EXECUTION PATH CLASSIFICATION FOR VULNERABILITY ANALYSIS AND DETECTION

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

[1]Information Security & Critical Infrastructure Protection (INFOSEC) Laboratory
Dept. of Informatics, Athens University of Economics & Business (AUEB), Greece
{geostergiop, panagiotis.petsanas, dgrit}@aueb.gr

[2]Dept. of Informatics, Aristotle University of Thessaloniki, Greece
{katsaros}@csd.auth.gr

Keywords: Code Exploits, Software Vulnerabilities, Source Code Classification, Fuzzy Logic, Tainted Object Propagation

Abstract. Various commercial and open-source tools exist, developed both by the industry and academic groups, which are able to detect various types of security bugs in applications' source code. However, most of these tools are prone to non-negligible rates of false positives and false negatives, since they are designed to detect a priori specified types of bugs. Also, their analysis scalability to large programs is often an issue. To address these problems, we present a new source code analysis technique based on execution path classification. We develop a prototype tool to test our method's ability to detect different types of information-flow dependent bugs. Our approach is based on classifying the Risk of likely exploits inside source code execution paths using two measuring functions: Severity and Vulnerability. For an Application Under Test (AUT), we analyze every single pair of input vector and program sink in an execution path, which we call an Information Block (IB). Severity quantifies the danger level of an IB using static analysis and a variation of the Information Gain algorithm. On the other hand, an IB's Vulnerability rank quantifies how certain the tool is that an exploit exists on a given execution path. The Vulnerability function is based on tainted object propagation. The Risk of each IB is the combination of its computed Severity and Vulnerability measurements through an aggregation operation over two fuzzy sets using a Fuzzy Logic system. An IB is characterized of a high risk, when both its Severity and Vulnerability rankings have been found to be above the low zone. In this case, our prototype tool called Entroine reports a detected code exploit. The tool was tested on 45 Java vulnerable programs from NIST's Juliet Test Suite, which implement 3 different types of exploits. All existing code exploits were detected without any false positive.

1 Introduction

Automated detection of potential vulnerabilities in source code is a well-known research area. Multiple techniques have been proposed, some of which have been proven effective in detecting a priori known flaws (e.g. Time Of Check, Time Of Use errors, widely known as TOCTOUs). Yet, the National Institute of Software and Technology (NIST) published recently a report [1], which indicates that most tools still generate non-negligible numbers of false negatives and false positives, whereas their analysis scalability to very big programs is questionable. Nevertheless, many tools shine on specific types of vulnerabilities, but it is clear that there is no overall "best" tool with a high detection rate in multiple exploit categories [2, 28].

To cope with these issues, we propose here a different approach for detecting vulnerabilities in source code, which is based on the classification and the criticality assessment of an application's execution paths, with each path representing a sequence of program points from one location to another location of the program's control flow graph. We implemented our method in a prototype tool for the analysis of Java code to test our analysis technique. The tool, called Entroine analyses the code of an Application Under Test (AUT) for possible flaws by classifying the execution paths based on their Entropy Loss, thus producing data, which are processed by a mathematical fuzzy logic system. More precisely, Entroine processes structures called information blocks (IBs), with each of them containing information for

execution paths, variables and program instructions on the paths. Only a subset of all possible execution paths is examined: the paths from locations associated with input vectors to locations corresponding to information flow sinks. IBs are classified in two different groups of sets as follows:

- the *Severity* sets that quantify the danger level for the execution paths (the impact that an exploit would have, if it would be manifested in the path),
- the *Vulnerability* sets that quantify detected vulnerabilities based on a variable usage analysis (tainted object propagation and validation of sanitization checks, in which the data context of variables is checked).

The method consists of the following three components: (a) A static analysis, based on the BCEL library [31, 32], which creates the control flow graph that is parsed to get information about variable usages. It is thus possible to detect input data vectors, control-flow locations and instructions that enforce context checks on variable data. Entroine then maps the execution paths for the AUT variables and, more specifically, only those locations, where the program execution can follow different paths (execution flow branching points). (b) A classification approach that combines output from (a) to create IBs. Each IB is classified using statistical Entropy Loss and the two fuzzy membership sets (Severity and Vulnerability). (c) A Fuzzy Logic system for quantifying the overall Risk for each IB, based on linguistic variables, and Severity and Vulnerability classification ratings.

The main contributions of this paper are summarized as follows:

1. We introduce a program analysis technique for our classification system. Based on the control flow graph and our Fuzzy Logic ranking system only a limited number of execution paths and statements have to be analyzed.
2. We propose a Risk classification of program locations using two membership functions, one for the identified Severity (Entropy Loss) and another one for the Vulnerability level.
3. We present our prototype tool. By using the Vulnerability and Severity classifications, we realized that the number of false positives is reduced. In addition, Entroine warned for elevated danger levels in program locations where a false negative could have been occurred.
4. We provide test results from applying Entroine to the Juliet Test Suite [33] that has been proposed by NIST to study the effectiveness of code analysis tools [28]. Entroine detected all common weaknesses used upon, without having reported any false positive.

In Section 2, we report recent results in related research. In Section 3, we expose the theoretical underpinnings of our method. Section 4 provides technical details for the implementation of our method in Entroine and section 5 presents our experiments and reports metrics and detection coverage in all tests.

2 Related Work

Previously proposed analysis techniques based on tainted object propagation such as the one in [3] mostly focus on how to formulate various classes of security vulnerabilities as instances of the general taint analysis problem. These approaches do not explicitly model the program's control flow and it is therefore possible to miss-flag sanitized input, thus resulting in false positives. Furthermore, there is no easy general approach to avoid the possibility of false negatives. This type of analysis does not suffer a potential state space explosion, but its scalability is directly connected to the analysis sensitivity characteristics (path and context sensitivity) and there is an inherent trade-off between the analysis scalability and the resulting precision/recall.

Regarding well-known static analysis tools, it is worth to mention FindBugs [9], which is used to detect more than 300 code defects that are usually classified in diverse categories, including those analyzed by tainted object propagation. The principle of most of the FindBug's bug detectors is to identifying low-hanging fruits, i.e. to cheaply detect likely defects or program points where the programmer's attention should be focused [4].

Other tools, such as [5, 6, 7] and [36] are well-known for their capability to detect numerous bugs, but related research in [8] has shown that their main focus is centered around specific bug types like null pointer exceptions, explicit import-export and not those, for which a taint analysis is required (XSS, OS executions etc.). In [8], a relatively low detection rate is reported for many of the above mentioned tools for some variants of important bug types (null pointer exception, user injections and non-black final instance). To the best of our knowledge, none of the above mentioned tools implements a mechanism to cope with the possibility of false negatives.

Pixy [10], a prototype implementing a flow-sensitive, inter-procedural and context-sensitive dataflow, alias and literal analysis, is a new tool that further develops pre-existing analyses. It mainly aims to detect cross-site scripting vulnerabilities in PHP scripts, but a false positive rate is at a-round 50% (i.e., one

false positive for each detected vulnerability) has been reported and no mechanism has been implemented to mitigate the problem.

Other researchers try to detect code flaws using program slicing. Introduced in [11] is a program slicing technique and applied it to debugging. The main drawback is that the slice set often contains too many program entities, which in some cases can correspond to the whole program.

Presented in [13], this technique that uses a threshold to prune the computed backward slice. A limitation is that this technique does not account for the strength of the dependences between program entities, nor the likelihood for each program entity to be a failure cause. Another limitation is that the slice sets can be sometimes very large. Finally, no information is provided by any of these techniques for how to start searching for a code flaw.

Researchers in [15], focus exclusively on specific flaws found in web applications as in [16], where various analysis techniques are combined to identify multi-module vulnerabilities.

None of these techniques attempts to profile the danger level in the program's behavior. In [12, 34, 35], we have presented the APP_LogGIC tool for source case analysis, but we focused on logical errors instead of program exploits and our ranking method was not based on a statistical classification of the source code.

```
public void bad() throws Throwable {
        String data = "", test = "";
        /* FLAW: data from .properties */
        data = properties.getProperty("data");
        if(System.getProperty("os.name").indexOf("win") >= 0) {
                String osCommand = "c:\\WINDOWS\\SYSTEM32\\cmd.exe /c dir ";
        }
        /*POTENTIAL FLAW: command injection */
        Process proc = Runtime.getRuntime().exec(osCommand + data);
}
```

Fig. 1. NIST's command injection example

3 Methodology

Let us consider the example shown in Figure 1 from the Juliet Test Suite, a collection of programs for testing source code analyzers [33].

Example 1. Variable data in Figure 1 is assigned data originated from the source `properties.GetProperty`. Then, it is used in the sink instruction `getRuntime().exec` without being checked previously or having sanitized the variable's contents, as it should have happened. Our method will detect and analyze the execution path starting from the invocation of `getProperty("data")` and ending with the `exec()` call, thus revealing the exploit present in the source code.

Definition 1. Given the set T of all transitions in the control flow graph of an AUT, an information block IB is a structure, containing a set of instructions I and a set of transitions Ti $\subseteq$ T enabled at the corresponding program points, along with information about data assignments on variables used in sets I and Ti.

Our method outputs a Risk value (ranging from one to five) that denotes the overall danger level of an IB. The Risk is quantified by means of a source code classification system using Fuzzy Logic to flag exploits [14]. This classification technique aims to confront two important problems: the large data sets of the AUT analysis and the possible false positives and false negatives when trying to detect specific vulnerabilities. Regarding the first mentioned problem, the Entroine tool can help auditors to focus only to those instructions and paths that appear having a relatively high rating in its classification system. The second mentioned problem can be alleviated through Entroine's ratings that implement general criteria, which take into account the possibility of an exploit in execution paths (Vulnerability) and a path's danger level (Severity). Two measuring functions, namely Severity and Vulnerability create fuzzy sets reflecting gradually varying danger levels. Each IB gets a membership degree in these sets, which represents its danger level and it is thus classified within a broad taxonomy of exploits which is based on a wide variety of publications [26, 27, 37]. Membership sets act as code filters for the IBs.

Example 2. Figure 2 below depicts the Entroine's output for the program of Figure 1. Our tool detected data entry points (input vectors) and the relevant execution path, stored in the "Lines Executed" field (line numbers correspond to lines inside the source code Class file, depicting the execution path's instructions). Then, the IB has been classified in relevant Severity and Vulnerability ranks by analyzing checks and relations between variables. Fig. 2's *Input Ranking* depicts the rank assigned based on the input vector classification, in this case, the `readLine()` instruction. Similar, Sink ranking depicts the rank assigned in the sink instruction where the exploit manifests: a rank 5 OS Injection `exec()` instruction.

```
Input Ranking:           4
Sink Ranking:            5
Starting at line:        54
Source:                  getProperty
Input into variable:     data bad, test bad
Sink method:             exec
Execution arguments:     osCommand, data,
Severity Rank#:          5
Lines executed:          106 83 82 81 80 79 78 77 76 75 73
                         62 60 57 56 54
Dangerous variables:     proc , test , data,
Connections between the dangerous variables: proc <-- data
Vulnerability Rank#:     4
```

Fig. 2. Information block (IB) example (from Entroine)

In the following section, we describe in detail the way that the Severity and Vulnerability classification ranks are computed.

3.1 Source code profiling for exploit detection

Entroine classifies source code using two different classification mechanisms: Severity and Vulnerability. Entroine aggregates results from both to produce a distinct, overall Risk value for dangerous code points.

Severity.

For an information block IB, `Severity(IB)` measures the membership degree of its path π in a Severity fuzzy set. `Severity(IB)` reflects the relative *impact* on an `IB`*'s* execution path π, if an exploit were to manifest on π. According to [22] by the National Institute of Software and Technology (NIST), the impact of an exploit on a program's execution can be captured by syntactical characteristics that determine the program's execution flow, i.e. the input vectors and *branch conditions* (e.g. conditional statements). Variables used in each transition of the execution path are weighted based on how they affect the control flow. Thus, variables that directly affect the control flow or are known to manifest exploits (e.g. they input data, used in branch conditions or affect the system) are considered dangerous.

Definition 2. Given the information block IB, with a set of variables and their execution paths, we define Severity as

$$\texttt{Severity(IB)} = v \in [0,5]$$

measuring the severity of `IB` on a Likert-type scale from 1 to 5.

Likert scales are a convenient way to quantify facts [17] that in our case refer to a program's control flow. If an exploit were to manifest on an execution path within an IB, the scale-range captures the intensity of its impact in the AUT's execution flow. Statistical Entropy Loss classifies execution paths and their information blocks in one of five Severity categories, one (1) through five (5). Categories are then grouped into Fuzzy Logic sets using labels: *high* Severity (4-5), *medium* (3) or *low* (1 or 2).

Entropy Loss as a statistical function for Severity measurement.

Evaluation of the membership degree of each execution path in the Severity set can be based on a well-defined statistical measure. To assign Severity ranks, continuous weights are estimated using Prior Entropy and Entropy Loss. Finally, a fuzzy relational classifier uses these estimations to establish correlations between Severity ranks and execution paths.

Expected Entropy Loss, which is also called *Information Gain*, is a statistical measure [19] that has been utilized as a feature selection technique for information retrieval [20]. Feature selection increases both effectiveness and efficiency, by removing non-relevant information based on corpus statistics [30].

Our method is based on selected *features*, i.e. source code instructions, which are tied to specific types of vulnerabilities (section 4.2). For example, the `exec()` instruction is known to be tied to OS injection vulnerabilities. Thus, Entroine uses `exec()` as a feature to classify vulnerable IBs as a detected type of OS Injection.

Expected entropy loss is computed for each information block and ranks each IB based on methods/features detected in its execution path. Each Severity rank corresponds to a specific taxonomy subset comprised of source code methods. A total of five (5) ranks/method sets depicting five different levels of danger exist (0 being the lowest to 4 being the highest). For example, subset 0 is comprised of instructions that are only detected in very low risk vulnerabilities, e.g. it contains the java.awt.Robot.keyRelease() method. This classification method considers features to be more important if they are effective discriminators of a specific danger level/rank (a.k.a. method subset) [18].

This technique was also used for source code classification in [21] and [18]. Here, we use the same technique, in order to classify execution paths and corresponding IBs into danger levels.

In the following paragraphs, we provide a brief description of the theory [19]. Let C be the event that indicates whether an execution path must be considered dangerous, depending on the path's transitions, and let *f* be the event that the path does indeed contain a specific feature *f* (e.g. the exec() instruction). Let $\bar{C}$ and $\bar{f}$ be their negations and *Pr( )* their probability (computed as in section 4.3.1). The prior entropy is the probability distribution that expresses how certain we are that an execution path belongs to a specific category, before feature f is taken into account:

$$e = -Pr(C)\, lgPr(C) - Pr(\bar{C})\, lgPr(\bar{C})$$

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

$$e_f = -\Pr(C \mid f) \lg \Pr(C \mid f) - \Pr(\bar{C} \mid f) \lg \Pr(\bar{C} \mid f)$$

whereas the posterior entropy, when the feature is absent is

$$e_{\bar{f}} = -\Pr(C \mid \bar{f}) \lg \Pr(C \mid \bar{f}) - \Pr(\bar{C} \mid \bar{f}) \lg \Pr(\bar{C} \mid \bar{f})$$

Thus the expected overall posterior entropy is given by

$$e_f \Pr(f) + e_{\bar{f}} \Pr(\bar{f})$$

and the expected entropy loss for a given feature *f* is

$$e - e_f \Pr(f) - e_{\bar{f}} \Pr(\bar{f})$$

The expected entropy loss is always non-negative and higher scores indicate more discriminatory features.

Example 3. Let us compute the expected Entropy Loss used for the Severity classification of the program in Example 1. Our Severity function will classify the path's features (input vectors, sinks, branch statements like `exec()` and `getProperty()`) according to a taxonomy of features made up of dangerous coding methods (Section 4.2). Five probabilities `Pr(C)` were computed, one for each of the five Severity ranks and the IB was classified at the Severity rank 4 (maximum danger level-set of the five). The IB's prior entropy e was then calculated for the same ranks. Prior entropy represents the current classification certainty of Entroine, i.e. the level of confidence that it has assigned the correct Severity rank. Finally,

the entropy loss (information gain) was calculated for each one of the detected input vectors and sinks in the execution path, for the variable data. We are interested in the highest and the lowest observed Entropy Loss (Information Gain) values:

1. The higher value of information gain is observed, the more the uncertainty for a dangerous security characteristic is lowered and the classification to a specific Rank category is therefore more robust. Also, a relatively high information gain coupled with a high probability `Pr(C|f)` for sanitization provides information about features within paths that lower the Vulnerability level.
2. The lowest value of information gain (highest entropy) provides information on the most widespread and distributed security methods by showing the level of danger diffused in the AUT's execution paths.

Figure 3 below depicts the Entropy Loss output for the example path. The conclusion drawn from this output is the following:

- The highest entropy loss (information gain) is detected in method `getProperty`. This shows that `getProperty` is a defining characteristic for this Rank 5 exploit.
- The lowest entropy loss in method `exec()` has highest probability of appearance, Pr(C)=1, which basically means that `exec()` is being used in all execution paths analyzed as potentially dangerous. This elevates the Severity level of the detected exploit significantly, because `exec()` is prone to OS injection.

```
Prior path Entropy =                          0.8112781
Source code Severity rank:                    5
Calculation exec. time:                       2 seconds
Entropy Loss for getInputStream - Rank 5: 0.31127812
Entropy Loss for getProperty - Rank 5:    0.81127812
Entropy Loss for exec in Rank 5:              0.0
```

Fig. 3. OS injection Rank 5 classification – Entropy calculation example

Vulnerability.

Vulnerability sets define categories based on the type of detection and the method's propagation rules and each category reveals the extent to which variable values are sanitized by conditional checks [22]. As a measuring function, Vulnerability assigns IBs into vulnerability sets thus quantifying how certain the tool is about an exploit manifesting in a specific variable usage.

Definition 3. Given the information block IB, with a set of variables and their execution paths, we define Severity as

$$\texttt{Vulnerability(IB)} = v \in [0,5]$$

Ratings here also use a Likert scale [17] from 1 to 5. Similarly to the `Severity(IB)` function, our fuzzy logic system classifies IBs in similar categories: "high" vulnerability, "medium" or "low".

Vulnerability function - Object Propagation and Control Flow heuristics.

Our control flow based heuristics for assigning Vulnerability ratings to information blocks are complemented by a *tainted object propagation* analysis. Tainted object propagation can reveal various types of attacks that are possible due to user input that has not been (properly) validated [3]. Variables holding input data (*sources*) are considered tainted. If a tainted object (or any other object derived from it) is passed as a parameter to an exploitable instruction (a *sink*) like the instructions executing OS commands we have a vulnerability case [3].

Variables and checks enforced upon them are analyzed for the following correctness criterion: all input data should be sanitized before their use [22]. Appropriate checks show: (i) whether a tainted variable is used in sinks without having previously checked its values, (ii) if data from a tainted variable is passed along and (iii) if there are instances of the input that have never been sanitized in any way.

Entroine checks tainted variable usage by analyzing its corresponding execution paths and conditions enforced on them (if any) for data propagation. The tool uses explicit taint object propagation rules for

the most common Java methods, such as `System.exec()`. These rules are outlined in Section 4.3.2, where the technical implementation details are discussed.

Example 4. Again, as an example, we will show how our method analyzes the program of Figure 1 to decide in which Vulnerability rank to classify the IB of Example 2. Our tainted object analysis detects that (i) an input vector assigns data to variable data and, then, (ii) data is never checked by a conditional statement like an `if-statement` or any other instruction known to sanitize variable data. Then (iii) variable data is used in a sink (`exec()`) without further sanitization of its contents. Thus, our method will not detect any transition that lowers the Vulnerability level of the execution path in Figure 1 and will therefore assign a high rating (4) on the Vulnerability scale for the IB containing this variable-execution path pair.

Using Severity and Vulnerability thresholds, Entroine can focus only on a subset of paths for exploit detection, thus limiting the number of paths needed to traverse during its tainted propagation analysis. The execution path set is pruned twice: (i) once based on Severity measurements and the type of instruction used ("safe" paths are discarded), and (ii) again when possible exploits have been detected by using a Vulnerability rank as threshold.

3.2 Risk

According to OWASP, the standard risk formulation is an operation over the likelihood and the impact of a finding [23]:

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

We adopt this notion of risk into our framework for exploit detection. In our approach, for each `IB` an estimate of the associated risk can be computed by combining `Severity(IB)` and `Vulnerability (IB)` into a single value called Risk. We opt for an aggregation function that allows taking into account membership degrees in a Fuzzy Logic system [14]:

Definition 3. Given an AUT and an information block IB with specific input vectors, corresponding variables and their execution paths $\pi \in P$, function Risk(IB) is the aggregation

$$\text{Risk(IB)}=\text{aggreg(Severity(IB),Vulnerability(IB))}$$

with a fuzzy set valuation

$$\text{Risk(IB)} = \{\text{Severity(IB)}\} \cap \{\text{Vulnerability(IB)}\}$$

Aggregation operations on fuzzy sets are operations by which several fuzzy sets are combined to produce a single fuzzy set. Entroine applies defuzzification [24] on the resulting set, using the Center of Gravity technique. *Defuzzification* is the computation of a single value from two given fuzzy sets and their corresponding membership degrees, i.e. the involvedness of each fuzzy set presented in Likert values.

Risk ratings have the following interpretation: for two information blocks `IB1` and `IB2`, if Risk(`IB1`) > Risk(`IB2`), then `IB1` *is more dangerous than* IB2, in terms of how respective paths $\pi 1$ and $\pi 2$ affect the execution of the AUT and if the variable analysis detects possible exploits. In the next section, we provide technical details for the techniques used to implement the discussed analysis.

The risk of each information block is plotted separately, producing a numerical and a fuzzy result. It is calculated by the Center of Gravity technique [24] via its Severity and Criticality assigned values. Aggregating both membership sets, produces a new membership set and, by taking the "center" (sort of an "average"), Entroine produces a discrete, numerical output.

4 Design and Implementation

In this section, the technical details on how Entroine was developed will be presented. The tools architecture, workflow along with technical details on how Severity and Vulnerability are calculated.

4.1 Entroine's architecture and workflow

Entroine consists of three main components: a static source code analyzer (depicted with the colors orange and green in Figure 4 below), an Information Block constructor and, finally, the fuzzy logic system to compute the Risk (grey and yellow colors).

- *Static Analysis:* Static code analysis uses the Java compiler to create Abstract Syntax Trees for the AUT. It provides information concerning every single method invocation, branch statements and variable assignments or declarations found in the AUT. Compiler methods (`visitIf()`, `visitMethodInvocation()`, etc.) were overridden, in order to analyze branch conditions and sanitization checks of variables. The following sample output shows the AST meta-data gathered for variable "sig_3" in a class named "Sub-system114":

  ```
  DECLARE::12::double::sig_3::0::Main22::Subsystem114.java
  ```

  The ByteCode Engineering Library (Apache BCEL) [32] is used to build the AUT's control flow graph and to extract the program's execution paths. BCEL is a library that analyzes, creates, and manipulates class files.
- *Information Block Creator:* This component combines information obtained from the static analysis to create IBs that contain pairs of execution path – input vector. Information blocks are then assigned Severity and Vulnerability ranks.
- *Fuzzy Logic system:* The Fuzzy Logic system is implemented using the jFuzzyLogic library [14]. We use it to aggregate Severity and Vulnerability sets to quantify the danger level for each IB. This aggregation classifies each IB to an overall Risk rank.

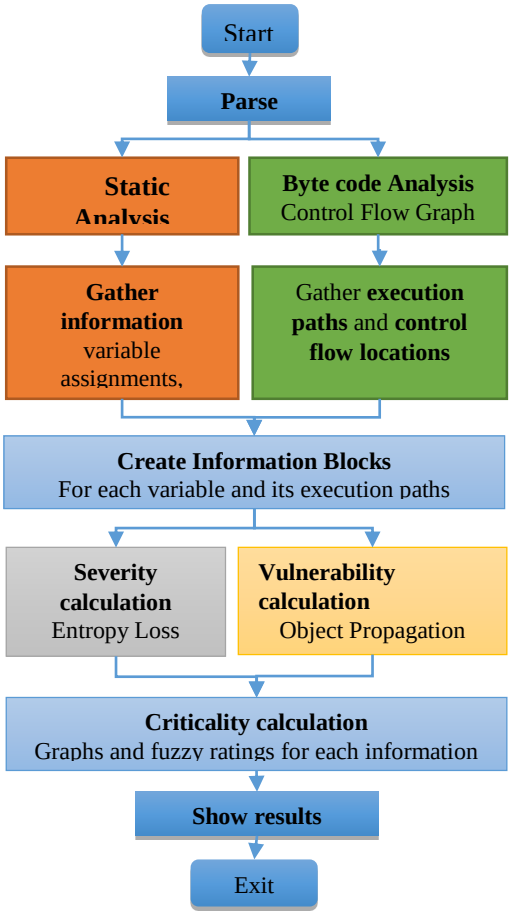

Fig. 4. Entroine's processing flowchart

4.2 Taxonomy of dangerous instructions

Following Oracle's JAVA API and documentation [25, 26, 27], three categories of Java instructions were used to classify execution paths in IBs. (i) *Control Flow instructions*, (ii) *Input Vector instructions* and

(iii) potentially exploitable methods (sinks). 159 Java methods were reviewed and gathered from formal publications and organizations that specifically classify exploits and, consequently, any instructions used in them [26, 27, 37].

The taxonomy's methods were grouped into *5 categories* of Severity corresponding to the taxonomy's Severity ranks. We based the Severity classification ranks for ranking instructions on the well-known international Common Vulnerability Scoring System (CVSS) scoring system [37] and the Common Weakness Enumeration system [38]. CVSS classifies potential vulnerabilities into danger levels and is used by NIST's National Vulnerability Database (NVD) to manage an entire database of vulnerabilities found in deployed software. The NVD is also using the CWE system as a classification mechanism that differentiates CVEs by the type of vulnerability they represent. The Common Weakness Scoring System (CWSS) [40] provides a mechanism for prioritizing software weaknesses. It applies scores into vulnerabilities based on a mathematical formula of characteristics.

Entroine uses all three of these systems, NVD-CVSS, CWE and CWSS ranking to assign source code instructions to specific danger levels, according the type of vulnerability, in which they participate, its general SWSS score and corresponding ranking value in similar CVSS vulnerabilities that we found.

The method followed to classify source code methods is the following: Each source code instruction in the taxonomy is assigned to a specific feature set which represents a danger level. The correct feature set was chosen using the CVSS scores of vulnerabilities from the NVD repository [42]. The steps are:

1. For each instruction, check the lowest and highest ratings of known vulnerabilities that use it.
2. Use the CVSS 3.0 scoring calculator 2 and calculate the lowest and highest possible vulnerability scores for each instruction, based on the characteristics of known vulnerabilities.
3. Rank each instruction in a feature set corresponding to a danger level using the output calculation. Instructions with score 7 or above were grouped in Set 5. Instructions with score 6 to 7 in Set 4, those with score 5 to 6 in Set 3, those with score 4 to 5 in Set 2, those with score 1 to 4 in Set 1.

Example: The `Runtime.exec()` instruction is widely-known to be used in many OS command injection exploits. CWE and NIST provide a multitude of critical vulnerabilities based in this instruction (e.g. the CWE-78 category). Also the CVSS 3.0 scoring system [41] ranked the use of `exec()` to execute code with application level privileges very high in its scale (9.3 out of 10). Thus, Entroine's taxonomy classifies the `exec()` instruction into its very high (5) danger level category. Similar notion has been followed in organizing the rest of Entroine's taxonomy instructions into Severity levels. This way, we limit our personal intuition, in an effort to support that Entroine's ranking system is justified.

Due to space limitations, only two small Java Class group examples are given. The complete classification system can be found in the provided link at the end of the article. The symbol § corresponds to chapters in Java documentation [26]:

Control Flow Statements.

According to a report [28], Boolean expressions determine the control flow. Such expressions are found in the following statements:

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)

Input Vector Methods.

Java has numerous methods and classes that accept data from users, streams or file [27]. Most of them concern byte, character and stream input/output. Entroine takes into account 69 different ways of entering data into an AUT. A small example is given below in Table 1**.**

Table 1. Example group: Input Vector methods taxonomy

java.io.BufferedReader	java.io.BufferedInputStream
java.io.ByteArrayInputStream	java.io.DataInputStream
java.lang.System	javax.servlet.http.
java.io.ObjectInputStream	java.io.StringReader

Based on [27] and common programming experience, monitoring specific Java objects seems to be an adequate, albeit not entirely thorough, way of tracing user data inside Java applications.

Exploitable Methods (sinks).
Based on CWE, NVD [37] and common knowledge, we know that specific methods are used in exploits. We consider them as potential sinks and thus, Entroine examines them carefully. As mentioned earlier, Entroine's taxonomy of exploitable methods was based on the exploit classification and relevant source code by NIST's NVD in their CWE taxonomy [37]. Entroine takes into account 90 methods known to be exploitable as sinks, according to NIST CWEs. It then classifies them according to CWE's rank and its corresponding CVSS-CWE and CWSS rank. An example is given at Table 2.

Table 2. Example group - Sink methods taxonomy

java.lang.Runtime	java.net.URLClassLoader
java.lang.System	java.sql.Statement
java.io. File	java.net.Socket

4.3 Classification and ranking system

As explained in Section 3, the Fuzzy Logic system from [14] is used in Entroine, which provides a means to rank possible logical errors. In order to aid the end-user, Severity and Vulnerability values are grouped into 3 sets (Low, Medium, High), with an approximate width of each group of 5/3 = 1,66~1,5 (final ranges: Low in [0…2], Medium in (2…3,5] and High in (3,5…5]).

Calculating Severity (Entropy Loss and Feature Selection).
Entroine's classification system for execution paths uses Entropy Loss (aka Information Gain) to capture the danger level in AUT's execution paths. It takes into consideration specific method execution appearances against the total number of instructions in a given set of transitions and applies Severity ranks to execution paths and the corresponding information blocks.

Entroine detects, evaluates and classifies instructions found in execution paths. Severity ratings are applied by classifying each information block into and corresponding path into one of five Severity levels, according to the Prior Entropy and Entropy Loss of features in every execution path.

Since each information block refers to a specific execution path and its variables, the necessary metrics are calculated based on a ratio between execution paths considered dangerous (e.g. command execution instructions like `exec()`) and the total number of paths detected for each applicaiton. Similarly to [18], probabilities for the expected entropy loss of each feature are calculated as follows:

$$\Pr(C) = \frac{\text{numberOfPathsInCategory}}{totalNumberOf\text{ExecutionPaths}}$$

$$\Pr(\bar{C}) = 1 - \Pr(C)$$

$$\Pr(f) = \frac{\text{numberOfPathsWithFeatureF}}{totalNumberOf\text{Paths}}$$

$$\Pr(\bar{f}) = 1 - \Pr(f)$$

$$\Pr(C|f) == \frac{\text{numberOfPathsForSpecificRankWithFeatureF}}{totalNumberOf\text{Paths}WithFeatureF}$$

$$\Pr(\bar{C}|f) = 1 - \Pr(C|f)$$

$$\Pr(C|\bar{f}) == \frac{\text{numberOfPathsForSpecificRankWithoutFeatureF}}{totalNumberOf\text{Paths}WithoutFeatureF}$$

$$\Pr(\bar{C}|\bar{f}) = 1 - \Pr(C|\bar{f})$$

numberOfPathsWithFeatureF denotes to the sum of execution paths inside the application under test that contain a specific method in them regardless of category that the path belongs to (low, medium or high), while *numberOfPathsForSpecificRankWithFeatureF* represents the sum of execution paths which belong to a specific danger level category (e.g. Severity rank 3 set), based on a specific feature F.

Entropy Loss is computed separately for each source code feature characterized by a specific token. Only tokens that are part of a variable's execution paths are analyzed. For example, in the expression `"data = properties.getProperty ("data");"` the tokens will be: `"data"`, `"getProperty"` and `"properties"`.

The taxonomy of Java instructions in section 4.2 defines various features used in place of *f* in the above equations. An example of Entroine's features classification is given in Table 3. For a complete list, the reader is referred to the link at the end of the article.

Table 3. Severity classification examples

Rank	**Example** of classified methods	**Category**
Low	javax.servlet.http.Cookie (new Cookie)	0
Low	java.lang.reflection.Field (new Field)	1
Medium	java.io.PipedInputStream (new PipedInputStream)	2
High	java.io.FileInputStream (new FileInputStream)	3
High	java.sql.ResultSet:: getString()	4

Severity is evaluated relative to the Technical Impact 5-point scale, as defined by NIST's Common Weakness Scoring System (CWSS). For better granularity, Entroine's scale is a 10-point scale, from 0 to 4 corresponding to the four Severity method subsets that depict different danger levels. The Fuzzy membership sets created by combining the CWSS scale points with the 5-point Severity rank scale are presented below in Figure 5:

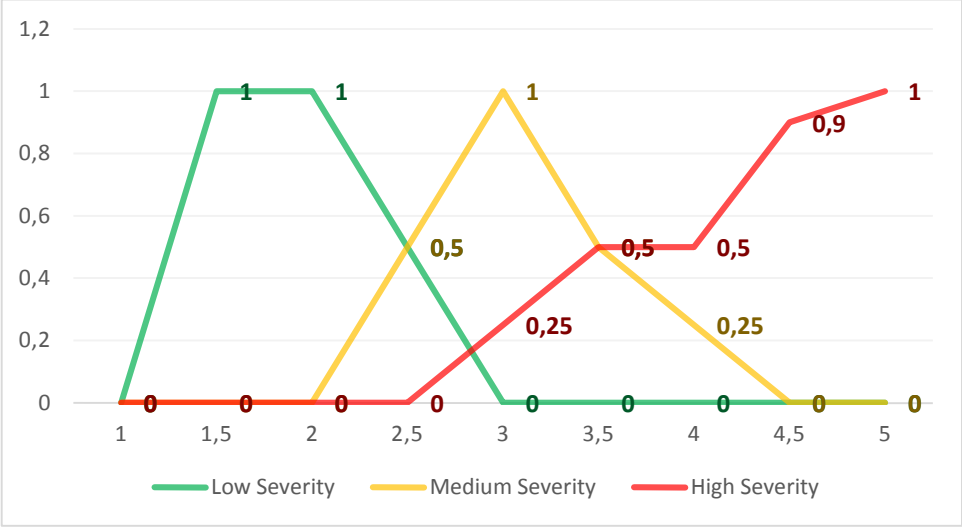

Fig. 5. Severity membership sets for Low, Medium and High rank values

The participation of values into membership sets for each part of the 1-5 scale are depicted below in Table 4:

Table 4. Participation percentages of Severity values in Fuzzy sets

0-4 ranks (values)	% participation – Low set	% participation – Medium set	% participation – High set
1	0	0	0
1.5	1	0	0
2	1	0	0
2.5	0.5	0.5	0
3	0	1	0.25
3.5	0	0.5	0.5
4	0	0.25	0.5
4.5	0	0	0.9
5	0	0	1

Calculating Vulnerability (Control Flow analysis and Tainted Object Propagation).
To calculate Vulnerability, Entroine runs a Tainted Propagation algorithm that classifies the likelihood of an exploit happening in an execution path. Entroine uses BCEL [31, 32] to traverse the program's Control Flow Graph bottom-to-top, in order to gather variable execution paths. Entroine's propagation rules are the following:

- The highest entropy loss (information gain) is detected Variables assigned data from expressions (e.g. +, -, method return) whose output depends on tainted data, are tainted.
- Literals (e.g. hardcoded strings, true declarations) are never tainted.
- If an object's variable gets tainted, only data referred by that variable are considered tainted, not all object properties.
- Methods that accept tainted variables as parameters are considered tainted.
- The return value of a tainted function is always tainted, even for functions with implicit return statements (e.g. constructors).

Table 5 depicts the check rules for exploit detection.

Table 5. Vulnerability check rules and their categories

Rank	Example of classified methods	Category
Low	No improper checks of variables	1
Low	Sinks NOT linked to input vectors	2
Medium	Propagation to methods	3
High	Improper checks on variables with input data – Variables used in sinks	4
High	No checks - variables used in sinks	5

Precalculated membership sets exist in the Fuzzy logic system that classify each IB Vulnerability value into a "Safe" or "Vulnerable" state. These two subsets are defined with fuzzy boundaries; i.e. the distinction between a "safe" flaw and a flaw that can lead to exploits is not clear. For this reason, empirical observations and examples from the National Vulnerability Database (NVD) [42] led us to believe that vulnerability scores vary due to context differentiations. To this end, we applied an empirical distribution of Vulnerability ranks into "Safe" and "Vulnerable" that leaves plenty of room for modeling uncertainty in lower ranks (1.5 to 2.5 out of 5 is a grey area). Same as with Severity, the fuzzy membership sets that classify Vulnerability ranks are depicted below in Figure 6:

Fig. 6. Vulnerability membership sets for Safe and Vulnerable rank values

The participation of values into membership sets for each part of the 1-5 scale are depicted below in Table 6:

Table 6. Participation percentages of Vulnerability values in Fuzzy sets

0-4 ranks (values)	% participation – Safe set	% participation – Vulnerable set
1	1	0
1.5	1	0.15
2	0.5	0.3
2.5	0.25	0.45
3	0	0.6
3.5	0	0.75
4	0	0.9
4.5	0	1
5	0	1

Risk.

Risk represents a calculated value assigned to each information block `IB` and its corresponding variables, by aggregating the above mentioned Severity and Vulnerability ratings. Membership of an IB in Risk sets is calculated using Fuzzy Logic's IF-THEN rules. For clarity, all scales (Severity, Vulnerability and Risk) are divided in the same sets: "Low", "Medium" and "High". An example of how Risk is calculated using Fuzzy Logic linguistic rules is given:

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

Table 7 shows the fuzzy logic output for Risk, based on the aggregation of Severity and Vulnerability.

Table 7. Severity x Vulnerability = R - Risk sets

Vulnerability \ Severity	Low	Medium	High
Safe	Trifle	Trifle	Average
Vulnerable	Average	High	High

Risk output is basically the output of the fuzzy logic process, depicted in a way that provides a crisp, single-valued number to ease the end-users understanding. Fuzzy Logic is basically a logical system that expresses situations with multivalued results that do not have distinct boundaries. In our case, Fuzzy Logic is used to understand the danger-level set (rank) that an IB belongs to. Since an execution path is comprised by multiple method execution that may belong to different danger-level rank sets, pinpointing the exact set to classify a path can be tricky. Even more, combining that classification output (Severity) with Vulnerability outputs leads to even bigger "fuzziness" concerning the overall Risk of each IB.

The fuzzy membership sets that define the Overall Risk sets are depicted below in Figure 7:

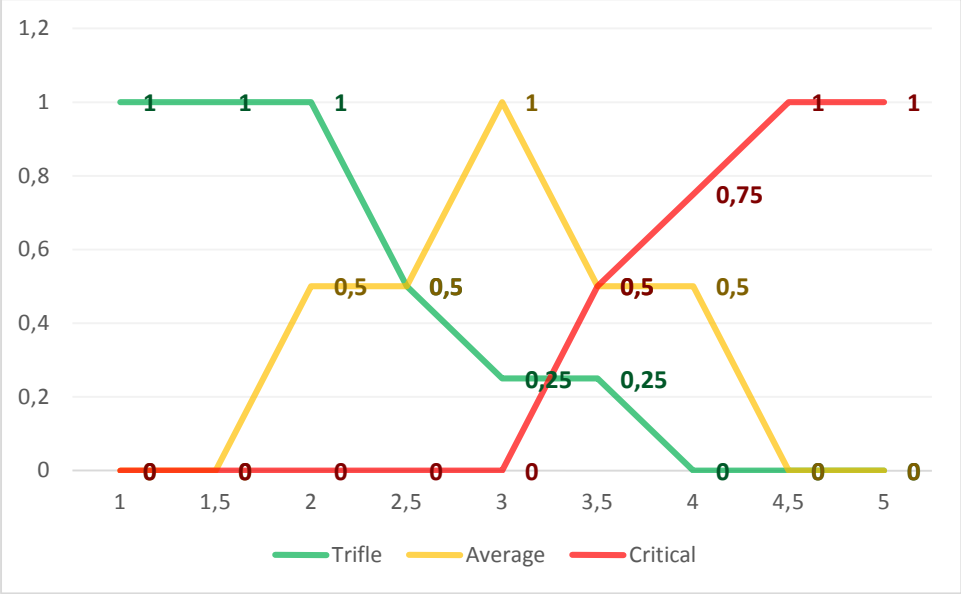

Fig. 7. Risk membership sets for Trifle (Low), Average and Critical (High) Risk values

The participation of values into membership sets are also depicted below in Table 8 for clarification purposes:

Table 8. Participation percentages of Risk values in Fuzzy sets

% participation	Trifle	Average	Critical
1	1	0	0
1.5	1	0	0
2	1	0.5	0
2.5	0.5	0.5	0
3	0.25	1	0
3.5	0.25	0.5	0.5
4	0	0.5	0.75
4.5	0	0	1
5	0	0	1

5 Experiments and results

5.1 Entroine's example run

Let us say that Entroine runs on a small sample application based on the code in Figure 1. The method steps would be the following:

1. Control Flow Graph analysis shows that there are 5 execution paths from an input vector (method that accepts method to a sink (method that executes data).
2. 5 IBs are created, one for each execution path containing all necessary info about the code traversed in each IB's execution path.
3. Severity calculations
    a. The overall prior entropy of the program is computed (`Pr(C)`) based on the prior categorization of each IB into Severity ranks/subsets.
    b. For each feature (method) in the taxonomy, the program's Entropy Loss is calculated for the gathered paths, based on the number of paths that contain each taxonomy method, as seen in Figure 8 below.
    c. The highest entropy loss detected points to the Severity rank (method subset) that best describes the corresponding IBs and the overall danger level of the application; if a vulnerability were to manifest on these paths.
4. Vulnerability calculations
    a. IBs are assigned to specific Vulnerability ranks based on the analysis of control flow statements and the sanitization of input data.
5. Severity and Vulnerability output values are fed to the Fuzzy Logic system.
6. The fuzzy membership of Risk is calculated, based on fuzzy set system.
7. Defuzzification is applied on the resulting Risk membership set to get a specific value for each detection. This is done to aid end-users in understanding Risk output (users understand a 3.7/5 scaled output quite better than a membership percentage in danger levels).
8. User can select IBs flagged and check their Risk analysis.

```
//////  Entropy Loss calculation //////

Entropy Loss calculation duration: 0 milliseconds
Entropy Loss for getInputStream in Rank 0: 0.0
Entropy Loss for getInputStream in Rank 1: 0.0
Entropy Loss for getInputStream in Rank 2: 0.0
Entropy Loss for getInputStream in Rank 3: 0.31127812445913283
Entropy Loss for getInputStream in Rank 4: 0.31127812445913283
Entropy Loss for readLine in Rank 0: 0.0
Entropy Loss for readLine in Rank 1: 0.0
Entropy Loss for readLine in Rank 2: 0.0
Entropy Loss for readLine in Rank 3: 0.12255624891826566
Entropy Loss for readLine in Rank 4: 0.12255624891826566
Entropy Loss for getProperty in Rank 0: 0.0
Entropy Loss for getProperty in Rank 1: 0.0
Entropy Loss for getProperty in Rank 2: 0.0
Entropy Loss for getProperty in Rank 3: 0.0
Entropy Loss for getProperty in Rank 4: 0.8112781244591328
Entropy Loss for exec in Rank 0: 0.0
Entropy Loss for exec in Rank 1: 0.0
Entropy Loss for exec in Rank 2: 0.0
Entropy Loss for exec in Rank 3: 0.0
Entropy Loss for exec in Rank 4: 0.0
```

Fig. 8. Screenshot of Entroine Severity calculation outputs in Eclipse Console

5.2 Experimental results on the Juliet Test Case

In order to test our approach, we needed appropriate AUTs to analyze. We considered whether we should use open-source software or "artificially made" programs, such as those usually used for benchmarking program analysis tools. Both options are characterized by various positive characteristics and limitations.

In choosing between real AUTs and artificial code for our purpose, we endorsed NSA's principles from [28, 29] were it states that "the benefits of using artificial code outweigh the associated dis-advantages". Therefore, for preliminary experimentation with Entroine we have opted using the Juliet Test Case suite, a formal collection of artificially-made programs packed with exploits [33].

The Juliet Test Suite is a collection of over 81.000 synthetic C/C++ and Java programs with a priori known flaws. The suite's Java tests contain cases for 112 different CWEs (exploits). Each test case focuses on one type of flaw, but other flaws may randomly manifest. A bad() method in each test-program manifests an exploit. A good() method implements a safe way of coding and has to be classified as a true negative. Since Juliet is a synthetic test suite, we mark results as true positive, if there is an appropriate warning in flawed (bad) code or false positive, if there is an appropriate warning in non-flawed (good) code, similarly to [1].

This testing methodology is developed by NIST. We focus on exploits from user input, whereas other categories are not examined (e.g. race conditions). Table 9 below provides a list of all Weakness Class Types used in the study. The middle column depicts the categories of exploits on which Entroine is tested (e.g. HTTP Response/Req Header-Servlet (add): exploits that manifest on servlets when adding HTTP headers in responses and requests):

Table 9. Weakness Classes – CWE

Weakness - CWE	Types of weaknesses analyzed	No. of tests
CWE-113	HTTP Response/Req HeaderServlet (add) HTTP Response/Req Cookie Servlet HTTP Response/Req HeaderServlet (set)	15
CWE-78	Operating System Command_ Injection	15
CWE-89	SQL Injection_connect_tcp SQL Injection_Environment_ execute SQL Injection_Servlet_execute	15

We ran Entroine on a set of vulnerable sample programs from the CWE categories depicted in Table 9. Our test data set consists of 45 total Juliet programs, 15 cases from each CWE category depicted in Table 9. Each bad method with an exploit will have to produce a True Positive (TP), whereas all good methods will have to represent True Negatives (TN). Overall 178 tests (TP+TN) were included in all programs: 45 exploits and 133 cases of safe implementations (TNs). Entroine flags detections when both Severity and Vulnerability ranks for an IB are ranked above the Low zone (Risk >=3). Table 10 shows the overall results of our tests and, consequently, the accuracy of the tool:

Table 10. TP, TN, FP detection rate (80 samples)

Weakness Class - CWE	TP Rate	TN Rate	TP +TN	All tests	No. of programs
CWE samples	45/ 45	133/ 133	178	178	45
Accuracy	TP = 100% , FP = 0%				

Table 11 provides a more detailed view of the results shown in Table 10. Table 11 depicts all tests per category of Juliet programs whereas Table 7 is an overall look on the results. 15 differentiated tests from each category where chosen for Entroine's preliminary proof-of-concept:

Table 11. Detection rates for each Weakness Type

Weakness Class - CWE	TP	TN	TP + TN	All tests	No. of programs
CWE-89: SQL Injection	15/15	51/51	66	66	15

CWE-78: OS Command Injection	15/15	28/28	43	43	15
CWE-113: HTTP Response Split	15/15	54/54	69	69	15

6 Conclusions

Entroine is in pre-alpha stage. Tests act as proof-of-concept statistics, since testing real-world, big applications is not yet feasible due to package complexity, external libraries, etc. State explosion remains an issue, a problem inherited by the used analysis techniques. Yet, state explosion seems manageable using source code classification, in order to focus on specific variable paths. This is aided by Severity ranking.

Another limitation of Entroine is that it cannot detect errors based on variables' context. This can be realized by introducing semantic constructs to analyze information behind input data. A formal comparison with known tools is, therefore, needed.

We plan to use our technique to test real-world code used in cyber-physical systems (e.g. high level code that manipulates devices through SCADA systems). This will work as an adequate extension to previous work of ours [39].

Entroine runs relatively fast in comparison to what it has to do. Table 12 depicts execution times.

Table 12. Entroine's execution times

Execution time (per 15 tests)	~129 sec
Entropy Loss calculation (per test)	~1 msec
Static analysis (per test)	~5 sec

All tests were ran on an Intel Core i5 4570 PC (3.2 GHz, 8GB RAM). A link to Entroine's taxonomy and example files can be found at: http://www.infosec.aueb.gr/Publications/Entroine_files.zip

References

1. Okun V., Delaitre O., Black P.: Report on the Static Analysis Tool Exposition (SATE) IV, NIST Special Publication 500-297 (2013)
2. Rutar N., Almazan, C., Foster, S.: A Comparison of Bug Finding Tools for Java. In: Proc. of the 15th International Symposium on Software Reliability Engineering. IEEE Computer Society, USA (2004)
3. Livshits V., Lam M.: Finding security vulnerabilities in Java applications with static analysis. In: Proc. of the 14th Usenix Security Symposium (2005)
4. Ayewah, N. Hovemeyer, D. Morgenthaler, J., Penix, J., Pugh, W.: Using Static Analysis to Find Bugs. In: Software, IEEE , vol.25, no.5, pp.22-29 (2008)
5. CodePro, https://developers.google.com/java-dev-tools/codepro/doc/
6. UCDetector, http://www.ucdetector.org/
7. Pmd, http://pmd.sourceforge.net/
8. Tripathi A., Gupta A.: A controlled experiment to evaluate the effectiveness and the efficiency of four static program analysis tools for Java programs. In: Proc. of the 18th International Conference on Evaluation & Assessment in Software Engineering. ACM (2014)
9. Hovemeyer D., Pugh W.: Finding bugs is easy. In SIGPLAN Not. 39, 12, pp. 92-106 (2004)
10. Jovanovic N., Kruegel C., Kirda E.: Static analysis for detecting taint-style vulnerabilities in web applications. In: Journal of Computer Security, No. 5, IOS Press (2010)
11. Weiser M.: Program Slicing. In: Proc. of the International Conference on Software Engineering, pp. 439–449 (1981)
12. Stergiopoulos G., Tsoumas V., Gritzalis D.: On Business Logic Vulnerabilities Hunting: The APP_LogGIC Framework. In: Proc. of the 7th International Conference on Network and System Security. Springer, 236-249 (2013)
13. Zhang X., Gupta N., Gupta R.: Pruning Dynamic Slices with Confidence. In: Proc. of the Conference on Programming Language Design and Implementation, pp. 169–180 (2006)
14. Cingolani P., Alcala-Fdez J.: jFuzzyLogic: A robust and flexible Fuzzy-Logic inference system language implementation". In: Proc. of the IEEE International Conference on Fuzzy Systems, 1-8 (2012)

15. Doupe A., Boe B. Vigna G.: Fear the EAR: Discovering and Mitigating Execution after Redirect Vulnerabilities. In: Proc. of the 18th ACM Conference on Computer and Communications Security. ACM, USA, pp. 251-262 (2011)
16. Balzarotti D., Cova M., Felmetsger V., Vigna G.: Multi-module vulnerability analysis of web-based applications. In: Proc. of the 14th ACM Conference on Computer and Communications security. ACM, USA, 25-35 (2007)
17. Albaum G., The Likert scale revisited. In: Market Research Society Journal, vol. 39, pp. 331-348 (1997)
18. Ugurel S., Krovetz R., Giles C., Pennock D., Glover E., Zha H.: What's the code?: automatic classification of source code archives. In: Proc.of the 8th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining, ACM, USA pp. 632-638 (2002)
19. Abramson N.: Information Theory and Coding. McGraw-Hill, USA (1963)
20. Etzkorn L., Davis, C: Automatically identifying reusable OO legacy code. In: IEEE Computer, pp. 66-71 (1997)
21. Glover E., Flake G., Lawrence S., Birmingham W., Kruger A., Giles L., Pennoek D.: Improving category specific web search by learning query modification. In: Proc. of the IEEE Symposium on Applications and the Internet, IEEE Press, USA, pp. 23-31 (2001)
22. Stoneburner G., Goguen A.: SP 800-30. Risk Management Guide for Information Technology Systems. Technical Report. NIST, USA (2002)
23. OWASP: The OWASP Risk Rating Methodology, www.owasp.org/ index.php/OWASP_Risk_Rating_Methodology
24. Leekwijck W., Kerre E.: Defuzzification: Criteria and classification. In: Fuzzy Sets and Systems, vol. 108, issue 2, 159-178 (1999)
25. Java API: Java Standard Edition 7 API Specification, http://docs.oracle.com/javase/7/docs/api/
26. Gosling J., Joy B., Steele G., Bracha G., Buckley A.: The Java Language Specification, Java SE 8 Edition, http://docs.oracle.com/javase/specs/jls/se8/ html/index.html
27. Harold E.: Java I/O, Tips and Techniques for Putting I/O to Work. O'Reilly (2006)
28. National Security Agency (NSA): On Analyzing Static Analysis Tools. Center for Assured Software, National Security Agency (2011)
29. National Security Agency (NSA): Static Analysis Tool Study-Methodology. Center for Assured Software (2012)
30. Yang Y., Pederson J.: A comparative study on feature selection in text categorization. In: Proc. of the 14th International Conference on Machine Learning (ICML'97), 412-420 (1997)
31. BCEL, Apache Commons BCEL project page. http://commons. apache.org/proper/commons-bcel/
32. Dahm, Markus, J. van Zyl, and E. Haase: The bytecode engineering library (BCEL) (2003)
33. Boland T., Black P.: Juliet 1.1 C/C++ and Java Test Suite. In: Computer, vol. 45, no. 10, pp. 88-90 (2012)
34. Stergiopoulos, G., Tsoumas, B., Gritzalis, D.: Hunting application-level logical errors. In: Proc. of the Engineering Secure Software and Systems Conference. Springer (LNCS 7159), 135-142 (2012)
35. Stergiopoulos G., Katsaros P., Gritzalis D.: Automated detection of logical errors in programs". In: Proc. of the 9th International Conference on Risks and Security of Internet and Systems, Springer (2014)
36. Coverity SAVE audit tool, http://www.coverity.com
37. Mell P., Scarfone, K., Romanosky S.: Common Vulnerability Scoring System. In: Security & Privacy, IEEE, vol.4, no.6, pp.85-89 (2006)
38. The Common Weakness Enumeration (CWE), Office of Cybersecurity and Communications, US Dept. of Homeland Security, http://cwe.mitre.org
39. Stergiopoulos G., Theoharidou M., Gritzalis D.: Using logical error detection in RemoteTerminal 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, Springer, USA (2015)
40. CWE - CWSS, https://cwe.mitre.org/cwss/cwss_v1.0.1.html
41. CWSS 3.0 scoring system, https://www.first.org/cvss/specification-document
42. National Vulnerability Database (NVD), https://nvd.nist.gov/