Program analysis with risk-based classification of dynamic invariants for logical error detection

George Stergiopoulos[a], Panayiotis Katsaros[b], Dimitris Gritzalis[a]

[a] *Information Security & Critical Infrastructure Protection Laboratory, Dept. of Informatics, Athens University of Economics & Business, 76 Patission Ave., GR-10434, Athens, Greece*
[b] *Dept. of Informatics, Aristotle University of Thessaloniki, Greece*

Abstract

The logical errors in programs causing deviations from the intended functionality cannot be detected by automated source code analysis, which mainly focuses on known defects and code vulnerabilities. To this end, we introduce a combination of analysis techniques implemented in a proof-of-concept prototype called PLATO. First, a set of dynamic invariants is inferred from the source code that represent the program's logic. The code is instrumented with assertions from the invariants, which are subsequently valuated through the program's symbolic execution. The findings are ranked using a fuzzy logic system with two scales characterizing their impact: (i) a *Severity scale* for the execution paths' characteristics and their *Information Gain*, (ii) a *Reliability scale* based on the measured Computational Density. Real, as well as synthetic applications with at least four different types of logical errors were analyzed. The method's effectiveness was assessed based on a dataset from 25 experiments. Albeit not without restrictions, the proposed automated analysis seems able to detect a wide variety of logical errors, while it filters out the false positives.

Keywords: logical errors, dynamic invariants, symbolic execution, fuzzy logic, information gain, computational density.

Email addresses: `geostergiop@aueb.gr` (George Stergiopoulos), `katsaros@csd.auth.gr` (Panayiotis Katsaros), `dgrit@aueb.gr` (Dimitris Gritzalis)

1. Introduction

The automated detection of program errors and vulnerabilities has mainly focused on static analysis techniques and software model checking, which have been found effective for known defects (e.g. Time Of Check - Time Of Use errors, null pointer dereferences etc.) and some types of exploitable vulnerabilities (e.g. unsanitized input data, buffer overflows etc.) [1, 2, 3, 4, 5, 6, 7]. On the other hand, there is not analogous progress on the automated detection of errors causing *deviations from the program's intended functionality* reflected in the application's functional requirements, i.e. what the programmer wants the code to do. Such errors do not follow a priori known patterns of defects and are commonly referred to as *logical errors*. A representative example is the following [8]: "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 (e.g. by pressing the "BACK" button, then the "FORWARD" one and re-entering a coupon code in the form)".

The automated detection of logical errors in an Application under Test (AUT) will have to be based on an abstraction of its operational logic. To this end, we opt the dynamic analysis of representative execution scenarios of the AUT aiming to infer *dynamic invariants*, i.e. properties that are *likely true* at a certain program point or points. Such invariants reveal information about the program's aimed behaviour, its particular implementation and its environment (program inputs) [9]. The inferred dynamic invarinats are used to instrument the code with assertions that are subsequently evaluated through symbolic execution of the AUT to detect potential logical errors. This form of analysis cannot be neither sound, nor complete, and for this reason we propose a post-processing of the findings, in order to come up with a meaningful result. The outlined method has been implemented in a prototype tool called PLATO and consists of the following steps:

1. The dynamic analysis of the AUT takes place with the Daikon tool [9]

that monitors the execution of scenarios; monitored scenarios have to adequately cover the intended functionality of the AUT, as specified in its user manual or in additional documentation of its business logic.

2. By analyzing the AUT's source code and its Abstract Syntax Tree, PLATO gathers: (i) execution path branching points and (ii) methods that play a key role in multiple code vulnerabilities from the classification in [10].
3. The dynamic invariants from Step 1 are then filtered based on the information derived from Step 2. Only the invariants with variables that affect the execution flow or are used in "dangerous" methods are kept.
4. An instrumented version of the AUT with assertions encoding the aforementioned invariants is symbolically executed. PLATO relies on custom-made extensions of the Java PathFinder (JPF) [6, 7] to detect possible logical errors, i.e. assertions that are violated in execution paths having the same prefix with other execution paths, in which they are satisfied.
5. PLATO combines the outputs from Steps 3 and 4 through a classification approach that ranks the assertions indicating possible logical errors. Two fuzzy membership classifiers, namely the Severity and the Reliability, are used, for classifying the overall Risk of the detections. The Severity ranking is based on a classification technique called *Information Gain*, while the Reliability depends on the measured *Computational Density.*

The same combination of tools was also used in a related method [8] and earlier versions of our analysis [11, 12, 13], but in the current work we have eventually kept only basic concepts from our previous research. The dynamic invariants are now evaluated using new formal classifiers, which replace those in [11, 12, 13] and the spurious invariants are minimized, due to the tool's novel classification system.

In overall, the main contributions of this article are summarized as follows:

1. We present PLATO, a proof-of-concept prototype for detecting logical errors, race conditions and code vulnerabilities in AUTs.

2. We show how most types of information flow dependent logical errors are detected by classifying invariant violations and their corresponding execution paths.
3. We evaluate PLATO on three AUTs with logical errors that manifest different types of vulnerabilities: (i) A multi-threaded airline control ticketing system that has been previously used in a controlled experiment for program analysis techniques. (ii) A real application that handles field sensors in a SCADA system with logical errors found in NIST's test suite from [14]. (iii) An aggregated AUT test-bed for evaluating PLATO's Severity classifications on code examples from NIST's test suite.

A first description of the analysis implemented in PLATO was published in [15]. The present article is a more complete exposition of the analysis framework and in addition presents a correlation analysis with data for invariant violations from 25 experiments. The performed experiments are based on 8 different AUTs along with their mutated variants and the analysis results show that invariant violations are correlated with the high severity values computed by PLATO.

The paper is structured as follows. In Section 2 we review the related work. Section 3 introduces the analysis building blocks. Section 4 discusses the method's workflow and its implementation in PLATO. The performed experiments and the correlation analysis results are described in Section 5. Finally, the paper concludes with a critical review of the achievements, along with comments on the method's applicability and a comparison with other approaches.

2. Related Work

The logical errors have been also researched in the field of advanced debugging techniques, but in debugging we do not aim to a fully automated program analysis. For example, delta debugging [16] relies on an hypothesis-trial-result loop towards isolating failure causes by systematically narrowing down failure-inducing circumstances.

Another related area of research is program slicing [17], a technique that has been also used for logical error detection [18]. These techniques are not scaled easily and they do not take into account the strength of dependencies between the program's entities to assess their likelihood of being the cause of a failure. In effect, with program slicing we can only find the program's entities related to some failure, but not its cause [19].

In [20], the authors propose an automated software testing approach based on the generation of new test inputs to direct execution along alternative program paths. This approach can only detect logical errors leading to program crashes, assertion violations, and non-termination, whereas our method relies on directed dynamic monitoring of executions in order to profile the semantic differences between similar execution paths.

The principles of logical error detection by evaluating dynamic invariants [9, 21] were first introduced in [8]. Only certain types of logical errors were addressed for web applications, whereas that approach was driven by symbolic input for model checking the dynamic invariants within the AUT. Our work differs in the following aspects: (i) we aim to detect logical errors, for any type of standalone application with inputs that can range over infinite domains (in [8], only input in the form of a `web.xml` file was considered), (ii) in addition to evaluating dynamic invariants for the method exit points, we evaluate invariants for the method entry points, which may also manifest deviated behaviour, (iii) many more invariants have to be handled for standalone applications, and therefore we introduce a risk-based system for their classification, in order to filter the possible false positives (our classifier is trained using collections of known code vulnerabilities).

In [22], the authors introduce a technique based on symbolic execution to improve the validity of dynamic invariants computed by Daikon. Such an approach is an interesting prospect towards refining and filtering spurious invariants in PLATO. However, the work in [22] aims to the generation of new test cases for Daikon, whereas PLATO detects logical errors that can cause malfunctioning or code vulnerabilities in most AUTs. Therefore, we apply invariant filtering to keep only the invariants that can lead to serious vulnerabilities.

3. Analysis Building Blocks

We present the analysis building blocks of PLATO: (i) the generation of likely invariants by dynamic analysis for profiling the AUT's functionality, (ii) the symbolic execution of the instrumented AUT with dynamic invariants and (iii) the classification of the results with our fuzzy logic system.

3.1. Dynamic invariants for profiling the functionality of an AUT

Dynamic invariants are logical rules for variables, such as `p!=null` or `var =="string"`, which hold true at certain point(s) of a program in *all* monitored executions. Daikon [9, 21] is a well-known tool for the dynamic analysis of programs towards computing likely invariants.

When focusing on representative use-case scenarios, while taking into account all possible restrictions and prerequisites for an AUT, such an analysis and the generated invariants are used for profiling the intended functionality. If the AUT's use cases are provided in a use-case diagram, an adequate coverage of the AUT's functionality can be achieved by monitoing executions for each possible flow of events, i.e. diagram path in all documented use cases [23]. For example, to cover a branching-point of a use case, we need to analyze with Daikon at least two different execution scenarios. All possible combinations of input should be tested, which can be derived from the business rules associated with the AUT's functions. For example, let us consider an AUT that can accept commands used to control pressure pipes. Daikon's analysis would have to include all possible combinations of commands.

3.2. Verification of dynamic invariants for logical error detection

Any imperative program denoted by $P = (X, L, \ell_0, T)$ [24] defines 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. The state of P is a valuation of the variables in X at some control location during the program execution. 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 defined in $X \cup X'$, with X the program's state variables at control location ℓ and X' the state variables at control location ℓ'.

NIST's recommendation in [25] states that the impact of a source code point can be captured by: (i) the *input vectors*, i.e. the program's entry points and variables representing user input, (ii) the *branch conditions* and (iii) the *sinks*, i.e. points with methods using variables with user input. Therefore, the set L comprises the points that determine the execution control flow (conditional expressions in branches and loops) and those that are critical for the program's fuctionality (methods that manipulate user input).

Let us denote with $u.X$ the set of all possible states for P. For a constraint ρ over $X \cup X'$ and a pair of states $(s, s') \in u.X \times u.X'$, we write $(s, s') \models \rho$ if the variables valuations in X and X' satisfy the constraint ρ. A *finite computation* of P is any sequence $(\ell_0, s_0), (\ell_1, s_1), ..., (\ell_k, s_k) \in (L \times u.X)$, where ℓ_0 is the initial location, s_0 is the initial state that assigns values to all variables in X, and for each $i \in \{0, ..., k-1\}$, 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 is some state s, such that (ℓ, s) appears in a computation. An execution path or, simply, path of the P is any sequence $\pi = (\ell_0, \rho_0, \ell_1), (\ell_1, \rho_1, \ell_2), ..., (\ell_{k-1}, \rho_{k-1}, \ell_k)$ of transitions.

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 $(\ell_{k+1}, s_i), (\ell_{k+1}, s_j)$ with $s_i \neq s_j$ and for the dynamic invariant r_k, $(s_{i-1}, s_i) \models r_k$ in 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.

PLATO converts the dynamic invariants into Java assertions that are used for instrumenting the source code. Let us consider that the invariant `p!=null` holds true at the entrance of a method. PLATO creates the assertion `[assert (p!=null);]` and instruments it at the beginning of that method. Dynamic invariants are instrumented according to two filtering criteria: (i) invariants that concern variables which affect the execution flow (conditionals in branches and loops) and (ii) invariants related to methods which are tied to known code vulnerabilities [26], [27]. The latter takes place using a taxonomy in PLATO (Section 4.3) that classifies methods according to their danger level based on [26], the Oracle's Java Taxonomy [28, 29] and reports from code audits [4]).

The execution paths are monitored with a state listener inside JPF, called `PlatoListener`. Our listener processes a log of the program's assertions, along with their valuations for the checked execution paths, as well as the execution branches in order to execute different transitions, while JPF is backtracking to previous states. For any dynamic invariant r_k, JPF's symbolic execution uses `PlatoListener` to gather all execution paths in which the invariant is evaluated. For some state s_j in a path, such that $(s_{j-1}, s_j) \not\models r_k$, we are interested to another path with the same prefix, such that $s_i \neq s_j$ and $(s_{i-1}, s_i) \models r_k$.

JPF's symbolic execution can start propagating information from any point in the program via attributes associated with program variables, operands etc [30]. During JPF's path traversal `PlatoListener` monitors all visited states to evaluate the asserted invariants. It utilizes JPF's implementation of the Choco constraint solver/decision procedure[1], for linear/non-linear constraints [32].

3.3. Fuzzy logic classification of invariant violations

The criterion of Definition 1 simply *could indicate a logical error* that in no way is located necessarily at the point of the violated invariant. Moreover, the dynamic analysis by Daikon is neither sound, not complete; an invariant violation does not provide sufficient evidence for characterizing the finding as a logical error. The true logical errors may have varying impacts to the functionality of the AUT, while the program's execution might or might not be diverted to exploitable program states. Thus, violations of asserted invariants will have to be assessed based on various characteristics and their position within the source code.

PLATO uses two advanced classification functions and a fuzzy set theory

[1] Choco [31] is a Java library for constraint programming.

approach for classifying the invariant violations found. For each detection, the result is a quantitative truth value ranging between 0 and 1, which measures the error's impact to the execution of the AUT. Such a truth value corresponds to a membership percentage in a fuzzy set [33], where the selected fuzzy sets represent different levels of danger. Thus, a logical error that does not affect critical functionality and does not divert the execution to exploitable states is not characterized as having a high impact. On the other hand, a logical error that for example is found in an if-statement of a user authentication module could lead to exploitable program states and it is therefore classified as extremely dangerous. Thus, violations of asserted invariants are classified into two different groups of sets, along with the corresponding execution paths:

- the *Severity* sets quantifying the danger level, i.e. the impact that an exploitable error would have;
- the *Reliability* sets, which quantify the overall reliability based on the size and the complexity of the code (measured by the Cyclomatic Density metric) that is traversed along an execution path.

PLATO's ratings are used to implement criteria that take into account the possibility of a true logical error in some transition. This is the only way to characterize a false positive, since true logical errors cannot be characterized based on predefined error patterns. Moreover, PLATO's classification system provides a means to cope with the large sets with execution paths, which have to be processed. The code auditor can thus focus only to those path transitions that appear having high ratings.

3.3.1. Severity

The severity measurement plays the role of a code auditor who inspects the program by tracing execution paths with specific characteristics to find flaws. The traced paths include: (i) the program's input vectors, (ii) the branch conditions with variable checks and (iii) the sinks.

For an execution path π, $Severity(\pi)$ measures π's membership degree in a Severity fuzzy set that reflects how dangerous is a logical error, if it were to manifest in path π, i.e. its relative *impact*.

Definition 2. Given the execution path π, we define

$$Severity(\pi) = \nu \in [1, 5]$$

to measure the severity of π on a Likert-type scale from 1 to 5.

In a Likert-type scale [34], a measurement specifies the level of danger of a logical error and the scale's range captures the intensity of its impact in the program's execution flow. An execution path π is weighted based on how its transitions and the corresponding methods affect the program's execution. The execution paths are classified in one out of five Severity categories ranked from one to five, based on their measured *Statistical Information Gain* (aka Expected Entropy Loss) [35]. Categories are then grouped into Fuzzy Logic sets using labels: *high* severity (4-5), *medium* (3) or *low* (1 or 2).

The Expected Information Gain is very successful in feature selection for information retrieval [36]; data are classified based on specific selected features while non-informative terms are removed according to corpus statistics [37].

The Information Gain has been also used in [38] and [39] for classifying source code, but in our analysis it allows classifying execution paths and their corresponding methods into danger levels. More specifically, PLATO looks for specific characteristics (features) based on a taxonomy of methods that have been reported to be tied to known types of vulnerabilities [26],[40]. The taxonomy includes five subsets of methods involved in similar types of vulnerabilities and therefore considered to have the same level of impact. A number of the Likert scale has been assigned to each set that depicts the danger level: set 1 contains the least dangerous methods while set 5 contains the most dangerous methods, which are known to be involved in many critical vulnerabilities. For example, the method `System.exec()` is known to be tied to OS code injection vulnerabilities [26] and it is therefore grouped into set 5. Table 1 provides example methods of the taxonomy and their classification into its sets.

Rank	Example methods	Set of methods
Low	javax.servlet.http.Cookie (new Cookie())	Set 1 (Level 1)
Low	java.lang.reflection.Field.set()	Set 2 (Level 2)
Medium	java.io.PipedInputStream (new PipedInputStream())	Set 3 (Level 3)
High	java.io.FileInputStream (new FileInputStream())	Set 4 (Level 4)
High	java.sql.PreparedStatement.prepareStatement()	Set 5 (Level 5)

Table 1: Severity classification examples - User input methods

The methods in Table 1 are used for user input in the AUT and their classification ranks reflect the different danger levels. For example, an invocation of `javax.servlet.http.Cookie()` creates a Cookie object to hold text data. This input vector may lead to various vulneratilities, whose impact varies from exposure of information (CWE-315, CWE-539, CWE-614) up to serious XSS injection attacks (CWE-79) [10]). However, the creation of an object does not provide enough evidence to raise an alarm, hence this method is ranked as a Level 1 danger. On the other hand, the `java.sql.PreparedStatement.prepareStatement()` is ranked as a very dangerous (Rank 5) method, since malicious data concatenated into the parameters of the `prepareStatement()` invocation result directly in a SQL Injection vulnerability. The technical details of our taxonomy of methods are further discussed in Section 4.3.

For each execution path, every set gets its own Information Gain measurement. High information gain values mean that the set is an effective discriminator [39]. Severity (π) thus shows which set of the taxonomy best characterizes the path π and it is the set that exhibits the highest overall Expected Information Gain (EIG). Since each set is tied to a specific level of impact (danger), then this level also indicates the danger level of the execution path.

The related theory comes from [35]. Let $Pr(C)$ be the probability of a transition in the path that indicates that the path is considered dangerous ($\overline{C}$ is the negation of C). $Pr(C)$ is quantified as the ratio of the dangerous methods over the total number of methods found in the path [39]. Let f be the event that a specific method or statement exists in the path from those included in PLATO's taxonomy and $\overline{f}$ its negation. The *prior entropy* e is the probability distribution that expresses how certain we are that an execution path is considered dangerous, before f is taken into account:

$$e = -Pr(C) \lg Pr(C) - Pr(\overline{C}) \lg Pr(\overline{C}) \tag{1}$$

where *lg* is the logarithm with base 2. The *posterior entropy*, when f has been detected in the path is

$$e_f = -Pr(C|f) \lg Pr(C|f) - Pr(\overline{C}|f) \lg Pr(\overline{C}|f) \tag{2}$$

whereas the posterior entropy, when the feature is absent is

$$e_{\overline{f}} = -Pr(C|\overline{f}) \lg Pr(C|\overline{f}) - Pr(\overline{C}|\overline{f}) \lg Pr(\overline{C}|\overline{f}) \tag{3}$$

Thus, the *expected overall posterior entropy* (EOPE) is given by

$$EOPE = e_f Pr(f) + e_{\overline{f}} Pr(\overline{f}) \tag{4}$$

and the EIG for f is

$$EIG = e - e_f Pr(f) - e_{\overline{f}} Pr(\overline{f}) \tag{5}$$

The EIG is always non-negative and higher scores indicate more discriminatory features. The higher the EIG for a given set of methods *f*, the more certain we are that this set f best describes the execution path.

3.3.2. Reliability

The Reliability function quantifies how reliable is an execution path, i.e. the likelihood of manifestating an exploitable behaviour in a variable usage.

Definition 3. Given the execution path π, with a set of state variables, we define Reliability as

$$\mathit{Reliability}(\pi) = \nu \in [1, 5]$$

to measure the reliability of π on a Likert scale from 1 to 5.

Similarly to the *Severity* function, our fuzzy logic system classifies execution paths in categories: *highly* safe (4-5), *medium* (3) or *low* (1 or 2).

The inherent risk in an AUT is connected to the complexity of its source code [41]. A broadly accepted measure is the well-known *Cyclomatic Complexity* [42], which however does not take into account the size of the analyzed code. The original McCabe metric is defined as

$$. \ V(G) = e - n + 2$$

where $V(G)$ is the cyclomatic complexity of the flow graph G of a program, e is the number of edges and n is the number of nodes. $V(G)$ can be computed by applying the following steps [43]:

1. increment by one for every IF, CASE or other alternate construct;
2. increment by one for every DO, DO-WHILE or other repetitive construct;
3. add two less than the number of logical alternatives in a CASE;
4. add one for each logical operator (AND, OR) in an IF.

A more effective evaluation of the inherent risk of an AUT can be based on a combination of the (cyclomatic) complexity and the code's size [44]. Modules with both a high complexity and a large size tend to have the lowest reliability. Modules with smaller size and high complexity are also a reliability risk, because they feature very terse code, which is difficult to change or to be modified.

PLATO implements heuristics that assign Reliability ratings to execution paths through an analysis based on McCabe's algorithm and the computation of the *Cyclomatic Density.* This is given as the ratio of the Cyclomatic Complexity to the logical lines-of-code, which measures the number of executable "statements" in the path (some statements like for example the variable assignment are excluded) [45]. The higher the Cyclomatic density value, the denser the logic. Thus, low output values from the Reliability classification function reflect reliable paths, whereas high values reflect complex, error-prone code. Related research in [44, 45] propose that Cyclomatic Density values should be in the range of .14 to .42, in order to keep them simple and comprehensible.

Table 2 depicts the classification categories applied for execution paths when using the Reliability classification function.

Rank	Example of classified methods	Category
Safe	Cycl. Density <= 0.1	1
Safe	Cycl. Complexity Density >0.1 && Cycl. Density <= 0.2	2
Medium	Cycl. Complexity Density >0.2 && Cycl. Density <= 0.3	3
Error-Prone	Cycl. Complexity Density >0.3 && Cycl. Density <= 0.4	4
Error-Prone	Cycl. Density >0.4	5

Table 2: Reliability categories based on Cyclomatic Density values

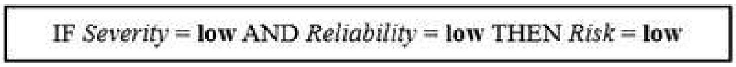
3.3.3. Combining Severity and Reliability ratings to quantify Risk

According to OWASP [26], the standard risk formulation is an operation over the likelihood and the impact of a finding:

$$Risk = Likelihood * Impact$$

An estimate of the risk associated with an execution path π can be computed by combining $Severity(\pi)$ and $Reliability(\pi)$ through applying an aggregation operation. Such an operation can combine several fuzzy sets to produce a single fuzzy set. Risk ratings have the following interpretation: for two execution paths π_1 and π_2, if $Risk(\pi_1) > Risk(\pi_2)$, then π_1 is more dangerous than π_2, in terms of the likelihood of these paths to divert execution to non-intended states and to cause an unexpected behaviour. The Risk rank of an execution path π is calculated using fuzzy logic's IF-THEN rules (Figure 1).

The membership sets of our fuzzy logic classification system are shaped as follows. We use pairs of the form (a, b), where a depicts the rank value and b depicts the membership percentage of that rank in the corresponding set. For example, $Severity - Medium = (2.5, 1)$ means that an output rank of 2.5 is a



IF *Severity* = **low** AND *Reliability* = **low** THEN *Risk* = **low**

Figure 1: Example of a Fuzzy Logic rule

member of the $MediumSeverity$ set with 100% certainty. In this way, PLATO plots ranks 1 to 5 into membership sets. The rest of all intermediate values are plotted based on points (a, b) as follows:

1. The *Severity* set: the [1..5] impact scale is partitioned into groups: Low =(0,1) (3,0), Medium =(1.5,0) (2.5,1) (3.5,0), High =(3,0) (5, 1).
2. The *Reliability* set: the [1..5] scale is partitioned into groups: Low =(0, 1) (1, 1) (3,0), Medium =(0, 0) (3, 1) (5, 0), High =(0,0) (5,1).

Thus, the fuzzy estimation of Risk values for a detected logical error is computed from tables with all possible values for Severity and Reliability. Initially, the appropriate IF-THEN rules are invoked. The result is essentially a membership function and a truth value controlling the output set, i.e. the linguistic variables Severity and Reliability. The membership percentages concerning Risk indicate the Risk group (Low, Medium or High) in which a logical error belongs to. Howver, this is still a fuzzy result. A "defuzzification" step [46] is applied at the end to get a single quantitative value. In this step, all IF-THEN output results are combined to give a single fuzzy Risk value for a logical error detection.

PLATO derives the aggregated Risk output using the Center of Gravity technique: a single value is computed from the two fuzzy sets and their corresponding membership degrees, i.e. the involvedness of each fuzzy set presented in Likert values. Severity and Reliability values are grouped into the same 3 sets as those in the Severity and Reliability scales (Low, Medium, High), with an approximate width of each group of (5/3) = around 1,5 (final ranges: Low in [0, 2], Medium in (2, 3.5] and High in (3.5, 5]).

Table 3 shows the fuzzy logic output for Risk, based on the aggregation of Severity and Reliability. Thus, when Severity is "Medium" but Reliability is "Error-Prone", then Risk is considered as HIGH, since Severity output "shifts" towards its worst-case, due to the reliability output.

Severity Reliability	Low	Medium	High
Safe	Low	Low	Medium
Medium	Low	Medium	High
Error-Prone	Medium	High	High

Table 3: Severity x Reliability = R - Risk sets

High Severity rankings have more weight than Reliability rankings: rightmost maximum values are taken into account in set aggregation, which assign more weight on Severity values (Reliability ratings provide only a generic view of the execution path's overall complexity).

Our Fuzzy Logic system has been implemented using the jFuzzyLogic library [47]. The technical details can be found in [47], [12] and [13].

4. Method and tool architecture

4.1. The method's workflow

The analysis building blocks of Section 3 are part of a workflow for the detection of logical errors with the following steps:

1. **Use case scenarios.** We assume the existence of use case scenarios that exercise the functionality of the AUT. These use-case scenarios will have to cover the functionality of the AUT sufficiently.
2. For each use-case scenario, a **dynamic analysis** with the Daikon tool is performed. A set of inferred dynamic invariants is obtained that characterize the functionality of the AUT.
3. The Daikon invariants are loaded in PLATO and are then processed as follows:
    - First, they **are filtered**, in order to use only those invariants that refer to the high-risk transitions, i.e. (i) statements that affect the

program's execution flow, and (ii) methods connected to the manifestation of exploitable behaviour (e.g. method `System.exec()` for executing OS commands with user input).

- **The AUT code is instrumented** with the remaining invariants in the form of Java assertions [26].
- The **instrumented code is symbolically executed** by JPF. A sufficient number of execution paths have to be covered, far more than the initially available use-case scenarios. JPF relies on the `PlatoListener` in order to detect invariants satisfying Definition 1.

4. **PLATO gathers `PlatoListener` detections and classifies** them into Severity and Reliability levels. A Risk value is then computed using Fuzzy Logic. The more suspicious an invariant violation and its corresponding execution path is, the higher it scores in the Risk scale.

PLATO accepts input from Daikon (step 2) and automates the source code analysis in step 3.

4.2. PLATO's architecture

The analysis building blocks of Section 3 have been implemented in PLATO by the set of components in Figure 2 with the shown input–output dependencies:

1. PLATO's source code analysis by the components in orange is based on the Abstract Syntax Tree (AST) derived by the Java compiler. Compiler methods such as `visitIf()` and `visitMethodInvocation()` have been overridden for analyzing branch conditions and variable sanitization checks, as well as to provide information for method invocations, variable assignments and declarations. The following example shows the meta-data gathered for the variable `sig_3` in a class named `Subsystem114`:

$$DECLARE :: 12 :: \textit{double} :: \textit{sig\_}3 :: 0 :: Main22 :: \textit{Subsystem}114.java$$

2. The components in green implement PLATO's filtering and verification of inferred dynamic invariants w.r.t. program states (ℓ, s) at certain locations in code gathered from symbolic execution of the AUT.

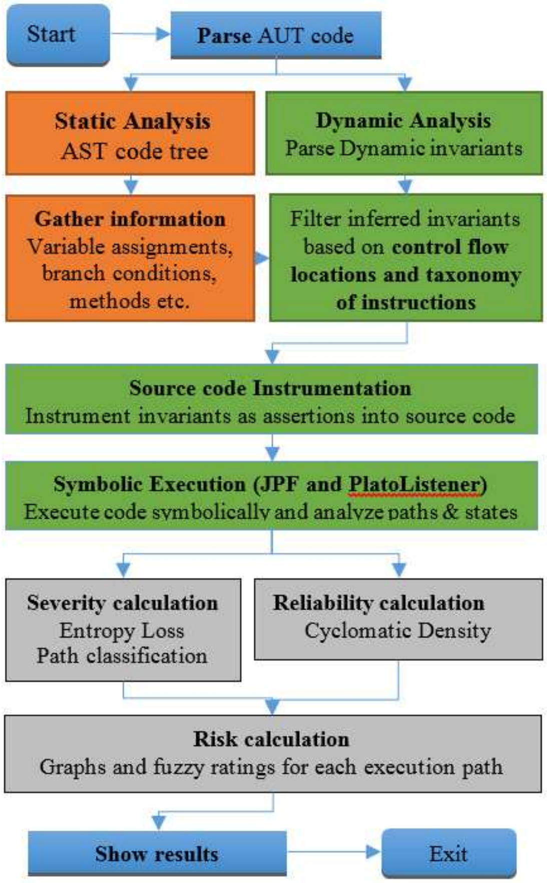

Start
Parse AUT code
Static Analysis
AST code tree
Dynamic Analysis
Parse Dynamic invariants
Gather information
Variable assignments, branch conditions, methods etc.
Filter inferred invariants based on control flow locations and taxonomy of instructions
Source code Instrumentation
Instrument invariants as assertions into source code
Symbolic Execution (JPF and PlatoListener)
Execute code symbolically and analyze paths & states
Severity calculation
Entropy Loss
Path classification
Reliability calculation
Cyclomatic Density
Risk calculation
Graphs and fuzzy ratings for each execution path
Show results
Exit

Figure 2: PLATO's processing flowchart

3. The components in grey implement PLATO's fuzzy logic system that combines all information gathered from (1) and (2), and assesses the Risk of execution paths, based on their position and the analysis findings. The Fuzzy Logic system has been implemented using the jFuzzyLogic library [47].

4.3. Classifying execution paths

Following Oracle's JAVA API documented in [27, 28, 29], we propose three categories of Java methods for the classification of execution paths. Severity ranking depends on (i) input vectors and (ii) potentially exploitable methods (sinks), while Reliability ranking is based on (iii) control flow checks.

A Taxonomony of methods for Severity calculations

We reviewed more than 159 Java methods that were grouped into sets representing similar danger levels. These sets are used as features in the Information Gain algorithm. Classified methods were assessed based a set of known security flaws by NIST [14] and other source code test-beds.

We propose five different sets, corresponding to danger levels from 1 to 5. The taxonomy was based on rankings of bugs and vulnerabilities recorded in NIST's National Vulnerability Database (NVD) [40][2]. NVD provides scores that represent the innate characteristics of each vulnerability using the CVSS scoring system. Thus, each method has been assigned to the set representing the appropriate danger level. This was implemented with the following approach:

1. For each method, we checked the lowest and highest ratings of NVD vulnerabilities that use it.
2. The characteristics of the identified vulnerabilities were then inputed into the CVSS 3.0 scoring calculator [3], to compute the lowest and highest possible vulnerability scores.
3. Each method was then added in the set corresponding to the result of previous step. Methods detected in vulnerabilities with scores 7 or above were grouped in Set 5. Methods with score 6 to 7 in Set 4, those with score 5 to 6 in Set 3, those with score 4 to 5 in Set 2 and those with score 1 to 4 in Set 1.

Example: The `java.lang.Runtime.exec()` method [29] is widely-known to be used in many OS command injection exploits. NVD vulnerabilities that use this method have an impact rating in the range from 6.5 up to 10 out of 10. For the characteristics of these records, the CVSS scoring calculator outputted a rating of high (7) to very high (10). Thus, in PLATO's taxonomy the `System.exec()` method was classified in the very high (5/5) danger level.

[2] bugs were gathered from the NVD repository: https://web.nvd.nist.gov/view/vuln/search-advanced

[3] https://www.first.org/cvss/calculator/3.0

In the following paragraphs, we present the types of Java methods in PLATO's taxonomy. Tables 4 and 5 provide examples for each type of method. For the full taxonomy, the reader can access the link at the end of this article.

- **Input Vector Methods**

  According to [27], for tracing user input inside Java applications we can monitor only the execution of specific methods. PLATO takes into account 69 methods from those that accept data from users, streams or files [29]. Some of these methods are shown in Table 4 (the asterisk is a wild-card for all methods included in class).

java.io.BufferedReader.readLine()	java.io.BufferedInputStream.read()
java.io.ByteArrayInputStream.read()	java.io.DataInputStream.readByte()
java.lang.System.getenv()	java.io.StringReader.read()

Table 4: Example group - Input Vector Methods taxonomy

- **Exploitable Methods (sinks)**

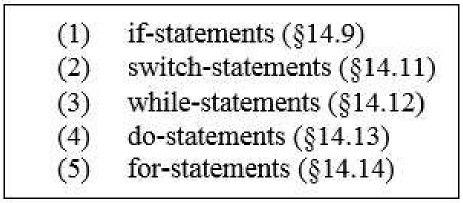
  Java methods that act as data sinks are known to be used in code exploits. Table 5 provides examples of such sink methods. All of them utilize directly some sort of data that can be considered tainted and, therefore, dangerous. PLATO takes into account 90 methods gathered from [14].

java.lang.Runtime.exec()	java.net.URLClassLoader (new URLClassLoader())
java.lang.System.load()	java.sql.Statement.executeQuery()
java.lang.System.setProperty()	javax.script.ScriptEngineeval()
java.io.File (new File())	java.net.Socket (new Socket())

Table 5: Example group - Sink methods taxonomy

Statements and methods for Reliability calculations

Reliability calculations take into consideration Java statements that affect the program's control flow. The statements shown in Figure 3 include boolean expressions that determine the control flow [27, 8].



(1) if-statements (§14.9)
(2) switch-statements (§14.11)
(3) while-statements (§14.12)
(4) do-statements (§14.13)
(5) for-statements (§14.14)

Figure 3: Example types of methods and statements included in PLATO's taxonomy

All such methods were gathered from the official Java documentation and are used for computing the Cyclomatic Density with the algorithm of Section 3.3.2.

4.4. Symbolic execution with PLATO's JPF listener

`PlatoListener` is the listener extension implemented in JPF to provide data and execution paths to PLATO. It monitors all executed program transitions, looking for invocations of `AssertionViolation`. Upon having detected one such invocation, `PlatoListener` saves the execution path leading to the assertion violation and compares it with all other saved paths to check the criterion of Definition 1. For every such detection, the values of the assertion variables are stored, along with the Java Class file and references to transitions executed prior to the violated assertion.

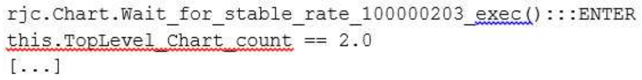
5. Experimental results

We are not aware of any stand-alone suite or revision(s) of software with a reported set of logical errors to use as a testing ground. Moreover, our experiments were restricted by the limitations of JPF's symbolic execution support. For this reason, we selected real applications with source code having various types of logical flaws, as well as "synthesized" applications. We present experimental results from: (1) software artifacts from the SIR Object Database [48] that have been previously used in controlled experiments on program analysis, (2) a test based on a suite by NIST [14] with logic flaws leading to exploits found in an application for handing field sensors in a SCADA system, (3) a test-bed application based on the source code of NIST's Juliet Test Case suite. Moreover, we present a correlation analysis with data from another 5 AUTs. First, we provide a motivating example, in order to introduce our experimentation approach to validate PLATO's effectiveness.

5.1. An example AUT

Let us consider an AUT with documented functionality.

***Profiling AUT's functionality in the form of dynamic invariants*.**
Example 1: Daikon observes the values of the variables while executing the program and reports invariants that hold true throughout all AUT executions. A dynamic invariant from Daikon in one of our tests is:



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

Figure 4: PLATO's processing flowchart

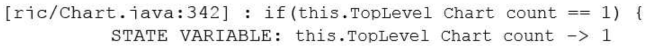
The dynamic invariant of Figure 4 shows that upon an invocation of `exec()`, the value of the variable `TopLevel_Chart_count` is equal to 2.

***Analyzing source code – invariant filtering and instrumentation*.**
Invariants are filtered and only those that refer to control locations or to paths with specific methods are kept (high Severity). Key points in code for invariant filtering are detected by PLATO's parser that returns the AUT's AST. Invariants are then parsed by their method type, variable or class type. The selected invariants are then instrumented as Java assertions in the AUT's source code.
Example 2. The dynamic invariant in Figure 4 is transformed into a Java assertion and it is then instrumented at the point indicated by Daikon (start of `Wait_for_stable_rate_100000203_exec()`).

```
assert this.TopLevel_Chart_count == 2.0;
```

Monitoring execution paths and program states.

JPF executes the instrumented source code and `PlatoListener` stores all assertion violations encountered along with the corresponding execution paths and backtracked states. In our example, we recall from Figure 4 that the invariant `this.TopLevel_Chart_count == 2.0` must be true each time the execution enters the method `Wait_for_stable_rate_ 100000203_exec()`. An example JPF-`PlatoListener` output is shown in Figure 5.



```
[rjc/Chart.java:342] : if(this.TopLevel Chart count == 1) {
           STATE VARIABLE: this.TopLevel Chart count -> 1
```

Figure 5: PLATO's processing flowchart

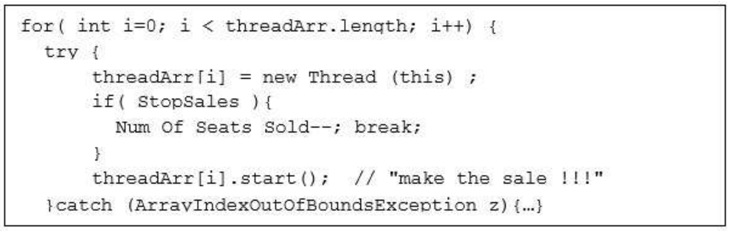
Example 3. Let us assume that Fig. 5 shows a path/transition at the beginning of the `Wait_for_stable_rate_100000203_exec()` method. The invariant refers to a variable used in an IF-Statement and it is therefore chosen for further analysis by PLATO's filtering system. It must be *true* every time the execution enters the `rjc.Chart.Wait_for_stable _rate_100000203_exec()` method. However, it is clear from the memory read in Fig. 5, that the invariant is violated, because `TopLevel_Chart _count` is equal to 1. An execution path with the same prefix was found to have `TopLevel_Chart_count = 2`. This contradiction is flagged as a possible logical error.

Classifying violations of dynamic invariants.

Example 4. The assertion in Example 2 was found to be both satisfied and violated in two execution paths with the same prefix. The information gain along the path to the invariant is then computed and the path is classified to Severity and Reliability ranks. These calculations ultimately assigned a Risk rating of 3.5 out of 5 (High Risk) for this finding.

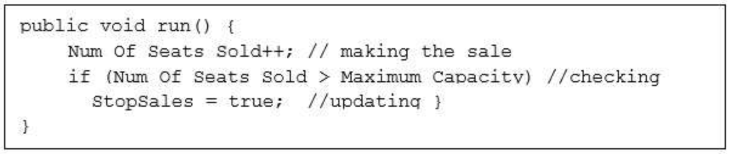
5.2. Experiment 1: airline test from the SIR Object Database

We selected an AUT from the SIR repository [48], which exhibits the characteristics of a multithreaded activity requiring arbitration. The experiment's AUT was a multi-threaded Java program for an airline to sell tickets. The logical error manifested leads to a race condition causing the airline application to sell more tickets than the available airplane seats. The fact that this is a known and well-documented error gave us the chance to validate the effectiveness of our method in detecting a subset of logical errors that produce race conditions. **The logical error**. When the program sells a ticket, it checks if the agents had previously sold all seats. If yes, the program stops processing additional transactions. Variable `StopSales` indicates that all tickets were sold and that issuing new tickets should be stopped. The logical error manifests when `StopSales` is updated by selling posts and, at the same time more tickets are sold by the running threads (agents). The AUT's code is shown in Figures 6 and 7.



```
for( int i=0; i < threadArr.length; i++) {
  try {
      threadArr[i] = new Thread (this) ;
      if( StopSales ){
        Num Of Seats Sold--; break;
      }
      threadArr[i].start();  // "make the sale !!!"
  }catch (ArrayIndexOutOfBoundsException z){…}
```

Figure 6: SIR AUT example code able to create i threads (agents) which sell tickets



```
public void run() {
     Num Of Seats Sold++; // making the sale
     if (Num Of Seats Sold > Maximum Capacity) //checking
       StopSales = true;  //updating }
}
```

Figure 7: Code: Make sale & check if limit was reached (update ”StopSales”)

PLATO's analysis for this test returned the output shown in Figure 8. The results obtained in each step of the workflow are:

Step 1-2. There was only one tested function point. Daikon inferred among other cases the following invariant:

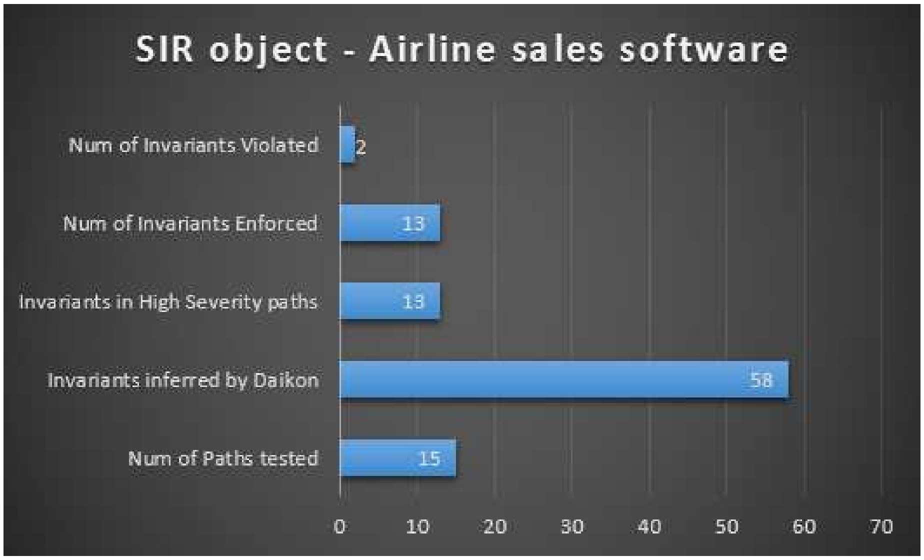


Figure 8: Airline sales: No of inferred invariants, chosen assertions and violations

```
Num_Of_Seats_Sold <= this.Maximum_Capacity
```

Step 3. An assertion violation was detected during the symbolic execution of the AUT for the method `runBug()`: two execution paths were found where the mentioned invariant was satisfied and violated respectively, thus implying a possible logical error.

Step 4. Our method classified the path in which the invariant assertion was violated with a Severity = 5 score and a Reliability = 3, thus yielding a total Risk value of 4.5.

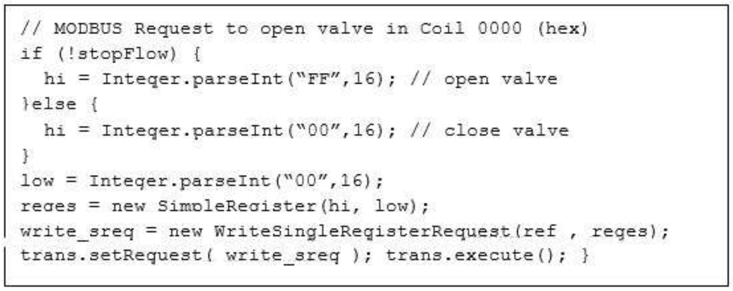
5.3. Experiment 2: remote SCADA RTUs (logical error CWE 840)

This experiment involved a type of logical error from NIST's Juliet Test Suite [14] classified as business logic vulnerabilities (CWE-840 group): vulnerabilities that depend on logical errors made by the programmer.

The programs in the Juliet Test Suite include two methods: a `bad()` method in which the logical error manifests and a `good()` method that implements a safe way of coding. We mark results as true positive, if there is an appropriate warning pinpointing the flawed (bad) method, or false positive if the tool flags the non-flawed (good) part of the source code.

The AUT was a real application built on top of the chosen synthetic test. The software can handle Remote Test Unit machinery (RTU) through SCADA systems in a sample critical infrastructure. Specifically, it can be used to open and close gas pipes according to pressure measurements, as well as to control pressure valves through an RTU unit and using its communication protocol. To communicate and control SCADA systems, the AUT uses a java library for communicating with SCADA equipment named JAMOD [49]. JAMOD applies the MODBUS protocol to connect software interfaces with RTUs in SCADA systems, using registers that manipulate control circuits (coils, holding registers etc.). In Figures 9 and 10 we provide excerpts from the AUT's source code. Real RTUSs can be handled, like the L75 RTU unit [50] that provides remote control actuation of quarter-turn valves and other rotary devices. The flaw manifests in the code controlling the RTU.

The tests were carried out using a MODBUS simulator named MOD_RSSIM [51], since acquiring a fully operational SCADA control was not feasible.



```
// MODBUS Request to open valve in Coil 0000 (hex)
if (!stopFlow) {
  hi = Integer.parseInt(“FF”,16); // open valve
}else {
  hi = Integer.parseInt(“00”,16); // close valve
}
low = Integer.parseInt(“00”,16);
reges = new SimpleRegister(hi, low);
write_sreq = new WriteSingleRegisterRequest(ref , reges);
trans.setRequest( write_sreq ); trans.execute(); }
```

Figure 9: High-level code able to manipulate a gas shaft in an L75 RTU unit

Software-to-RTU command code – an example.

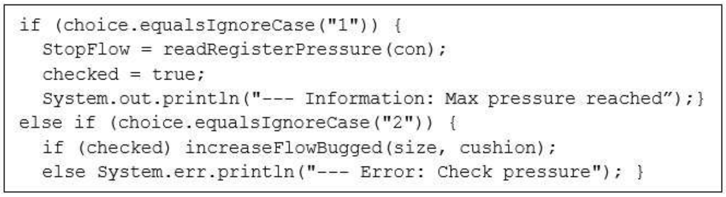
Let Coil 1 be the valve shaft motion control circuit. A write data command with message “FF00” either starts or continues valve movement, if it is already moving. A write data command with message "0000" will stop valve movement. Other data values will return an exception response [50]. Using this, the hexadecimal version of the MODBUS command is able to control and open gas pipes using the function "Write Coil" on bus 5 would be: 05, 05, 00, 00, FF, 00, 8D, BE.

The application tested in this case study works as follows: Each time an "open" (FF00) command is deployed, the application checks an RTU sensor to see if it can increase pipe pressure by opening one more valve without reaching the pressure limits. If pressure levels are too high, the AUT will send a "close" (0000) command and will not open another pipe. If pressure remains high, the AUT can send a second "close" command to reduce pressure. Figure 9 depicts the high-level methods of the AUT that are used to control the flow of a valve with the aforementioned commands. When variable `stopFlow` is false, an "open valve" request is sent, otherwise a "close valve".

The logical error.

The control system must not utilize more than a specific amount of gas pipes, otherwise high pressure could cause a rupture, leaks, even a failure to the entire distribution system. Each time the program sends an RTU request to open a valve, *it must check if the pressure in pipes has reached a maximum or not.* If true, it should stop the flow. Variable `stopFlow` controls this functionality. Inside the AUT's code, the logical error manifests at the part where the boolean variable `StopFlow` is updated. The software checks a pressure sensor to see if it should allow further increase in gas pressure or not. This check should take place each time a valve shaft is ordered to open. However, a higher number of pipes than the allowed maximum can be opened, by bypassing the aforementioned pressure check using alternate execution routes. This happens due to sensitive information that remains unclear in objects (i.e. there is a sequence of actions where variable `StopFlow` is not updated properly).

To better understand this execution deviation, refer to Figure 10. Due to an erroneous initialization of variable `checked`, the control-flow statement at line 6 (`if(checked)` can be bypassed if two consecutive “increase flow” invocations are sent within 3 to 4 seconds (i.e. before connection resets).



```
if (choice.equalsIgnoreCase("1")) {
  StopFlow = readRegisterPressure(con);
  checked = true;
  System.out.println("--- Information: Max pressure reached”);}
else if (choice.equalsIgnoreCase("2")) {
  if (checked) increaseFlowBugged(size, cushion);
  else System.err.println("--- Error: Check pressure"); }
```

Figure 10: Source code example - Checks imposed to handle pressure limits

This logical error is an error taken from [14] that was also classified in NIST’s vulnerability database [10].

We deployed the method presented in Section 4 to analyze the execution of the aforementioned hypothetical scenario, for handling pressure in pipes. Figure 11 depicts the invariant violations and overall statistics of the experiment.

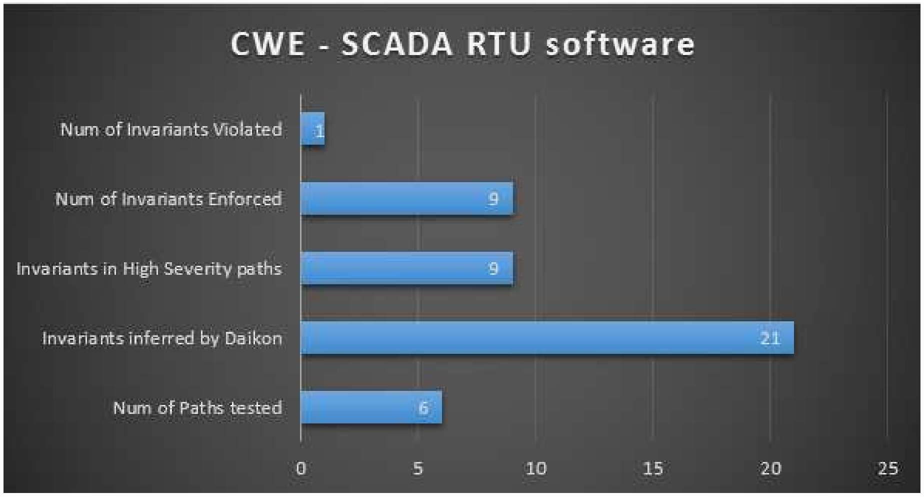


Figure 11: CWE 840-RTU: No of inferred invariants, chosen assertions and violations

The results for each step (concerning the flaw) were the following :

Step 1. The AUT’s functionality has two flows currently available:

A. Exec choice(1), Exec choice(2).

B. Exec choice(2), Exec choice(1), Exec choice(2).

Step 2. The dynamic analysis of these flows yielded 40 dynamic invariants. Amongst them, the following invariant refers to a hidden logical error:

```
Bug.readRegisterPressure():::ENTER
this.checked == false
```

meaning that, upon entering method `readRegisterPressure()`, the variable checked should always be FALSE.

Step 3. The dynamic invariants where instrumented as assertions inside the source code and the software was executed symbolically. An assertion violation was detected for method `readRegisterPressure()`: two execution paths were found in which the variable `checked` had a TRUE and FALSE value respectively, thus violating the invariant assertion from Step 2.

Step 4. Our tool classified this invariant with Severity = 5, due to the methods `sendRequest()` and `WriteSingleRegister` that manipulate SCADA commands, and Reliability = 2. The estimated total Risk value for this specific dynamic invariant was 3.5 (High Risk) .

5.4. Experiment 3: Testing the execution path classification system (Severity)

To test the proposed classification scheme, we created a test-bed application based on the source code of NIST's Juliet Test Case suite. For our purposes we created an test suite that is, essentially, an aggregation of multiple Juliet test filled with various vulnerabilities of different danger-levels; ranging from medium information leakage to serious OS execution injection. The test suite had both true positives and true negatives. The CWE vulnerability types that were manifested in the analyzed test suite were: CWE-840 (business logic errors), CWE-78 (OS Command Injection) and CWE-315 (Cleartext Storage of Sensitive Information in a Cookie).

Test scores and Information Gain output for dangerous methods detected in execution paths are provided in Table 6. We can see that PLATO's classification system yielded an overall Severity Rank = 3 out of 5 for the aggregated test suite,

Table 6: Classification of Juliet Test Suite experiment - Information Gain results on methods

Entire Source code - Prior Entropy	0.4021791902022729
Entire Source code - Prior Severity rank	3
Entropy Loss for println (Rank 2)	0.16229218908241483
Entropy Loss for readLine (Rank 4)	0.24229218908241482
Entropy Loss for addCookie (Rank 2)	0.16229218908241483
Entropy Loss for exec (Rank 4)	0.24229218908241482
Entropy Loss for getPassword (Rank 2)	0.16229218908241483
Entropy Loss for getUserName (Rank 2)	0.16229218908241483

but specific paths were ranked with a Severity Rank = 5 out of 5. It is interesting that the execution paths that scored the highest were manifesting the most dangerous vulnerability of all flaws present in the AUT (CWE-79, OS command injection). This can be seen from the fact that te most dangerous (Rank 4) methods detected in dangerous paths presented the highest Gain (0.2423) on both occasions, thus scoring higher than all the rest. In Table 6, the first highest-ranked method is the `exec()` (Rank 4) method which is a sink utilized for OS command injection exploits.

In overall, PLATO's Severity mechanism: (i) detected the paths that are prone to vulnerabilities, due to dangerous methods, and (ii) successfully ranked them according to the danger-level of the flaw present in their source. For a complete list of dangerous execution paths detected and their ranking, the reader is referred to Appendix A.

5.5. Data analysis and statistical comparison of findings

Our statistical comparison of analysis findings was based on an extensive dataset consisting of (i) additional case studies from NIST's Juliet Test suite - eight different AUTs in total, (ii) mutated source code drived from the processed AUTs, and (iii) error-free source code of the AUTs after having removed the bugs. The ultimate aim was to accumulate statistical evidence for potential connections between various analysis aspects, such as the extent to which the

Table 7: Data from logical error detection experiments-Part 1

METRICS / TESTS	NASA	SIR	RTU	CWE 226
Invariants Violated	1	2	1	1
Inv/s in High Severity paths	8	13	9	34
Total invariants inferred	120	58	21	36
loaded code	1274	1285	1053	1285
Instructions executed	4357	72431	7631	7631
Paths tested	6540	15	6	6
choice generators	1	1243	743	1

Table 8: Data from logical error detection experiments-Part 2

METRICS / TESTS	DART_SMART	Rational	RTU-safe	CWE78
Invariants Violated	0	1	0	2
Inv/s in High Severity paths	0	0	0	2
Total invariants inferred	24	85	21	12
loaded code	1094	1241	1053	1274
Instructions executed	5013	3613	7631	4357
Paths tested	74	1	6	6540
choice generators	1	27	743	1

code loaded affects the number of inferred invariants (scalability) and how the number of control-flow branches affects the number of invariant violations.

The original eight sample tests depicted in Tables 7 and 8 were used to develop a dataset of 25 experiments.

Data obtained were analyzed by evaluating the Correlation metric that shows how much two variables share similar changes in value range. Correlation results for two variables X and Z can indicate that they are positively correlated, negatively correlated, or not correlated at all. We do not to use the Covariance metric, because Covariances are hard to compare when they are calculated for variables with different value scales. Instead, Correlation is a scaled version of Covariance which normalizes the Covariance and ends up with values between

Table 9: Correlation results on the produced dataset.

Correlation	Invariants Violated	Invariants in High Severity paths	Total Invariants inferred
Invariants Violated	1		
Inv/s in High Severity paths	0,1	1	
Total invariants inferred	0	0,01	1
loaded code	0,5	0,4	0,5
Instructions executed	0,5	0,2	0,07
Paths tested	0,3	-0,1	0,3
choice generators	0,4	0	-0,2

-1 and 1, no matter what are the original unit values of variables.

As shown in Table 9, the dataset consists of seven variables, with three of them used for Correlation analysis. Invariant violations seem to be connected with the choices in execution flow and thus, indirectly, with the instructions in these paths. The logical errors and relevant invariant violations seem to be slightly correlated with the amount of tested paths. While more paths may lead to more invariant violations, the well coded programs can still have numerous paths without any violated invariant. On the other hand, bad coding can lead to invariant violation even with just a handful of execution paths. Violated invariants have some correlation with the number of inferred high-severity invariants, but the high severity invariants are correlated only with the amount of code loaded and with the number of instructions executed per AUT. The latter happens because invariants are considered of high severity only when they manifest in paths with dangerous method invocations according to PLATO's taxonomy.

It is important to note that with model checking or symbolic execution alone, it would be not possible to detect the injected logical errors. This is so, because the semantic differences in the traversed paths were transparent to JPF's symbolic execution.

6. Discussion and concluding remarks

6.1. Method applicability and state explosion issues

PLATO can detect logical errors manifested as code vulnerabilities that are recorded in NIST's vulnerability suites. The applicability of the method depends on how thoroughly are analyzed the input vectors and dynamic invariants of an AUT. In our tests, the rate of successful logical error detections was close to 100% success, but the sample on which PLATO was tested is still relatively small, in order to support such a high detection rate.

To sidestep the problem of state explosion, Daikon and JPF can target only specific methods, whereas the Severity ranking can pinpoint the really dangerous methods. As shown in Table 10, for a given experiment it was made possible, using method paths instead of comparing invariants with all program states, to reduce the initial data set (Daikon traces, states and paths) from 155 MB to only 13MB, and to speed up the analysis by 80%.

Table 10: Execution times for PLATO RJC experiment

	Full paths and states	Method paths and assertions
Size	155 MB	13MB
Time elapsed	18 min (RJC)	4 min (RJC)

6.2. Advantages and Limitations

PLATO depends on the soundness of the likely dynamic invariants provided by Daikon and one of its limitations is the need for data from execution scenarios. However, this is an inherent problem of all heuristic approaches that rely on repetitive observations to form rules. On the other hand, if we want to automatically detect flaws related to the logic of AUTs, the heuristic analysis implemented in Daikon is the only way to model their intended functionality.

The second limitation is the targeted analysis of the functionality of AUTs. If PLATO would have to analyze AUTs of thousands of lines in entirety, problems would arise, mostly due to JPF's inability to handle large, complex applications and also due to state explosion. This can be mitigated by breaking down large applications to separate functionalities to analyze each of them separately.

From our experience with using PLATO in practice, we end up with the following conclusions:

- PLATO can indeed detect logical errors in AUTs of reasonable size and complexity.

- The experiments have shown that our method can also provide valid detections for other types of flaws beyond logical errors, such as the detected race condition.

- Logical errors can be manifested in very diverse contexts and the detection of a priori known error types (e.g. race conditions), which are then classified as logical errors, is not a reliable approach. With PLATO's deductive approach, we do not only detect diverse logical errors, but the tool also provides insight on the impact of each error, which is context-specific.

References

[1] Codepro (2015).
URL https://developers.google.com/java-dev-tools/codepro/doc/

[2] Ucdetector (2015).
URL http://www.ucdetector.org/

[3] Pmd (2015).
URL http://pmd.sourceforge.net/

[4] D. Hovemeyer, W. Pugh, Finding bugs is easy, ACM Sigplan Notices 39 (12) (2004) 92–106.

[5] Coverity save audit tool (2015).
URL `http://www.coverity.com`

[6] C. S. Păsăreanu, W. Visser, Verification of java programs using symbolic execution and invariant generation, in: Model Checking Software, Springer, 2004, pp. 164–181.

[7] The java pathfinder tool (2015).
URL `http://babelfish.arc.nasa.gov/trac/jpf/`

[8] V. Felmetsger, L. Cavedon, C. Kruegel, G. Vigna, Toward automated detection of logic vulnerabilities in web applications, in: USENIX Security Symposium, 2010, pp. 143–160.

[9] M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, C. Xiao, The daikon system for dynamic detection of likely invariants, Science of Computer Programming 69 (1) (2007) 35–45.

[10] The common weakness enumeration (cwe) community (2015).
URL `https://cwe.mitre.org/`

[11] G. Stergiopoulos, B. Tsoumas, D. Gritzalis, Hunting application-level logical errors, in: Engineering Secure Software and Systems, Springer, 2012, pp. 135–142.

[12] G. Stergiopoulos, B. Tsoumas, D. Gritzalis, On business logic vulnerabilities hunting: The app_loggic framework, in: Network and System Security, Springer, 2013, pp. 236–249.

[13] G. Stergiopoulos, P. Katsaros, D. Gritzalis, Automated detection of logical errors in programs, in: Proc. of the 9th International Conference on Risks & Security of Internet and Systems, 2014.

[14] T. Boland, P. E. Black, Juliet 1.1 c/c++ and java test suite, Computer (10) (2012) 88–90.

[15] G. Stergiopoulos, P. Katsaros, D. Gritzalis, T. Apostolopoulos, Combining invariant violation with execution path classification for detecting multiple types of logical errors and race conditions, in: Proceedings of the 13th International Joint Conference on e-Business and Telecommunications, 2016, pp. 28–40. doi:10.5220/0005947200280040.

[16] A. Zeller, Isolating cause-effect chains from computer programs, in: Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering, ACM, 2002, pp. 1–10.

[17] M. Weiser, Program slicing, in: Proceedings of the 5th international conference on Software engineering, IEEE Press, 1981, pp. 439–449.

[18] X. Zhang, N. Gupta, R. Gupta, Pruning dynamic slices with confidence, in: ACM SIGPLAN Notices, Vol. 41, ACM, 2006, pp. 169–180.

[19] G. K. Baah, Statistical causal analysis for fault localization.

[20] P. Godefroid, N. Klarlund, K. Sen, Dart: Directed automated random testing, SIGPLAN Not. 40 (6) (2005) 213–223. doi:10.1145/1064978.1065036.
URL http://doi.acm.org/10.1145/1064978.1065036

[21] The daikon invariant detector manual (2015).
URL http://groups.csail.mit.edu/pag/daikon/

[22] L. Zhang, G. Yang, N. Rungta, S. Person, S. Khurshid, Feedback-driven dynamic invariant discovery, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis, ISSTA 2014, ACM, New York, NY, USA, 2014, pp. 362–372. doi:10.1145/2610384.2610389.
URL http://doi.acm.org/10.1145/2610384.2610389

[23] P. Zielczynski, Traceability from use cases to test cases.
URL http://www.ibm.com/developerworks/rational/library/04/r-3217/

[24] R. Jhala, R. Majumdar, Software model checking, ACM Comput. Surv. 41 (4) (2009) 21:1–21:54. doi:10.1145/1592434.1592438.
URL http://doi.acm.org/10.1145/1592434.1592438

[25] G. Stoneburner, A. Goguen, A. Feringa, Risk management guide for information technology systems, Nist special publication 800 (30) (2002) 800–30.

[26] R. A. Martin, S. Barnum, Common weakness enumeration (cwe) status update, ACM SIGAda Ada Letters 28 (1) (2008) 88–91.

[27] E. R. Harold, Java I/O, " O'Reilly Media, Inc.", 2006.

[28] J. Gosling, B. Joy, G. L. Steele Jr, G. Bracha, A. Buckley, The Java Language Specification, Pearson Education, 2014.

[29] Java platform, standard edition 7 api specification (2015).
URL http://docs.oracle.com/javase/7/docs/api/

[30] C. S. Păsăreanu, P. C. Mehlitz, D. H. Bushnell, K. Gundy-Burlet, M. Lowry, S. Person, M. Pape, Combining unit-level symbolic execution and system-level concrete execution for testing nasa software, in: Proceedings of the 2008 International Symposium on Software Testing and Analysis, ISSTA '08, ACM, New York, NY, USA, 2008, pp. 15–26. doi:10.1145/1390630.1390635.
URL http://doi.acm.org/10.1145/1390630.1390635

[31] C. Prud'homme, J.-G. Fages, X. Lorca, Choco3 Documentation, TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S. (2014).
URL http://www.choco-solver.org

[32] Nasa jpf symbolic pathfinder – tool documentation, [online] http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/

jpf-symbc/doc (2015).
URL http://babelfish.arc.nasa.gov/trac/jpf/wiki/projects/jpf-symbc/doc

[33] I. Perfilieva, J. Močkoř, Mathematical principles of fuzzy logic, Springer Science & Business Media, 1999.

[34] G. Albaum, The likert scale revisited, Journal-Market research society 39 (1997) 331–348.

[35] N. A. Abramson, Introduction to Information Theory and Coding, McGraw Hill, 1964.

[36] L. H. Etzkorn, C. G. Davis, Automatically identifying reusable oo legacy code, Computer 30 (10) (1997) 66–71.

[37] Y. Yang, J. O. Pedersen, A comparative study on feature selection in text categorization, in: ICML, Vol. 97, 1997, pp. 412–420.

[38] E. J. Glover, G. W. Flake, S. Lawrence, W. P. Birmingham, A. Kruger, C. L. Giles, D. M. Pennock, Improving category specific web search by learning query modifications, in: Applications and the Internet, 2001. Proceedings. 2001 Symposium on, IEEE, 2001, pp. 23–32.

[39] S. Ugurel, R. Krovetz, C. L. Giles, What's the code?: automatic classification of source code archives, in: Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, 2002, pp. 632–638.

[40] National vulnerability database, [online] http://nvd.nist.gov (2015).
URL http://nvd.nist.gov

[41] P. Chhabra, L. Bansal, An effective implementation of improved halstead metrics for software parameters analysis (2014).

[42] M. Bray, K. Brune, D. A. Fisher, J. Foreman, M. Gerken, C4 software technology reference guide-a prototype., Tech. rep., DTIC Document (1997).

[43] W. J. Hansen, Measurement of program complexity by the pair:(cyclomatic number, operator count), ACM SIGPLan Notices 13 (3) (1978) 29–33.

[44] L. Rosenberg, T. Hammer, Metrics for quality assurance and risk assessment, Proc. Eleventh International Software Quality Week, San Francisco, CA.

[45] Using code quality metrics in management of outsourced development and maintenance (2015).
URL http://www.mccabe.com/pdf/McCabeCodeQualityMetrics-OutsourcedDev.pdf

[46] W. Van Leekwijck, E. E. Kerre, Defuzzification: criteria and classification, Fuzzy sets and systems 108 (2) (1999) 159–178.

[47] P. Cingolani, J. Alcala-Fdez, jfuzzylogic: a robust and flexible fuzzy-logic inference system language implementation., in: FUZZ-IEEE, Citeseer, 2012, pp. 1–8.

[48] G. Rothermel, S. Elbaum, A. Kinneer, H. Do, Software-artifact infrastructure repository (2006).

[49] W. D., Jamod java modbus implementation, [online] http://jamod.sourceforge.net/ (2015).
URL http://jamod.sourceforge.net/

[50] Flowserve l75 series electric actuator, fcd lmaim7502-00, [online] http://jamod.sourceforge.net/ (July 2005).
URL http://jamod.sourceforge.net/

[51] Plcsimulator modbus plc simulator, [online] http://www.plcsimulator.org/ (2014).
URL http://www.plcsimulator.org/

Appendix A.

Output for the Experiment 4 concerning the Severity classification mechanism is depicted below:

```
AUT Prior Entropy = 0.4021791902022729,
AUT Source code Severity Prior overall rank: 3
////// Information Gain calculation //////
Information Gain calculation duration: 3 milliseconds
Information Gain for getInputStream in Rank 0: 0.0
Information Gain for getInputStream in Rank 1: 0.0
Information Gain for getInputStream in Rank 2: 0.0
Information Gain for getInputStream in Rank 3: 0.0
Information Gain for getInputStream in Rank 4: 0.0
Information Gain for println in Rank 0: 0.0
Information Gain for println in Rank 1: 0.0
Information Gain for println in Rank 2: 0.16229218908241483
Information Gain for println in Rank 3: 0.0
Information Gain for println in Rank 4: 0.0024051879625567596
Information Gain for readLine in Rank 0: 0.0
Information Gain for readLine in Rank 1: 0.0
Information Gain for readLine in Rank 2: 0.004915013910784527
Information Gain for readLine in Rank 3: 0.0
Information Gain for readLine in Rank 4: 0.24229218908241482
Information Gain for getProperty in Rank 0: 0.0
Information Gain for getProperty in Rank 1: 0.0
Information Gain for getProperty in Rank 2: 0.0
Information Gain for getProperty in Rank 3: 0.0
Information Gain for getProperty in Rank 4: 0.0
Information Gain for write in Rank 0: 0.0
Information Gain for write in Rank 1: 0.0
Information Gain for write in Rank 2: 0.0
Information Gain for write in Rank 3: 0.0
Information Gain for write in Rank 4: 0.0
Information Gain for addCookie in Rank 0: 0.0
Information Gain for addCookie in Rank 1: 0.0
Information Gain for addCookie in Rank 2: 0.16229218908241483
Information Gain for addCookie in Rank 3: 0.0
Information Gain for addCookie in Rank 4: 0.0024051879625567596
Information Gain for exec in Rank 0: 0.0
Information Gain for exec in Rank 1: 0.0
Information Gain for exec in Rank 2: 0.004915013910784527
Information Gain for exec in Rank 3: 0.0
Information Gain for exec in Rank 4: 0.24229218908241482
Information Gain for load in Rank 0: 0.0
Information Gain for load in Rank 1: 0.0
Information Gain for load in Rank 2: 0.0
Information Gain for load in Rank 3: 0.0
Information Gain for load in Rank 4: 0.0
Information Gain for getPassword in Rank 0: 0.0
Information Gain for getPassword in Rank 1: 0.0
Information Gain for getPassword in Rank 2: 0.16229218908241483
Information Gain for getPassword in Rank 3: 0.0
Information Gain for getPassword in Rank 4: 0.0024051879625567596
Information Gain for nextInt in Rank 0: 0.0
Information Gain for nextInt in Rank 1: 0.0
Information Gain for nextInt in Rank 2: 0.0
Information Gain for nextInt in Rank 3: 0.0
Information Gain for nextInt in Rank 4: 0.0
Information Gain for getUserName in Rank 0: 0.0
Information Gain for getUserName in Rank 1: 0.0
Information Gain for getUserName in Rank 2: 0.16229218908241483
Information Gain for getUserName in Rank 3: 0.0
Information Gain for getUserName in Rank 4: 0.0024051879625567596

////// Execution paths //////

Execution path detected
Input Ranking: 4
Sink Ranking: 5
Starting at line: 83
Source: readLine
Input into variable: data bad
Ending at line: 135
Sink method: exec
Execution arguments:
Severity Rank: 5
Lines that are executed: 135 83
Dangerous variables: data bad,

Execution path detected
Input Ranking: 4
Sink Ranking: 3
Starting at line: 55
Source: getPassword
Input into variable: data2 bad
Ending at line: 68
Sink method: addCookie
Execution arguments:
Severity Rank: 3
Lines that are executed: 68 55
Dangerous variables: data2 bad,
```

Execution path detected

Input Ranking: 4

Sink Ranking: 3

Starting at line: 37

Source: getProperty

Input into variable: dRata bad

Ending at line: 91

Sink method: println

Execution arguments:

Severity Rank: 3

Lines that are executed: 91

Dangerous variables: dRata bad,