Source code profiling and classification for automated detection of logical errors

George Stergiopoulos
Information Security & Critical Infrastructure Protection Laboratory
Dept. of Informatics, Athens University of Economics & Business
76, Patission Ave., Athens, Greece
geostergiop@aueb.gr

Panagiotis Katsaros
Dept. of Informatics
Aristotle Univ. of Thessaloniki
University Campus
Thessaloniki, Greece
katsaros@csd.auth.gr

Dimitris Gritzalis
Information Security & Critical Infrastructure Protection Laboratory
Dept. of Informatics, Athens University of Economics & Business
76, Patission Ave., Athens, Greece
dgrit@aueb.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 code profiling method based on token classification. Our method combines an information flow analysis, the crosschecking of dynamic invariants with symbolic execution, and code classification heuristics with the use of a fuzzy logic system. 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 in detail the heuristics used to overcome the problem of state space explosion and that of the large data sets. Code metrics and test results are provided to demonstrate the effectiveness of the proposed approach. This paper extends the work that appears in an article currently submitted to an international conference with proceedings. In this adequately extended version of our method we present classification mechanisms that can take into account multiple user input and provide a detailed description of the used source code classification techniques and heuristics.

Categories and Subject Descriptors

D.3.3 [**Software Engineering**]: Software/Program Verification - *Class invariants, Correctness proofs, Formal methods, Model checking*.

General Terms

Algorithms, Reliability, Experimentation, Security, Human Factors, Theory, Verification.

1. INTRODUCTION

During software development, technical design requirements are prepared by the staff. These guidelines are based on business requirements gathered, that define how the application will be written, i.e. what the programmer wants his code to do and what not to do. Usually, threats and vulnerabilities are identified during the software development Life Cycle, using modern techniques for static and dynamic analysis of programs. These techniques have been proven effective in detecting a priori known flaws (such as injections or buffer overflows), but they do not go far enough in the detection of *logical errors* (i.e., erroneous translation of software requirements causing unintended program behavior, due to execution flow deviations). As an example, we consider the following [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.

In previous publications, we worked towards addressing this problem by extracting the programmed behavior of *Applications Under Test* (AUT) with code profiling techniques. Potential logical errors are then detected and classified by applying heuristics on the gathered data. This work is an extended version of a paper currently submitted to an international conference and is pending review. This updated version presents new classification mechanisms and provides a much clearer view into source code classification and heuristics. Our approach is based on previous research [5-7] and consists of the following steps:

(a) 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 Daikon tool from MIT [12].

(b) A preliminary analysis with the JPF tool from NASA and custom-made methods gathers a set of execution paths and program states along these paths

(c) Static analysis and classification of source code that recognizes input data vectors, decision points and instructions that enforce context checking on variables. This classification not only creates a map of all program points, in which execution can follow different paths (execution flow branching points), but can also group instructions about specific variables and assign ranks on each one. This rank depicts how dangerous a variable is, based on it usage inside the source code.

(d) One group of logical errors is detected by crosschecking the information gathered in (b) with the dynamic invariants collected in (a). Invariants are checked upon multiple execution paths and their accessed program states.

(e) Another type of logical errors, those that manifest due to faulty input data manipulation, is detected by a tainted object propagation analysis which uses the above mentioned classification technique. “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] [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 extend our classification method to cover multiple input vectors (source code data entry points). We present a technique were information extracted from an AUT is grouped into instructions with common variables and weighted accordingly. This way it adequately shows how dangerous the use of a source code variable might be in an AUT.
- 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], with thousands of downloads per week. Our tests involve the injection of logically malformed data based on code metrics, which pinpoint probable code locations for representative faults [15]. The injected logical errors 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 source code classification used by APP_LogGIC and link it to our Fuzzy Logic ranking system. In Section 5, we present the method implementation in the APP_LogGIC tool and we discuss the problems faced and the found solutions. Section 6 focuses on the results of our experiments with the two AUTs. We conclude with a review of the main aspects of our approach and a discussion on possible future research plans.

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. Finally, it is not shown that the approach can scale smoothly to larger, standalone 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 and we applied the method on lab test-beds. In [7], the Fuzzy Logic ranking system was formally defined and further 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.

Zeller [31] developed a state-altering technique, the delta debugging technique, which requires one passing and one failing execution that are identical in terms of the program paths. The intuition behind this technique is that any difference between the two identical executions is probably the cause of the failure.

Zhang, Gupta, and Gupta [32] developed a variant of delta debugging called predicate switching that also alters the states of predicates during program execution. Given a failing execution, the goal of predicate switching is to find the predicate that if switched (i.e., from false to true or true to false) causes the program to execute successfully. The first limitation of state-altering techniques is that they do not deal with the problem of semantic consistency because by altering a program state, the techniques do not guarantee that the new execution is an actual execution [35].The second limitation of state-altering techniques is that they require the execution of the program each time a state is altered. The second limitation of state-altering techniques is that they require the execution of the program each time a state is altered [35]. While trying to detect logical errors, we came across similar problems. Our method uses a one-time execution coupled with dynamic analysis of the AUT which seems to bypass these issues.

Other related work, which tries to detect errors in source code, uses program slicing. Weiser [33] defined program slicing and applied it to debugging. The drawback in Weiser’s technique is that the slice set often contains many program entities (in some cases the whole program). Zhang, Gupta, and Gupta [34] present a technique that uses a threshold to prune the backward slice computed for a particular program entity. By pruning the backward slice, their technique can reduce the size of the computed slice set. The first limitation of the above slicing techniques is that they do not account for the strength of the dependences between program entities and how likely each program entity is the cause of the failure. The second limitation of these techniques is that the slice sets can sometimes be very large. The third limitation is that the techniques do not provide the developer with information on how to start searching for the fault. The fourth limitation is that the techniques only compute program entities that are associated with failures instead of finding program entities that caused the failure [35].

In this work, the method that we first proposed in [6] and [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.

3. PROFILING THE BEHAVIOR BEHIND THE SOURCE CODE

Judging from experiments, requirements analysis [17], and previous research [5] [6] [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 Logical 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, ℓ_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 (ℓ_0, s_0), (ℓ_1, s_1), ... , (ℓ_k, s_k) $\in$ (L × u.X), where ℓ_0 is the initial location, s_0 is an initial state, and for each i $\in$ {0, ..., k −1}, there is a transition (ℓ_i, ρ, ℓ_{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 π = (ℓ_0, ρ_0, ℓ_1), (ℓ_1, ρ_1, ℓ_2), ... , (ℓ_{k-1}, ρ_{k-1}, ℓ_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≥0 the transition (ℓ_k, ρ_k, ℓ_{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) $\models r_k$ in π_i and (s_{j-1}, s_j) $\not\models r_k$ 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 and 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.

3.3.1 Severity (critical source code points)

Depending on the logic realized by some transition (ℓ_k, ρ_k, ℓ_{k+1}), k≥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 (ℓ_k, ρ_k, ℓ_{k+1}). *Severity*(ℓ_k, ρ_k, ℓ_{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 [29] 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 5.5.

3.3.2 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, τ = (ℓ, ρ, ℓ') and (ℓ', s) $\in$ (L×u.X), we define Vulnerability as

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

Ratings here also use a Likert scale [29] from 1 to 5. Same as with Severity(τ), our fuzzy logic method evaluates transitions as being of "high" Vulnerability, "medium" or "low".

Tables 2-3 in Section 5.5 show the considered severity and vulnerability levels, while a more detailed presentation of the fuzzy logic system is given in [7].

3.3.3 *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 ∈ *u*.X representing an accessed state and τ ∈ T an executed transition associated with the dynamic invariants r, function *Risk(τ, s, r)* is the aggregation

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

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.

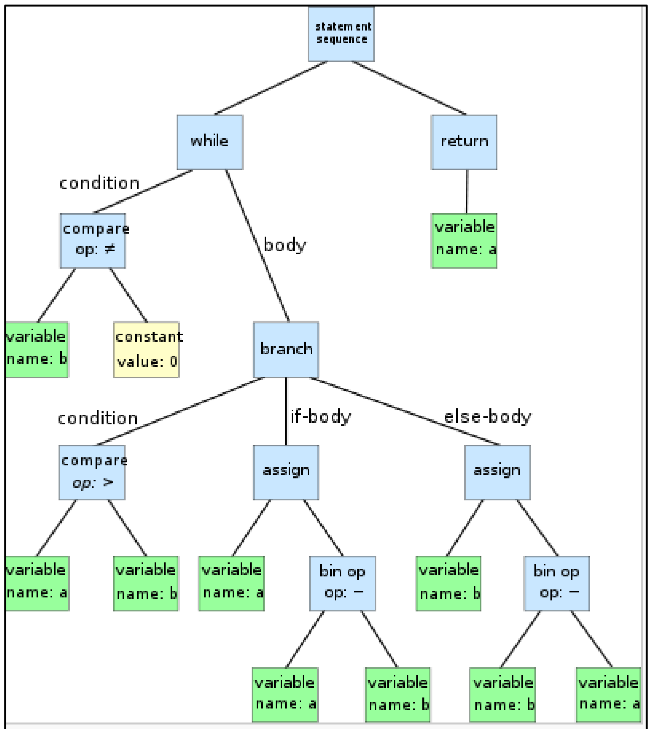
Risk ratings have the following interpretation: for two tuples $vs_1=(\tau_1, s_1, r_1)$ and $vs_2=(\tau_2, s_2, r_2)$, if $\text{Risk}(vs_1) > \text{Risk}(vs_2)$, then vs_1 is more dangerous than vs_2, in terms of how τ_1 and τ_2 affect the execution of the AUT and if the analysis detects a manifested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

4. DISSECTING THE SOURCE CODE: AST CLASSIFICATION AND HEURISTICS

There exist source code points where execution flow can follow different paths. According to them, invariants are classified as important or not for logical error detection; i.e. invariants about variables used in control flow points should be checked for violations whereas others can be discarded most of the times [5] [6] [7]. Also, there exists another group of source code points that APP_LogGIC utilizes: Points where input data from users enters an application (input vectors) is stored or used (sinks).

APP_LogGIC detects, evaluates and classifies both these code point groups. To do that, it utilizes the JavaC compiler to contruct and parse an abstract syntax tree (AST) representation of an AUT's source code. Fig. 1 depicts a sample source code and its AST tree.

Along with the above mentioned two categories, the method also analyzes all source code instructions associated with them. The following section presents the types of data extracted and used by APP_LogGIC.



An abstract syntax tree for the following code for the Euclidean algorithm:

```
while b ≠ 0
    if a > b
        a := a − b
    else
        b := b − a
return a
```

Figure 1. Source code and its AST representation

4.1 Types of data gathered

Using the AST tree, our method parses source code and gathers information concerning six (6) categories of instructions, namely: (1) control flow locations, (2) input vector locations, (3) variable declarations, (4) variable assignments, (5) method declarations, and (6) method invocations. As far as control flow (1) and input vectors location is concerned, these categories involve specific objects, methods and classes. According to Oracle Java [29] [30], these are the following:

(1) Control flow locations

According to [29], boolean expressions determine the control flow in several kinds of statements.

a. if-statements (§14.9)
b. switch-statements (§14.11)
c. while-statements (§14.12)
d. do-statements (§14.13)
e. for-statements (§14.14)

(2) Input vector locations

Java has numerous methods and classes that accept data from users [30]. They can be broadly summarized in the following two categories:

a. Byte and number-oriented I/O (streams)

BufferedInputStream	BufferedOutputStream
ByteArrayInputStream	ByteArrayOutputStream
DataInputStream	DataOutputStream
FileInputStream	FileOutputStream
FilterInputStream	FilterOutputStream
LineNumberInputStream	ObjectInputStream
ObjectOutputStream	PipedInputStream
PipedOutputStream	PrintStream
PushbackInputStream	SequenceInputStream
StringBufferInputStream	

b. Character and Text I/O (readers – writers)

BufferedReader	BufferedWriter
CharArrayReader	CharArrayWriter
FileReader	FileWriter
FilterReader	FilterWriter
InputStreamReader	LineNumberReader
OutputStreamWriter	PipedReader
PipedWriter	PrintWriter
PushbackReader	StringReader
StringWriter	

c. Miscellaneous I/O

Console	Scanner
Javax.Swing	Channels

Based on [30] and common programming experience, monitoring these sets of Java objects seems to be an adequate, albeit not entirely thorough, way of tracing user data inside Java applications.

5. THE APP_LOGGIC TOOL

5.1 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. 2 below depicts the following methods:

(a) 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).

(b) 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).

(c) The *Information Extraction Method* analyzes branches in the source code and rates them using the function Severity(τ).

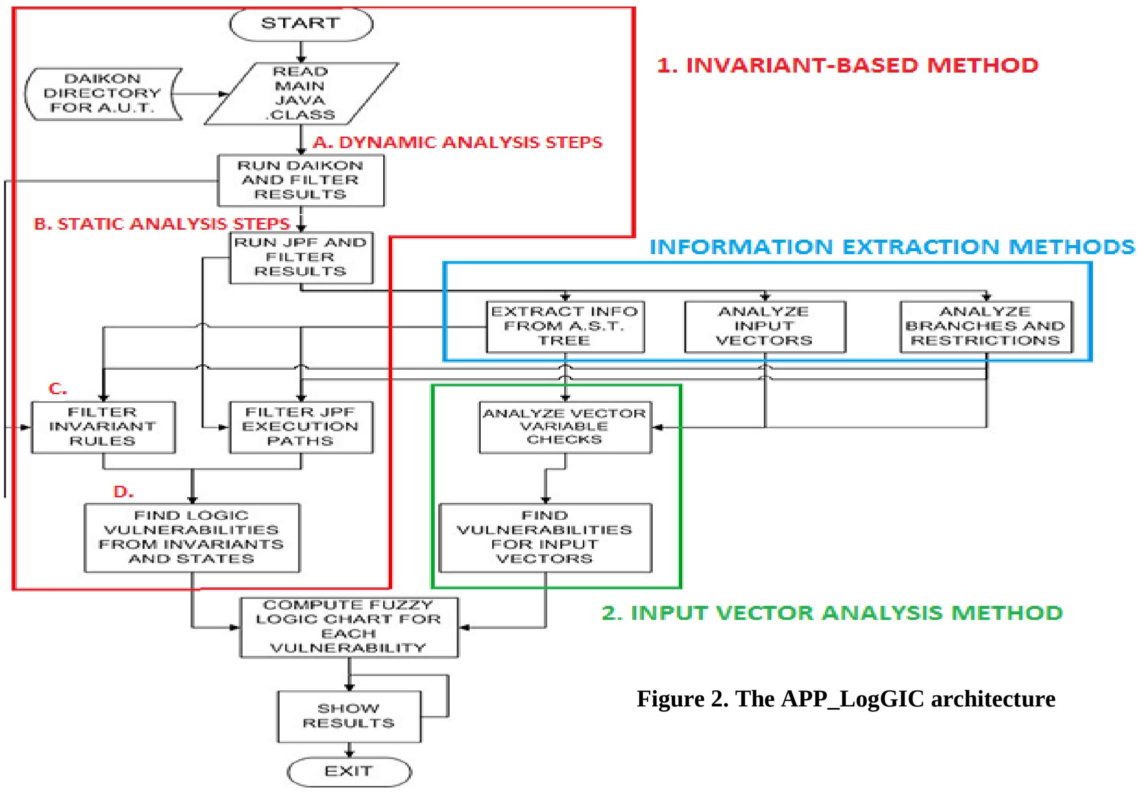

START
DAIKON DIRECTORY FOR A.U.T.
READ MAIN JAVA .CLASS
1. INVARIANT-BASED METHOD
A. DYNAMIC ANALYSIS STEPS
RUN DAIKON AND FILTER RESULTS
B. STATIC ANALYSIS STEPS
RUN JPF AND FILTER RESULTS
INFORMATION EXTRACTION METHODS
EXTRACT INFO FROM A.S.T. TREE
ANALYZE INPUT VECTORS
ANALYZE BRANCHES AND RESTRICTIONS
C.
FILTER INVARIANT RULES
FILTER JPF EXECUTION PATHS
ANALYZE VECTOR VARIABLE CHECKS
D.
FIND LOGIC VULNERABILITIES FROM INVARIANTS AND STATES
FIND VULNERABILITIES FOR INPUT VECTORS
COMPUTE FUZZY LOGIC CHART FOR EACH VULNERABILITY
2. INPUT VECTOR ANALYSIS METHOD
SHOW RESULTS
EXIT

Figure 2. The APP_LogGIC architecture

(d) *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.

5.2 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.

5.2.1 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_exe
c():::ENTER
this.TopLevel_Chart_count == 2.0
[...]
```

Figure 3. Invariants produced by Daikon's dynamic analysis

Daikon runs the program, observes the values that the program computes and then reports, as in Fig. 3, 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. 3 shows that, upon invocation of method `Wait_for_stable_rate_exec()`, the value of variable TopLevel_Chart_count is equal to '2'.

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.

5.2.2 Verifying dynamic invariants - logical error detection

Let us consider the invariant shown in Fig. 3. APP_LogGIC checks if there are execution paths with the same prefix and some differing program state corresponding to the shown dynamic invariant. If true, 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 sign of a possible logical error in `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].

The current implementation of APP_LogGIC fully classifies if-statements detected in source code. AST analysis produces meta-data describing source code branches and stores them in the following form:

```
BRANCH::93::3::TopLevel_Chart_count::==::0::
Chart.java
```

Figure 4. APP_LogGIC AST branch analysis data

Fig. 4 depicts information gathered by APP_LogGIC concerning an if-statement location. The first token represents the type of control flow location (in this case, a *BRANCH*). The second token is the source code line in which the statement can be found. The third token is the Severity rank appointed to this branch based on APP_-LogGIC's Fuzzy Logic rules. All control locations and their respective variables get a Severity rating, depending on how much they can affect the execution of an AUT. More information on how Severity ratings are applied by APP_LogGIC's Fuzzy Logic system appears in Section 5.5, Table 2.

Switch-statements are currently not fully implemented in the tool. Also, due to Daikon's inability to produce invariant rules for loops (for, do, while statements), APP_LogGIC does not analyze these control flow locations.

5.3 Input Vector Analysis Method

All AUT meta-data gathered by APP_LogGIC using its AST tree focuse on the types of data presented in Section 4.1. Information is stored in multiple files, using a token based approach similar to the one presented in Fig. 4 above.

```
DECLARE::12::double::sig_3::0::Main22::Subsy
stem114.java
ASSIGN::17::sig_3::e_and_edot_2[(int)(1)]::M
ain22::Subsystem114.javada
```

Figure 5. APP_LogGIC AST branch analysis data

Figure 5 above depicts AST meta-data gathered about variable `sig_3` in class `Subsystem114` of the ReactionJetController (RJC) AUT.

A method based on 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 execution 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.

5.3.1 Verifying Input Vectors - Tainted object detection

APP_LogGIC checks whether a *tainted variable* is accessed in a conditional statement without having previously checked its initial values, using the following algorithm. To do that, the general algorithm followed is the following:

1. Get input vector X and its method from list L of detected input vectors.
2. REPEAT while list L is not empty.
   (a) Detect DECLARATIONS of variable `A`, used in vector X.
      - Save variable `A` initialization data type.

   (b) Detect ASSIGNMENTS of data to variable `A` in same method/class.
      - If an ASSIGNMENT edits variable `A's` data context, flag A as safe (`checkEnforced = TRUE`).

   (c) Detect ASSIGNMENTS of variable `A` to other variables or methods.
      - If ASSIGNMENTS of `A` to other variables or methods exist AND `A` is not flagged as safe (checkEnforced = FALSE)
         - Add all new variables tainted by A to list L.

   (d) Detect control-flow location and sanitization checks where variable `A` is used as a left-operand in the if-statement.
      - If variable `A` is not safe (checkEnforced = FALSE), flag variable high in the Vulnerability scale (see Table 3 for classification rankings).
3. REPORT input vectors from L that are not flagged as safe.

More information, as to how rank values are assigned, is provided in Tables 2-3 of Section 5.5.

APP_LogGIC currently cannot detect errors based on the application variables' context. This needs semantic constructs to analyze information behind input data. Instead, our method uses a broader way to flag dangerous input vectors: If an instruction directly edits a variable holding input data, we consider that the programmer has implemented controls on that input data, thus flag variable as safe.

5.3.2 Classifying potentially dangerous Methods

In order to create a list of instruction that are of importance when checking for security exploits in Java AUTs, we user the *Juliet Test Suite*. This is a collection of test cases in the Java language, provided by the National Institute of Standards and Technology (NIST). The Juliet Test Suite is a collection of over 81,000 synthetic C/C++ and Java programs with known flaws. These programs are useful as test cases for testing the effectiveness of static analyzers and other software assurance tools, and are in the public domain [36]. As far as Java tests are concerned, it contains examples for 112 different CWE. Each test case focuses on one type of flaw, but other flaws may randomly manifest. An example can be found in the following Table 1.

Any type of vulnerability can be checked with APP_LogGIC's Input Vector analysis method simply by inserting instructions of interest into its MethodFinder AST tree analyzer. The following table depicts an example set of instructions checked by APP_LogGIC for our proof-of-concept tests.

The method combines information gathered in order to assign ranks that either increase or decrease the Severity rank of specific variables in instructions. The overall Severity is used as an input for detecting how dangerous the use of a variable inside the AUT. In Section 6 we provide a detailed test case classification example produced by APP_LogGIC using Juliet Suite's OS injection tests.

Table 1. Juliet Suite set used by Input Vector Analysis method for OS command injection detection

Juliet Suite - Critical Java instructions for OS injection	
Java Package	**Instruction**
java.io	readLine
Runtime	exec
ResultSet	getString
System	Getenv
Cookie	getValue
HttpServletRequest	getParameter
System	getProperty
Properties	getProperty

5.4 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]).

5.4.1 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 2 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 2. Formal presentations on the ranking system and its conditions can be found in [7].

Table 2. App_LogGIC's Severity ranks in the Likert scale

Linguistic Value	Severity 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 <u>once</u> on an "IF" branch	3
High	Severity for variables used in a conditional branch <u>twice or more</u> on an "IF" branch and/or a "SWITCH" branch	4
High	Severity for variables used as a data sink <u>and</u> in a conditional branch on an "IF" branch and/or a "SWITCH" branch	5

Table 3. APP_LogGIC Vulnerability levels in the Likert scale

Linguistic value	Vulnerability 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

5.4.2 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 3.

5.4.3 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 (Fig. 6). For clarity, all scales (Severity, Vulnerability and Risk) share the same linguistic characterization: "*Low*", "*Medium*" and "*High*".

Fig. 6 below shows how Risk is calculated. For the complete set of the defined formal Fuzzy Logic rules, please refer to [7].

IF Severity IS low AND Vulnerability IS low THEN Risk IS

Figure 6. Example of a Fuzzy Logic rule

Table 4 that follows, depicts the fuzzy logic output for Risk, based on the aggregation of Severity and Vulnerability.

Table 4. Risk for each variable = Severity x Vulnerability

Severity / Vulnerability	Low	Medium	High
Low	Low	Low	Medium
Medium	Low	Medium	High
High	Medium	High	High

6. EXPERIMENTS AND TEST RESULTS

We found 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 three different open-source applications:

(i) A classification example of source code vulnerable to OS command injection.

(ii) *The Apollo Lunar Lander Reaction Jet Controller* (RJC) provided along with SPF by the Java Pathfinder team in NASA Ames Research Center [9].

(iii) An SSH framework called JSCH from the JCraft company [18].

6.1 Example Classification: OS injection vulnerability

As mentioned earlier in Section 4.1, APP_LogGIC is able to extract information about six categories of instructions using the compiled AST tree.

```
public void bad() throws Throwable
{
   String data;
   data = ""; /* Initialize data */

(...)

/* FLAW: Read data from .properties file
*/
   data =properties.getProperty("data");
   String osCommand;

if(System.getProperty("os.name").toLowerC
ase().indexOf("win") >= 0)
{
       /* running on Windows */
       osCommand =
       "c:\\WINDOWS\\SYSTEM32\\cmd.exe /c
       dir ";
}

(...)

/* POTENTIAL FLAW: command injection */
   Process process =
   Runtime.getRuntime().exec(osCommand +
   data);
   process.waitFor();
}
```

Figure 7. Juliet Suite OS command injection test case

Let us consider this exploit shown above in Fig. 7. Variable `data` is initialized. Then it is given data from an outer source (input vector, namely `properties.getProperty`. At last, it is used as a sink in the following instruction, `getRuntime().exec`, without checking nor applying any sanitization variable `data's` data context. APP_LogGIC extracted the following information, concerning this case:

```
ASSIGN::20::data::""::bad::test.java
ASSIGN::33::data::properties.getProperty::da
ta::bad::test.java
VECTOR::33::2::data::PROPERTIES::"data"::tes
t.java
DECLARE::69::Process::process::Runtime.getRu
ntime().exec::osCommand+data::bad::test.java
VECTOR::69::2::exec::GETTEXT::Runtime::test.
java
```

Figure 8. Extracted information on variable data

By grouping information as shown in Fig. 8, APP_LogGIC detected the input vector and raised the Severity value to rank **two** (shown in bold above). Another two ranks are added since instruction exec is considered dangerous by APP_LogGIC (last line in Fig. 8). APP_ LogGIC found no sanitization checks in the code above, effectively checking the variable's content. Thus, the Severity rank given was not lowered. This raised a flag and provided the user of our tool with enough information to detect the potential flaw without having to go through the entire code of the AUT. Ranks given as output by our method, considering variable data, can be seen in Tables 5-6 below. To compare the results, the reader is advised check them against Tables 2-3.

Table 5. Severity rank assigned for Juliet Test case.

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

Table 6. Vulnerability rank assigned for Juliet Test case.

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

6.2 Invariant tests: RJC Application

In order 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 source code was a change in the AUT's execution flow from its intended path to an erroneous one, thus causing a logical error.

Our approach was based on recent results from the field of fault injection, which show that the key issue for the injection of 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.

6.2.1 Injection metrics

In order to inject faults into RJC, we had to pinpoint source code methods with relatively high representativeness. To do that, 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 in Java. 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 representativeness 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].

Table 7. Highest metric scores for NASA's RJC

	Lines of code	Cyclomatic complexity	Average methods/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

Based on Table 7, 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:

Table 8. Severity rank for RJC injection by APP_LogGIC

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

Table 9. Vulnerability rank for RJC injection by APP_LogGIC

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

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. After injection, three (3) 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 acquired can be downloaded using the link at the end of this article.

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 min, an improvement of up to 80%.

6.3 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 did not 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:

Table 10. Severity rank for JSCH input vectors by APP_LogGIC

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

Table 11. Vulnerability 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

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.

6.4 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 (Table 12). Both of the analyzed applications are relatively small in comparison to other AUT (in the order of many GB), so it is something to consider for future research or implementation.

Table 12. Execution times for APP_LogGIC experiments

	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)

APP_LogGIC ran on an Intel Core 2 Duo E6550 PC (2.33 GHz, 4GB RAM).

7. 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.

On the other hand, the method suffers by a number of limitations. The types of vulnerabilities found are limited to those which violate explicit, simple invariant rules like "`a == -1`" or "`b == "string"`". Complex rules generated by Daikon need deep semantic analysis to manipulate and understand. 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 during design for a more efficient reasoning of the source code [24]. The idea is to explore these challenges using semantic constructs such as XBRL [25] or OWL [26] to describe complex programming logic more soundly, without having to deal with Daikon's restrictions.

Another venue is to test this method on control systems used in critical infrastructures (CI) or manufacturing facilities. Widely used programmable logic controllers (PLC) control functions in critical infrastructures such as water, power, gas pipelines, and nuclear facilities [37][38]. Possible 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 [39-42].

Also, testing PLC will probably need information and business process analysis of their execution. Thus, this research venue will probably intertwine with the research idea proposed above (i.e. searching AUT for logic errors using design artifacts provided by developers).

Note: Method invocation path files, files containing the AST tree mapping by APP_LogGIC, together with execution snapshots for RJC and JSCH, can be found at:
http://www.cis.aueb.gr/Publications/APP_LogGIC-2014.zip

8. REFERENCES

[1] Dobbins J., "Inspections as an up-front quality technique". In *Handbook of Software Quality Assurance* (3rd ed.), Prentice Hall USA 217-252, 1998.

[2] McLaughlin B., *Building Java Enterprise Applications, Vol. 1: Architecture (O'Reilly Java)*. O'Reilly Media, Inc., 2002.

[3] Peng W., Wallace D., *Software Error Analysis*. In NIST Special Publication 500-209 NIST, 7-10, 1993.

[4] Kimura M., "Software vulnerability, Definition, modeling, and practical evaluation for e-mail transfer software". In *International Journal of Pressure Vessels and Piping*. Vol. 83, Issue 4, 256–261, 2006.

[5] Felmetsger V., Cavedon L., Kruegel C., Vigna J., "Toward automated detection of logic vulnerabilities in web applications". In *Proc. of the 19th USENIX Symposium*. USENIX Association, USA, 10-10, 2010.

[6] Stergiopoulos G., Tsoumas B., Gritzalis D., "Hunting application-level logical errors". In *Proc. of the Engineering Secure Software and Systems Conference*. Springer, 135-142, 2012.

[7] Stergiopoulos G., Tsoumas B., Gritzalis D., "On business logic vulnerabilities hunting: The APP_LogGIC Framework". In *Proc. of the 7th International Conference on Network and System Security*. Springer, Spain, 236-249, 2013.

[8] Pasareanu C., Visser W., "Verification of Java Programs Using Symbolic Execution and Invariant Generation". In *Proc. of SPIN* (2004). Springer, Spain, 164-181, 2004.

[9] *The Java PathFinder tool*, NASA Ames Research Center, NASA, USA http://babelfish.arc.nasa.gov/trac/jpf/

[10] Doupe A., Boe B. Vigna G., "Fear the EAR: Discovering and Mitigating Execution after Redirect Vulnerabilities". In *Proc. of the 18th ACM Conference on Computer and Communications Security*. ACM, USA, 251-262, 2011.

[11] Balzarotti D., Cova M., Felmetsger V., Vigna G., "Multi-module vulnerability analysis of web-based applications". In: *Proc. of the 14th ACM Conference on Computer and Communications security*. ACM, USA, 25-35, 2007.

[12] Ernst M., Perkins J., Guo P., McCamant S., Pacheco C., Tschantz M., Xiao C., "The Daikon system for dynamic detection of likely invariants". In *Science of Computer Programming*, vol. 69, pp. 35-45, 2007.

[13] *The Daikon Invariant Detector Manual*, http://groups.csail.mit.edu/pag/daikon/

[14] Brumley D., Newsome J., Song D., Wang H., Jha S., "Towards automatic generation of vulnerability-based signatures". In *IEEE Symposium on Security and Privacy*, May 2006.

[15] Natella R., Cotronneo D., Duraes J., Madeira H., "On Fault Representativeness of Software Fault Injection. In *IEEE Transactions on Software Engineering*, vol. 39, no. 1, 80-96, 2013.

[16] *Foundations of Fuzzy Logic*, *Fuzzy Operators*, Mathworks, http://www.mathworks.com/help/toolbox/fuzzy/bp78l6_-1.html

[17] *Systems Engineering Fundamentals*. Supplementary text prepared by the Defense Acquisition University Press, Defense Acquisition University, USA, 2001.

[18] *JSCH SSH framework*, JCraft, http://www.jcraft.com/jsch/

[19] Cingolani P., Alcala-Fdez J., "jFuzzyLogic: A robust and flexible Fuzzy-Logic inference system language implementation". In *IEEE International Conference on Fuzzy Systems*, 1-8, 2012.

[20] Leekwijck W., Kerre E., "Defuzzification: Criteria and classification". In *Fuzzy Sets and Systems*, vol. 108, issue 2, 159-178, 1999.

[21] Stoneburner G., Goguen A., *Risk Management Guide for Information Technology Systems*. Technical Report SP800-30. NIST, USA, 2002.

[22] Burns A., Burns R., *Basic Marketing Research*. Pearson Education, USA, 245, 2008.

[23] Fenton N., Pfleeger S., *Software Metrics: A Rigorous and Practical Approach*, PWS Publishing, 1998.

[24] Giannakopoulou D., Pasareanu C., Cobleigh J., "Assume-guarantee verification of source code with design-level assumptions". In *Proc. of the 26th International Conference on Software Engineering*. IEEE Computer Society, 211-220, 2004

[25] Najmi F., RosettaNet N., Bedini I., Telecom F., Chiusano J., Hamilton B., Martin M., *ebXML Registry Information Model*. OWL 2 Web Ontology Language Document Overview, W3C Recommendation, 2005.

[26] Jhala R., Majumdar R., "Software model checking". In *ACM Computing Surveys*, vol. 41 issue 4, 2009.

[27] The OWASP Risk Rating Methodology, www.owasp.org/index.php/OWASP_Risk_Rating_Methodology

[28] Albaum G., "The Likert scale revisited". *In Market Research Society Journal*, vol. 39, pp. 331-348, 1997.

[29] Gosling J., Joy B., Steele G., Bracha G., Buckley A., *The Java Language Specification*, Java SE 8 Edition, 2013 http://docs.oracle.com/javase/specs/jls/se8/html/index.html

[30] Harold E., *Java I/O, Tips and Techniques for Putting I/O to Work*. O'Reilly, 2006.

[31] Zeller A., "Isolating cause-effect chains from computer programs". In *Proc. of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering*, 2002.

[32] Zhang X., Gupta R., Gupta N., "Locating Faults through Automated Predicate Switching". In *Proc. of the 28th International Conference on Software Engineering*, 2006.

[33] Weiser M., "Program Slicing". In *Proc. of the International Conference on Software Engineering*, pp. 439–449, 1981.

[34] Zhang X., Gupta N., Gupta R., "Pruning Dynamic Slices with Confidence". In *Proc. of the Conference on Programming Language Design and Implementation*, pp. 169–180, 2006.

[35] Baah G., *Statistical causal analysis for fault localization*. Georgia Institute of Technology, 2012.

[36] Boland T., Black P., "Juliet 1.1 C/C++ and Java Test Suite". In *Computer*, vol. 45, no. 10, pp. 88-90, 2012.

[37] Theoharidou M., Kotzanikolaou P., Gritzalis D., "Risk-based criticality analysis". In *Proc. of the 3rd IFIP International Conference on Critical Infrastructure Protection*, Springer, USA, 2009.

[38] Theoharidou M., Kotzanikolaou P., Gritzalis D., "A multi-layer criticality assessment methodology based on interdependencies". In *Computers & Security*, Vol. 29, No. 6, pp. 643-658, 2010.

[39] Kotzanikolaou P., Theoharidou M., Gritzalis D., "Cascading effects of common-cause failures on Critical Infrastructures". In *Proc. of the 7th IFIP International Conference on Critical Infrastructure Protection*, Springer, pp. 171-182, USA, 2013.

[40] Kotzanikolaou P., Theoharidou M., Gritzalis D., "Accessing n-order dependencies between critical infrastructures". In *International Journal of Critical Infrastructures*, Vol. 9, Nos. 1-2, pp. 93-110, 2013.

[41] Theoharidou M., Kotzanikolaou P., Gritzalis D., "Risk assessment methodology for interdependent critical infrastructures". In *International Journal of Risk Assessment and Management*, vol. 15, nos. 2/3, pp. 128-148, 2011.

[42] Kotzanikolaou P., Theoharidou M., Gritzalis D., "Risk assessment of multi-order interdependencies between critical ICT infrastructures". In *Critical Information Infrastructure Protection and Resilience in the ICT Sector*, pp. 151-170, IGI Global, 2013.