Scalable Fault Tree Construction and Analysis

May 3rd, 2017

Student: Will Dower
Advisor: Avi Silberschatz
Additional Guidance: Ennan Zhai

1. Abstract

Currently, the best way for a cloud-computing service such as Amazon Web Services or Microsoft Azure to identify likely points of failure in its network is to sit and wait for something to break. In a network the size of Amazon’s, for example, the only way the company might be made aware of the fact that five servers all depend on the same switch to reach the internet would be if that switch failed. Thus, companies are forced to passively react to the consequences of hidden dependencies, rather than fix them before they become a problem. Such companies could benefit from a tool that would allow them to assemble dependency graphs of their network components and use them to identify risk groups, or sets of components which could take down the network if they failed simultaneously. Companies could use this information to strengthen their networks, and end users of cloud services could use it to analyze several cloud services and compare the relative risk in using them.

A vital step in the construction of such a tool is the assembly and analysis of the fault tree, a data structure that aims to represent the hierarchy of dependencies upon which a given service runs. The fault tree generator is Python code I wrote which generates a fault tree in the form of a Boolean satisfiability formula from a list of dependency data, and then identifies the most prevalent risk groups in the tree.

2. Background

2.1 Definitions

A fault tree is a tree model of a computer system with a root node which represents the status of the entire system and a node for each of the system’s components and their vulnerabilities, with logical AND and OR gates connecting each child node to its parents. An AND gate connecting child nodes to their parent indicates that each of the child nodes would have to fail to take down the parent
node, while an OR gate indicates that any of the children failing would also cause the parent to fail. A risk group is a set of components which, if they all failed, would result in the root node failing (that is, the cloud system being unable to function). A minimal risk group or cutset is a risk group in which the set cannot have components removed without ceasing to be a risk group; that is, it is a group of vulnerabilities which must all fail in tandem in order to damage the root node, without any extraneous members. The vulnerabilities which make up risk groups also have a weight, which is some number indicating the relative likelihood that the vulnerability will trigger. An example of a fault tree (without displayed weights) can be seen in Fig. 1.

In this fault tree, \{Vul1, Vul2, Vul3\} is a risk group because if all three vulnerabilities triggered, the network would fail, but some vulnerabilities could be removed from the set and this property would remain. Accordingly, \{Vul 1\} is a minimal risk group because if Vulnerability 1 triggers, the file system goes down and thus so does the entire data access service, but no vulnerabilities can be removed from the set and still leave a risk group. \{Vul 3\} is also a minimal risk group, since both of the service’s paths to the internet depend on a component that Vulnerability 3 knocks out. In other systems, there may be cutsets consisting of multiple vulnerabilities instead of just one, representing failures that must occur simultaneously in order for the system to fail.
2.2 INDaaS

Independence-as-a-service, or INDaaS, is a conceptual architecture for discovering risk groups within a cloud service, proposed by Zhai et al as a method for businesses to audit the reliability of large-scale cloud networks [1]. INDaaS would provide the following functions:

- Dependency data acquisition, using a variety of tools to discover network, hardware, and software dependencies
- A fault tree generator that could take in the dependency data and discover its most volatile risk groups
- A private set intersection cardinality protocol to allow a user to compare the results of INDaaS’s findings for multiple cloud systems, without revealing the actual structure of the tested systems to each other or the end user

This project deals with implementing the fault tree generator, using existing techniques to gather dependency data for input.

2.3 Dependency Data

The dependency data that forms the input for the fault tree generator is gathered from a variety of sources in order to provide the most complete picture possible of a system’s makeup and vulnerabilities. These sources include numerous currently existing tools such as NSDMiner for network dependencies [3], rdepends, a Linux utility for software package dependencies [4], and lshw, a Linux tool for discovering hardware specifications [5]. This stage of INDaaS forms dependency files which have three sections:

- A listing of all the various paths a cloud system can use to reach the internet, in which paths are lists of network hardware components leading from a cloud server to the internet
- A listing of all the system components, and the vulnerabilities associated with those components, where vulnerabilities can exist due to software or hardware issues
- A listing of all vulnerabilities and their scores, where the score is an integer between 1-10 assigned to the vulnerability using NIST’s National Vulnerability Database scoring system [2]

The paths to the internet are expressed such that the failure of any component in the path will also cause the entire path to fail. Likewise, each vulnerability in a component will take down the component
should it trigger. All paths, components and vulnerabilities are formatted into XML-like tags for a standardized overview of the network’s dependency data.

```xml
# Network path information
# All the network paths follow this format, i.e., src, dst and route
# src means the source of the path
# dst means the destination of the path
# route lists all the components between src and dst
<src="S1" dst="Internet" route="Agg1,DNS1"/>
<src="S1" dst="Internet" route="Agg2,DNS1"/>
<src="S2" dst="Internet" route="Agg1,DNS1"/>
<src="S2" dst="Internet" route="Agg2,DNS1"/>

# ---- IP address information ----
# Note that each item follows the format: unique id, name, IP address,
# and vulnerabilities (related to software components)
{S1, "Server-1", "172.28.228.21", vul="v4,v5"}
{S2, "Server-2", "172.28.228.22", vul="v1"}
{Agg1, "Agg-1", "10.0.0.1", vul="v3"}
{Agg2, "Agg-2", "10.0.0.2", vul="v1"}
{DNS1, "DNS-1", "75.142.33.98", vul="v2,v3"}

# ---- Vulnerability scoring ----
{name="v1" score="3"}
{name="v2" score="4"}
{name="v3" score="1"}
{name="v4" score="10"}
{name="v5" score="7"}
```

Fig 2: An example dependency data file.

3. Fault Tree Generator Implementation

3.1 Difficulties

The goal of the project is to determine which vulnerabilities in a fault tree can take down the root service should they all trigger simultaneously. Stated another way, the goal is to use dependency information to assemble a Boolean satisfiability formula, in which the formula’s clauses represent paths to the internet, variables represent the cloud network’s vulnerabilities, and assigning a value of 1 to those variables indicates a vulnerability triggering. Unfortunately, Boolean satisfiability problems are NP-hard. The whole reason a system like INDaaS is needed to automatically assemble and analyze dependencies is because cloud systems are by definition extremely large and contain multitudes of interconnected dependencies; finding cutsets of important vulnerabilities in these systems is computationally demanding.
Furthermore, the problem becomes even more difficult if the user is attempting to find more than one cutset – for instance, a useful metric for a would-be cloud network customer would be the top ten “riskiest” risk-groups present in a datacenter. Thus, a fault tree analyzer must not only solve an NP-hard problem, but solve it repeatedly and reasonably quickly, while factoring in the weights of the variables in its consideration.

The original INDaaS prototype suggested a fault tree analyzer algorithm which worked by essentially randomly triggering vulnerabilities and seeing if the larger system still worked. The process would be slow, non-deterministic, and unable to confirm the risk groups it returned to be minimal. This project is an attempt to supply a more robust and efficient process for fault tree analysis.

3.2 The Maxino SAT-solver

Luckily, there do exist algorithms that can at least partially mitigate the computational problems with fault trees. Maxino [6] is a tool which can solve Boolean satisfiability formulas reasonably quickly, and with verifiable accuracy [7]. To use the SAT-solver, the user specifies a number of clauses consisting of Boolean variables, and the relative cost of satisfying each clause. Maxino will then analyze the input and determine the least-costly set of variables that must trigger in order for every clause to be satisfied.

Consider the following Boolean formula:

\[ \Phi = ( x_1 \lor \neg x_2 \lor x_4 ) \land ( \neg x_1 \lor \neg x_2 \lor x_3 ) \land ( x_1 \lor x_2 ) \]

Variables \( x_1, x_2, x_3 \) and \( x_4 \) have weights of 6, 7, 8, and 9 respectively. The corresponding input file for the SAT-solver would be:

```
-----------------------
p wcnf 4 7 31
31 1 -2 4 0
31 -1 -2 3 0
31 1 2 0
6 -1 0
7 -2 0
8 -3 0
9 -4 0
-----------------------
```

Fig 3: Input for the Maxino SAT-solver.

Here, the top line contains a bit of header information, as well as the numbers 4, 7, and 31, which represent the number of variables, the number of clauses, and a weight which distinguishes hard clauses, equal to the sum of all the other weights plus one. Then come the Boolean formula’s hard clauses – the ones for which Maxino is actually solving. Each variable is represented by a unique integer,
and a clause line contains all the variable numbers in the clause (with negative numbers representing the negation of the variable) after the hard clause weight. Last comes the soft clauses, which are extra clauses consisting of the negation of a single variable and that variable’s weight. Each line is terminated by a zero. The hard clauses represent the actual Boolean formula, and each has the same hard weight; the soft clauses exist to ensure that the SAT-solver factors in the cost to trigger each variable. The end result is a list of which variables must trigger to solve all clauses at the least cost.

A fault tree graph can easily become a Boolean formula solvable by Maxino. The pathways to the internet become clauses in the formula, in which all the components in the pathway are replaced by the vulnerabilities that can affect those components. Those vulnerabilities then become the Boolean variables for the formula. Should such a formula be solved, it would reveal the vulnerabilities which must trigger to take down every single path to the internet concurrently.

3.3 ft_analyzer

To create the fault tree generator, I wrote a Python script to take in a dependency data file and piece together the fault tree in the form of a Boolean satisfiability problem, and then run Maxino on the tree to discover the cutsets. The output is the cutset with the highest likelihood of failure — the “riskiest” risk group.

The first step was to use regex matching to assemble dictionaries of component and vulnerability values. Next, a list of all the vulnerabilities along a path could be assembled by replacing the path’s components with the vulnerabilities of those components. The lists of path vulnerabilities would then be used to construct the clauses of a Maxino input file. The weights of the hard clauses would be the sum of all the vulnerability weights. The weights of the soft clauses could not be the

Fig. 4: A fault tree can be converted to a Boolean formula for analysis.
actual vulnerability weights, because while a higher vulnerability weight represents a higher likelihood of failure, Maxino tries to find the lowest-costing variable set, and the idea is to return the set with the highest failure chance. Weights therefore had to be manipulated using the formula:

$$A(x) = -100 \times \log\left(\frac{W(x)}{20}\right),$$

where variable x is a vulnerability, W(x) is that vulnerability’s score, and A(x) is the adjusted score for the purposes of SAT-solving. This formula ensures that vulnerabilities with higher weights, when divided by 20, will be a smaller number when the log is taken. The result is then multiplied by -100 to make it a positive integer which Maxino can work with.

A desired functionality of the fault tree generator was for it to be able to return not only the “riskiest” risk group, but a list of risk groups, ranked from highest to lowest. The goal is for the end user to be able to receive, for example, the top ten risk groups in a cloud network. The Maxino tool has no such functionality, however, so I simply looped over the input file for as many repetitions as necessary. Every time a cutset was found, I would add it to the input file as a hard clause, thereby ensuring that when Maxino was run on it again, the original cutset would be too costly to consider. From the command line, the user can specify r, the number of cutsets they wish to receive, and the script will respond accordingly.

The final output of the script is an XML file showing the cutsets in order from highest to lowest. Each cutset consisted of a list of that cutset’s vulnerabilities, and a “weight” calculated by using the formula:

$$\prod \left( \frac{W(x)}{20} \right),$$

where x is a vulnerability and W(x) is that vulnerability’s weight. The weights were calculated in this way for the same reason the weights were adjusted in the SAT-solver input file; simply summing up the vulnerability weights would lead to an unintuitive situation where less-probable cutsets would have a higher number for the reported weight, so instead they are all divided into a decimal less than one and multiplied together to ensure less-probable cutsets have correspondingly smaller weights.

The final output of the ft_analyzer script, when run on the example file given in Fig. 2 with r=3, is the following:

```xml
<?xml version="1.0" ?>
<cutsets>
  <cutset items="v2" weight="0.2"/>
  <cutset items="v1, v4" weight="0.075"/>
  <cutset items="v1, v5" weight="0.0525"/>
</cutsets>
```

Fig 5: ft_analyzer output.

The cutsets are thus identified and nicely packaged in XML for use in later stages of analysis.
4. Tests

In order to test the fault tree generator, I fed it a variety of inputs and timed its progress. I tested using three dependency data files representing datacenters containing 128, 1024 and 3456 servers, respectively. These files described relatively simple components and paths – each component had only one vulnerability, and each vulnerability had a weight of one. I then ran the script to find the time it would take to assemble the top ten, top one hundred and top one thousand risk groups from each of these files. I also had time to run an $r=10,000$ test on the file for the datacenter with 3456 servers, but I didn’t quite get to test the other two (and I suspect that there wouldn’t even be 10,000 cutsets to find on the smaller inputs).

Before I received the test files, I also generated a few of my own. I used a script to create 100 components, each of which contained 1-3 of 100 random vulnerabilities with random weights, and then 9 paths to the internet consisting of 1-3 random components. I used the randomly generated files to test the time it would take to find 10,000 risk groups, in order to supply another student with an input for his project, which dealt with the next step of analysis after the fault tree. This test had the added benefit of giving me a seriously huge task for the script on a more complex input. I include the results from those tests as well.

Examples of the input files I tested are below:

```plaintext
# paths from servers to the internet
<src="S1" dst="Internet" route="ToR1,Agg1,Core1"/>
<src="S1" dst="Internet" route="ToR1,Agg2,Core1"/>
<src="S1" dst="Internet" route="ToR1,Agg3,Core2"/>
<src="S1" dst="Internet" route="ToR1,Agg4,Core2"/>

# component descriptions
{S1, "Server-1", "172.28.228.1", vul="v1"}
{S2, "Server-2", "172.28.228.2", vul="v2"}
{S3, "Server-3", "172.28.228.3", vul="v3"}
{S4, "Server-4", "172.28.228.4", vul="v4"}

# vulnerability descriptions
{name="v1" score="1"}
{name="v2" score="1"}
{name="v3" score="1"}
{name="v4" score="1"}
```

Fig 6: A segment from a dependency input test file.
I timed all the runs using Linux's `time` shell command. I tested my own randomly-generated files on my personal Linux machine, and I used the Zoo machines to test the three files representing datacenters. I realize that the different environments and lack of repeated trials for these results means the data cannot be considered as anything other than rough figures, but these tests are still good benchmarks for how fast the script can chew through different numbers of cutsets.

### 4.1 Test Results

The time it took to complete the tests for the three input files representing datacenters were as follows:

<table>
<thead>
<tr>
<th>File</th>
<th>Repetitions</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>topo-128</td>
<td>10</td>
<td>0.167 seconds</td>
</tr>
<tr>
<td>topo-128</td>
<td>100</td>
<td>0.800 seconds</td>
</tr>
<tr>
<td>topo-128</td>
<td>1000</td>
<td>3 minutes, 25.582 seconds</td>
</tr>
<tr>
<td>topo-1024</td>
<td>10</td>
<td>9.356 seconds</td>
</tr>
<tr>
<td>topo-1024</td>
<td>100</td>
<td>13.376 seconds</td>
</tr>
<tr>
<td>topo-1024</td>
<td>1000</td>
<td>1 minute, 39.992 seconds</td>
</tr>
<tr>
<td>topo-3456</td>
<td>10</td>
<td>2 minutes, 50.627 seconds</td>
</tr>
<tr>
<td>topo-3456</td>
<td>100</td>
<td>3 minutes, 8.066 seconds</td>
</tr>
</tbody>
</table>
As may be expected, the time to complete the cutset list increased at an exponential rate as the number of cutsets increased by multiples of ten. The obvious bottleneck in the program is Maxino’s processing of the Boolean formula. As more and more cutsets are assembled, it becomes harder and harder for Maxino to find cutsets it hasn’t already identified – I noticed while watching the $r=10000$ jobs complete that the Maxino subprocess would begin by finding cutsets in fractions of a second, but eventually it would take upwards of 20 full seconds to find the next entry. This is because every cutset is added to the Maxino input file upon each repetition, making each successive run force Maxino to deal with more and more clauses to solve.

However, the expected use of the script will be to infrequently audit cloud services, meaning that even a 12-minute runtime is not particularly bad, especially considering that the program is taking a stab at solving an NP-hard problem with every repetition. Furthermore, if a system has a large number of variables, it is likely that many cutsets will contain some of the same vulnerabilities – that is, there may be multiple cutsets which can be mitigated by replacing or updating one component. For this reason, asking for a cutset list of greater than one hundred or so might not be very useful anyway.

The times it took to find 10,000 cutsets in my randomly-generated files (named bf1, bf2, and bf3) were somewhat less clearly related. They were as follows:

<table>
<thead>
<tr>
<th>File</th>
<th>Repetitions</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>bf1</td>
<td>10,000</td>
<td>4 hours, 37 minutes, 54 seconds</td>
</tr>
<tr>
<td>bf2</td>
<td>10,000</td>
<td>13 hours, 38 minutes, 30 seconds</td>
</tr>
<tr>
<td>bf3</td>
<td>10,000</td>
<td>3 hours, 05 minutes, 55 seconds</td>
</tr>
</tbody>
</table>

Table 2: Results from running ft_analyzer on randomly-generated test files.
get 10,000 cutsets out of topo-3456. It would seem that even minor fluctuations in the dependency chains can massively affect the time it takes to solve for all clauses – which, in a way, demonstrates how well-hidden the consequences of dependency chains can be.

I would have attempted to run more 10,000-cutset tests to get a better idea of the average time it takes to solve such problems, but since an individual run could take the better part of a day to complete (and I imagine using the entire Zoo to run a dozen or so tests simultaneously would be frowned upon), I simply didn’t have the time. Luckily, as noted above, in most situations where a script like this would be used, the user probably won’t require more than a hundred or so cutsets anyway.

5. Conclusion

5.1 Possible Improvements

There is room for improvement upon the design of the fault tree generator tool. The most obvious is the Maxino SAT-solver – given its inability to return more than one cutset, I had to use a somewhat roundabout method of appending more clauses onto the input file, which at the least added file I/O overhead to each loop in the script. It is likely that better algorithms and tools for approaching satisfiability problems will appear with time, but until then, I count my blessings for the fact that one even as rudimentary as Maxino exists, as it is in fact solving an NP-hard problem with reasonable speed.

I also originally intended to build the entire ft_analyzer script as one single Python module, and though I later refactored it to be more modular, this redirection probably introduced some inefficiencies and ugliness into the code that I didn’t have time to fix by rebuilding the whole thing from the ground up with modularity in mind.

5.2 Final Thoughts

The ft_analyzer script is ultimately an iteration upon the original INDaaS approach which reinterprets the minimal risk group problem as a Boolean formula problem, thereby supplying a fairly useful solution to INDaaS’s fault tree analysis step for the first time. Using the script, useful data can be pulled from the tangles of cloud service networks, and volatile points of failure can be identified before Microsoft or Amazon lose thousands of dollars due to downtime.
References


