Automated detection of logical errors in programs

George Stergiopoulos [1], Panagiotis Katsaros [2], Dimitris Gritzalis [1]

[1] Information Security and Critical Infrastructure Protection Laboratory
Dept. of Informatics, Athens University of Economics & Business (AUEB), Greece
{geostergiop,dgrit}@aueb.gr
[2] Dept. of Informatics, Aristotle University of Thessaloniki, Greece
{katsaros}@csd.auth.gr

Abstract. Research and industrial experience reveal that code reviews as a part of software inspection might be the most cost-effective technique a team can use to reduce defects. Tools that automate code inspection mostly focus on the detection of a priori known defect patterns and security vulnerabilities. Automated detection of logical errors, due to a faulty implementation of applications' functionality is a relatively uncharted territory. Automation can be based on profiling the intended behavior behind the source code. In this paper, we present a new code profiling method that combines an information flow analysis, the crosschecking of dynamic invariants with symbolic execution, and the use of fuzzy logic. Our goal is to detect logical errors and exploitable vulnerabilities. The theoretical underpinnings and the practical implementation of our approach are discussed. We test the APP_LogGIC tool that implements the proposed analysis on two real-world applications. The results show that profiling the intended program behavior is feasible in diverse applications. We discuss the heuristics used to overcome the problem of state space explosion and of the large data sets. Code metrics and test results are provided to demonstrate the effectiveness of the approach.

Keywords: risk, logical errors, source code profiling, static analysis, dynamic analysis, input vectors, fuzzy logic.

1 Introduction

Software development is the process of deriving an executable program description from a set of given requirements that reflect the intended program behavior, i.e. what the programmer wants his code to do and what not to do. It is an intellectual activity that translates intended functionality - in the form of requirements - into source code. Modern techniques for static and dynamic analysis of programs have been proven effective in detecting a priori known flaws, but they do not go far enough in the detection of logical errors: erroneous translation of software requirements causing unintended program behavior, due to execution flow deviations.

These errors are not a priori known like the flaws that can be detected by most static analysis and software model checking techniques. As an example, we consider the fol-

lowing [5]: "a web store application allows, using coupons, to obtain a one-time-discount-per-coupon on certain items; a faulty implementation can lead to using the same coupon multiple times, thus eventually zeroing the price". Automated detection of such program behavior is a relatively uncharted territory.

We address this problem by extracting the programmed behavior of Applications Under Test (AUTs) with code profiling techniques. Potential logical errors are then detected and classified by applying heuristics on the gathered data. Our approach is based on previous research [5-7]. In this paper, we augment and reconstruct the presented method in order to be able to make tests on complex, real-world applications. We change the static analysis, from scripted execution of possible paths to symbolic execution with various types of data listeners for the source code variables. Additionally, we formally define logical errors and lay the foundations of the method. We develop a new parser for Daikon's invariants in order to present test results using diverse, complex AUTs: a Jet Controller from NASA and an SSH framework; thus proving that our method can be utilized to detect different kinds of errors. The method consists of the following steps:

1. For an AUT, a representation of its programmed behavior is generated in the form of *dynamic invariants*, i.e. source code rules in the form of assert statements. Invariants are collected by dynamic analysis of the AUT with the MIT Daikon tool [12].
2. A preliminary analysis with the JPF tool from NASA and custom-made methods gathers the following data: (i) a set of execution paths and program states along these paths and (ii) input data vectors and a map of all program points, in which execution can follow different paths (execution flow branching points).
3. Logical errors are then detected by crosschecking information gathered with the dynamic invariants collected, during the steps (1) and (2). Invariants are checked upon multiple execution paths and their accessed program states.
4. Logical errors due to faulty input data manipulation are also detected by a tainted object propagation analysis. "Tainted" input data are traced throughout the source code and the applied sanitization checks are verified.

The main contributions of this paper are summarized as follows:

- We elaborate on our approach that was first discussed in [6] and [7], to show how the programmed behavior of an AUT can be validated efficiently and can be used as a map for logical error detection.
- We introduce fuzzy logic membership sets used to classify logical errors: (i) *Severity*, with values from a scale quantifying the impact of a logical error, with respect to how it affects the AUT's execution flow and (ii) *Vulnerability*: with values from a scale quantifying the likelihood of a logical error and how dangerous it is. The proposed fuzzy sets aim to automate reasoning based on the analysis findings, similarly to a code audit process.
- We analyze two real-world, open source applications with diverse characteristics: the *Reaction Jet Control (RJC)* application from NASA's Apollo Lunar Lander and an SSH framework called *JSCH* from the JCraft company [18. Tests involve the injection of logically malformed data based on code metrics [15], which divert the AUT's execution paths to non-intended states.

In Section 2, we review previous work on the used techniques. Section 3 provides the background terminology and some definitions needed to describe our approach. In Section 4, we present the method implementation in the APP_LogGIC tool and we discuss the problems faced and the found solutions. Section 5 focuses on the results of our experiments with the two AUTs. Finally, we conclude with a review of the main aspects of our approach and a discussion on possible future research directions.

2 Related Work

In [5], the authors describe how they used the Daikon tool [13] to infer a set of behavioral specifications called likely invariants that represent the behavioral aspects during the execution of web applets. They use NASA's Java Pathfinder (JPF) [8, 9] for model checking the application behavior over symbolic input, in order to validate whether the Daikon results are satisfied or violated. The analysis yields execution paths that, under specific conditions, can indicate the presence of certain types of logic errors that are encountered in web applications. The described method is applicable only to single-execution web applets. Also, it is not shown that the approach can scale to larger, stand-alone applications.

A variant of the same method is used in [6] and [7], where we presented a first implementation of the APP_LogGIC tool. In [6], we specifically targeted logical errors in GUI applications. We presented a preliminary deployment of a Fuzzy Logic ranking system to address the problem of false positives, problem relative to multiple areas of code analysis. Authors in [31] deal with similar issues concerning false positives/negatives in static analysis. We applied the method on lab test-beds. In [7], the Fuzzy Logic ranking system was formally defined and developed.

The research presented in [10], focuses exclusively on specific flaws found in web applications. In [11], the authors combine analysis techniques to identify multi-module vulnerabilities in web applications, but they do not address the problem of profiling source code behavior or logical errors per se.

In our current work, the method that we first proposed in [6-7] is evolved to a more complete and effective approach with the capacity to be tested on real-world complex applications, instead of test-beds and simple GUI AUT. Also, we move towards limiting false positives through classification and heuristics.

3 Profiling the behavior behind the source code

Judging from experiments, requirements analysis [17] and previous research [5-7] on profiling the logic behind an AUT, we need: (i) a set of parsable logical rules (dynamic invariants) referring to the intended program functionality, (ii) a set of finite execution paths and variable valuations with adequate coverage of the AUT functionality, (iii) the boolean valuation of the logical rules over the set of execution paths to enable detection of logical errors and (iv) a classification system for source code instructions to filter variables in branch conditions and data input vectors.

3.1 Extracting intended program functionality as Rules (Dynamic Invariants)

The functionality of an AUT is captured in the form of dynamic invariants generated by the Daikon tool from MIT. Invariants are logical rules for variables, such as `p!=null` or `var=="string"` that hold true at certain point(s) of a program in all monitored executions. Dynamic invariants represent the programmed behavior. If the monitored executions are representative use-case scenarios of the AUT, then the generated dynamic invariants refer to the AUT's intended functionality. Intuitively, if an execution path is found that violates a (combination of) dynamic invariant(s), this means that a possible logical error exists, which affects the variable(s) referred in the invariant.

3.2 Program states and their variables

In order to verify Daikon invariants we need to crosscheck them with a set of finite execution paths and variable valuations, with adequate coverage of the AUT functionality. In this section, we introduce formal definitions for the used data sets.

An imperative program $P = (X, L, \ell_0, T)$ defines [27] a set X of typed variables, a set L of control locations, an initial location $\ell_0 \in L$, and a set T of transitions. Each transition $\tau \in T$ is a tuple (ℓ, ρ, ℓ'), where $\ell, \ell' \in L$ are control locations, and ρ is a constraint over free variables from $X \cup X'$, where X denotes values at control location ℓ and X' denotes the values of the variables in set X at control location ℓ'. For verification purposes, the set L of control locations comprises the source code points, which control the execution flow of a program, i.e. conditional statements (such as branches and loops).

State of a program P is a valuation of the variables in X. The set of all possible states is denoted as $u.X$. We shall represent sets of states using constraints. For a constraint ρ over $X \cup X'$ and a valuation $(s, s') \in u.X \times u.X'$, we write $(s, s') \models \rho$ if the valuation satisfies the constraint ρ. We focus on AUTs with an explicitly provided *initial state* that assigns specific values to all variables in X. Finite computation of the program P is any sequence $(\ell_0, s_0), (\ell_1, s_1), \ldots, (\ell_k, s_k) \in (L \times u.X)$, where ℓ_0 is the initial location, s_0 is an initial state, and for each $i \in \{0, \ldots, k-1\}$, there is a transition $(\ell_i, \rho, \ell_{i+1}) \in T$ such that $(s_i, s_{i+1}) \models \rho$. A location ℓ is reachable if there exists some state s such that (ℓ, s) appears in some computation. An *execution path* or, simply, *path* of the program P is any sequence $\pi = (\ell_0, \rho_0, \ell_1), (\ell_1, \rho_1, \ell_2), \ldots, (\ell_{k-1}, \rho_{k-1}, \ell_k)$ of transitions, where ℓ_0 is the initial location.

3.3 Source code profiling for logical error detection

According to NIST [21], the impact that a source code point has in a program may be captured by the program's Input Vectors (entry points and variables with user data) and *Branch Conditions* (e.g. *conditional statements* like if-statements). These characteristics determine the program's execution flow. Our approach studies how the AUT's execution is affected by crosschecking the truth values of the extracted dynamic invariants. A logical error is defined as follows:

Definition 1. A *logical error* manifests if there are execution paths π_i and π_j with the same prefix, such that for some $k \geq 0$ the transition $(\ell_k, \rho_k, \ell_{k+1})$ results in states (ℓ_{k+1}, s_i), (ℓ_{k+1}, s_j) with $s_i \neq s_j$ and for the dynamic invariant r_k, $(s_{i-1}, s_i) \vDash r_k$ in π_i and $(s_{j-1}, s_j) \nvDash r_k$ in π_j, i.e. r_k is satisfied in π_i and is violated in π_j.

If a program error located in some transition does not cause unstable execution in the analyzed paths, it does not manifest as a logical error according to Def. 1. For this reason, our framework adopts a notion of risk for logical error detection. Risk is quantified by means of a fuzzy logic classification system based on two measuring functions, namely Severity and Vulnerability. These functions complement invariant verification and act as source code filters for logical error detection.

Our fuzzy logic approach also aims to confront two inherent problems in automated detection of code defects: the large data sets of the processed AUT execution paths and the possible false positives. Regarding the first mentioned problem, APP_LogGIC helps the code auditor to focus only to those transitions in the code that appear having high ratings in our classification system. Regarding false positives, due to the absence of predefined error patterns, APP_LogGIC's ratings implement criteria that take into account the possibility of a logical error in some transition.

Severity (critical source code points)

Depending on the logic realized by some transition $(\ell_k, \rho_k, \ell_{k+1})$, $k \geq 0$ a logical error might be of high severity or not. We consider that all program transitions have a severity measurement and we define the measuring function Severity for quantifying the relative impact of a logical error in the execution of the AUT, if it were to manifest with the transition $(\ell_k, \rho_k, \ell_{k+1})$. *Severity*$(\ell_k, \rho_k, \ell_{k+1})$ measures the membership degree of the transition in a fuzzy logic set. Variables from states (ℓ_k, s_k) and (ℓ_{k+1}, s_{k+1}) that are used in the transition are weighted based on how they affect the execution flow. Those variables that directly affect the control-flow (e.g. they are part of the AUT's input vectors and are used in branch conditions) are considered dangerous: if a logical error were to manifest because of them, it causes an unintended behavior.

Definition 2. Given a transition $\tau \in T$ enabled at a source code point, we define Severity as

$$\text{Severity}(\tau) = v \in [0, 5]$$

measuring the severity of τ on a Likert-type scale [28] from 1 to 5. If a logical error were to manifest at a source code point, the scale-range captures the intensity of its impact in the AUT's execution flow. A fuzzy logic method evaluates transitions as being of high Severity (4 or 5), medium (3) or low (1 or 2). Technical details about the criteria used in severity assignments are presented in section 4.5.

Vulnerability (logical error likelihood and danger based on its type)

Vulnerability is a measuring function quantifying the likelihood of a logical error in a given transition and how dangerous it is, based on its type. Vulnerability memberships are evaluated by taking into account: (i) the violations of dynamic invariants by the reached program states and (ii) input from an information flow analysis revealing the extent to which variable values are sanitized by conditional checks [21].

Definition 2: Given a tuple (τ, s, r), where r is a dynamic invariant, $\tau = (\ell, \rho, \ell')$ and $(\ell', s) \in (L \times u.X)$, we define Vulnerability as

$$\text{Vulnerability}(\tau, s, r) = v \in [0, 5]$$

Ratings here also use a Likert scale [28] from 1 to 5. Same as with Severity(τ), our fuzzy logic method evaluates transitions as being of "high" Vulnerability, "medium" or "low".

Tables 1 and 2 in Section 4.5 show the considered severity and vulnerability levels, while a more detailed presentation of the fuzzy logic system is given in [7].

Quantifying the risk associated with program transitions.
According to OWASP, *the standard risk formulation is an operation over the likelihood and the impact of a finding* [6]:

$$\text{Risk} = \text{Likelihood} * \text{Impact}$$

We adopt this notion of risk into our framework for logical error detection. In our approach, Severity(τ) reflects the relative *Impact* of the transition τ at some source code point, whereas Vulnerability(τ, s, r) encompasses the *Likelihood* of a logical error in τ. Given the dynamic invariant r for τ, an estimate of the risk associated with τ can be computed by combining Severity(τ) and Vulnerability(τ, s, r) into a single value called Risk. There may be many different options for combining the values of the two measuring functions. We opt for an aggregation function that allows taking into account membership degrees in a Fuzzy Logic system [16]:

Definition 3. Given an AUT and a set of paths with $s \in u.X$ representing an accessed state and $\tau \in T$ an executed transition associated with the dynamic invariants r, function *Risk(τ, s, r)* is the aggregation

$$\text{Risk}(\tau, s, r) = \textit{aggreg}(\text{Severity}(\tau), \text{Vulnerability}(\tau, s, r))$$

with a fuzzy set valuation

$$\text{Risk}(\tau, s, r) = \{\text{Severity}(\tau)\} \cap \{\text{Vulnerability}(\tau, s, r)\}$$

Aggregation operations on fuzzy sets are operations by which several fuzzy sets are combined in a desirable way to produce a single fuzzy set. APP_LogGIC applies defuzzification [20] on the resulting set, using the Center of Gravity technique. Defuzzification is the computation of a single value from two given fuzzy sets and their corresponding membership degrees, i.e. the involvedness of each fuzzy set presented in Likert values.

Risk ratings have the following interpretation: for two tuples $vs_1=(\tau_1, s_1, r_1)$ and $vs_2=(\tau_2, s_2, r_2)$, if $\text{Risk}(vs_1) > \text{Risk}(vs_2)$, then vs_1 is more dangerous than vs_2, in terms of how τ_1 and τ_2 affect the execution of the AUT and if the analysis detects a manifested logical error. In the next section, we provide technical details for the techniques used to implement the discussed analysis.

4 Design and implementation of the APP_LogGIC tool

4.1 APP_LogGIC's architecture

APP_LogGIC flags possible logical errors based on information for their impact on the program's behavior and their location in code. The more suspicious a source code point is, the higher it scores in the Fuzzy Logic system. Fig. 1 depicts the following methods:

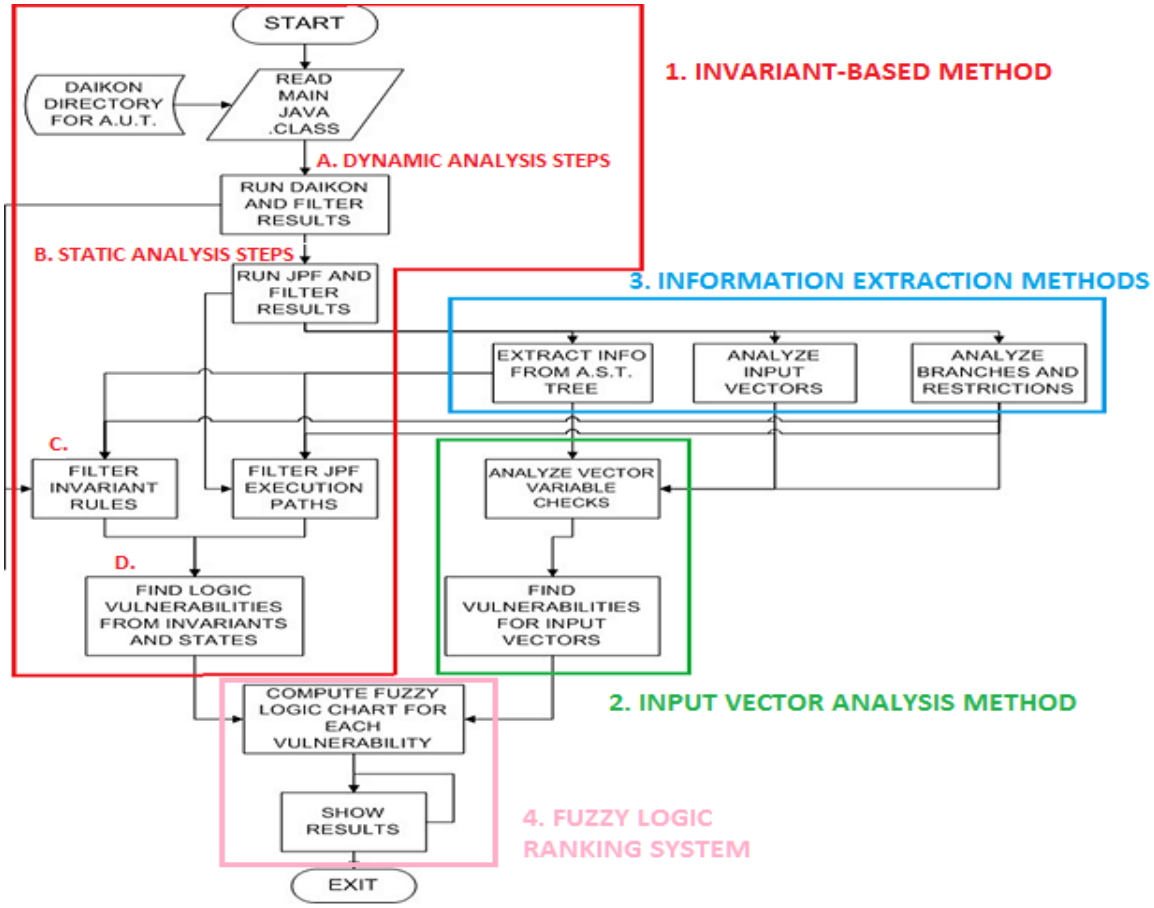

Fig. 1. The APP_LogGIC architecture

1. The *Invariant-Based Method* extracts dynamic invariants and verifies them against tuples (ℓ, s) of program states at specific code locations gathered from AUT executions. For every checked state s and dynamic invariant r a vulnerability rating is then applied using the function Vulnerability(τ, s, r).
2. The *Input Vector Analysis Method* analyzes input vectors and applies a Vulnerability rating on variables of program states that hold input data, as in (a).

3. The *Information Extraction Method* analyzes branches in the source code and rates them using the function Severity(τ).
4. *Fuzzy Logic ranking system*: APP_LogGIC combines all information gathered from (a), (b) and (c), and assesses the Risk of source code points and states based on their position and the analysis findings.

4.2 Invariant-Based Method

To automate verification of dynamic invariants for logical error detection we need: (i) a set of parsable rules representing the programmed behavior for the AUT, (ii) a set of execution paths and information for the contents of the state variables and (iii) a complete analysis of the AUT's source code to gather input vectors and map all possible points, in which execution flow can be diverted.

Extracting the programmed behavior – Dynamic Invariants
Daikon performs dynamic analysis and produces *dynamic invariants* which describe the AUT's programmed behavior. If the tool is run for a sufficient set of use-cases that covers the expected AUT's functionality, then the extracted programmed behavior matches the programmer's intended behavior. An example dynamic invariant generated from our tests is:

```
rjc.Chart.Wait_for_stable_rate_100000203_exec():::ENTER
this.TopLevel_Chart_count == 2.0
[...]
```

Fig. 2. Dynamic invariants produced by Daikon Dynamic Analysis

Daikon runs the program, observes the values that the program computes, and then reports, as in Fig. 2, assertions about source code variables that hold true throughout all AUT executions (much like "laws of conduct" for correct execution [12] [13]). The dynamic invariant of Fig. 2 shows that, upon invocation of method `Wait_for_stable_rate_exec()`, the value of the variable `TopLevel_Chart_count` is equal to '2'.

Invariant rules are filtered and only those that refer to control flow points and input vector points of the source code are kept. APP_LogGIC has a built-in Daikon parser that creates method objects with invariant objects based on the tokens of the parsed invariants. Thus, we have a fast way to parse invariants by method type, variable or class type.

Gathering execution paths and program states
Execution paths and program states are gathered using the Java Pathfinder tool (JPF) from NASA Ames Research Center [9]. The JPF core is a Virtual Machine (VM) for Java bytecode [9]. The default instruction set makes use of execution choices. JPF identifies points in programs from where execution flow can follow different paths and then systematically explores all of them [9].

Compared to our previous work in [6] [7], we have changed the static analysis from scripted execution of possible paths to symbolic execution (as in [5]). Symbolic Path-

Finder (SPF) [8] combines symbolic execution with model checking and constraint solving for test case generation. This provides us a large number of execution paths along with program states (Figure 3 depicts an example execution instruction record) while, at the same time, helps to avoid the error-prone process of manually configuring multiple application runs.

SPF's results are then used to check if Daikon's dynamic invariants hold true along the executions paths or not. If APP_LogGIC detects two different versions of an execution path that (as in Def. 1) differ in some state, such that one path satisfies a Daikon invariant while the other violates it, then a logical error is flagged and the membership in the Vulnerability Fuzzy Logic set is increased.

```
[rjc/Chart.java:342] :
if (execute_at_initialization_464 == 1) {
VARIABLE: execute_at_initialization_464 -> 1
```

Fig. 3. SPF output: instruction executed and variable content

In order to gather the sets of execution paths and states for given inputs of the AUT (store, access, update of data), we had to re-code SPF's basic listener, namely the Java class named: gov.nasa.jpf.symbc.SymbolicListener. Since SPF's model checking is based on listener objects, we extended the @override executeInstruction() and instructionExecuted() methods implemented to watch for and collect data during instruction invocation.

Verifying dynamic invariants - logical error detection

Let us consider the invariant shown in Fig. 2: APP_LogGIC checks if there are execution paths with the same prefix and some differing program state corresponding to the shown dynamic invariant. In this case, APP_LogGIC tries to find a path/state combination that violates that assertion upon entering the `exec()` method (variable's value is not '2.0') and, simultaneously, a second combination that satisfies it. This contradiction, if present, is a clear sign of a possible logical error inside `exec()` and variable `TopLevel_Chart_count`.

APP_LogGIC uses Severity ranks and focuses on dynamic invariants that refer to variables used in conditional statements (branch conditions), which are responsible for execution path deviations; if there is a possibility for a logical error manifestation, then this may happen in a branch condition since conditional branching is a decision-making point in the control flow [5]. Information for the Vulnerability rating methods is provided in Section 4.5 below.

4.3 Information Extraction Method

In order to gather input vectors and all source code points where execution flow can follow different paths we were based on the JavaC compiler and an appropriate abstract syntax tree (AST) representation. The JavaC Treescanner methods (`visitIf()`, `visitMethodInvocation()` etc.) were overridden, in order to detect and analyze sanitization checks of input data. Sanitization checks are control flow points, in which the data context of variables is checked.

4.4 Input Vector Analysis Method

A tainted object propagation analysis complements the dynamic invariant method for logical error detection. All variables that hold input data (input vectors) and the checks enforced upon them are analyzed for their role in conditional statements (as in section 3.3) and for the following correctness criterion: all input data should be sanitized before their use [21]. This analysis shows: (i) whether a *tainted variable* (i.e. a variable that contains potentially dangerous input data) is accessed in a conditional statement without having previously checked its initial values, (ii) if data from a tainted variable is passed along in methods and other variables and (iii) instances of user input that are never checked or sanitized in any way.

APP_LogGIC checks tainted variables by analyzing the conditions enforced on their content. For example, if an input vector variable is used only in the conditional statement `if(a != null)` and then variable `a` is used in a command without further sanitization of its contents, then this check is flagged as ineffective and APP_ LogGIC gives a high rating on the Vulnerability scale for that variable. More information for how rank values are assigned is provided in the tables of Section 4.5.

4.5 The Fuzzy Logic ranking system

As explained in Section 3, a Fuzzy Logic system add-on [19] is used in APP_LogGIC and ranks possible logical errors. In order to aid the APP_LogGIC end-user, Severity and Vulnerability values are grouped into 3 sets (Low, Medium, High), with an approximate width of each group of (5/3) = 1,66~1,5 (final ranges: Low in [0…2], Medium in (2…3,5] and High in (3,5…5]).

Severity (impact of a source code point on execution flow)
As a program transition we consider any instruction at a source code point that accesses variable values of the program's state. By measuring the Severity of a transition, we also assign the given Severity rating to the accessed variables; e.g., the IF-statement `if (isAdmin == true){...}` represents a check on `isAdmin`: This conditional branch is a control flow point where unintended execution deviations may occur [5]. Thus, the involved transition is classified as important (rating 3-5 on the scale). The variable isAdmin and its transition are rated as **Medium (3)**. A variable is assigned only one rating, depending on how the variable is used in transitions throughout the AUT. Table 1 below depicts the Likert ratings for Severity. For example, if two transitions exist, an if-statement and a data input transition, then a variable used in both transition will get an overall Severity value of five (5) as it can be shown on the last line of Table 1. Due to lack of space, formal presentations on the ranking system and its conditions can be found in [7].

Linguistic Value	Condition	Severity Level
Low	Random variable Severity	1
Low	Random variable Severity	2
Medium	Severity for variables used as data sinks (i.e. data originated from user input)	3
Medium	Severity for variables used in a conditional branch once on an "IF" branch	3
High	Severity for variables used in a conditional branch twice or more on an "IF" branch and/or a "SWITCH" branch	4
High	Severity for variables used as a data sink and in a conditional branch on an "IF" branch and/or a "SWITCH" branch	5

Table 1. App_LogGIC's Severity ranks in the Likert scale

Vulnerability

By measuring the Vulnerability of a tuple (τ, s, r) as seen in Section 3.3, we also assign the given Vulnerability rating to the accessed variables used in transition τ and the corresponding program state. Similar to Severity, a variable is assigned only one overall Vulnerability rating, depending on how the variable is used in transitions throughout the AUT. Rating conditions are presented in Table 2 below.

Linguistic value	Condition	Vulnerability level
Low	No invariant incoherencies / No improper checks of variables.	0
Medium	Multiple propagation of input data using only general, insufficient checks on variable content. (**Input Vector Method**)	2
Medium	Sound checks in variable contents but multiple propagation to method variables with relatively improper checks (**Input Vector Method**)	3
High	Improper/insufficient checks on variables holding input data – Variables also used in branch conditions (**Input Vector Method**)	4
High	Invariant enforcement AND invariant violation in alternate versions of same execution path (**Invariant-Based Method**)	5

Table 2. APP_LogGIC Vulnerability levels in the Likert scale

Risk

Risk represents a calculated value assigned to each tuple (τ, s, r) and its corresponding variables, by aggregating the aforementioned Severity and Vulnerability ratings. Our tool produces a set of graphs where the combined risk factor is drawn. It is calculated using Fuzzy Set Theory: Fuzzy Logic's linguistic variables in the form of IF-THEN rules (Figure 4). For clarity, all scales (Severity, Vulnerability and Risk) share the same linguistic characterization: "*Low*", "*Medium*" and "*High*".

Fig. 4 shows how Risk is calculated. The complete analysis of how formal Fuzzy Logic rules are calculated and defined is provided in [7]. Table 3 depicts the fuzzy logic output for Risk, based on the aggregation of Severity and Vulnerability.

IF Severity IS low AND Vulnerability IS low THEN Risk IS low

Fig. 4. Example of a Fuzzy Logic rule

Vulnerability \ **Severity**	Low	Medium	High
Low	Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	High

Table 3. Risk for each variable = Severity x Vulnerability

5 Experiments and test results

To the best of our knowledge, there is no commercial test-bed or open-source revision of an AUT with a reported set of existing logical errors. For this reason, our experiments were based exclusively on formal fault injection into two different open-source applications: (i) The *Apollo Lunar Lander Reaction Jet Controller* (RJC) provided along with SPF by the Java Pathfinder team in NASA Ames Research Center [9] and (ii) an SSH framework called *JSCH* from the JCraft company [18].

To cope with the inherent analysis scalability problems, we switched to method invocation paths instead of entire execution paths. This is consistent with the Daikon analysis, since Daikon dynamic invariants only describe a program's execution during entry and exit of a method invocation. As a consequence, the size of the data set for the RJC AUT was reduced from 155MB to 73MB and the execution of the APP_LogGIC analysis was speed up by ~5 minutes, an improvement of up to 80%.

5.1 Invariant tests: RJC Application

To validate APP_LogGIC's effectiveness, we injected two faults into NASA's RJC application. A malformed Java object was created that was initialized with an invalid value. The result of injecting the object in the code was a change in the AUT execution flow from its intended path to an erroneous one, thus causing a logical error.

Our approach was based on recent results from research on fault injection, which show that the key issue when injecting software faults is *fault representativeness* [15]: there is a strong relationship between fault representativeness and fault locations, rather than between fault representativeness and the various types of faults injected. To pinpoint source code methods into RJC with relatively high representativeness we used common software engineering metrics. According to [15], fault-load representativeness can be achieved by looking at the following metrics: *Lines of Code* and *Cyclomatic Complexity* which represent respectively the number of statements and the number of

paths in a component [15] [23]. *The Average methods per Class* counts the number of methods defined per type in all Class objects. If this metric scores high, it benefits these experiments since method invocation paths will be more complex and, therefore, likely more error-prone. This metric synergizes well with Cyclomatic Complexity in the RJC experiments. With the above mentioned metrics, we detected methods in RJC that have high representtativeness and then we injected logic errors in them. Our analysis was based on the CodePro Analytix tool from Google. More specifically, we evaluated the system behavior when one of its components is faulty and not the behavior of the faulty component itself. We did not consider additional metrics, as metrics tend to correlate with each other. On the other hand, the used metrics suffice in order to detect key points in the source code for fault injection [15].

	Lines of Code	**Cyclomatic Complexity**	**Average methods per Type**
Rjc.Chart.java	10,48	3,31	**29**
Rjc.Chart_1.java	13,68	3,31	**29**
Rjc.Chart_2.java	13,68	3,31	**29**
Rjc.Reaction_Jet_Control0.java	**99,50**	**7,50**	2
Rjc.Reaction_Jet_Control1.java	**85,50**	**7,50**	2

Table 4. Highest metric scores for NASA's RJC

As we can see in Table 4, these five classes have the highest ratings in RJC source code. `Reaction_Jet_Control` classes have the highest Lines of Code and Complexity values. Yet, their average methods per type are significantly low. Also, they have no execution-defining branch statements inside their code able to diverge the execution of RJC. To this end, we decided to inject the faulty values in the `rjc.Chart.Wait_for_stable_rate_100000203_exec`() method within Chart.java. JPF provided the needed method invocation paths that were used by APP_LogGIC to check the Daikon-generated dynamic invariants. **8063** method invocation paths were satisfying the invariant "`TopLevel_Chart_count == 2`" and three injected paths were violating it. APP_LogGIC detected the dynamic invariant violation for both of the two fault injections. Variable `TopLevel_Chart_count` held injected data and was also used in an if-statement: APP_LogGIC's Fuzzy Logic system classified the logical error with the following ratings:

Medium	Severity for variables used in a CB **ONCE** on: o An "IF" branch o A'SWITCH' branch	3

Table 5. Severity **rank** for RJC injection by APP_LogGIC

High	Invariant enforcement AND invariant violation in alternate versions of same execution path (**Invariant-Based analysis**)	5

Table 6. Vulnerability **rank** for RJC injection by APP_LogGIC

A total of 6,240 control flow locations (such as if-statements) were gathered and analyzed from symbolic execution. Also, 515,854 method invocations and variable Store and Invoke instructions were processed. The injected paths had 8,064 comparisons. Before injection, all 8,064 paths were found satisfying the rule. As mentioned earlier, after injection, three paths were found having different states (variable `TopLevel_Chart_count` had different values while entering and exiting method `exec()`). Both injected faults were discovered and all possible deviated execution paths were detected. Data sets can be downloaded via the link at the end of this work.

5.2 Tainted object propagation tests: JSCH framework

JSCH [18] is an SSH2 framework licensed under a BSD-style open-source license. Here, we tested APP_LogGIC's capability to detect logical errors manifesting from input data. We didn't have to inject any logical errors in JSCH since, to some extent, some were already present in examples provided along with the framework's code. JSCH uses SSH connections and built-in encryption for security. Yet, the examples provided with its source code have improper sanitization of user input.

Using Tainted Object analysis, AST trees and the Java compiler, APP_LogGIC created a map of the AUT (variable assignments, declarations, method invocations etc.). The analysis followed the tainted input and gathered the variables were input data could reside (a.k.a. *sinks*) to detect whether sanitization checks are been enforced or not. APP_LogGIC detected variables without proper sanitization and ranked these input vectors accordingly:

Medium	Severity for variables used as data sinks (i.e. data originated from user input)	3

Table 7. Severity **rank** for JSCH input vectors by APP_LogGIC

High	No check or improper checks in variables depended on input data and used in branch conditions.	4

Table 8. Vulnerability **rank** for JSCH input vectors by APP_LogGIC

APP_LogGIC found out that sanitization checks in JSCH were only comparing *initialization data to actual variable data*. This is a common logical error [21], since such checks can only show that variable data is updated compared to their initial value, but lack further content checks.

The tool detected eleven (11) sinks where data was stored without proper sanitization. Its Fuzzy Logic system calculated which of these points are dangerous based on their position and utilization inside the source code; it then detected and ranked four of them as potentially dangerous. Indeed, out of the eleven aforementioned variables used in sinks, the four variables that were detected by APP_LogGIC where the only ones that did not have proper sanitization checks enforced on their data.

Method applicability issues

Even though APP_LogGIC's result had a 100% success rate in flagging dangerous and injected points for logical errors, yet, the sample upon which APP_LogGIC was tested still remains very small to claim such a high average detection rate. The applicability of the method presented depends on how thoroughly the input vectors and dynamic invariants are analyzed. At the moment, APP_LogGIC can only analyze simple invariants and two types of input vectors. Yet, judging from the parsable syntax of dynamic invariants, one can safely deduct that, with the right parser, most dynamic invariants can be verified. This program could evolve into a potentially valuable tool: program tests created by developers using APP_LogGIC in various stages of the development cycle, could help detect logical errors and reduce the costly process of backtracking to fix them.

State explosion remains a major issue, since it is a problem inherited by the used analysis techniques. Yet, state explosion is manageable using source code classification. Both Daikon and JPF can be configured to target specific source code methods of interest rather than analyze the entire source code of an AUT. Severity ranking helps this. The use of method invocation paths downsized the initial data set for RJC from 155MB to 73MB and speeded up execution almost 80% in comparison with experiments using the entire execution paths, as shown in Table 9.

	Execution – Full paths and states	**Execution – Method invocation paths and states**
Size	155 MB	73MB
Time elapsed	~ 18 min (RJC) ~ 6 min (JSCH)	~ 4 min (RJC) ~ 6 min (JSCH)
Errors detected	2 out of 2 injections (RJC)	2 out of 2 (RJC)

Table 9. Execution times for APP_LogGIC experiments

Both of the analyzed applications are relatively small in comparison to other AUT (on the order of many GB). The AUT size will be considered in future research. APP_LogGIC ran on an Intel Core 2 Duo E6550 PC (2.33 GHz, 4GB RAM).

6 Conclusions

Preliminary results show that profiling the intended behavior of applications is feasible (up to a certain complexity level) even in real-world applications. Logic profiling aside, the use of Fuzzy Logic provides some advantages: (i) It reduces the data to be analyzed by focusing on high impact (dangerous) source code points (Severity ranking) and (ii) it is a way to treat false positives by assessing logical errors and ignoring irrelevant dynamic invariants (Vulnerability ranking): Errors in non-critical points of the source code which do not divert execution can be discarded; i.e. invariant violations referring to source code points that do not somehow affect any conditional branches (such as if-statements) during execution.

The method suffers by a number of limitations. Complex invariant rules generated by Daikon need deep semantic analysis in order to be usable. Also, Daikon does not support analysis of loops ("While" and "For"). On top of that, Daikon's dynamic execution must cover as much AUT functionality as possible, if a logical error is to be discovered. Otherwise, dynamic invariants generated will not correctly describe the AUT behavior intended by its programmer. We plan to explore different approaches using design artifacts provided by developers for a more efficient reasoning of the source code [24] such as XBRL or OWL to describe programming logic.

Another venue is to test this method on control systems used in critical infrastructures (CI) or manufacturing facilities. Widely used programmable logic controllers control functions in critical infrastructures such as water, power, gas pipelines, and nuclear facilities [25-26]. Logic errors might lead to weaknesses that make it possible to execute commands not intended by their programmer. The effect of this attack might lead to cascading effects amongst numerous interconnected CI, or can have an impact on a number of other infrastructures, including mobile systems, etc. [27-30].

Acknowledgment

This research has been co-financed by the European Union (European Social Fund ESF) and Greek national funds through the Operational Program "Education and Lifelong Learning" of the National Strategic Reference Framework (NSRF) - Research Funding Program: THALES AUEB - Software Engineering Research Platform.

References

1. Dobbins J. 1998. Inspections as an up-front quality technique. In *Handbook of Software Quality Assurance*, Prentice Hall, 217-252.
2. McLaughlin, B. 2002. *Building Java Enterprise Applications, Vol. 1: Architecture*. O' Reilly.
3. Peng, W. and Wallace, D. 1993. *Software Error Analysis*. NIST Pub. *500-209*, 7-10.
4. Kimura, M. 2006. Software vulnerability, Definition, modeling, and practical evaluation for e-mail transfer software. In *Int. Journal of Pressure Vessels & Piping*. Vol. 83, No. 4, 256–261.
5. Felmetsger, V., Cavedon, L., Kruegel, C. and Vigna, J. 2010. Toward automated detection of logic vulnerabilities in web applications. In *Proc. of the 19th USENIX Symp*. USA, 10-10.
6. Stergiopoulos, G., Tsoumas, B. and Gritzalis, D. 2012. Hunting application-level logical errors. In *Proc. of the Engineering Secure Software and Systems Conf*. Springer, 135-142.
7. Stergiopoulos, G., Tsoumas, V. and Gritzalis, D. 2013. On Business Logic Vulnerabilities Hunting: The APP_LogGIC Framework. In *Proc. of the 7th International Conference on Network and System Security*. Springer, 236-249.
8. Pasareanu, C. and Visser, W. 2004. Verification of Java Programs Using Symbolic Execution and Invariant Generation. In *Proc. of SPIN*. Springer, 164-181.
9. *The Java PathFinder tool*, NASA Ames RC, http://babelfish.arc.nasa.gov/trac/jpf/
10. Doupe, A., Boe, B. and Vigna, G. 2011. Fear the EAR: Discovering and Mitigating Execution After Redirect Vulnerabilities. In *Proc. of the 18th ACM Conference on Computer and Communications Security*. ACM, 251-262.

11. Balzarotti, D., Cova, M., Felmetsger, V. and Vigna, G. 2007. Multi-module vulnerability analysis of web-based applications. In *Proc. of the 14th ACM Conference on Computer and Communications Security*. ACM, 25-35.
12. Ernst, M., Perkins, J., Guo, P., McCamant, S., Pacheco, C., Tschantz, M. and Xiao, C. 2007. The Daikon system for dynamic detection of likely invariants. In *Science of Computer Programming*, vol. 69, 35-45.
13. *The Daikon Invariant Detector Manual*, http://groups.csail.mit.edu/pag/daikon/
14. Brumley, D., Newsome, J., Song, D., Wang, H. and Jha S. 2006. Towards automatic generation of vulnerability-based signatures. In *IEEE Symposium on Security and Privacy*.
15. Natella, R., Cotronneo, D., Duraes, J. and Madeira, H. 2013. On Fault Representativeness of Software Fault Injection. In *IEEE Trans. on Software Engineering*. Vol. 39, no. 1, 80-96.
16. *Foundations of Fuzzy Logic, Fuzzy Operators*, Mathworks http://www.mathworks.com/help/toolbox/fuzzy/bp78l6_-1.html
17. *Systems Engineering Fundamentals*. 2001. Supplementary text prepared by the Defense Acquisition University Press, Defense Acquisition University, USA.
18. *JSCH SSH framework*, JCraft, http://www.jcraft.com/jsch/
19. Cingolani, P. and Alcala-Fdez J. 2012. jFuzzyLogic: a robust and flexible Fuzzy-Logic inference system language implementation. In *Proc. of the IEEE International Conference on Fuzzy Systems*. IEEE, 1-8.
20. Leekwijck, W. and Kerre, E. 1999. Defuzzification: Criteria and classification. In *Fuzzy Sets and Systems*. Vol. 108, no. 2, 159-178.
21. Stoneburner G. and Goguen A. 2002. SP 800-30. *Risk Management Guide for Information Technology Systems*. Technical Report. NIST, USA.
22. Burns, A. and Burns, R. 2008. *Basic Marketing Research*. Pearson Education, 245.
23. Fenton, N., Pfleeger S. 1998. *Software Metrics: A Rigorous and Practical Approach*, PWS.
24. Giannakopoulou, D., Pasareanu, C. and Cobleigh, J. 2004. Assume-guarantee verification of source code with design-level assumptions. In *Proc. of the 26th International Conference on Software Engineering*. IEEE, 211-220.
25. Kotzanikolaou P., Theoharidou M., Gritzalis D. 2013. Accessing n-order dependencies between critical infrastructures. In *Int. Journal of Critical Infrastructures*. Vol. 9 (1-2), 93-110.
26. Theoharidou M., Kotzanikolaou P. and Gritzalis D. 2011. Risk assessment methodology for interdependent critical infrastructures. In *Int. Journal of Risk Assessment and Management*. Vol. 15 (2/3), 128-148.
27. Kandias M., Mitrou L., Stavrou V., Gritzalis D. 2013. Which side are you on? A new Panopticon vs. privacy. In *Proc. of 10th International Conference on Security and Cryptography*. SciTePress, 98-110.
28. Theoharidou M., Kandias M., and Gritzalis D. 2012. Securing Transportation-Critical Infrastructures: Trends and Perspectives. In *Proc. of the 7th IEEE International Conference in Global Security, Safety and Sustainability*. Springer, 171-178.
29. Mylonas A., Dritsas S, Tsoumas V., and Gritzalis D. 2011, Smartphone Security Evaluation - The Malware Attack Case. In *Proc. of the 8th International Conference on Security and Cryptography*, SciTepress, 25-36.
30. Theoharidou M., Mylonas A., Gritzalis D. 2012. A risk assessment method for smartphones. In *Proc. of the 27th IFIP Int. Information Security and Privacy Conference*, Springer, 428-440.
31. Chatzieleftheriou, G., Katsaros, P. 2011, Test driving static analysis tools in search of C code vulnerabilities. In *Proc. of the 35th IEEE Computer Software and Applications Conference Workshops*, IEEE, 96-103.