Book Title
Book Editors
IOS Press, 2003

Scenario Graphs Applied to Security (Summary Paper)

Jeannette M. Wing
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213 US

Abstract. Traditional model checking produces one counterexample to illustrate a violation of a property by a model of the system. Some applications benefit from having *all* counterexamples, not just one. We call this set of counterexamples a *scenario graph*. In this paper we present two different algorithms for producing scenario graphs and explain how scenario graphs are a natural representation for *attack graphs* used in the security community.

Keywords. Scenario graphs, attack graphs, counterexamples, model checking, security.

1. Overview

Model checking is a technique for determining whether a formal model of a system satisfies a given property. If the property is false in the model, model checkers typically produce a single counterexample. The developer uses this counterexample to revise the model (or the property), which often means fixing a bug in the design of the system. The developer then iterates through the process, rechecking the revised model against the (possibly revised) property.

Sometimes, however, we would like *all* counterexamples, not just one. Rather than produce one example of how the model does not satisfy a given property, why not produce all of them at once? We call the set of all counterexamples a *scenario graph*. For a traditional use of model checking, e.g., to find bugs, each path in the graph represents a counterexample, i.e., a failure scenario. In our application to security, each path represents an attack, a way in which an intruder can attack a system. Attack graphs are a special case of scenario graphs.

This paper gives two algorithms for producing scenario graphs and summarizes our findings in the application of scenario graphs to security. The first algorithm was published in [6]; the second in [5].

2. Relation of This Paper to Previously Published Work

All the ideas in this paper can be found in greater detail in Oleg Sheyner's doctoral dissertation [11]. Portions of his dissertation have appeared in conference and jour-

nal papers [6,9,7,5,10]. Thus, this paper does not present any new work. Except for Sheyner's thesis, however, no published paper has both algorithms presented in one place or presents them in the broader context of scenario graphs. Previously published papers have focused exclusively on attack graphs.

3. Algorithms for Generating Scenario Graphs

We present two algorithms for generating scenario graphs. The first is based on symbolic model checking and produces counterexamples for only safety properties, as expressed in terms of a computational tree logic. The second is based on explicit-state model checking and produces counterexamples for both safety and liveness properties, as expressed in terms of a linear temporal logic.

Both algorithms produce scenario graphs that guarantee the following informally stated properties:

- *Soundness*: Each path in the graph is a violation of the given property.
- *Exhaustive*: The graph contains all executions of the model that violate the given property.
- *Succinctness of states*: Each node in the graph represents a state that participates in some counterexample.
- *Succinctness of transitions*: Each edge in the graph represents a state transition that participates in some counterexample.

These properties of our scenario graphs are not obvious, in particular for the second algorithm. See [11] for formal definitions and proofs.

3.1. Symbolic Algorithm

Our first algorithm for producing scenario graphs is inspired by the symbolic model checking algorithm as implemented in model checkers such as NuSMV [8]. Our presentation and discussion of the algorithm in this section is taken almost verbatim from [9].

In the model checker NuSMV, the model M is a finite labeled transition system and p is a property written in *Computation Tree Logic* (CTL). In this section, we consider only safety properties, which in CTL have the form $\mathbf{AG}f$ (i.e., $p = \mathbf{AG}f$, where f is a formula in propositional logic). If the model M satisfies the property p, NuSMV reports "true." If M does not satisfy p, NuSMV produces a counterexample. A single counterexample shows a scenario that leads to a violation of the safety property.

Scenario graphs depict ways in which the execution of the model of a system can lead into an unsafe state. We can express the property that an unsafe state cannot be reached as:

$$\mathbf{AG}(\neg unsafe)$$

When this property is false, there are unsafe states that are reachable from the initial state. The precise meaning of *unsafe* depends on the system being modeled. For security, *unsafe* might mean that an intruder has gained root access to a host on a network.

We briefly describe the algorithm (Figure 1) for constructing scenario graphs for the property $\mathbf{AG}(\neg unsafe)$. The first step is to determine the set of states S_{reach} that

Input:

S – set of states
$R \subseteq S \times S$ – transition relation
$S_0 \subseteq S$ – set of initial states
$L : S \rightarrow 2^{AP}$ – labeling of states with propositional formulas
$p = \mathbf{AG}(\neg unsafe)$ – a safety property

Output:

Scenario graph $G_p = \langle S_{unsafe}, R^p, S_0^p, S_s^p \rangle$

Algorithm: $GenerateScenarioGraph(S, R, S_0, L, p)$

1. $S_{reach} = reachable(S, R, S_0, L)$

(* Use model checking to find the set of states S_{unsafe} that violate the safety property $\mathbf{AG}(\neg unsafe)$. *)

2. $S_{unsafe} = modelCheck(S_r, R, S_0, L, p)$.

(* Restrict the transition relation R to states in the set S_{unsafe} *)

3. $R^p = R \cap (S_{unsafe} \times S_{unsafe})$.
   $S_0^p = S_0 \cap S_{unsafe}$.
   $S_s^p = \{s | s \in S_{unsafe} \wedge unsafe \in L(s)\}$.
4. Return $G_p = \langle S_{unsafe}, R^p, S_0^p, S_s^p \rangle$.

Figure 1. Symbolic Algorithm for Generating Scenario Graphs

are reachable from the initial state. (This is a standard step in symbolic model checkers, where S_{reach} is represented symbolically, not explicitly.) Next, the algorithm computes the set of reachable states S_{unsafe} that have a path to an unsafe state. The set of states S_{unsafe} is computed using an iterative algorithm derived from a fix-point characterization of the **AG** operator [2]. Let R be the transition relation of the model, i.e., $(s, s') \in R$ if and only if there is a transition from state s to s'. By restricting the domain and range of R to S_{unsafe} we obtain a transition relation R^p that encapsulates the edges of the scenario graph. Therefore, the scenario graph is $\langle S_{unsafe}, R^p, S_0^p, S_s^p \rangle$, where S_{unsafe} and R^p represent the set of nodes and set of edges of the graph, respectively, $S_0^p = S_0 \cap S_{unsafe}$ is the set of initial states, and $S_s^p = \{s | s \in S_{unsafe} \wedge unsafe \in L(s)\}$ is the set of success states.

In symbolic model checkers, such as NuSMV, the transition relation and sets of states are represented using ordered binary decision diagrams (BDDs) [1], a compact representation for boolean functions. There are efficient BDD algorithms for all operations used in our algorithm.

3.2. Explicit-State Algorithm

Our second algorithm for producing scenario graphs uses an explicit-state model checking algorithm based on ω-automata theory. Model checkers such as SPIN [4] use explicit-state model checking. Our presentation and discussion of the algorithm in this section is taken almost verbatim from [5].

Figure 2 contains a high-level outline of our second algorithm for generating scenario graphs. We model our system as a Bücchi automaton M. Bücchi automata are finite state machines that accept infinite executions. A Bücchi automaton specifies a subset of

Input:
M – the model Bücchi automaton
p – an LTL property
Output:
Scenario graph $M_p = M \cap \neg p$
Algorithm: $GenerateScenarioGraph(M, p)$
1. Convert LTL formula $\neg p$ to equivalent Bücchi automaton N_p.
2. Construct the intersection automaton $I = M \cap \neg N_p$.
I accepts the language $L(M) \setminus L(p)$, which is precisely the set of of executions of M forbidden by p.
3. Compute SCC, the set of strongly-connected components of I that include at least one *acceptance* state.
4. Return M_p, which consists of SCC plus all the paths to any component in SCC from any initial state of I.

Figure 2. Explicit-State Algorithm for Generating Scenario Graphs

acceptance states. The automaton accepts any infinite execution that visits an acceptance state infinitely often. The property p is specified in *Linear Temporal Logic* (LTL). The property p induces a language $L(p)$ of executions that are permitted under the property. The executions of the model M that are *not* permitted by p thus constitute the language $L(M) \setminus L(p)$. The scenario graph is the automaton, $M_p = M \cap \neg p$, accepting this language. The construction procedure for M_p uses Gerth et.al.'s algorithm [3] for converting LTL formulae to Bücchi (Step 1). The Bücchi acceptance condition implies that any scenario accepted by M_p must eventually reach a strongly connected component of the graph that contains at least one acceptance state. Such components are found in Step 3 using Tarjan's classic strongly connected component algorithm [12]. This step isolates the relevant parts of the graph and prunes states that do not participate in any scenarios.

4. Attack Graphs are Scenario Graphs

In the security community, Red Teams construct *attack graphs* to show how a system is vulnerable to attack. Each path in an attack graph shows a way in which an intruder can compromise the security of a system. These graphs are drawn by hand, usually on huge floor-to-ceiling, wall-to-wall white boards. Since they are drawn by hand, they are prone to error: they might be incomplete (missing attacks), they might have redundant paths or redundant subgraphs, or they might have irrelevant nodes, transitions, or paths.

The correspondence between scenario graphs and attack graphs is simple. For a given desired security property, we generate the scenario graph for a model of the system to be protected. An example security property is that an intruder should never gain root access to a specific host. Since each scenario graph is property-specific, in practice, we might need to generate many scenario graphs to represent the entire attack graph that a Red Team might construct manually. Our main advantage is that we automate the process of producing attack graphs: (1) Our technique scales beyond what humans can do by hand; and (2) since our algorithms guarantee to produce scenario graphs that are sound,

exhaustive, and succinct, our graphs are not subject to the errors that humans are prone to make.

5. Practical Experience

We built a suite of tools that allows us to model networked systems [10]. We write XML input specifications that model the following kinds of information of a system: connectivity between hosts on the network, services running on each host, firewall rules, host-based and network-based intrusion detection systems, and most importantly, the actions an intruder might take in attempting to attack a system. We use our modifications of NuSMV and SPIN, reflecting our two algorithms, to produce attack graphs.

In practice we found that the explicit-state algorithm has good performance: the speed to generate the attack graph is linear in the number of reachable state transitions [11]. We also found that for our limited number of examples, our explicit-state algorithm is better than our symbolic algorithm in terms of time to generate graphs. In all of our examples, our models are large due to their large number of state variables, but at the same time they have a very small reachable state space. Thus we have a double whammy against the symbolic algorithm: Small reachable state spaces are better for explicit-state model checking, and larger numbers of state variables are worse for symbolic model checking.

These performance results, however, are not definitive. For one, we did not try to fine tune the implementation of our symbolic model checking algorithm. But most importantly, our application to security biases our experimental results in favor of our explicit-state algorithm. For other applications, the symbolic algorithm might be the better choice.

6. Future Work

We are now producing scenario graphs so large that humans have a hard time interpreting them. We plan to address the problem of size in several ways:

- Apply optimization techniques from the model checking literature to reduce the size of scenario graphs. For example, we can use symmetry and partial-order reduction techniques.
- Find ways to compress either or both the internal representation of the scenario graph and the external one displayed to the user.
- Design and implement new graph-based analyses on scenario graphs. For example, for attack graphs, we can determine which actions need to be prevented to guarantee an intruder cannot succeed (for a given property). We reduce our problem to the Minimum Hitting Set Problem (MHS), reduce MHS to the Minimum Set Covering Problem (MSC), and then adapt a greedy approximation algorithm for the MSC to perform this analysis [7]. We would like to explore more such analyses for scenario graphs in general.
- Rather than actually construct the scenario graph and display it to the end user, we could simply produce the answer to a domain-specific query on the internal representation of the scenario graph.

Finally, we are also interested in pursuing further uses of attack graphs, e.g., in using them in conjunction with on-line intrusion detection systems and in using them to help with alert correlation.

Acknowledgments

Oleg Sheyner deserves all the credit for the technical contributions summarized in this paper. The initial idea of using model checking to produce attack graphs is due to my collaboration with Somesh Jha.

This research is sponsored in part by the Army Research Office under contract no. DAAD190110485 and DAAD19-02-1-0389, the National Science Foundation under grant no. CCR-0121547 and CNS-0433540, and the Software Engineering Institute through a US government funding appropriation.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the sponsoring institutions, the US Government or any other entity.

References

[1] Randal E. Bryant, "Graph-based algorithms for boolean function manipulation," IEEE Transactions on Computers, C-35(8):677–691, August 1986.

[2] Edmund M. Clarke, Orna Grumberg, and Doron Peled, *Model Checking*, MIT Press, 2000.

[3] Rob Gerth, Doron Peled, MOshe Y. Vardi, and Pierre Wolper, "Simple on-the-fly automatic verification of linear temporal logic," *Proceedings of the 6th Symposium on Logic in Computer Science*, Amsterdam, July 1991, pp. 406–415.

[4] Gerald J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*, Addison-Wesley, 2004. *http://www.spinroot.com/spin/whatispin.html*.

[5] Todd Hughes and Oleg Sheyner, "Attack Scenario Graphs for Computer Network Threat Analysis and Predication," *Complexity*, 9(2):15–18, November/December 2003.

[6] Somesh Jha and Jeannette M. Wing, "Survivability Analysis of Networked Systems," *Proceedings of the International Conference on Software Engineering*, Toronto, May 2001.

[7] Somesh Jha, Oleg Sheyner, and Jeannette M. Wing, "Minimization and Reliability Analysis of Attack Graphs," *Proceedings of the Computer Security Foundations Workshop*, Nova Scotia, June 2002, pp. 49–63.

[8] NuSMV: a new symbolic model checker, *http://afrodite.itc.it:1024/ nusmv/*.

[9] Oleg Sheyner, Joshua Haines, Somesh Jha, Richard Lippman, and Jeannette M.Wing, "Automated Generation and Analysis of Attack Graphs," *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, May 2002.

[10] Oleg Sheyner and Jeannette M. Wing, "Tools for Generating and Analyziing Attack Graphs," *Proceedings of Formal Methods for Components and Objects*, Lencture Notes in Computer Science 3188, 2004, pp. 344–371.

[11] Oleg Sheyner, "Scenario Graphs and Attack Graphs," CMU Computer Science Department technical report CMU-CS-04-122, Ph.D. dissertation, April 2004.

[12] R.E. Tarjan, "Depth first search and linear graph algorithms," *SIAM Journal of Computing*, 1(2):146–160, June 1972.