability and Maintainability*. IEEE Computer Society, Los Alamitos, CA, USA, 458–461.

Diimitrios Georgakopoulos, Mark Hornick, and Amit Sheth. 1995. An Overview of Workflow Management: From Process Modeling to Workflow Automation Infrastructure. *Distributed and Parallel Databases* 3, 2 (April 1995), 119–153.

Joseph Lorenzo Hall. 2008. Improving the security, transparency and efficiency of California's 1% manual tally procedures. In *Proceedings of the 2008 USENIX/ACCURATE Electronic Voting Technology Workshop (EVT'08)*. USENIX Association, Berkeley, CA, USA, 1–12.

Joseph Lorenzo Hall, Emily Barabas, Gregory Shapiro, Coye Cheshire, and Deirdre K Mulligan. 2012. Probing the Front Lines: Pollworker Perceptions of Security & Privacy. In *Proceedings of the 2012 Workshop* *on Electronic Voting Technology / Workshop on Trustworthy Elections*. USENIX Association, Berkeley, CA, USA, 2:1–2:15.

Joseph Lorenzo Hall, Luke W. Miratrix, Philip B. Stark, Melvin Briones, Elaine Ginnold, Freddie Oakley, Martin Peaden, Gail Pellerin, Tom Stanionis, and Tricia Webber. 2009. Implementing Risk-Limiting Post-Election Audits in California. In *Proceedings of the 2009 Electronic Voting Technology Workshop / Workshop on Trustworthy Computing*. USENIX Association, Berkeley, CA, USA, 19:1–19:24.

Mario Heiderich, Tilman Frosch, Marcus Niemietz, and Jörg Schwenk. 2011. The Bug That Made Me President a Browser– and Web-Security Case Study on Helios Voting. In *E-Voting and Identity: Proceedings of the Third International Conference, VoteID 2011 (Lecture Notes in Computer Science)*. 89–103.

Guy Helmer, Johnny Wong, Mark Slagell, Vasant Honavar, Les Miller, and Robyn Lutz. 2002. A Software Fault Tree Approach to Requirements Analysis of an Intrusion Detection System. *Requirements Engineering* 7, 4 (December 2002), 207–220.

Elizabeth A. Henneman, George S. Avrunin, Lori A. Clarke, Leon J. Osterweil, Chester Andrzejewski, Jr., Karen Merrigan, Rachel Cobleigh, Kimberly Frederick, Ethan Katz-Bassett, and Philip L. Henneman. 2007. Increasing Patient Safety and Efficiency in Transfusion Therapy Using Formal Process Definitions. *Transfusion Medicine Reviews* 21, 1 (January 2007), 49–57.

L. Howard Holley and Barry K. Rosen. 1980. Qualified data flow problems. In *Proceedings of the 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming languages*. ACM, New York, NY, USA, 68–82.

Jeffrey Hunker and Christian W. Probst. 2011. Insiders and Insider Threats—An Overview of Definitions and Mitigation Techniques. *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications* 2, 1 (2011), 4–27.

William A. Hyman and Erin Johnson. 2008. Fault Tree Analysis of Clinical Alarms. *Journal of Clinical Engineering* 33, 2 (April 2008), 85–94.

Radu Iosif, Matthew B. Dwyer, and John Hatcliff. 2005. Translating Java for Multiple Model Checkers: The Bandera Back End. *Formal Methods in System Design* 26, 2 (March 2005), 137–180.

Fatih Karayumak, Michaela Kauer, Maina Olembo, Tobias Volk, and Melanie Volkamer. 2011a. User Study of the Improved Helios Voting System Interfaces. In *Proceedings of the First Workshop on Socio-Technical Aspects in Security and Trust*. IEEE Computer Society Press, Los Alamitos, CA, USA, 37–44.

Fatih Karayumak, Maina Olembo, Michaela Kauer, and Melanie Volkamer. 2011b. Usability Analysis of Helios - An Open Source Verifiable Remote Electronic Voting System. In *Proceedings of the 2011 Electronic Voting Technology Workshop / Workshop on Trustworthy Elections*. USENIX Association, Berkeley, CA.

Tadayoshi Kohno, Adam Stubblefield, Aviel D. Rubin, and Dan S. Wallach. 2004. Analysis of an Electronic Voting System. In *Proceedings of the 2004 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Los Alamitos, CA, USA, 27–40. DOI:http://dx.doi.org/10.1109/SECPRI.2004.1301313

Costas Lambrinoudakis, Vassilis Tsoumas, Maria Karyda, and Spyros Ikonomopoulos. 2003. Secure Electronic Voting: The Current Landscape. In *Secure Electronic Voting*. Advances in Information Security, Vol. 7. Kluwer Academic Publishers, Boston, MA, Chapter 7, 101–122.

Eric L. Lazarus. 2010. Change Result of Election Successfully. (2010).

N.G. Leveson, S.S. Cha, and T.J. Shimeall. 1991. Safety verification of Ada programs using software fault trees. *Software, IEEE* 8, 4 (jul 1991), 48 –59.

Gavin Lowe. 1996. *Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, Berlin, Germany, Chapter Breaking and Fixing the Needham-Schroeder Public-Key Protocol using FDR, 147–166.

Declan McCullagh. 2007. E-voting predicament: Not-so-secret ballots. (August 2007). http://www.cnet.com/news/e-voting-predicament-not-so-secret-ballots/

John P. McDermott. 2001. Attack Net Penetration Testing. In *Proceedings of the 2000 Workshop on New Security Paradigms (NSPW)*. ACM, New York, NY, USA, 15–21.

Rebecca T. Mercuri and Peter G. Neumann. 2003. Verification for Electronic Balloting Systems. In *Secure Electronic Voting*. Advances in Information Security, Vol. 7. Kluwer Academic Publishers, Boston, MA, Chapter 3, 31–42.

Shin-ichi Minato. 1996. *Binary Decision Diagrams and Applications for VLSI CAD*. Kluwer Academic Publishers, Boston, MA, USA.

Lilian Mitrou, Dimitris Gritzalis, Sokratis Katsikas, and Gerald Quirchmayr. 2003. Electronic Voting: Constitutional and Legal Requirements, and Their Technical Implications. In *Secure Electronic Voting*. Advances in Information Security, Vol. 7. Kluwer Academic Publishers, Boston, MA, Chapter 4, 43–60.

A. P. Moore, R. J. Ellison, and R. C. Linger. 2001. *Attack Modeling for Information Security and Survivability*. Technical Report. Carnegie Mellon University, Software Engineering Institute.

Igor Nai Fovino, Marcelo Masera, and Alessio De Cian. 2009. Integrating Cyber Attacks Within Fault Trees. *Reliability Engineering and System Safety* 94, 9 (2009), 1394–1402.

National Association of Secretaries of State (NASS). 2007. Survey Post Election Audits. (Sept. 2007).

Office of the California Secretary of State 2007. *Top to Bottom Review of Electronic Voting Machines*. Office of the California Secretary of State, Sacramento, CA.

Leon J. Osterweil, George S. Avrunin, Bin Chen, Lori A. Clarke, Rachel Cobleigh, Elizabeth A. Henneman, and Philip L. Henneman. 2007. Engineering Medical Processes to Improve Their Safety. In *Situational Method Engineering: Fundamentals and Experiences*. IFIP International Federation for Information Processing, Vol. 244. Springer, Boston, MA, 267–282.

G.J. Pai and J. Bechta Dugan. 2002. Automatic synthesis of dynamic fault trees from UML system models. In *Proceedings of the 13th International Symposium on Software Reliability Engineering*. IEEE Computer Society, Los Alamitos, CA, USA, 243–254.

Sean Peisert, Matt Bishop, Sidney Karin, and Keith Marzullo. 2007. Toward Models for Forensic Analysis. In *Proceedings of the Second International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE)*. IEEE Computer Society, Los Alamitos, CA, USA, 3–15.

Sean Philip Peisert. 2007. *A Model of Forensic Analysis Using Goal-Oriented Logging*. Ph.D. Dissertation. Department of Computer Science and Engineering, University of California, San Diego.

Shari Lawrence Pfleeger, Joel B. Predd, Jeffrey Hunker, and Carla Bulford. 2010. Insiders Behaving Badly: Addressing Bad Actors and Their Actions. *IEEE Transactions on Information Forensics and Security* 5, 1 (Mar. 2010), 169–179.

Huong Phan, George Avrunin, Matt Bishop, Lori A. Clarke, and Leon J. Osterweil. 2012. A Systematic Process-Model-Based Approach for Synthesizing Attacks and Evaluating Them. In *Proceedings of the 2012 USENIX/ACCURATE Electronic Voting Technology Workshop*. USENIX Association, Berkeley, CA, USA, 10:1–10:16.

Cynthia Phillips and Laura Painton Swiler. 1998. A Graph-Based System for Network-Vulnerability Analysis. In *Proceedings of the 1998 New Security Paradigms Workshop*. ACM, New York, NY, USA, 71–79.

Nayot Poolsapassit and Indrajit Ray. 2007. Investigating Computer Attacks Using Attack Trees. In *Advances in Digital Forensics III*. IFIP International Federation for Information Processing, Vol. 242. Springer, Boston, MA, 331–343.

Christian W. Probst, Jeffrey Hunker, Dieter Gollmann, and Matt Bishop (Eds.). 2010. *Insider Threats in Cyber Security*. Advances in Information Security, Vol. 49. Springer, New York, NY, USA.

Elliot Proebstel, Sean Riddle, Francis Hsu, Justin Cummins, Freddie Oakley, Tom Stanionis, and Matt Bishop. 2007. An Analysis of the Hart Intercivic DAU eSlate. In *Proceedings of the USENIX Workshop on Accurate Electronic Voting Technology*. USENIX Association, Berkeley, CA, USA, 3:1–3:12.

RABA Innovative Solution Cell (RiSC). 2004. *Trusted Agent Report Diebold AccuVote-TS Voting System*. RABA Technologies, Columbia, MD.

Mohammad S. Raunak, Bin Chen, Amr Elssamadisy, Lori A. Clarke, and Leon J. Osterweil. 2006. Definition and Analysis of Election Processes. In *Software Process Change*. Lecture Notes in Computer Science, Vol. 3966. Springer, Berlin, 178–185.

Indrajit Ray and Nayot Poolsapassit. 2005. Using Attack Trees to Identify Malicious Attacks from Authorized Insiders. In *Computer Security – ESORICS 2005*. Lecture Notes in Computer Science, Vol. 3679. Springer, Berlin, 231–246.

Ali M. Rushdi and Omar M. Ba-rukab. 2005. Fault-Tree Modelling of Computer System Security. *International Journal of Computer Mathematics* 82, 7 (July 2005), 805–819.

Roy G. Saltman. 2003. Public Confidence and Auditability in Voting Systems. In *Secure Electronic Voting*. Advances in Information Security, Vol. 7. Kluwer Academic Publishers, Boston, MA, Chapter 8, 31–42.

Anandarup Sarkar, Sean Kohler, Sean Riddle, Bertram Ludaescher, and Matt Bishop. 2014. Insider Attack Identification and Prevention Using a Declarative Approach. In *2014 IEEE Security and Privacy Workshops*. IEEE Computer Society, Los Alamitos, CA, USA, 251–264.

Bruce Schneier. 1999. Modeling Security Threats. *Dr. Dobb's Journal* 22, 12 (December 1999), 4–6.

Walter A. Shewhart. 1931. *Economic Control of Quality of Manufactured Product*. D. Van Nostrand Company, New York, NY.

O. Sheyner, J. Haines, S. Jha, R. Lippmann, and J.M. Wing. 2002a. Automated generation and analysis of attack graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. 273 – 284.

Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippmann, and Jeannette M. Wing. 2002b. Automated Generation and Analysis of Attack Graphs. In *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. IEEE Computer Society, Los Alamitos, CA, USA, 273–284.

Borislava I. Simidchieva, Sophie J. Engle, Michael Clifford, Alicia Clay Jones, Sean Peisert, Matt Bishop, Lori A. Clarke, and Leon J. Osterweil. 2010. 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)*. USENIX Association, Berkeley, CA, USA, 6:1–6:16.

Borislava I. Simidchieva, Matthew S. Marzilli, Lori A. Clarke, and Leon J. Osterweil. 2008. Specifying and Verifying Requirements for Election Processes. In *Proceedings of the International Conference on Digital Government Research*. Digital Government Society of North America, 63–72.

John A. Simpson and Edmund S. C. Weiner (Eds.). 1991. *The Oxford English Dictionary* (2nd ed.). Clarendon Press, Oxford, UK.

Rachel L. Smith, George S. Avrunin, Lori A. Clarke, and Leon J. Osterweil. 2002. PROPEL: An Approach Supporting Property Elucidation. In *Proceedings of the Twenty-Fourth International Conference on Software Engineering*. ACM, New York, NY, USA, 11–21.

Drew Springall, Travis Finkenauer, Zakir Durumeric, Jason Kitcat, Harri Hursti, Margaret MacAlpine, and J. Alex Halderman. 2014. Security Analysis of the Estonian Internet Voting System. In *Proceedings of the 23rd ACM Conference on Computer and Communication Security*. ACM, New York, NY, USA, 703–715.

TGDC. 2007. *Voluntary Voting System Guidelines Recommendations to the Election Assistance Commission*. Technical Report. Technical Guidelines Development Committee, Election Assistance Commission, Washington, DC, USA.

Roberto Tiella, Adolfo Villafiorita, and Silvia Tomasi. 2006. Specification of the Control Logic of an eVoting System in UML: the ProVotE Experience. In *Proceedings of 5th International Workshop on Critical Systems Development Using Modeling Languages*.

VerifiedVoting. 2013. Post Election Audit. (2013). https://www.verifiedvoting.org/resources/post-election-audit

Adolfo Villafiorita, Komminist Weldemariam, and Roberto Tiella. 2009. Development, Formal Verification, and Evaluation of an E-Voting System With VVPAT. *IEEE Transactions on Information Forensics and Security* 4, 4 (Dec. 2009), 651–661.

J.R. Ward, M.N. Lyons, S. Barclay, J. Anderson, P. Buckle, and P.J. Clarkson. 2007. Using Fault Tree Analysis (FTA) in Healthcare: a Case Study of Repeat Prescribing in Primary Care. In *Proceedings of Patient Safety Research: Shaping the European Agenda*.

Komminist Weldemariam, Richard A. Kemmerer, and Adolfo Villafiorita. 2009. *Specification and Analysis of the Electronic Voting Process for the ES&S Voting System*. Technical report. Dept. of Computer Science, University of California at Santa Barbara, Santa Barbara, CA, USA.

Komminist Weldemariam and Adolfo Villafiorita. 2008. Modeling and Analysis of Procedural Security in (e)Voting: the Trentino's Approach and Experiences. In *Proceedings of the 2008 USENIX / ACCURATE Electronic Voting Technology Workshop*. USENIX Association, Berkeley, CA, USA, 1–10.

Oliver Wiegert. 1998. *Business Process Modeling and Workflow Definition with UML*. SAP AG.

Alexander Wise, Aaron G. Cass, Barbara Staudt Lerner, Eric K. McCall, Leon J. Osterweil, and Jr. Stanley M. Sutton. 2000. Using Little-JIL to Coordinate Agents in Software Engineering. In *Proceedings of the Fifteenth IEEE International Conference on Automated Software Engineering*. IEEE Computer Society, Los Alamitos, CA, USA, 155–163.

Scott Wolchok, Eric Wustrow, Dawn Isabel, and J. Alex Halderman. 2012. Attacking the Washington, D.C. Internet Voting System. In *Proceedings of the 16th International Conference on Financial Cryptography and Data Security (Lecture Notes in Computer Science)*, Angelos D. Keromytis (Ed.), Vol. 7397. Springer Berlin Heidelberg, Berlin, Germany, 114–128.

Christian Wolter, Philip Miseldine, and Christoph Meinel. 2009. Verification of Business Process Entailment Constraints Using SPIN. In *First International Symposium on Engineering Secure Software and Systems (LNCS)*. Springer, Leuven, Belgium, 1–15.

Alec Yasinsac, David Wagner, Matt Bishop, Ted Baker, Breno de Medeiros, Gary Tyson, Michael Shamos, and Mike Burmester. 2007. *Software Review and Security Analysis of the ES&S iVoteronic 8.0.1.2 Voting Machine Firmware*. Security and Assurance in Information Technology Laboratory, Florida State University, Tallahassee, FL.

Ka-Ping Yee. 2007. *Building Reliable Voting Machine Software*. Technical Report EECS-2007-167. Dept. of Electrical Engineering and Computer Science, University of California at Berkeley, Berkeley, CA, USA.

Tao Zhang, Mingzeng Hu, Xiaochun Yun, and Yongzheng Zhang. 2005. Computer Vulnerability Evaluation Using Fault Tree Analysis. In *Information Security Practice and Experience (Lecture Notes in Computer Science)*, Vol. 3439. Springer, Berlin, 302–313.

Received Month Year; revised Month Year; accepted Month Year