Formal Analysis and Verification of Systems Security Models with Gnosis

Brian Monahan and Nick Papanikolaou
Cloud and Security Lab, HP Labs
{brian.monahan,nick.papanikolaou}@hp.com

***Abstract*—Emergent context-dependent non-functional requirements, such as those involving systems security activities and processes are, almost by definition, difficult to assess for their adequacy. One cannot easily anticipate and measure the effectiveness of systems defences in advance of actual field deployment until it is, of course, too late and the damage has been done. Our approach to security requirements assessment involves explicitly building systems security models using Gnosis, a process modelling simulation language developed at HP Labs. Gnosis models capture security situations which typically include aspects of the threat environment. In this paper we present the core aspects of this approach and discuss our latest work on developing explicit-state model checking of properties of multiple simulation runs.**

I. Introduction

One of the greatest challenges in modern computing is that of understanding and reasoning about complex system behaviour. Simulation is one of the most widely used approaches for dealing with this challenge, and numerous different simulation techniques and tools have been developed for a wide range of domains. The effectiveness of any given simulation depends on the accuracy and amount of detail contained in the model used to produce that simulation; therefore, a great deal of skill and experience is required to develop simulations which actually provide meaningful information and insight into the possible behaviours of a system. Our focus here is not on the development of such models, but rather on the information that can be gleaned about system behaviour from *the results of several simulations*, even when there is very little (or even no) knowledge of the original model used to produce them.

We argue that one can extract meaningful finite-state models from the results of simulations, which can reveal patterns and structure that is not necessarily obvious in the original model used for these simulations. This claim is of fundamental importance, as it means that simulations may reveal more information about a system than one might think; furthermore, the techniques we present here can be generalised so as to apply to any other experimental setup in which system behaviour is contained in ordered logs or timelines, and in which it is desirable to detect common patterns and underlying structure.

The types of simulations for which the techniques we present in this report find greatest utility are those where there is inherent non-deterministic behaviour, and also explicit specification of the probabilities of particular actions/events occurring. It is these types of simulations which give several different, and hence interesting, results (specifically, they give rise to different traces; this term is defined below). In order to account for the probabilities of different action patterns in simulations, our technique for extracting such patterns assigns explicit probabilities to actions and events.

Emergent context-dependent non-functional requirements, such as those for systems security activities and processes are, almost by definition, difficult to assess for their adequacy. One cannot easily anticipate and measure the effectiveness of system defences in advance of actual field deployment - until it is of course too late and the damage has been done. Our approach to security requirements assessment involves explicitly building systems security models using the Gnosis process modelling simulation language, developed at HP Labs, to capture security situations which typically include aspects of the threat environment. Such models are typically highly concurrent and naturally stochastic. Subsequent experimental simulation then allows us to empirically explore a range of different scenarios. We do this by performing multiple runs of the ”what-if” alternatives that arise from discretely varying particular parameters of the model. Statistical analysis of those multiple runs for each scenario can then be performed to help find out what combinations of options there might be that improve overall security utility. The downside is that this approach often produces a very significant volume of data that is difficult to analyze and understand in security terms. Our paper briefly discusses some work-in-progress towards addressing this issue, essentially by extracting smaller, more compact process models that characterize certain particular features. The anticipated advantages of our approach would be empirically derived models that are necessarily simpler than the original simulation and potentially more tractable for analysis using, for example, probabilistic model-checking techniques.

A. Related Work

The current Gnosis simulation framework does not have support for formal methods or verification in its current form. We are conscious of the relevance and linkages between our work and related work on automata chains and process mining.

In the context of resource-based logics such as SCRP (Synchronous Calculus of Resource and Process — see [7], [8]), Matthew Collinson developed some preliminary tools for basic resource-oriented model checking of Core Gnosis.

The authors Grastien, Cordier, Largouët [9] have devised a formal theory of automata chaining, namely, ways to join together automata with common states. This is related, but much more extensive, to our method of joining together the automata extracted from different simulation runs. In our approach, we match state labels from different runs and only identify them if their labels are identical; an area for future work is to investigate more sophisticated ways of merging automata together, particularly for special or corner cases where additional transitions (often just ϵ-transitions) are needed to merge more complex automata.

We are aware of related work by Van der Aalst and others on *process mining* [11]. The key difference between our approach and that advocated by the creators of process mining lies in their preference for the use of Petri net models rather than finite-state automata. However, we note that it is possible to convert Petri net models to automata, and that in [11] a tool is described that can be used to model-check LTL formulae over process models extracted from logs. Certainly the objectives of process mining are very much in alignment with the ideas presented here, and we envisage carrying over relevant ideas into our work.

II. The Gnosis Simulation Framework

Gnosis is a process modeling and simulation language developed within HP Labs to abstractly model systems security in operational terms at a variety of different scales and contexts. Our general starting point is that security is a process or activity that can involve a multitude of significant characteristics and interactions. Typically, these interaction arise between the modelled elements of a systems context involving people, process and technology. Mathematically, such aspects are characterized within our framework incorporating notions of Process, Resource, Location, and Environment. For us, a modelled system may equally represent a large-scale organizational business process, a cluster of back-end database servers, or some composite of software components. Abstractly, their principle behaviours may all be very similar, meaning that the models would also possess similar form and structure.

Besides including conventional means for data representation and manipulation, Gnosis includes constructs for describing processes, functions, resources, and locations. The resulting models are thus directly executable in terms of discrete process simulation. Gnosis also provides a good degree of stochastic capability; events may be drawn from a stock of probability distributions including uniform, normal, negative exponential, and Weibull; for example, these allow us to compactly model stochastic queuing and Markov chain process phenomena. A brief illustration of Gnosis is given in the appendix.

A semantics of (Core) Gnosis is discussed and developed in terms of the process calculus SCRP mentioned above (See [7], [8] for further details).

A. Gnosis traces

Each run of the Gnosis simulator on an input model `model.gn` will produce two files of interest, the trace file `model.tr` and corresponding dump file `model.csv`. In order to extract meaningful information from a run of `model.gn`, we need to link together the information found in these two files; in particular, we need to know, for each step in the trace (or at least specified steps in the trace; where each step assumed to appear as a single printed line), what the corresponding values are of all the variables in the model — namely, the system state.

A naïve approach to linking trace information with the states of variables at different points during execution would be to include explicit `print` statements outputting the values of variables in different parts of the original model `model.gn`. However, our approach here is to extract information from simulation runs without modifying the source model.

III. Extracting Emergent Structure from Gnosis Traces

We have conceived a method of analysing essential properties of multiple runs of a simulation model, particularly in the context of the Gnosis modelling and simulation framework. Our method extracts information from the output of each simulation run, namely information about state changes, and constructs a graph corresponding to a finite state automaton with a simple state transition function. The graphs from multiple runs can be combined into one bigger automaton, and this latter construction is a particularly useful representation of characteristic behaviours in the original model. The automata produced by the procedure are amenable to subsequent analysis and reasoning via model-checking techniques.

Our method presupposes the existence of:

- a simulation model M (which includes statements that produce observable outputs when run),
- a list $L = v_1, v_2, \ldots, v_k$ of "watch variables" (which are variables in the model whose state changes produce observable output),
- a simulator (in our experiments we have been focusing on the Gnosis tool), and
- a means of running it repeatedly on the simulation model to produce "dump files" (containing printouts of the system state at discrete time steps in a trace).

The method itself only makes use of the output of Gnosis when the model M is simulated; in particular our solution just processes the dump files produced. Let $D = d_1, d_2, \ldots, d_n$ denote the set of dump files generated by running Gnosis n times on model M. In the case of Gnosis, in order to run the algorithms presented in this section we need to make use of both *execution traces* and *dump files*, the latter being listings of the states of variables in a model at specified time instants in a particular run.

The algorithm in this section extracts state changes from a single simulation run and produces a graph consisting of these changes.

The method processes each trace t_i where $(1 \leq i \leq n)$, as described in Figure 1.

IV. Applications and Future Work

The advantages of this approach are manifold. First, Gnosis simulation models tend to be extremely large and complex.

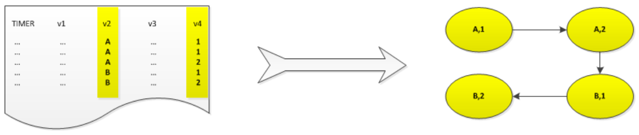

Fig. 1. Processing a trace from a single simulation run to generate an automaton. In the example, the set of watch variables is {`v2,v4`} and no filter/formula has been applied.

Algorithm 1 `process_single_run`(t_i, W): The algorithm for processing a single simulation run.

```
 1: for all lines in the file $t_i$ do
 2:     Initialize, for all $i$, $currentstate[v_i] \leftarrow state[v_0]$.
 3:     for all watch variables $v_1, \ldots, v_i, \ldots, v_k \in W$ do
 4:         if $state[v_i] \neq currentstate[v_i]$ then
 5:             Add transition $currentstate[v_i] \Rightarrow state[v_i]$ to graph/automaton
 6:             $currentstate[v_i] \leftarrow state[v_i]$
 7:         end if
 8:     end for
 9: end for
10: Assemble the list of all state changes and construct a directed graph $G_i$ whose nodes are labelled by the values of $v_1, v_2, \ldots, v_k$, and whose edges correspond to transitions.
```

Identifying entire regions of Gnosis models with a single state descriptor can be extremely useful in practice, as it allows the user to abstract away from many details; in fact, the finite-state automata generated by our technique can be easily visualised and this would surely assist in the human understanding of any Gnosis model.

So the first great advantage of our method is the conciseness it provides in relation to existing techniques for understanding Gnosis models. The most important advantage is that the method paves the way for model checking of Gnosis models, or parts thereof. Model checking is a well-established means of gaining assurance and confidence in the correctness of systems, and most importantly it is useful for automatically detecting conceptual flaws or errors in system designs. We note here that in order for an analysis using our method to be truly beneficial, the initial Gnosis simulation model has to be a very accurate representation of the real-world problem under consideration. Our method creates an abstraction of an existing model, so there is always the danger of obtaining overly general conclusions or results that are divorced from important aspects of the real-world system or problem under consideration. Despite the above caveat, we believe that our method of analysis can bring significant benefits if combined with a suitable model-checking algorithm and a suitable logic for specifying properties of multiple runs of models.

We envisage a number of applications of these techniques in the context of security analytics and in the analysis of security properties of complex, concurrent system models. Here are some of these:

- Analysis of security logs of cloud infrastructures
- Detection of patterns of malicious behaviour in system behaviour
- Comparison of attacker models

V. Conclusions

In this paper we have presented a method, and experimental implementation, for extracting emergent structure from system simulation models. We have documented algorithms for extracting state changes from individual simulation runs, and combining these state changes together when processing multiple runs. The purpose of this work is to obtain meaningful representations of complex simulation models and we envisage many different practical applications in future work.

References

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

[2] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In Jörg Desel, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Lectures on Concurrency and Petri Nets, Advances in Petri Nets*, volume 3098, pages 87–124. Springer-Verlag, 2004.

[3] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie. *Systems and Software Verification: Model-Checking Techniques and Tools*. Springer, 2001.

[4] Y. Beres, J. Griffin, S. Shiu, M. Heitman, D. Markle, and P. Ventura. Analysing the performance of security solutions to reduce vulnerability exposure window. In *Proceedings of the 2008 Annual Computer Security Applications Conference*, pages 33–42. IEEE Computer Society Conference Publishing Services (CPS), 2008.

[5] E. M. Clarke and P. Zuliani. Statistical model checking for cyber-physical systems. In *ATVA 2011: 9th International Symposium on Automated Technology for Verification and Analysis*, volume 6996 of *Lecture Notes in Computer Science*, pages 1–12, 2011.

[6] Edmund Clarke, Orna Grumberg, and Doron Peled. *Model Checking*. MIT Press, 1999.

[7] Matthew Collinson, Brian Monahan, and David Pym. Semantics for structured systems modelling and simulation. In *Proceedings of Simutools 2010*. ACM Digital Library, 2010. ISBN: 78-963-9799-87-5.

[8] Matthew Collinson, Brian Monahan, and David Pym. *A Discipline of Mathematical Systems Modelling*. College Publications, 2012.

[9] Alban Grastien, Marie-Odile Cordier, and Christine Largouët. Incremental diagnosis of discrete-event systems. In *Sixteenth International Workshop on Principles of Diagnosis (DX-05)*, 2005.

[10] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan and S. Qadeer, editors, *Proc. 23rd International Conference on Computer Aided Verification (CAV'11)*, volume 6806 of *LNCS*, pages 585–591. Springer, 2011.

[11] Wil M. P. Van Der Aalst. *Process Mining*. Springer Verlag, 2011.

Appendix

A Simple Gnosis Example

This appendix contains a very simple example of a Gnosis model, given purely to illustrate some basics of the language and to give some sense of what a model might contain. The example concerns just the supply part of a systems patching scenario - this includes patch generation, patch testing and patch application. A much more involved account of systems patching and vulnerability management can be found in [4].

The initial section of a model describes the parameters and stochastic variates that it uses.

```
 -- Title : A mini systems-patching example
 -- Seed  : 123456789

param runTime = 1000

param avgPatchRate = 10.6
param patchRate = negexp(avgPatchRate)
param lowerApplyTime = 3.6
param upperApplyTime = 7.2
param applyTime =
   uniform(lowerApplyTime, upperApplyTime)

param avgTest = 23.5
param testRange = 7.3
param testTime =
  uniform (avgTest - testRange, avgTest + testRange)

param probGoodPatch = 0.8
param patchOK = bernoulli (probGoodPatch)

param testers = 3
param patchers = 5

share testStaff (testers);  share patchStaff (patchers)

var patchesCreated = 0;  var patchesTestedOK = 0
var badPatches = 0;      var patchesApplied = 0

bin newPatches;  bin patchApply
```

Next, various processes that represent systems activities are defined - note that there is a “measure” process which periodically outputs ’dump’ information about the values of variables and the use of resources.

```
process generatePatch {
  launch generatePatch after patchRate
  put 1 into newPatches
  patchesCreated += 1
}

process checkPatch {
  repeat {
    get 1 from newPatches

    claim 1 testStaff;
      hold (testTime);

      if [patchOK > 0] {
        patchesTestedOK += 1
        put 1 into patchApply
      }
      or else
      {
        badPatches += 1
      }

    release 1 testStaff
  }
}

process applyPatch {
  repeat {
    get 1 from patchApply

    claim 2 patchStaff;
      hold (applyTime);
    release 2 patchStaff

    patchesApplied += 1
  }
}

process measure {
  repeat {
    hold(10)
    dump()                    // output values of variables
  }
}
```

Finally, the model launches instances of these processes and performs the simulation for a defined duration, as specified by `runTime`.

```
launch generatePatch
do 2 { launch checkPatch }
do 3 { launch applyPatch }
launch measure                       // capture measurements

hold (runTime)      // let processes run for specified time
close                                // close the simulation
```

When models are simulated, a CSV file is produced containing data values for variables, together with an explicit trace file, a fragment of which is reproduced here:

```
0               | *MAIN*     - ShareDefn: creating resource 'testStaff' with 3
0               | *MAIN*     - ShareDefn: creating resource 'patchStaff' with 5
0               | *MAIN*     - CountBinDefn: creating bin of initial size 0 named as 'newPatches'
0               | *MAIN*     - CountBinDefn: creating bin of initial size 0 named as 'patchApply'
0               | *MAIN*     - Launching process 'generatePatch.1' (generatePatch) at time 0
0               | *MAIN*     - Launching process 'checkPatch.1' (checkPatch) at time 0
0               | *MAIN*     - Launching process 'checkPatch.2' (checkPatch) at time 0
0               | *MAIN*     - Launching process 'applyPatch.1' (applyPatch) at time 0
0               | *MAIN*     - Launching process 'applyPatch.2' (applyPatch) at time 0
0               | *MAIN*     - Launching process 'applyPatch.3' (applyPatch) at time 0
0               | *MAIN*     - Launching process 'measure.1' (measure) at time 0
0               | *MAIN*     - Hold issued for 1000
0               | generatePatch.1 - Launching process 'generatePatch.2' (generatePatch) at time 14.3
0               | generatePatch.1 - PutCountBin: putting 1 units into bin 'newPatches'
0               | checkPatch.1 - GetCountBin: taking 1 units from bin 'newPatches'

  ....
  ....
  ....

982.278359198   | applyPatch.3 - Hold issued for 5.54678759405
987.751063231   | generatePatch.71 - Launching process 'generatePatch.72' (generatePatch) at time 99
987.751063231   | generatePatch.71 - PutCountBin: putting 1 units into bin 'newPatches'
987.751063231   | checkPatch.2 - GetCountBin: taking 1 units from bin 'newPatches'
987.751063231   | checkPatch.2 - ClaimShare: claiming 1 units for resource 'testStaff'
987.751063231   | checkPatch.2 - Hold issued for 19.5354026393
987.825146792   | applyPatch.3 - ReleaseShare: releasing 2 units for resource 'patchStaff'
990             | measure.1  - Hold issued for 10
992.939507824   | generatePatch.72 - Launching process 'generatePatch.73' (generatePatch) at time 99
992.939507824   | generatePatch.72 - PutCountBin: putting 1 units into bin 'newPatches'
995.179446215   | generatePatch.73 - Launching process 'generatePatch.74' (generatePatch) at time 10
995.179446215   | generatePatch.73 - PutCountBin: putting 1 units into bin 'newPatches'
996.399056487   | checkPatch.1 - PutCountBin: putting 1 units into bin 'patchApply'
996.399056487   | checkPatch.1 - ReleaseShare: releasing 1 units for resource 'testStaff'
996.399056487   | checkPatch.1 - GetCountBin: taking 1 units from bin 'newPatches'
996.399056487   | checkPatch.1 - ClaimShare: claiming 1 units for resource 'testStaff'
996.399056487   | checkPatch.1 - Hold issued for 24.9121582659
996.399056487   | applyPatch.2 - GetCountBin: taking 1 units from bin 'patchApply'
996.399056487   | applyPatch.2 - ClaimShare: claiming 2 units for resource 'patchStaff'
996.399056487   | applyPatch.2 - Hold issued for 6.4590931604
1000            | *MAIN*     - Simulation Closed.
```