Flanking Security in Critical Infrastructures

George Stergiopoulos

September 2015
Flanking security in Critical Infrastructures: 

*Time-based impact analysis*

*Risk mitigation*

*Detection of logical error in functionality*

George Stergiopoulos
OUTLINE

- Background
  - Basic Concepts
  - Research motivation
  - Existing approaches
  - Thesis' Objectives

- Contribution
  - Time-based analysis of impact
  - Risk mitigation using graph theory
  - Automatic detection of logical errors

- Conclusion
  - Publications
  - Future work
BASIC CONCEPTS

- **Critical Infrastructures (CIs):** “Physical and information technology facilities, networks, services and assets which, if disrupted or destroyed, would have a serious impact on the health, safety, security or economic well-being”.

- **Interdependency:** The state of one CI depends on the output (product, asset or service) of another. [Rinaldi, 2001]

- **Software logical errors:** Software flaws due to erroneous implementation of functionality into source code. [Felmtseger, 2010]

RESEARCH MOTIVATION

- Overlooked characteristics in CIs:
  1. Static analysis of impact between infrastructures.
     - Need to analyze evolution of impact to propose actions
  2. Absence of risk mitigation mechanisms.
     - Need to pinpoint points-of-failure in dependency graphs
  3. Human error critical in high-impact failures.
     - Use of third party software in Cis.

Infrastructure Dependencies + Third-party Software → Multiple security issues in critical infrastructures
EXISTING APPROACHES

CI interdependency analysis

- Sector-specific tools.
  - Mostly energy and water.

- Analysis best on economic relationships.
  - Purely economical.

Software functionality

- Detect a priori flaws.
  - e.g. null pointers, buffer overflows.

- Debugging techniques.
  - No automated program analysis.
  - Only detect specific types of flaws.
1. Time-based impact evolution in dependencies
   • Growth rates of failures

2. Risk mitigation using graph theory
   • Pinpoint CIs and mitigate risk

3. Logical errors in software
   • Functionality flaws lead to failures in infrastructures
1. Time-based impact evolution in dependencies
   • Growth rates of failures

2. Risk mitigation using graph theory
   • Pinpoint CIs and mitigate risk

3. Logical errors in software
   • Functionality flaws lead to failures in infrastructures
**TIME-BASED IMPACT EVOLUTION IN DEPENDENCIES**

- **RESEARCH QUESTION**: Most-Critical risk paths remain the same even if time passes?

- **SOLUTION**: Time-based analysis of **Risk evolution in dependencies** through time.
  - *Fuzzy Logic* analysis of impact through time.

- CI dependencies create risk paths.
  - Some paths more dangerous than others.
  - Protection of most-critical-paths a necessity.
PREVIOUS WORK ON DEPENDENCY ANALYSIS

- **Risk of dependency** = (**Likelihood** of occurrence) * **Impact**
- **Impact** $I_{i,j}$ and **Likelihood** $L_{i,j}$ of disruption on infrastructure node $Y_j$
- **N-order Risk** $DR_{Y_0,...,Y_n}$ for a chain of disruptions:

$$DR_{Y_0,...,Y_n} = \sum_{i=1}^{n} R_{Y_0,...,Y_i} = \sum_{i=1}^{n} (\prod_{j=1}^{i} L_{Y_{j-1},Y_j}) \cdot I_{Y_{i-1},Y_i}$$
TIME-BASED IMPACT EVOLUTION IN DEPENDENCIES

- Calculates dependency impact for time frames.
  - Gives estimate of impact progression from failures.

- **Worst-case** scenario and **Growth rate** of impact provided by risk assessments.

- Supports
  1. **Slow**,
  2. **Linear**
  3. **Fast**

  evolution of impact after failure.

![Graph showing impact evolution over time](image)
TIME-BASED IMPACT EVOLUTION IN DEPENDENCIES

- Preemptive analysis detects dangerous paths for each time-frame.
- Most critical CI and path in each time slot for cascading failures.
- Different impact growth rates (e.g. path A-E-F-G at 3 hours).
RISK MITIGATION USING GRAPH THEORY

1. Time-based impact evolution in dependencies
   • Growth rates of failures

2. Risk mitigation using graph theory
   • Pinpoint CIs and mitigate risk

3. Logical errors in software
   • Functionality flaws lead to failures in infrastructures
**RISK MITIGATION USING GRAPH THEORY**

- **RESEARCH QUESTION**: Which nodes offer greater overall benefit to apply risk mitigations measures?

- **SOLUTION**: Effective risk mitigation strategies using graph Centrality metrics.
  - Degree, Closeness, Betweenness, Eccentricity and Eigenvector metrics.

- Key CIs affect multiple depended infrastructures.
  - Some do not belong in most critical risk paths.
Centrality metrics from Graph Theory to describe importance of nodes according to paths.

Describe *importance* of interconnected infrastructure nodes in dependency Risk Graphs.

1. *Closeness*: Average shortest path of node with others.
2. *Eccentricity*: Greatest distance in all shortest paths.
3. *Betweenness*: No. of paths a node participates in.
4. *Bonacich (Eigenvector)*: Influence of node.
RISK MITIGATION USING GRAPH THEORY

- Feature selection.
- Metrics detect dangerous CI nodes amongst dependencies.
  - 2000 Tests
  - 32,950 CI nodes
  - 700 graphs
  - 774,015,270 paths

<table>
<thead>
<tr>
<th>INFORMATION GAIN</th>
<th>Inbound Test</th>
<th>Outbound Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Betweeness</td>
<td>0.259</td>
<td>0.277</td>
</tr>
<tr>
<td>Eccentricity</td>
<td>0.238</td>
<td>0.285</td>
</tr>
<tr>
<td>Closeness</td>
<td>0.387</td>
<td>0.345</td>
</tr>
<tr>
<td>Eigenvector</td>
<td>0.151</td>
<td>0.260</td>
</tr>
<tr>
<td>Intersection of all Centralities</td>
<td>0.176</td>
<td>0.248</td>
</tr>
<tr>
<td>Inbound degree (sinkholes)</td>
<td>-</td>
<td>0.302</td>
</tr>
<tr>
<td>Outbound degree</td>
<td>0.281</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1: Weka’s output ranking using the Information Gain algorithm

<table>
<thead>
<tr>
<th>GAIN RATIO</th>
<th>Inbound Test</th>
<th>Outbound Test</th>
</tr>
</thead>
<tbody>
<tr>
<td>Betweeness</td>
<td>0.08</td>
<td>0.101</td>
</tr>
<tr>
<td>Eccentricity</td>
<td>0.08</td>
<td>0.101</td>
</tr>
<tr>
<td>Closeness</td>
<td>0.14</td>
<td>0.120</td>
</tr>
<tr>
<td>Eigenvector</td>
<td>0.06</td>
<td>0.09</td>
</tr>
<tr>
<td>Intersection of all Centralities</td>
<td>0.458</td>
<td>0.550</td>
</tr>
<tr>
<td>Inbound degree (sinkholes)</td>
<td>-</td>
<td>0.103</td>
</tr>
<tr>
<td>Outbound degree</td>
<td>0.101</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 2: Weka’s output ranking using the Gain Ratio algorithm
RISK MITIGATION: EFFICIENCY OF SOLUTION

- Recursive selection of CI nodes for risk mitigation.
- Comparison with two mitigation strategies.
- Up to 43.7% risk reduction from algorithm.
1. Time-based impact evolution in dependencies
   • Growth rates of failures

2. Risk mitigation using graph theory
   • Pinpoint CIs and mitigate risk

3. Logical errors in software
   • Functionality flaws lead to failures in infrastructures
LOGICAL ERRORS IN FUNCTIONALITY

RESEARCH QUESTION: Can we detect flaws in software that lead to CI failures due to erroneous implementation of business logic functionality?

SOLUTION: Automated analysis of source code.
- Dynamic - Static analysis.
- Fuzzy Logic classification.

Use of high-level 3rd part software to control CI functionality and services.
- Software flaws can lead to serious failures.
- Worst-case: Errors in functionality of applications.
Software logical errors: erroneous translation of requirements causing unintended program behavior.

Profiling logic behind software needs:

(i) Logical rules intended program functionality.
(ii) Software Executions with adequate coverage.
(iii) Evaluations of logical rules over executions.
(iv) Classification of detections to impact-levels.

(i) Logical Rules (Invariants) → MIT’s Daikon tool
(ii) Executions → NASA’s JPF tool
(iii) Evaluations → JPF tool + PlatoListener
(iv) Classification → ???
Risk classification assigned for detections:

\[ Risk(x) = Severity(x) \cap Reliability(x) \]

**Reliability**: Likelihood that a logical error exists.

**Cyclomatic Density**

**Severity**: Impact of error in execution flow.

**Information Gain**

### Classification of Detections

<table>
<thead>
<tr>
<th>C = SxR</th>
<th>Low</th>
<th>Medium</th>
<th>High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Low</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>Medium</td>
<td>Low</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>High</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>
✓ Plato tool detects logical errors in high-level code.

<table>
<thead>
<tr>
<th>Step</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Gather documentation for the software under test.</td>
</tr>
<tr>
<td>2.</td>
<td>For each app functionality, perform dynamic analysis.</td>
</tr>
<tr>
<td>3.</td>
<td>Infer dynamic invariants that describe functionality.</td>
</tr>
<tr>
<td>4.</td>
<td>Instrument source code with the dynamic invariants.</td>
</tr>
<tr>
<td>5.</td>
<td>Symbolically execute instrumented source code with JPF tool.</td>
</tr>
<tr>
<td>6.</td>
<td>Check for violations of invariants.</td>
</tr>
<tr>
<td>7.</td>
<td>Classify and output the Risk rank of violations.</td>
</tr>
</tbody>
</table>
1. Control of RTU/PLC units in gas pipes.
2. Fault injection in NASA software.
3. Airline ticketing service.
Three initiatives complement each other and solve different problems.

Different granularity level of threat analysis.

- Impact evolution detects extra high-risk paths-per-time frame.
- Detect dangerous CI nodes that affect multiple dependencies.
- Efficient risk mitigation strategies tried & tested.
- Detect logical errors in software that handles functionality.
- Predict failures to avoid catastrophic failures.

CONCLUSIONS: The road so far
Time-based analysis – I(t) tables

- Calculation of all possible I(t) values
- Pre-computed table - expected time-related impact values for fast evolving failures
- Worst impact at T = 12 hours

<table>
<thead>
<tr>
<th>Time Related Impact</th>
<th>Very Low</th>
<th>Low</th>
<th>Low</th>
<th>Medium</th>
<th>Medium</th>
<th>High</th>
<th>High</th>
<th>Very High</th>
<th>Very High</th>
</tr>
</thead>
<tbody>
<tr>
<td>Early</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>15 minutes</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>1 hour</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>3 hours</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>Medium</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12 hours</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>24 hours</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>48 hours</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Late</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 week</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>2 weeks</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>Y. Late</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 weeks</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>&gt; 4 weeks</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

```
foreach e_2 in set T do
  Set e_2 as t;
  /* For t > T always output maximum impact*/
  if t > T then
    I(t) = I;
  else if t ≤ T then
    foreach I in set I do
      if G is fast then
        I(t) = I \cdot \log_T t;
      else if G is linear then
        I(t) = I \cdot \left(\frac{t}{T}\right);
      else if G is slow then
        I(t) = I\left(\frac{t}{T}\right);
    end
  end
end
```
Example: Examined dependency input data $G = \text{fast}$ and $T = 12h$.

1. Fuzzy membership set Low membership percentages: \( \text{Low} = \{(1, 0.05) \ (2, 0.55) \ (3, 0.4)\} \) where the second value is each couple is the membership percentage of the corresponding impact value.

2. Subset of the rules used to calculate the Low output set of the values $I(t)$ are the following:

   17: IF Impact IS Low AND Time IS Early THEN Fuzzy Impact is Very Low;
   18: IF Impact IS Low AND Time IS Medium THEN Fuzzy Impact is Medium;
   19: IF Impact IS Low AND Time IS Late THEN Fuzzy Impact is High;
   20: IF Impact IS Low AND Time IS Very Late THEN Fuzzy Impact is High;

1. Fuzzy set mostly characterized by that value is the Low set.

2. Based on Table 2, this time belongs to the Medium time fuzzy set.
   - Thus, using rule 18, CIDA’s setup process will choose to output a Medium Fuzzy Impact value.
   - RightMostMembership defuzzification technique on all impact value.
TIME-BASED ANALYSIS: CALIFORNIA BLACK OUT SCENARIO

✓ California Blackout Scenario used to test the methodology implemented CIDA tool
1. Assess the cumulative dependency risk of all existing dependency paths
2. Compute all *centrality measures* for every node.
3. Select a subset of nodes for applying risk mitigation controls, based on centrality measures.
   A. Calculate Information Gain of each centrality measure for selected subset of nodes.
   B. Recurse by choosing highest output and reapplying calculations in new subset.
4. Evaluate results by comparing new graph to initial one (*risk of most critical path, max risk of all paths, or no. of paths with high risk*).
Selects nodes for risk mitigation

**Procedure choosingNodeSubsets** (C, \( IG_c(D) \), N, D)

**Inputs**
- High centrality subsets: \( C = \{C_C, C_I, C_B, C_Eg, C_E\} \)
- Head set of all nodes: \( N \)
- Number of nodes to be chosen from \( D \): PERCENT
- Subset of dangerous nodes: \( D \subset N \)
- Information gain of a centrality subset \( c \) over \( D \): \( IG_c(D) \)

**Output**
- Percentage of nodes from \( D \subset N \) for risk mitigation

**Step 1 – Start:**
Mark all sets in \( C \) as unused
Position at the root node

**Step 2 – Leaf:**
If \( C \) is empty then
  Output top nodes from \( D \) with highest centrality
  END
else
  if \( IG_c(D) = 0 \) \( \forall c \in C \) then
    Output top nodes from \( D \) with highest centrality
    END
  end if
end if

**Step 3 – Decide case:**
if nodes in \( D < \) PERCENT then
  Go to Step 4
else
  Go to Step 5
end if

**Step 4 – Backtrack:**
if at the root then
  STOP
else
  Return to the node above the current node
choose different \( c \in C \)
  Go to Step 3
end if

**Step 5 – Decision:**
if \( IG_{c_1}(D) = IG_{c_2}(D) \) \( \forall c_1, c_2 \in C \) then
  Select the leftmost \( c \) from \( C \)
else
  Calculate information gain \( IG_c(D) \) \( \forall c \in C \)
  Select \( c \) from \( C \) with the highest \( IG_c(D) \)
end if
\( N \leftarrow N \subset c \)
\( D \leftarrow D \subset c \)
\( C \leftarrow C - \{c\} \)
Go to Step 2
end procedure
Definition 1. A logical error manifests if there are execution paths $\pi_i$ and $\pi_j$ with the same prefix, such that for some $k \geq 0$ the transition $(\ell_k, \rho_k, \ell_{k+1})$ results in states $(\ell_{k+1}, s_i), (\ell_{k+1}, s_j)$ with $s_i \neq s_j$ and for the dynamic invariant $r_k$, $(s_{i-1}, s_i) \models r_k$ in $\pi_i$ and $(s_{j-1}, s_j) \not\models r_k$ in $\pi_j$, i.e. $r_k$ is satisfied in $\pi_i$ and is violated in $\pi_j$.

So, for each dynamic invariant $r_k$, PlatoListener relies on JPF’s symbolic execution to gather all execution paths that evaluate the dynamic invariant. Then, for each path with some state $s_j$ such that $(s_{j-1}, s_j) \not\models r_k$, PLATO compares it with other paths with the same prefix such that $s_i \neq s_j$ and $(s_{i-1}, s_i) \models r_k$. 
DETECTING ERRORS THAT CAUSE FAILURES. WHAT WE NEED?

JPF - Static analysis
Produces method execution paths

Daikon - Dynamic analysis Produces invariant rules

VARIABLE: ix_31 <- 0
VARIABLE: ix_31 -> 0

rjc.SimpleCounter.Main([I)V

VARIABLE: execute_at_initialization_933 -> 1

rjc.SimpleCounter.SimpleCounter_100000256_exec ()V

rjc.SimpleCounter.CounterState_100000257_exec()V

VARIABLE: TopLevel_SimpleCounter_count -> 0

VARIABLE: TopLevel_SimpleCounter_count <- 1

rjc.Chart.Chart_100000202_exec()::
ENTER

this.TopLevel_Chart_count == 2.0

this.TopLevel_Chart_Firefct1 == -0.005235987755982988

this.TopLevel_Chart_Firefct1 == this.TopLevel_Chart_Coastfct1
Severity classification based on Expected Information Gain measure.
- Successful in feature selection for information retrieval

Uses taxonomy of dangerous code instructions.
- Instructions tied to known vulnerability types
- 5 subsets/danger levels of instructions - Feature Sets to classify execution paths.

Severity ratings applied by classifying each execution path into a Severity set

<table>
<thead>
<tr>
<th>Rank</th>
<th>Example of classified methods</th>
<th>Feature Set (Category level)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>javax.servlet.http.Cookie</td>
<td>Set 1 (Level 1)</td>
</tr>
<tr>
<td>Low</td>
<td>java.lang.reflect.Field</td>
<td>Set 2 (Level 2)</td>
</tr>
<tr>
<td>Medium</td>
<td>java.io.PipedInputStream</td>
<td>Set 3 (Level 3)</td>
</tr>
<tr>
<td>High</td>
<td>java.io.FileInputStream</td>
<td>Set 4 (Level 4)</td>
</tr>
<tr>
<td>High</td>
<td>java.sql.ResultSet::getString</td>
<td>Set 5 (Level 5)</td>
</tr>
</tbody>
</table>
Instructions assigned to feature sets, i.e. danger levels.

Correct feature set inferred based on the CVSS scores in NVD repository.

Implemented using the following algorithm:

1. For each instruction, check lowest and highest ratings of NVD vulnerabilities that use this instruction.
2. Characteristics inputted in CVSS 3.0 scoring calculator 2.
   - Calculate lowest and highest possible vulnerability scores.
3. Instructions with score 7 or above grouped in Set 5.
   Instructions with score 6 to 7 in Set 4, those with score 5 to 6 in Set 3, those with score 4 to 5 in Set 2 and those with score 1 to 4 in Set 1.
Cyclomatic Complexity doesn’t consider size of code.

Modules with both high complexity and large size tend to have the lowest reliability.

Compute $V(G)$:

1. Increment one for every IF, CASE or other same construct;
2. Increment one for every DO, DO-WHILE or other same construct;
3. Add two less than number of logical alternatives in CASE;
4. Add one for each logical operator (AND, OR) in an IF.

<table>
<thead>
<tr>
<th>Rank</th>
<th>Example of classified methods</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>Cycl. Complexity Density $\leq 0.1$</td>
<td>1</td>
</tr>
<tr>
<td>Low</td>
<td>Cycl. Complexity Density $&gt;0.1$ &amp; &amp; Cycl. Complexity Density $\leq 0.2$</td>
<td>2</td>
</tr>
<tr>
<td>Medium</td>
<td>Cycl. Complexity Density $&gt;0.2$ &amp; &amp; Cycl. Complexity Density $\leq 0.3$</td>
<td>3</td>
</tr>
<tr>
<td>High</td>
<td>Cycl. Complexity Density $&gt;0.3$ &amp; &amp; Cycl. Complexity Density $\leq 0.4$</td>
<td>4</td>
</tr>
<tr>
<td>High</td>
<td>Cycl. Complexity Density $&gt;0.4$</td>
<td>5</td>
</tr>
</tbody>
</table>
PLATO: PROCESSING FLOWCHART

Start → Parse AUT code

Static Analysis
AST code tree

Gather information
Variable assignments, branch conditions, methods etc.

Dynamic Analysis
Parse Dynamic invariants

Filter inferred invariants based on control flow locations and taxonomy of instructions

Source code Instrumentation
Instrument invariants as assertions into source code

Symbolic Execution (JPF and PlatoListener)
Execute code symbolically and analyze paths & states

Severity calculation
Entropy Loss
Path classification

Reliability calculation
Cyclomatic Density

Risk calculation
Graphs and fuzzy ratings for each execution path

Show results → Exit
Error injection to source code with high representativeness

- **Code Metrics:** *Lines of Code, Cyclomatic Complexity, Average methods per Class.*
- Detect key points in the source code for fault injection.

Evaluation of system behavior when one of its components is faulty and not the behavior of the faulty component itself.

<table>
<thead>
<tr>
<th></th>
<th>Lines of code</th>
<th>Cyclomatic complexity</th>
<th>Average methods/type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rjc.Chart.java</td>
<td>10,48</td>
<td>3,31</td>
<td>29</td>
</tr>
<tr>
<td>Rjc.Chart_1.java</td>
<td>13,68</td>
<td>3,31</td>
<td>29</td>
</tr>
<tr>
<td>Rjc.Chart_2.java</td>
<td>13,68</td>
<td>3,31</td>
<td>29</td>
</tr>
<tr>
<td>Rjc.Reaction_Jet_Control0.java</td>
<td>99,50</td>
<td>7,50</td>
<td>2</td>
</tr>
<tr>
<td>Rjc.Reaction_Jet_Control1.java</td>
<td>85,50</td>
<td>7,50</td>
<td>2</td>
</tr>
</tbody>
</table>

- **PLATO’s Fuzzy Logic system classified violations with the following ratings:**
  - **Severity** = 2,
  - **Reliability** = 1.
Step 1. Functionality has two flows currently available:
   A. Exec choice(1), Exec choice(2).
   B. Exec choice(2), Exec choice(1), Exec choice(2).

Steps 2-3. Dynamic analysis: 40 invariants for selected functionality. Next invariant refers to hidden logical error:

   \[ \text{Bug.readRegisterPressure}():::ENTER \]
   \[ \text{this.checked} == \text{false} \]

Steps 4-5. Invariants instrumented in source code points.
   - Software was executed symbolically.

Step 6-7. Assertion violation detected - readRegisterPressure(): Two executions where variable was TRUE and FALSE respectively, implying a logical error.
   - Severity = 3 and Vulnerability = 5 yielding a Risk value of ~4 for detection.
**Step 1-3.** There is only one function point to test. Daikon inferred the following invariant amongst others:

```
Num Of Seats Sold <= this.Maximum Capacity
```

**Step 4-5.** Dynamic invariants instrumented in the source code and the software was symbolically executed in JPF.

**Step 6-7.** Assertion violation detected - runBug(): two executions were found where the mentioned invariant was enforced and violated respectively, thus implying a possible logical error.

- **Severity** = 5 score and a **Reliability** = 3, yielding a Risk value of 4.5
References


