# Combining Invariant Violation with Execution Path Classification for Detecting Multiple Types of Logical Errors and Race Conditions

George Stergiopoulos[1], Panayiotis Katsaros[2], Dimitris Gritzalis[1] and Theodore Apostolopoulos[1]
[1]*Information Security & Critical Infrastructure Protection Laboratory, Dept. of Informatics, Athens University of Economics & Business, 76 Patission Ave., GR-10434, Athens, Greece*
[2]*Dept. of Informatics, Aristotle University of Thessaloniki, Thessaloniki, Greece*

Keywords: Code Classification, Logical Errors, Dynamic Invariants, Source Code, Execution Path, Assertions, Vulnerability, Exploit, Automatic, Analysis, Information Gain, Fuzzy Logic.

Abstract: *Context:* Modern automated source code analysis techniques can be very successful in detecting a priori defined defect patterns and security vulnerabilities. Yet, they cannot detect flaws that manifest due to erroneous translation of the software's functional requirements into the source code. The automated detection of *logical errors* that are attributed to a faulty implementation of applications' functionality, is a relatively uncharted territory. In previous research, we proposed a combination of automated analyses for logical error detection. In this paper, we develop a novel business-logic oriented method able to filter mathematical depictions of software logic in order to augment logical error detection, eliminate previous limitations in analysis and provide a formal tested logical error detection classification without subjective discrepancies. As a proof of concept, our method has been implemented in a prototype tool called PLATO that can detect various types of logical errors. Potential logical errors are thus detected that are ranked using a *fuzzy logic system* with two scales characterizing their impact: (i) a *Severity scale*, based on the execution paths' characteristics and *Information Gain*, (ii) a *Reliability scale*, based on the measured program's *Computational Density*. The method's effectiveness is shown using diverse experiments. Albeit not without restrictions, the proposed automated analysis seems able to detect a wide variety of logical errors, while at the same time limiting the false positives.



## 1 INTRODUCTION

The sum of all functional requirements of an application reflect the intended program behavior; that is, what the programmer wants his code to do and what not to do. During software development, functional requirements are translated into source code. A *software error or fault* is the difference between a computed, observed, or measured value and the true, specified or theoretically correct value or condition inside the software code (Peng and Wallace, 1993). A *(software) vulnerability* is a weakness in a system or application that is subject to exploitation or misuse (Scarfone et al., 2008). It is also defined as a mistake in software that can be leveraged to gain access, violate a reasonable security policy or force software to exhibit unintended behavior (CVE, 2015).

Research on automated detection of software errors and vulnerabilities has mainly focused on static analysis and software model checking techniques that are effective in detecting a priori specified errors (e.g. Time Of Check - Time Of Use errors, null pointer dereferences etc.), bad coding patterns and some types of exploitable vulnerabilities (e.g. unsanitized input data, buffer overflows etc.). Yet, errors related to the intended program functionality, which are broadly called *logical errors*, are not a priori known. In a code auditing process, they cannot be analyzed as pattern-specific errors since they are rather application-specific. At the level of the program's execution flow, these errors will cause execution diversions that manifest as unintended program behaviour (Felmetsger et al., 2010).

Since logical errors in an *Application under Test* (AUT) are essentially execution deviations from its intended functionality, their automated detection needs to be based on some model of the AUT's operational logic. Such a model can be inferred in the form of likely invariants from the dynamic analysis of official executions of the AUT's functionality (i.e. execution of scenarios). *Dynamic invariants* are properties that are likely true at a certain point or points of the program and, in effect, reveal information about the goal behaviour, the particular imple-



Stergiopoulos, G., Katsaros, P., Gritzalis, D. and Apostolopoulos, T.
Combining Invariant Violation with Execution Path Classification for Detecting Multiple Types of Logical Errors and Race Conditions.
DOI: 10.5220/0005947200280040
In *Proceedings of the 13th International Joint Conference on e-Business and Telecommunications (ICETE 2016) - Volume 4: SECRYPT*, pages 28-40
ISBN: 978-989-758-196-0

mentation and the environment (inputs) under which the program runs (Ernst et al., 2007). Our method for the automated detection of logical errors extends previous research (Felmetsger et al., 2010) (Stergiopoulos et al., 2012) (Stergiopoulos et al., 2013)(Stergiopoulos et al., 2014) (Stergiopoulos et al., 2015b) (Stergiopoulos et al., 2015a) by combining methods utilized in vulnerability detection, albeit not for logical errors. The same combination of tools was used in the aforementioned articles, but there are important differences and at the end, only basic concepts from those works are kept. In PLATO, dynamic invariants are evaluated using different techniques. The tool is now capable of analyzing the full range of instrumented invariants, while keeping spurious invariants to a minimum using a new classification system that uses the Information Gain algorithm. Its present version implements two new formal classifiers, which replace the previously used empirical, text-based rules. Classification functions are trained using data collections of known code vulnerabilities from the National Institute of Standards and Technology (NIST) to classify source code paths using information gain algorithms.

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

1. We show how most types of information flow dependent logical errors can be detected by classifying invariant violations and their corresponding execution paths based on information gain. Dangerous source code methods recorded by major databases are used as indicators of risk, according to their appearance in real-world vulnerabilities. PLATO's logical error detections are classified in two different groups of sets as follows:
   - the Severity sets, quantifying the danger level of an execution path $\pi$ (the impact of an error, if it were to manifest on path $\pi$ during execution). Severity is based on an algorithm which uses Information Gain for classification from data mining.
   - the Reliability sets, quantifying a path $\pi$ with an invariant violation based on the size and complexity of the code traversed by path $\pi$.
2. To test the diversity of errors that can be detected, we develop and evaluate the PLATO tool on different AUTs containing logical errors that manifest different types of vulnerabilities: (i) A multithreaded airline control ticketing system; previously used in a controlled experimentation with program analysis. (ii) An aggregated AUT testbed that aims to evaluate PLATO's Severity classification system using multiple vulnerable code examples from NIST's source code vulnerability suite (Boland and Black, 2012). The choice of the experimental scenarios was based on analyzing real-world source code containing the most common types of logical errors recorded in software development, according to (Martin and Barnum, 2008) and specifically (cwe, 2016).

## 2 RELATED WORK

Recent developments in debugging techniques also focus on the detection of logical errors, but they do not aim to a fully automated program analysis. Delta debugging (Zeller, 2002) is a state-altering technique that systematically narrows the difference between the states of a failed program run from the states of a failure-free run, down to a small set of variables. The intuition is that any difference between the two execution paths could be the failure cause. Predicate switching is a variant of delta debugging (Zhang et al., 2006) that alters predicate truth values during of a program execution. Given a failing execution, the goal is to find the predicate that, if switched from false to true or the opposite, it causes the program to execute successfully. A limitation of state-altering techniques is that they do not address the problem of semantic consistency; there is no guarantee that by altering a state the new execution path will still be a valid program run (Baah, 2012). A second limitation is the usability of this technique, since the program has to re-run after every single state alternation. In our approach for detecting logical errors, state alternation is avoided through the use of dynamic invariants along with a one-time symbolic execution of the AUT.

In (Doupé et al., 2011), the authors focus exclusively on the detection of specific flaws found in web applications, whereas in (Balzarotti et al., 2007) web applications are analyzed for multi-module vulnerabilities using a combination of analysis techniques. However, both works do not address the problem of profiling the source code behavior or detecting logical errors per se.

In (Godefroid et al., 2005), authors present DART for automatically testing software that combines (1) automated extraction of interface using static source-code parsing, (2) automatic generation of a test driver to perform random testing and simulate a general environment and (3) dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to direct execution along alternative program paths. DART detects errors such as program crashes, assertion violations, and non-

termination. Although detections from DART and from our approach will certainly occasionally overlap, still DART cannot detect logical flaws that do not lead to one of the aforementioned errors (e.g. a program crash). If AUT execution terminates normally, DART cannot understand semantic differences in functionality during similar executions. Our approach utilizes a basic notion and theory of this paper, namely the fact that "directed search usually provides much better code coverage than a simple random search" (Godefroid et al., 2005). Our approach uses directed dynamic monitoring of executions to provide functionality coverage and extract dynamic invariants that can adequately describe functionality. Certain types of logical errors in web applications can be detected with the approach discussed in (Felmetsger et al., 2010). A set of likely invariants that characterize the execution of the AUTs is inferred using the Daikon tool (Ernst et al., 2007)(dai, 2015). The Daikon results are then used in JPF (Păsăreanu and Visser, 2004)(jpf, 2015) to model-check the behavior of the AUT over symbolic input. Our approach can be applied on any type of standalone application (even GUI applications), with no predefined mappings of inputs, which can range over infinite domains (in (Felmetsger et al., 2010), analysis is restricted to a web.xml file). To cope with this difference, input vectors and information flows are derived by monitoring user executions. Variants of our method were presented previously in (Stergiopoulos et al., 2012) and (Stergiopoulos et al., 2013). In (Stergiopoulos et al., 2012), we specifically targeted logical errors in GUI applications. We described a preliminary deployment of a Fuzzy Logic ranking system to mitigate the possibility of false positives and we applied the method on lab test-beds. In (Stergiopoulos et al., 2013), the Fuzzy Logic ranking system was formally defined and further developed. In this work, the classification mechanism that was proposed and evolved in (Stergiopoulos et al., 2012), (Stergiopoulos et al., 2013) and in (Stergiopoulos et al., 2014) changes to a different approach that experiments indicate to be capable to analyze real-world applications instead of test-beds and simple GUI AUTs while limiting subjectivity in detection classification, due to formal classification mechanisms and different invariant filtering techniques than the previous approaches. Specifically, the current classification method is based on well-known data mining techniques for source code classification ((Ugurel et al., 2002)), trained upon a internationally accepted dataset of example vulnerabilities (NIST's Juliet Suite (Boland and Black, 2012)) for dangerous source code and corresponding inferred invariants. Previous methodologies followed (Felmetsger et al., 2010) and contributed an empirical classification mechanism and test variations. (Stergiopoulos et al., 2015b) was the first publication to detect logical errors in real-world SCADA high-level software over the MODBUS protocol, but neither its classification system used any formal method, nor tests were thorough enough to adequately present a detection range.

# 3 ANALYSIS BUILDING BLOCKS

In this section, the main building blocks of PLATO's methodology are described, namely: (i) how the behaviour of an AUT is modeled using likely dynamic invariants, (ii) how the obtained likely invariants are verified through symbolically executing the AUT and (iii) how the results are classified using fuzzy logic to measure the impact and the size/complexity of the affected code, for each detection.

## 3.1 Overview

In this subsection, we will walk readers through key intuitions of the presented methodology before diving deeper into more technical aspects. Generally, the entire method is comprised of three steps:

- Software-under-Test is executed while Daikon's agent is monitoring its memory and code. Daikon then produces logical rules for variables called invariants the can describe software business logic.
- Invariants are filtered by PLATO based on a data classification algorithm that utilizes a formal mathematical technique called *Information Gain* to keep only invariants that refer to "dangerous" aspects of the software-under-test business logic. PLATO discards the rest. This *Severity* method for filtering invariants according to machine learning and data classification algorithms is the biggest novelty of this publication.
- Selected invariants are then inserted inside the software-under-test source code in the form of code assertions.
- NASA's JPF symbolically executes the newly instrumented code, trying to traverse as many valid execution paths as possible, while a listener is checking for invariant assertion violations or enforcements (i.e. if they hold true or not in numerous execution flows).
- If same invariant is found to be both enforced and violated in different versions of the same sub-execution flow, PLATO flags this event as a logical error detection.

### 3.2 Dynamic Invariants for Profiling the Behavior of the Source Code

The functionality of an AUT is captured in the form of dynamic invariants, generated by the Daikon tool (Ernst et al., 2007)(dai, 2015). These invariants are logical rules for variables (e.g. `p!=null` or `var=="string"`) that hold true at certain point(s) of a program in all monitored executions. As far as PLATOs tests is concerned, Daikons monitoring is a type of functional testing. Functionality test suites aim to verify that the AUT behaves correctly from a business perspective and functions according to its business requirements. A business requirement is "a condition or capability to which a system must conform" (Zielczynski, 2006). It is a specific business behaviour of an application as observed by a user. Functional test cases are used to validate the way an AUT performs in accordance to those requirements.

The generated dynamic invariants can reflect *the intended functionality of the AUT*, if they are derived from monitored executions of representative use-case scenarios. To achieve adequate coverage during functional testing, we adopt two typical rules of thumb:

First, we require a test case for each possible flow of events inside a use case (this corresponds to a diagram path in a UML use case diagram (Zielczynski, 2006)).

Second, we test as many variations (i.e. combinations of input) of each test case as possible (Zielczynski, 2006); i.e. we "fuzz" the UML's input data to cover multiple input scenarios. Experience has showed that no hardcore fuzzing is needed here, just indicative data input cases. These input variations are hidden in the statements or conditions that guide actions and activities in the AUT (business rules).

The validity of the inferred dynamic invariants (i.e. the inferred program behaviour) is tested against as many execution paths of the AUT as possible, using symbolic execution of the AUT. Intuitively, if there is an execution path, which violates a (combination of) dynamic invariant(s), then a logical error may exist, which affects the variable(s) referred in the invariant.

### 3.3 Symbolic Execution for Verifying Dynamic Invariants

PLATO converts the likely invariants into Java assertions and instruments them into the source code. For example, let us consider that the invariant `p!=null` holds true when the execution flow enters a method. In this case, PLATO creates the assertion `[assert (p!=null);]` and instruments it at the beginning of that method, just before any other method execution. Likely dynamic invariants are filtered according to two filtering criteria: invariants concerning variables which affect the execution flow and invariants related to source code methods which are tied to known application vulnerabilities (Martin and Barnum, 2008)(Harold, 2006). For the former, we particularly focus on the conditional expressions in branches and loops. The latter is implemented by using a taxonomy that classifies source code methods according to their danger level. This taxonomy is embedded in PLATO and is based on the taxonomies presented in (Martin and Barnum, 2008), the Oracle's Java Taxonomy (Gosling et al., 2014)(jap, 2015) and reports from code audits (Hovemeyer and Pugh, 2004). More information on this taxonomy is provided in Section 4.2.1 which covers technical details.

Daikon's invariants are then cross-checked with a set of finite execution paths and their variable valuations for each tested path. For this purpose, PLATO obviously needs execution paths that adequately cover the functionality of the AUT. PLATO leverages NASA's JPF tool to execute the AUT symbolically. Specifically, we developed an extension listener for Java Symbolic PathFinder's (SPF) to collaborate with PLATO, named PlatoListener. SPF symbolically executes Java byte-code programs (jpf, 2015). One of its main features is automated generation of test inputs to explore a high number of different execution paths of an AUT. Our PlatoListener realized a listener extension able to monitor AUT states and paths during SPF's constraint solving and path traversal. This way it managed to evaluate invariant assertions as instrumented and executed through the model checker.

### 3.4 Fuzzy Logic Classification of Detections

It is not true that all the logical errors can divert the programs' execution to exploitable states and that they have comparable impact on the functionality of an AUT. Thus, similarly to a code auditor's reasoning, PLATO classifies detections using a fuzzy set theory approach combined with two advanced classification functions. Every assertion violation along with a execution path are classified into two different groups of sets:

- the *Severity* sets, which quantify the danger level of the execution path, i.e. the impact that an exploitable error would have, if it would be manifested on that path;
- the *Reliability* sets, which quantify the overall reliability of an execution path based on the size and

the complexity of the code traversed in it (a code metric is used named Cyclomatic Density).

With this fuzzy logic approach, we also aim to confront two inherent problems in automated logical error detection: the large data sets of the processed AUT execution paths. PLATO helps the code auditor to focus only to those path transitions that appear having high ratings in the classification system.

#### 3.4.1 Severity

For an execution path $\pi$, *Severity*($\pi$) measures $\pi$'s membership degree in a Severity fuzzy set that reflects how dangerous is a flaw if it were to manifest in path $\pi$, i.e. its relative *impact*. Execution path $\pi$ is weighted based on how its transitions and corresponding executed source code methods affect the program's execution: if there are transitions in the path that are known to manifest exploitable behaviour, then $\pi$ is considered dangerous and is assigned higher Severity ranks.

**Definition 1.** Given the execution path $\pi$, we define

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

to measure the severity of $\pi$ on a Likert-type scale from 1 to 5.

Likert scales are a convenient way to quantify facts (Albaum, 1997) that, in our case, refer to a program's flow. If an exploitable behaviour were to manifest in an execution path, the scale-range captures the intensity of its impact in the program's control flow. In order to weight paths, Severity is based on the Statistical Information Gain, a measure used to classify execution paths in one out of five Severity categories that are ranked from one to five. Categories are then grouped into Fuzzy Logic sets using labels: *high* severity (4-5), *medium* (3) or *low* (1 or 2).

#### 3.4.2 Measuring Severity of Execution Paths using its Statistical Information Gain

Our Severity classification approach is based on the *Expected Information Gain* (aka Expected Entropy Loss) statistical measure (Abramson, 1964) that has been successful in feature selection for information retrieval (Etzkorn and Davis, 1997). Information Gain has been used before by Glover et al. (Glover et al., 2001) and Ugurel et al. (Ugurel et al., 2002) for classifying source code. Here, we use it to classify execution paths and their corresponding source code methods into danger levels.

To measure the Expected Information Gain of an execution path, we need characteristics (features) to look for. PLATO uses a taxonomy of dangerous source code methods. These methods are recorded to be tied to known vulnerability types (Martin and Barnum, 2008),(nvd, 2015). The taxonomy is divided into 5 subsets of source code methods that act as sets of attributes to classify execution paths. Each subset's code methods are considered to have the same impact level (i.e. they are known to be involved in similar types of vulnerabilities). Each set is characterized by a number on the Likert scale (1 to 5) depicting the danger level of its source code methods: Set 1 contains the least dangerous methods while Set 5 contains the most dangerous source code methods, known to be involved in many critical vulnerabilities. For example, the `System.exec()` source code method is known to be tied to OS injection vulnerabilities (Martin and Barnum, 2008). Therefore `exec()` is grouped in Set 5 of the taxonomy. Severity ratings are applied by classifying each execution path into one of these five Severity sets of attributes which correspond to specific impact levels.

In the following paragraphs, we provide a brief description of this theory (Abramson, 1964). Let $Pr(C)$ be the probability of a transition in the path that indicates that the path is considered dangerous. $Pr(C)$ is quantified as the ratio of the dangerous source code methods over the total number of methods in the path. Let $f$ be the event that a specific source code method or statement exists in the path. We also denote by $\overline{C}$ and $\overline{f}$ the negations of $C$ and $f$.

The *prior entropy* $e$ is the probability distribution that expresses how certain we are that an execution path is considered dangerous, before feature $f$ is taken into account:

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

where *lg* is the binary logarithm (logarithm to the base 2). The posterior entropy, when feature $f$ has been detected in the path is

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

whereas the posterior entropy, when the feature is absent is

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

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

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

and the expected Information Gain (EIG) for a given feature $f$ is

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

The higher the EIG for a given set of attributes of source code methods $f$, the more certain we are that this set $f$ best describes the execution path.

Table 1: Severity classification examples - Data input methods.

| Rank | Example of classified methods | Set of Attributes |
|---|---|---|
| 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) |

Similarly to (Ugurel et al., 2002), EIG is calculated based on ratios between source code methods in a path that are considered dangerous (e.g. methods executing data, like `exec()`) and the total number of source code methods executed in the transitions of each execution path. The taxonomy of Java source code methods acts as the sets of attributes (corresponding to the event $f$ in the above equations). Example source code methods of the taxonomy and their classification into sets of attributes are given in Table 1 below. Different classification ranks reflect the different danger level. More technical details on the taxonomy are given in Section 4.2.1.

Severity ($\pi$) basically tells us which set of attributes best characterizes a path $\pi$; the one that exhibits the highest overall EIG. Since each set of attributes $f$ is tied to a specific impact (danger) level, then this level also indicates the danger level of the corresponding execution path.

#### 3.4.3 Reliability

As a measuring function, Reliability is used to classify execution paths into Reliability sets. It quantifies how reliable an execution path is by computing the likelihood that an exploitable behavior is manifested in a variable usage.

**Definition 2.** Given the execution path $\pi$, with a set of state variables, we define Reliability as

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

to measure the reliability of $\pi$ on a Likert scale from 1 to 5.

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

#### 3.4.4 Measuring code Reliability with Cyclomatic Density

The inherent risk or risk build-up of an AUT is connected to its source code's complexity (Chhabra and Bansal, 2014). A broadly accepted measure is the well-known *Cyclomatic Complexity* (Bray et al., 1997) that measures the maximum number of linearly independent circuits in a program's control flow graph (Gill and Kemerer, 1991). 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 in the graph. McCabe showed that V(G) can be computed by applying the following steps (Hansen, 1978):

1. increment by one for every IF, CASE or other alternate execution 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.

However, Cyclomatic Complexity does not take into consideration the size of the analyzed code. Research conducted in the Software Assurance Technology Center of NASA has showed that the most effective evaluation of the inherent risk of an AUT should be based on a combination of the (cyclomatic) complexity and the code's size (Rosenberg and Hammer, 1998). 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.

To this end, PLATO implements heuristics that assign Reliability ratings to execution paths through a cyclomatic density analysis. The proposed method is based on McCabe's algorithm and the computation of the *Cyclomatic Density* for each execution path. The *Cyclomatic Density* is the ratio of the Cyclomatic Complexity to the logical *lines-of-code*, which measures the number of executable "statements" in the path (some statements are excluded like for example a variable assignment) (McC, 2015). This ratio represents the normalized complexity of the source code of an execution path $\pi$ and it is considered a statistically significant single-value predictor of code's maintainability (Rosenberg and Hammer, 1998)(McC, 2015). 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 (Rosenberg and Hammer, 1998)(McC, 2015) proposes that Cyclomatic Density values for the code to be simple and comprehensible should be in the range of .14 to .42 .

Table 2: Reliability categories based on Cyclomatic Density values.

| Rank | Example of classified methods | Lvl |
|---|---|---|
| Safe | Cycl.Density <= 0.1 | 1 |
| Safe | Cycl.Density >0.1 && Cycl.Density <= 0.2 | 2 |
| Average | Cycl.Density >0.2 && Cycl.Density <= 0.3 | 3 |
| ErrorProne | Cycl.Density >0.3 && Cycl.Density <= 0.4 | 4 |
| ErrorProne | Cycl. Density >0.4 | 5 |

Each path is assigned a density value. The higher the value, the more complex the logic of the traversed code is and therefore more likely to have logical errors lurking in its transitions (Rosenberg and Hammer, 1998)(McC, 2015). Table 2 depicts the classification categories for execution paths that can be applied using the Reliability classification function.

#### 3.4.5 Risk: Combining Severity and Reliability Ratings

According to OWASP, the standard risk formulation is an operation over the likelihood and the impact of a finding (Martin and Barnum, 2008):

$$Risk = Likelihood * Impact$$

We adopt this notion of risk to enhance the logical error classification approach. For each execution path , an estimate of the associated risk is computed by combining $Severity(\pi)$ and $Reliability(\pi)$. Aggregation operations combine several fuzzy sets to produce a single fuzzy set. The Risk rank of an execution path $\pi$ is calculated using Fuzzy Logic's IF-THEN rules. An example is given in Figure 1.

The fuzzy logic classification system uses the following membership sets for ranks 1 to 5. For each pair (a, b), 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 member if the Medium Severity set with 100% (1) certainty. This way PLATO plots ranks 1 to 5 into membership sets. The rest of all intermediate values are plotted based on the mathematical equation defined by these points (a, b):

1. The *Severity* set: partitions the [1..5] impact scale to groups Low, Medium and High as: Low =(0,1) (3,0), M =(1.5,0) (2.5,1) (3.5,0), H =(3,0) (5, 1).

2. The *Reliability* set: partitions the [1..10] time scale to groups Early, Medium, Late and Very Late periods as: Low =(0, 1) (1, 1) (3,0), Medium =(0, 0) (3, 1) (5, 0), High =(0,0) (5,1).

By using the pre-computed tables with all expected values for Severity and Reliability, it is now possible to assess the fuzzy estimation of the Risk values, for a given logical error detection.

Risk calculations are performed as follows: Initially, the appropriate IF-THEN rules are invoked and generate a result for each rule. Then these results are combined to output truth values. Each IF-THEN result is, essentially, a membership function and 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) that a logical error belongs to.

Table 3 shows the fuzzy logic output for Risk, based on the aggregation of Severity and Reliability.

Table 3: Severity x Reliability = R - Risk sets.

| **Sev/ty** **Rel/ty** | Low | Medium | High |
|---|---|---|---|
| Safe | Low | Low | Medium |
| Medium | Low | Medium | High |
| Error-Prone | Medium | High | High |

Risk, Severity and Reliability ratings are supplementary to invariant violations and do not provide the basic mechanism for logical error detection; they just provide a more clear view for the code auditor. Also, high Severity rankings have more weight than Reliability rankings. Rightmost maximum is found to have closer-to-the-truth ranking results since Severity ratings take into consideration source code methods executed inside path transitions whilst Reliability ratings provide only a generic view of the execution path's overall complexity.

The Fuzzy Logic system has been implemented using the jFuzzyLogic library (Cingolani and Alcala-Fdez, 2012).

## 4 A METHOD TO DETECT LOGICAL ERRORS IN SOURCE CODE

### 4.1 The Method's Workflow

The analysis building blocks described in section 3 and implemented in PLATO are part of our workflow

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

Figure 1: Example of a Fuzzy Logic rule.

for logical error detection with the following steps:

1. **Use Case Scenarios.** We assume the existence of a test suite with use case scenarios that exercises the functionality of the AUT. The selected use-case scenarios must cover the intended AUT's functionality to a sufficient degree. This can be quantified by appropriate coverage metrics.
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 based on the executed use case scenarios.
3. Daikon invariants are loaded in PLATO and are processed as follows:
   - The inferred dynamic invariants **are filtered** by PLATO, in order to use only those referring to high-risk transitions, i.e. (i) statements that affect the program's execution flow, and (ii) source code methods that are connected to the manifestation of exploitable behaviour (e.g. method System.exec() for executing OS commands with user input).
   - PLATO **instruments the AUT code** with the critical dynamic invariants, which are embedded into the code as Java assertions (Martin and Barnum, 2008).

   The **instrumented source code is symbolically executed** in NASA's JPF tool with our `PlatoListener` extension. A sufficiently large number of feasible execution paths has to be covered, far more than the initial use case scenarios covering the intended functionality. JPF relies on the `PlatoListener` so as to check for existing assertion violations and then flags the invariants involved.
4. **PLATO gathers `PlatoListener` detections and classifies** each of 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 analysis of the source in step 3. Finally, the PlatoListener is used in step 4 for monitoring JPF's symbolic execution.

## 4.2 Classifying Execution Paths

Following Oracle's JAVA API and the related documentation in ((Harold, 2006)(Gosling et al., 2014)(jap, 2015)), three categories of Java source code methods are proposed for the classification of execution paths with respect to their Severity and Reliability values. Severity ranking is based on (i) Input Vectors and (ii) potentially exploitable methods (sinks). Reliability ranking is based on (iii) Control Flow checks (e.g. if-statements).

### 4.2.1 A Taxonomony of Source Code Methods for Severity Calculations

About 159 Java methods were reviewed and then grouped into sets depicting danger levels. These sets are used as features in the Information Gain algorithm to compute the Severity rating of execution paths. Classified source code methods were gathered from NIST's Software Assurance Reference Dataset suites (SARD) (Boland and Black, 2012), a set of known security flaws together with source code test-beds.

Five sets of attributes are proposed, corresponding to five danger levels from 1 to 5. The taxonomy was based on rankings of bugs and vulnerabilities recorded in NIST's National Vulnerability Database (NVD) (nvd, 2015), the U.S. government repository of standards based vulnerability management data. NVD provides scores that represent the innate characteristics of each vulnerability using the CVSS scoring system (nvd, 2015), which is an open and standardized method for rating IT vulnerabilities.

Thus, each source code method in the taxonomy is assigned to the set of attributes representing the appropriate danger level. The correct set of attributes is inferred based on the CVSS scores in the NVD repository. This was implemented using the following algorithm:

1. For each source code method, we checked the lowest and highest ratings of NVD vulnerabilities that use this source code method [1].
2. The characteristics of the identified vulnerabilities are then inputed in the CVSS 3.0 scoring calculator [2], in order to calculate the lowest and highest possible vulnerability scores.

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

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

3. Each source code method was then added in a set of attributes corresponding to the result of previous step. Source code 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()` source code method (jap, 2015) is widely-known to be used in many OS command injection exploits. NVD vulnerabilities recorded using this source code method have an impact rating ranging from 6.5 up to 10 out of 10. Using the characteristics of these records, the CVSS scoring calculator outputted a rating of high (7) to very high (10). This was expected, because `exec()` is often used to execute code with application level privileges. Thus, the `System.exec()` method was classified in PLATO's taxonomy in the very high (5/5) danger level category.

Tables 4 and 5 provide examples for various types. For the full taxonomy, the reader can access the link at the end of this article.

Table 4: Example group - Input Vector Methods taxonomy.

| |
|---|
| java.io.BufferedReader.readLine() |
| java.io.ByteArrayInputStream.read() |
| java.lang.System.getenv() |

Table 5: Example group - Sink methods taxonomy.

| |
|---|
| java.lang.Runtime.exec() |
| java.sql.Statement.executeQuery() |
| java.lang.System.setProperty() |
| java.io.File (new File()) |

#### 4.2.2 Statements and Methods for Reliability Calculations

Computing the Cyclomatic Density of a source code is tied to the number of execution-branch statements inside the code. Thus, Reliability calculations take into consideration Java statements that affect the program's control flow.

- **Control Flow Statements**

  According to (Harold, 2006) and (Felmetsger et al., 2010), boolean expressions determine the control flow. Such expressions are found in the statements shown in Figure 2.

All source code methods from the mentioned types were gathered from the official Java documentation (jap, 2015)(Harold, 2006) and are used for the computations of the Cyclomatic Density algorithm of Section 3.4.4.

(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 2: Example types of methods and statements included in PLATO's taxonomy.

## 5 EXPERIMENTAL RESULTS

The choice of the experimental scenarios was based on analyzing real-world source code containing the most common types of logical errors recorded in software development, according to (Martin and Barnum, 2008) and specifically (cwe, 2016).

We are not aware of any commercial, stand-alone suite or open-source revision(s) of software with a reported set of existing logical errors to use as a testing ground. Also, testing is restricted by JPF's limitations in symbolically executing software. For this reason, our experiments were selected as to contain real-world implementations of source code with different types of logical flaws often detected in real-world code audits, in an effort to prove that diverse types of logical errors can be detected and ranked effectively.

### 5.1 Experiment 1: Real-world Airline Test from the SIR Object Database

The Software-artifact Infrastructure Repository (SIR) (Rothermel et al., 2006) is a repository of software artifacts that supports controlled experimentation with program analysis and software testing techniques (Do et al., 2005), (Wright et al., 2010).

Our method was tested against a real-world AUT from the SIR repository, which exhibits the characteristics of a multithreaded activity requiring arbitration. The AUT was a multi-threaded Java program for an airline to sell tickets. The fact that this is a known and well-documented error that can be detected using different techniques (e.g. model checking) does not cancel the goal of this experiment, which was to show that our method can also detect logical errors that result in race conditions. This particular race condition is not detected through model checking but rather through an inferred invariant violation, thus providing that invariant violation method can be used to detect the subset of logical errors that produce race conditions.

**The Logical Error.** The logical error manifested in this example leads to a race condition causing the

airline application to sell more tickets than the available airplane seats. Each time the program sells a ticket, it checks if the agents had previously sold all the seats. If yes, the program stops the processing of additional transactions. Variable `StopSales` indicates that all the available 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 time more tickets are sold by the running threads (agents). The AUT's code is shown in Figure 3.

```
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 3: SIR AUT example code able to create i threads (agents) which sell tickets.

PLATO's analysis for this test returned the results shown in Figure 4. We now present in detail the results obtained in the different steps of our workflow:

**Step 1-2.** There is only one function point to test.

- Single test-case: Functionality has only one flow of events (multithreaded server accepting ticket sale information and registering them).
- Numerous test-case variations: Infinite possible variations for input (number of seats and cushion to limit maximum saling of tickets).

Thus, we executed 20 variations of the test-case for the airline server functionality, while trying to utilize boundary values as input. Daikon monitored inputs ranging from 1 seat with 1 sailing agent limit up to 1000 seats with 1000 agents and extracted the following invariant amongst others:

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

**Step 3.** Dynamic invariants were instrumented in the source code and the software was symbolically executed in JPF. An assertion violation was detected for the method `runBug()`: two executions were found where the mentioned invariant was enforced 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.**

## 5.2 Experiment 2: Multiple Execution Path Classification for Logical Errors

To test the proposed classification method implemented inside PLATO, we needed to execute it on an appropriate test suite. We had two options: Either utilize open-source applications or "artificially made" programs, common in benchmarking various source code analysis tools. Both options have positive and negative characteristics.

To this end, we endorsed the National Security Agency's (NSA) comparison results from (Agency, 2011) and (Agency, 2012), which state that "the benefits of using artificial code outweigh the associated disadvantages" when testing source code analysis tools. Therefore, we created a test-bed application based on the source code provided by NIST's **Juliet Test Case** suite, a formal collection of artificially-made programs (Boland and Black, 2012) packed with well-known and recorded exploits. The Juliet Test Suite is "a collection of over 81.000 synthetic C/C++ and Java programs with a priori known flaws. The suites Java tests contain cases for 112 different CWEs (exploits)" (Boland and Black, 2012). Each test case focuses on one type of flaw, but some tests contain multiple flaws. Each test has a *bad()* method in each test-program that manifests an exploit. A *good()* method is essentially a safe way of coding (true negative).

For our purposes we created an test suite that is, essentially, an aggregation of multiple Juliet test filled with various vulnerabilities of different danger-level; ranging from medium information leakage to serious OS execution injection. The test suit had both true positives and true negatives. The CWE types of vulnerabilities that manifested inside the analyzed test suite, as defined in NIST's formal CWE taxonomy (CVE, 2015), 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 source code methods detected in execution paths are provided at Table 6. We can see from table 6 that PLATO's classification system yielded an overall **Severity Rank = 3 out of 5** for the aggregated test suite but ranked specific individual paths with a **Severity Rank = 5 out of 5.** Interestingly, 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) source code methods detected in dangerous paths in-

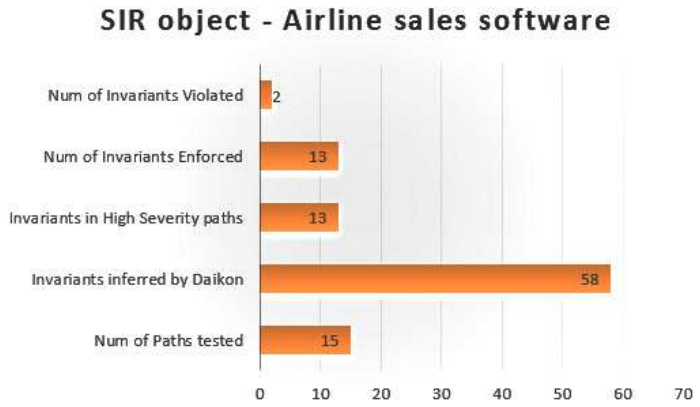


Figure 4: Airline sales: No of inferred invariants, chosen assertions and violations.

Table 6: Experiment 2: Information Gain classification.

| | |
|---|---|
| Entire Source code - Prior Entropy | 0.402179 |
| Entire Source code - Prior Severity | 3 |
| Entropy Loss for println(Rank 2) | 0.162292 |
| Entropy Loss for readLine(Rank 4) | 0.242292 |
| Entropy Loss for addCookie(Rank 2) | 0.162292 |
| Entropy Loss for exec(Rank 4) | 0.242292 |
| Entropy Loss for getPassword(Rank 2) | 0.162292 |
| Entropy Loss for getUserName Rank 2) | 0.162292 |

side the AUT presented the highest Gain (0.2423) on both occasions; thus scoring higher than all the rest. As seen in table 6, the first highest-ranked method is the exec() (Rank 4) method which is a sink utilized for OS command injection exploits.

PLATO's Severity mechanism (i) detected all paths prone to vulnerabilities due to the use of dangerous source code methods, and (ii) successfully ranked them to appropriate danger-levels based on their flaws; thus effectively representing their danger-level. We should underline here that this experiment was executed in order to demonstrate the classification capabilities of the Severity function. Data provided refer only to the classification mechanism.

# 6 CONCLUSIONS

Although detection rate was close to 100% success, still, the sample upon which PLATO was tested remains 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. Yet, publications in software engineering during the last years (Bastani et al., 2015) (Barr et al., 2015) continue indicate that Daikon is able to capture information flows and general execution logic behind the source code, if executed properly. The second most difficult problem in logical error detection, namely classifying information flows and invariant violations according to their impact on business logic, seems manageable if we utilize formal classification of these flows using functional characteristics from common experience in software development (e.g. method and libraries known to be error prone). This what PLATO does: It can classify likely invariants and their violations by utilizing a formal classifier, trained by NIST's source code error and vulnerability suite.

State explosion remains a major issue, since it is a problem inherited by the used analysis techniques, albeit now it seems manageable since we leverage JPF's memory management and implement all invariants as assertions instead of trying to analyze them in memory. Although not tested in really large applications due to JPF's aforementioned restrictions, our test results imply this. We are in the process of developing a fully-functional meta-tool that will be able to analyze any type of application by using targeted code analysis.

## 6.1 Advantages and Limitations

One of our method's limitations is the need for input data from live execution of AUTs, while Daikon infers the likely dynamic invariants. This is an inherent problem in all empirical methods, since empirical approaches rely on repetitive observations to form rules. Our method does not model the business logic of AUTs using formal methods, but is rather dependent on the soundness of the likely dynamic invariants provided by Daikon and the various executions of the AUT. If PLATO were to examine AUTs of thousands of source code lines in entirety, problems would arise, mostly due to JPF's inability to handle large, complex applications and also due to state explosion. Still, JPF is the best symbolic execution software currently available. Future test-beds can include more complex

utilization of JPF by creating an automated configuration mechanism to ”tweak” JPF per test-case and break testing into multiple executions.

Although empirical methods are often criticized for the lack of sound foundations in software engineering, it is obvious that, in order for a tool to detect flaws in the logic of applications, it needs to model knowledge that reflects intended functionality.

Also, (provided that it is executed in the correct manner and that it covers the entire functionality of an AUT) Daikon's output does reflect the AUT's intended functionality, since its dynamic invariants are properties that were true over the observed executions (Ernst et al., 2007). PLATO's results enforce this notion.

Based on the above notion and our tests, we drew some significant conclusions:

- PLATO can indeed detect logical errors in applications using reasonable limits in the size and complexity of AUTs, something no other tool can claim at the time this article was written.
- Results have shown that this method goes beyond logical error detection and can provide valid detections of other types of flaws. The unexpected detection of race conditions in one of our experiments, although it was an unintended side effect, proved this to be the case. As shown in previous results, limiting a variables value in an airplane ticket store not only led to a logical error that was essentially a race condition flaw, but also to a logical vulnerability that could lead the airline to sell more tickets that its seats.
- Logical errors must be detected using productive reasoning and not inductive because logical errors can manifest in widely different contexts. For example, a race condition can lead to a logical vulnerability and is indeed a subtype of logical programming errors, but it can also lead to other types of errors (null pointer exception, division by zero etc.) or even to no errors at all. Instead, PLATO's deductive approach, not only detects different types of logical errors but also provides insight on the impact of each error.

## REFERENCES


(2015). Common vulnerabilities and exposures, us-cert, mitre, CVE. MITRE, CVE-ID CVE-2014-0160.

(2015). The daikon invariant detector manual.

(2015). The java pathfinder tool.

(2015). Java platform, standard edition 7 api specification.

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

(2015). Using code quality metrics in management of outsourced development and maintenance.

(2016). Cwe-840: Business logic errors.

Abramson, N. A. (1964). *Introduction to Information Theory and Coding*. McGraw Hill.

Agency, N. S. (2011). *NSA, On Analyzing Static Analysis Tools*. National Security Agency.

Agency, N. S. (2012). *NSA,Static Analysis Tool Study-Methodology*. National Security Agency.

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

Baah, G. K. (2012). Statistical causal analysis for fault localization.

Balzarotti, D., Cova, M., Felmetsger, V. V., and Vigna, G. (2007). Multi-module vulnerability analysis of web-based applications. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 25–35. ACM.

Barr, E. T., Harman, M., McMinn, P., Shahbaz, M., and Yoo, S. (2015). The oracle problem in software testing: A survey. *Software Engineering, IEEE Transactions on*, 41(5):507–525.

Bastani, O., Anand, S., and Aiken, A. (2015). Interactively verifying absence of explicit information flows in android apps. In *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 299–315. ACM.

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

Bray, M., Brune, K., Fisher, D. A., Foreman, J., and Gerken, M. (1997). C4 software technology reference guide-a prototype. Technical report, DTIC Document.

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

Cingolani, P. and Alcala-Fdez, J. (2012). jfuzzylogic: a robust and flexible fuzzy-logic inference system language implementation. In *FUZZ-IEEE*, pages 1–8. Citeseer.

Do, H., Elbaum, S., and Rothermel, G. (2005). Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435.

Doupé, A., Boe, B., Kruegel, C., and Vigna, G. (2011). Fear the ear: discovering and mitigating execution after redirect vulnerabilities. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 251–262. ACM.

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

Etzkorn, L. H. and Davis, C. G. (1997). Automatically identifying reusable oo legacy code. *Computer*, 30(10):66–71.

Felmetsger, V., Cavedon, L., Kruegel, C., and Vigna, G. (2010). Toward automated detection of logic vulnera-

bilities in web applications. In *USENIX Security Symposium*, pages 143–160.

Gill, G. K. and Kemerer, C. F. (1991). Cyclomatic complexity density and software maintenance productivity. *Software Engineering, IEEE Transactions on*, 17(12):1284–1288.

Glover, E. J., Flake, G. W., Lawrence, S., Birmingham, W. P., Kruger, A., Giles, C. L., and Pennock, D. M. (2001). Improving category specific web search by learning query modifications. In *Applications and the Internet, 2001. Proceedings. 2001 Symposium on*, pages 23–32. IEEE.

Godefroid, P., Klarlund, N., and Sen, K. (2005). Dart: Directed automated random testing. *SIGPLAN Not.*, 40(6):213–223.

Gosling, J., Joy, B., Steele Jr, G. L., Bracha, G., and Buckley, A. (2014). *The Java Language Specification*. Pearson Education.

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

Harold, E. R. (2006). *Java I/O*. " O'Reilly Media, Inc.".

Hovemeyer, D. and Pugh, W. (2004). Finding bugs is easy. *ACM Sigplan Notices*, 39(12):92–106.

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

Păsăreanu, C. S. and Visser, W. (2004). Verification of java programs using symbolic execution and invariant generation. In *Model Checking Software*, pages 164–181. Springer.

Peng, W. W. and Wallace, D. R. (1993). Software error analysis. *NIST Special Publication*, 500:209.

Rosenberg, L. and Hammer, T. (1998). Metrics for quality assurance and risk assessment. *Proc. Eleventh International Software Quality Week, San Francisco, CA*.

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

Scarfone, K. A., Grance, T., and Masone, K. (2008). Sp 800-61 rev. 1. computer security incident handling guide. Technical report, Gaithersburg, MD, United States.

Stergiopoulos, G., Katsaros, P., and Gritzalis, D. (2014). Automated detection of logical errors in programs. In *Proc. of the 9th International Conference on Risks & Security of Internet and Systems*.

Stergiopoulos, G., Petsanas, P., Katsaros, P., and Gritzalis, D. (2015a). Automated exploit detection using path profiling - the disposition should matter, not the position. In *Proceedings of the 12th International Conference on Security and Cryptography*, pages 100–111.

Stergiopoulos, G., Theoharidou, M., and Gritzalis, D. (2015b). Using logical error detection in remote-terminal units to predict initiating events of critical infrastructures failures. In *Proc. of the 3rd International Conference on Human Aspects of Information Security, Privacy and Trust (HCI-2015), Springer, USA*.

Stergiopoulos, G., Tsoumas, B., and Gritzalis, D. (2012). Hunting application-level logical errors. In *Engineering Secure Software and Systems*, pages 135–142. Springer.

Stergiopoulos, G., Tsoumas, B., and Gritzalis, D. (2013). On business logic vulnerabilities hunting: The app_loggic framework. In *Network and System Security*, pages 236–249. Springer.

Ugurel, S., Krovetz, R., and Giles, C. L. (2002). 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*, pages 632–638. ACM.

Wright, H. K., Kim, M., and Perry, D. E. (2010). Validity concerns in software engineering research. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 411–414. ACM.

Zeller, A. (2002). Isolating cause-effect chains from computer programs. In *Proceedings of the 10th ACM SIGSOFT symposium on Foundations of software engineering*, pages 1–10. ACM.

Zhang, X., Gupta, N., and Gupta, R. (2006). Locating faults through automated predicate switching. In *Proceedings of the 28th international conference on Software engineering*, pages 272–281. ACM.

Zielczynski, P. (May 2006). Traceability from use cases to test cases. *IBM developerWorks*.