

# Security Analysis of Systems using Model-Integrated-Computing

Marty Henderson      Blake Sheridan

August 4, 2006

## Abstract

Our project dealt with modeling security systems and analyzing those models for possible weaknesses. To model these systems we have used Attack Trees, an approach to security modeling developed by Bruce Schneier, a professional in the fields of cryptography and security. Security analysis benefits from an attack tree modeling approach. A well designed modeling environment can find the biggest holes and easiest or cheapest methods of fortification. Using Vanderbilt's Generic Modeling Environment (GME), we implemented an attack tree modeling paradigm. Furthermore, we developed tools in the form of GME interpreters for analyzing systems, viewing systems more easily, and importing or exporting systems.

## 1 Introduction

As a system gets bigger it tends to develop bigger holes, just by the nature of the beast. The work of many salaried individuals is devoted entirely to keeping track of a system's security from possible assailants. These systems can vary from a large corporate building to a computer network. A question to bear in mind: what is the best method for overseeing the security of a system?

Bruce Schneier, a professional in the fields of cryptography and security, developed a security modeling approach, dubbed an "attack tree". The goal of an attack, e.g. breaking into a network, is drawn as the head node of a tree. Possible sub-goals for achieving the goal are then attached to this node as children nodes. In a similar fashion, the entire attack becomes a tree with the first steps for each method being leaves at the bottom of the tree.[1]

Security analysis benefits from an attack tree modeling approach. The visual representation of possible attack scenarios allows the analyst to better see the bigger picture; complexity of attacks, areas of weakness, etc. Moreover, computer modeling of these attack trees leads to

automated analysis of systems. A well designed modeling environment can find the biggest holes and easiest or cheapest methods of fortification.

Using Vanderbilt's Generic Modeling Environment (GME), we have implemented an attack tree modeling paradigm. Furthermore, we have developed tools in the form of GME interpreters for analysis of systems. Our Analysis interpreter finds vulnerable paths in the model. Our Collapse interpreter collapses sub trees for easier viewing. Our Dispatch interpreter imports and exports models.

## 2 Problems Encountered

We have encountered several problems throughout the course of this project. Most have come up while writing methods for the interpreter, except the problems we have had with writing OCL constraints. The first significant problems we encountered were in our initial attempts at OCL constraints. Our metamodel initially had two constraints against multiplicity and self connections. A node can neither have multiple parents nor connect to itself. Our first attempts at writing these constraints failed, because we were unfamiliar with OCL syntax. We learned enough about OCL syntax from the tutorial examples to complete these constraints.

### **Multiplicity:**

```
self.attachingConnections("src")->size < 2
```

### **Self Connection Constraint:**

```
self.connectionPoints("src")->theOnly().target()<>  
self.connectionPoints("dst")->theOnly().target()
```

We later developed a third OCL constraint to ensure that there are no cycles in the Attack Tree. This restraint was much more complicated than the first two mentioned and required two OCL functions to complete, as shown below.

### **Cyclic Constraint:**

```
self.CyclicFunction()<>true
```

### **CyclicFunction:**

```
if (self.GetTopParent(self) <> self) then
    false
else (
    if (self.attachingConnections("src")->size<1)then
        false
    else (
        true
    )
    endif
)
endif
```

### **GetTopParent:**

```
if (self.attachingConnections("src")->size < 1) then
    self
else (
    let connection = self.attachingConnections("src")->theOnly() in
    (
        let parent = connection.connectionPoints("dst")->theOnly()->target() in
        (
            if (child <> parent) then
                parent.GetTopParent(child)
            else (
                parent
            )
        )
        endif
    )
)
)
```

The bulk of the work for this constraint is done in the `GetTopParent()` function. This function will travel up the tree from the original node it is called on, until it either finds the top of the tree or loops back around and hits the same node it was called on, indicating that there is a cycle. The `CyclicFunction()` checks to see whether the node returned by `GetTopParent()` is the same as the original node. If it is, `CyclicFunction()` also ensures that the node has a

parent, because otherwise, it could simply be returning itself as the top parent without a cycle. This constraint is run whenever a connection is made between two nodes.

The second major problem that we had was writing the "Pathfinder" algorithm which finds all possible paths to the goal of an attack tree. Every sub tree of an attack tree can be an "and" tree or an "or" tree. If it is an "and" tree, then all of the child nodes have to be met in order for this node to be accomplished. If it is an "or" tree, then this node is accomplished if any of its child nodes are fulfilled. This variable made the "Pathfinder" algorithm very difficult to write. Whenever an "or" node is encountered, paths are added to the list of current paths, but when an "and" node is encountered, the list of current paths is multiplied by the number of paths under it. Every time that new paths are added, the list of nodes already visited on the way to the current node from the goal must be added to the new path in the list. Because of this difficulty in keeping up with the lists of paths and the various situations in which new paths must be added based on "and" or "or" nodes, this method was very difficult to plan and write.

Another problem that we ran into was the need to refactor our code several times. At the beginning of the project, we did not always know what was ahead of us or what certain data would be used for, so we did not initially store pertinent data such as a path to the goal in the best way. Several times throughout this project we have made major revisions to our code to resolve an unforeseen problem that has developed. After writing the Pathfinder algorithm, we decided that we should define our own PathData class for storing a path and all the data associated with it. The path itself is stored as a list of long integers representing the unique node ids for each node in the path. Other attributes are stored in the PathData class that describe important data related to the class such as how much it would cost an intruder to perform, how much damage it would cause, whether it is possible, whether special equipment is required, and whether it is detectable by the administrator, and whether it is legal.

Since the midpoint of our project, we have tried to determine how our project can behave that will be most beneficial to future users. We have made several major changes to make our project better represent real-life security threats. On the lowest level, we have changed the node attributes in our paradigm significantly. Before the changes, we started out with boolean attributes Legal, Detection, SpecialToolsNeeded and Possible and a double attribute CostToDefend. Legal indicated whether the attack was legal or illegal. Detection indicated whether it would be detected. SpecialToolsNeeded indicated whether special tools were needed for this attack. Possible indicated whether it was possible to infiltrate a given node. CostToDefend represented the cost for the company to fortify a node such that it was impossible to penetrate.

These seemed like good measures of security at first, but we soon realized that they were oversimplified. Legal is unneeded because virtually any significant attack will be illegal, and the company will want to stop the attack either way, so while legal may have some importance as an attribute, it is not worthy of being one of the few attributes represented in a node. The Detection attribute was unrealistic. It is likely that many attacks will be detected some of

the time, but may be missed if executed perfectly. To have a true or false value representing detection is too simple. It would more appropriately be a range of values. Possible was also unrealistic in the same way as detection. A node is not simply possible or impossible. A node is vulnerable to an intruder with a certain amount of skill and tools. This attribute can not be reduced to a simple boolean value. `SpecialToolsNeeded` was not very descriptive, because again, almost every attack is going to require some special tools. A better measure would be a range of how many special tools or a cost of the tools required. We replaced these boolean attributes with two field attributes: `TechnicalAbility` (ranging from 1 to 100) and `ProbabilityOfApprehension` (ranging from 0 to 1). We felt that these better represented the detectability, possibility, and technicality of the attack. Since we now were not representing the possibility of a node with a boolean, it didn't make sense to have a cost to defend attribute, because money spent defending a node would affect the difficulty of breaking the node (which is rolled into `TechnicalAbility`), but it would not make it 100 percent impossible to infiltrate. Therefore we removed `CostToDefend` from the paradigm.

These changes to the paradigm affected the way many of our algorithms worked also. We could no longer eliminate possible paths as early as we were before, because we could only eliminate a path based on the final value of its attributes, and not simply by using the possible attribute as we work our way through the tree. We also changed the interface to reflect the new attributes we have and how the paths could be filtered. Possibly the most significant affect these new attributes had was that we could not do fortification analysis in the same way. We could not now say that fortifying a specific node would prevent a given number of paths, because there is no absolute fortification of a node and it would not absolutely prevent a given number of paths, it would only change the difficulty for those paths.

Instead of continuing our fortification analysis in the same way, we created a new dialog that displays all the vulnerable paths, based on the filter inputs. These paths can be sorted in ascending or descending order by any of their attributes. When a path is selected, all of its attributes and nodes are displayed in the dialog. This dialog also gives you the options to highlight a path, add a path to the model for further examination, input new filters, or return to the model.

### 3 Results

There have been several changes to the Attack Tree paradigm over the course of this project. The paradigm now allows for Attack Tree models to serve as a container for an entire system, or to be nested within one another, representing subtrees within the main tree. Attack Trees may have any number of nodes within them and any number of models representing collapsed sub trees. Our paradigm originally only had one kind of connection, from a node to a node, but with the addition of collapsed sub trees, we had to add a second kind of connection, from a model to a node.

The first of three model interpreters is the primary analysis interpreter. After a user has created a model for analysis he or she can invoke the analyzer, which will correct minor errors in the model or abort on major ones. A correctly constructed model will have its icons set to reflect the states of all nodes - either goal node, and node, or node, or leaf node. Next, the user is presented with a GUI allowing him or her to select parameters for searching for paths. For example, a user can filter out all paths that would require an assailant to spend more than \$5000. (Note that the unit of cost need not be dollars, it can be whatever the user wishes to measure cost by.) After locating the vulnerable paths, these paths will be displayed in a second dialog window. Each path can be selected to see a list of attributes describing that path. The paths can also be sorted based on any of the parameters describing a path: cost to intruder, damage, return on investment, probability of apprehension, technical ability required, depth, and number of leaves. This dialog allows the user to add a path to their main model for further examination, rerun the interpreter with new filters, or return to the main model.

Our initial interface for filtering paths can be seen below.

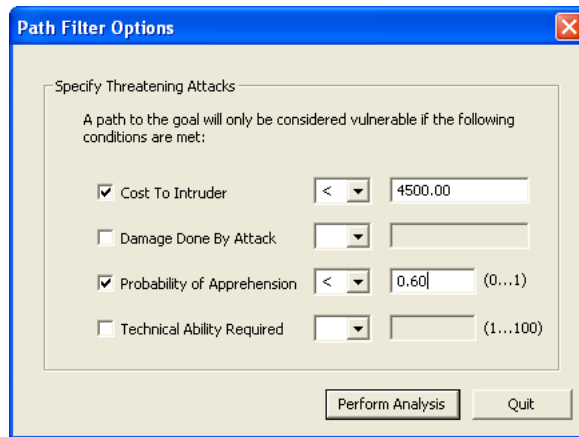


Figure 1: Path Filter Options interface

In this dialog, the user may select any number of four filtering options. These options will determine which paths to the goal are considered to be vulnerable. The user may limit these vulnerable paths based on the cost of the attack, the damage done by the attack, the technical ability required, or the probability of apprehension.

After selecting filter options, the bulk of the Analysis interpreter is run on the model. All algorithms in this project that involved searching the tree were done by means of a depth first search. The Pathfinder and Percolation algorithms do the majority of the work in finding paths to the goal. The percolation method runs first. This method is responsible for percolating the

relevant attributes of leaf nodes to their parent nodes, and then to their parent nodes, and so forth until the goal node is reached. Percolation is done differently depending on which attribute is being percolated and what type the parent node is. If the parent node is an "And" node, then when cost is percolated, the cost of the parent node will be the sum of all the costs of the child nodes. If the parent node is an "Or" node, then when cost is percolated, the cost of the parent node will be the minimum cost of all the child nodes, because only one child must be fulfilled to fulfill the parent node. Technical Ability required and Probability of Apprehension attributes are percolated by similar means. Damage done by attack is not percolated, because that is typically independent of child or parent nodes.

After percolation is complete, the pathfinder algorithm finds all paths to the goal of the system. Because the attributes of a vulnerable path have the potential to grow or decline as new nodes are added to the vulnerable paths, filtering of the paths cannot be done until the paths are completed. This means that our Pathfinder algorithm builds every possible path to the goal, regardless of that path's attributes. After all the paths are built, those paths are filtered based on the input from the user in the initial interface.

The Path Analysis Dialog can be seen below in Figure 2.

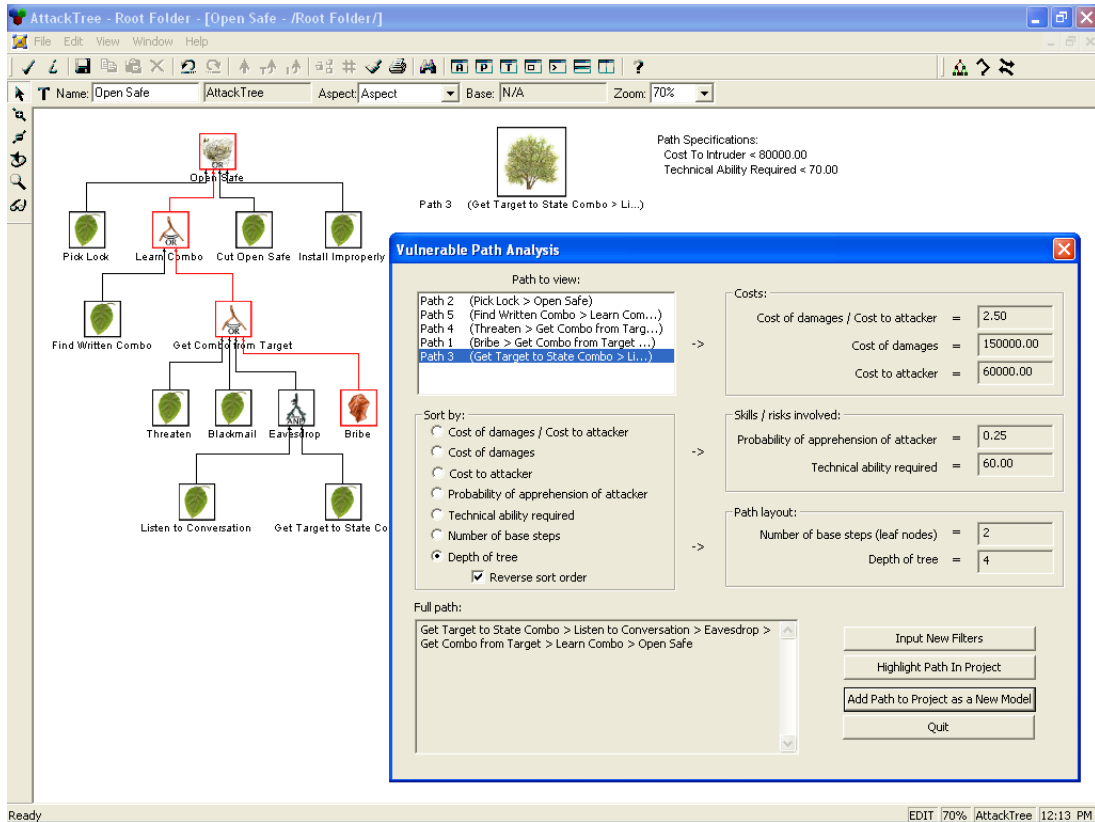


Figure 2: Path Analysis Dialog

In this dialog, the user can see all paths that were returned from the Analysis interpreter, based on the filter specifications they gave. These paths can be sorted by the radio buttons, in ascending or descending order. Each path can be selected from the listbox at the top. Whenever a path is selected, that paths attributes are displayed along the right side of the dialog. The nodes in that path are displayed at the bottom of the dialog. When a specific path is selected, that path can be highlighted in the model or be added to the main model as an embedded model to be viewed in greater detail later. These options are all meant to enable the user to see which paths present the greatest threat.

The Collapse interpreter enables the user to collapse entire sub trees of the graph into a single model for easier viewing. When this model is analyzed, the attributes of the collapsed model are percolated up to the collapsed model, so that it can fully represent the collapsed sub tree. This interpreter also allows these collapsed models to be expanded back to their initial

state.

The Dispatch interpreter allows the user to import or export a model in a variety of formats. The model can be imported from an XML document. The model can be exported as XML or as a Graphviz file, which can be transformed into an image.

## 4 Conclusions and Future Work

As this project stands at the conclusion of our internship, it would certainly be of use to someone looking to analyze their security measures. It is fully capable of modeling not only computer security systems, but any kind of security system that can be represented in tree format. However, there are also many enhancements that could still be made to this project.

Users are able to model their systems and use filters to search for specific threats. These filters allow the user to see specifically what different types of intruders have the power to do to their system. The vulnerable paths to the goal can be viewed, sorted by their attributes, highlighted, and examined as a new model, all through our user interface.

We feel that research in this area of security analysis and modeling in GME should be continued as there are many benefits to modeling and analysis with attack trees. This method of modeling provides an intuitive and convenient way for users to view their system from whatever layer of abstraction they find most useful. Similarly, users can collapse sub trees that they are not interested in focusing on at the moment into a single models.

We recommend that research be continued on the following improvements to our current model and analysis:

**Intruder Profiles:** The primary focus of future work should be to better diagnose the weaknesses of a modeled system. The first step in doing this would involve allowing the user to build and save enemy profiles in the initial user interface. These profiles would describe the capabilities of various possible intruders. Capabilities described may include technical ability, amount willing to spend, risk threshold, and level of access they currently have to the system.

A company might have separate profiles for the average script kiddie, professional cracker, and common burglar. These profiles could be loaded into the interface and then used as the filter specifications for vulnerable attacks. This would allow further fortification analysis. The Analysis interpreter could then determine the cheapest way to fortify a node or path such that at least one of its attributes would be out of reach to the current threat profile being considered.

**Model Real Life System:** Model and diagnose a real life system. Most likely a local computer network. This would provide many benefits and help to improve our code in many ways. First, in testing the code that extensively, we would find many of the bugs that we have missed. Also, we would be able to see how our program could be improved for easier use when used with a large system. Second, it would help us to see what security analyses

the company finds most helpful and which ones are not needed. As this project continually develops, the most important guideline is that it grows to better represent real-life systems and provides either reasonable security suggestions or the ability for the user to easily see what needs improvement.

## References

- [1] B. Schneier. Attack trees. *<http://www.schneier.com/paper-attacktrees-ddj-ft.html>*, Dec. 1999. 1 Aug. 2006.