

Detecting Real Software Faults Through Resource Consumption Anomalies: An Evaluation of the Tricorder Methodology

Caio Guimaraes Herrera[1(✉)], Simone R. S. Souza[1], Ricardo Ribeiro dos Santos[2], and Paulo Sergio Lopes de Souza[1]

[1] University of Sao Paulo (ICMC/USP), Sao Carlos, SP, Brazil
caio.herrera@outlook.com
[2] Federal University of Mato Grosso do Sul (UFMS), Campo Grande, MS, Brazil

Abstract. Software faults in production environments pose significant challenges to software reliability, as traditional testing approaches do not always account for non-deterministic or operational conditions. The Tricorder methodology addresses this gap by detecting software anomalies through the monitoring of resource consumption behavior, without requiring access to the application's internal logic. While previous studies validated the methodology against artificially inserted faults in controlled experiments, its effectiveness against real-world faults remains an open question. This paper presents the first evaluation of the Tricorder methodology on real software faults from the Defects4j dataset, using the TriAD (Tricorder-based Anomaly Detection) tool. A total of 55 faults were selected across seven open-source Java applications and eight fault classes. Results show that TriAD achieved a fault-detection rate of 33% and a control group correctness rate of 71%. A one-tailed Wilcoxon signed-rank test confirmed a statistically significant difference in the number of iterations required to detect actual faults versus control-group executions, producing an F1-score of 39.18%. These results demonstrate the potential of the Tricorder methodology to identify real software faults through resource-consumption anomalies, while highlighting challenges introduced by API constraints in the dataset that limited execution to unit tests. Future work will aim to evaluate TriAD in end-to-end application development scenarios.

Keywords: Software reliability · Anomaly detection · Software testing

1 Introduction

Software testing is essential for ensuring the quality of software systems, and remains one of the most widely used approaches in software engineering [1]. However, it is an expensive and time-consuming activity, as it requires detailed knowledge of program behavior and expected outputs to construct test cases. Moreover, traditional testing often fails to detect software faults caused by operational

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2027
O. Gervasi et al. (Eds.): ICCSA 2026, LNCS 16769, pp. 146–161, 2027.
https://doi.org/10.1007/978-3-032-30494-0_10

conditions or non-deterministic aspects of production environments [13,16]. In this context, reliability engineering supports the fault detection processes by providing strategies for monitoring and continuously testing these applications, even in production environments [21].

The literature emphasizes runtime behavior modeling combined with statistical profiling to monitor systems and detect anomalies without access to internal code logic [3,6,7]. These approaches support the continuous evaluation of a system's reliability to identify deviations in the consumption of machine resources, such as RAM, CPU and I/O.

In this scenario, the **Tricorder** methodology [14] is presented as a complementary approach to regular software testing that aims to identify software faults through the monitoring of resource consumption, the definition of expected profiles, and the detection of anomalies. To evaluate its effectiveness, the methodology was implemented using a set of scripts and applied to selected applications in a laboratory environment, with artificial faults added to them. The **Tricorder** methodology detected the majority of these faults, demonstrating its effectiveness. However, it still lacks exposure to real faults in applications that were not tailored for conducting laboratory experiments. There is a need to determine whether **Tricorder** would remain effective at detecting real software faults in a realistic scenario.

This paper provides the first exposition of the **Tricorder** methodology applied to real software faults using the **Defects4j** dataset. We implemented the methodology as a reliability tool, **TriAD** (Tricorder-based Anomaly Detection), adapted to the dataset's API to execute unit tests for each selected application. An experiment was conducted to evaluate the methodology's effectiveness in this scenario and its behavior during control-group runs.

The remainder of this paper is organized as follows: Sect. 2 details how the **Tricorder** methodology works. Section 3 presents previous and related work to this paper, as well as explains how the methodology has been validated to date. Section 4 briefly explains how the **TriAD** tool was built, while Sect. 5 details how the **Defects4j** dataset was used, what were the needed adaptations to the **Tricorder** methodology, and how was the experiment set up. Finally, Sect. 6 presents and discusses the obtained results, and Sect. 7 concludes this paper, while discussing limitations and opportunities for future work.

2 The Tricorder Methodology

The **Tricorder** methodology [14] is an approach for detecting software anomalies by monitoring resource consumption of a system under test (SUT) as an indication of a possible fault in the code. The methodology relies on a baseline SUT version with no known faults as the reference behavior for resource consumption. Then, a new version of the SUT is introduced, and the new consumption behavior will be compared with the defined reference. This approach should not be confused with the Tricorder platform, proposed by Sadowski et al. [17], which performs static program analysis; the two share only the name.

Tricorder is intended to be used as a complementary approach to regular testing activities, as it does not rely on any structural or code information from the SUT, nor is it concerned about the correctness between input and output data. Moreover, **Tricorder**'s independence from code knowledge is the main factor enabling its use in production environments.

To define the expected resource consumption behavior for a particular SUT, **Tricorder** allows the definition of different workloads, which serve as one of the input parameters for executing the methodology. As a result, for each provided workload, the following steps will be executed:

1. Begin with the reference SUT version, **S**, and execute it **N** times.
2. Each execution will be monitored, and resource consumption data will be sampled to create a profile.
3. The profiles for the reference version are stored for future use.
4. Then, a new version of the SUT, called **S'**, will also be executed **N** times, and for each execution, a new profile is created.
5. After the generation of the profiles, a clustering step is executed. Starting from the full set of profiles in **S**, the clustering algorithm groups them in an unsupervised manner.
6. Profiles for **S'** are added one by one to the set, and the clustering process is executed again.
7. If any iteration yields a group containing only **S'** profiles, **Tricorder** is considered to have detected an anomaly.

The clustering step is conducted using the **DAMICORE** methodology [18], which enables unsupervised grouping of generic data. The goal in this step is to detect similarities between different executions of the SUT, where each group represents a different way it can behave. Groups composed of reference profiles are assumed to represent regular behavior, since the reference version serves as the baseline. On the other hand, groups containing only profiles from the new version are considered a new behavior; in this context, an anomaly.

To validate the effectiveness of the **Tricorder** methodology for detecting resource consumption anomalies, Python 2 scripts were developed to implement it and used in laboratory experiments. Two main scripts implement the methodology: a **monitoring** script that executes the SUT and samples the resource consumption into usage profiles, and an **analysis** script that runs clustering and issues a verdict on the existence of an anomaly. The **analysis** script uses **Damicorepy** [5], a Python implementation of **DAMICORE** that allows customization of its parameters.

Figures 1 and 2 illustrate the expected behavior for the **Tricorder** scripts when dealing with the reference version and an anomalous version of the SUT, respectively.

To support the resource consumption sampling, Montes et al. provided a utility tool within the **monitoring** script, called **Thermometer** [14], that relies on the *PSUtil* library to collect CPU usage, allocated bytes in RAM, and the

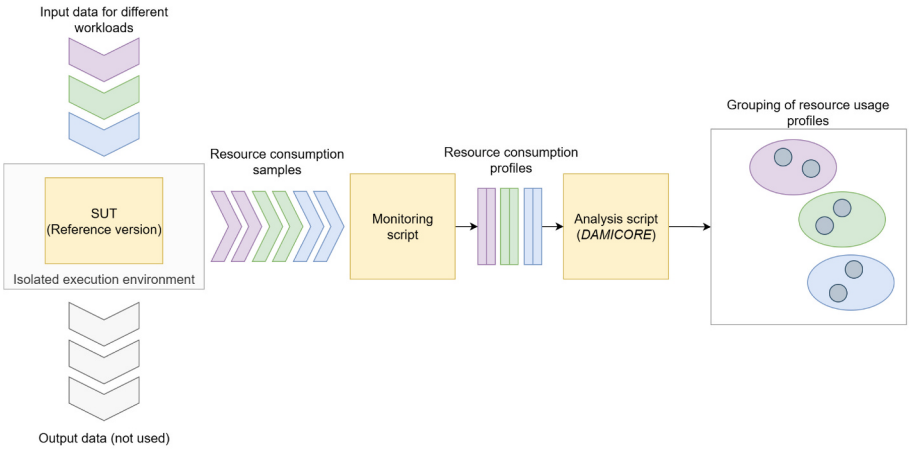

Fig. 1. Expected behavior for the **Tricorder** scripts, when running against the reference SUT version. Adapted from Herrera et al. [10].

number of I/O operations, which are used to build an execution profile. By default, metrics are sampled every 100 milliseconds. The authors reiterate that the **Tricorder** methodology allows the use of other metrics as well, and that the code can be extended to collect them.

Aiming to reduce false positives, Montes et al. also described and implemented a restrictive grouping strategy for the **analysis** script [14], in which an anomaly is reported only after being detected in two consecutive clustering iterations.

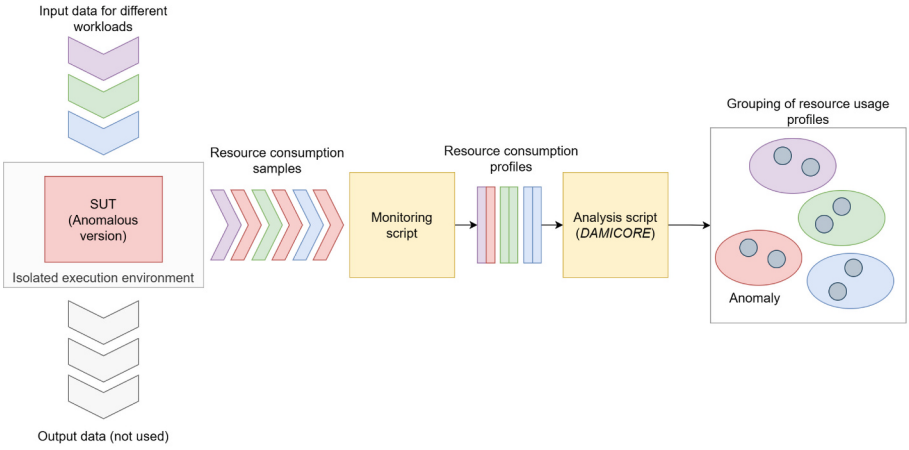

Fig. 2. Expected behavior for the **Tricorder** scripts, when running against an anomalous SUT version. Adapted from Herrera et al. [10].

3 Related Work

Based on the scripts presented in Sect. 2, a series of experiments was conducted to validate the effectiveness of the **Tricorder** methodology across different application domains and to investigate opportunities for performance improvement in its execution. Montes et al. started by instantiating **YATClient** and **YATServer** [14], a client-server application pair that simulates data stream traffic over the TCP protocol from the client to the server.

In this context, the **Tricorder** methodology monitored resource usage for the **YATServer** application. In the original application, faults were artificially inserted into the code, generating mutant versions. These new versions were used by **Tricorder** to detect anomalies in resource consumption behavior. Specifically, faults for seven distinct categories, inspired by real problems found in industrial environments [14], were inserted:

- **BCLEAN**: Excessive memory usage where allocated memory is not deallocated after use.
- **INFINITE**: An infinite loop causing unnecessary processing.
- **MONO**: A fault that limits parallelism, causing the application to run on a single thread.
- **SLEEP**: An unnecessary or excessively long pause in execution.
- **SWAP**: A fault in which the structure of the packages being sent does not have the same format as the expected structure to be received.
- **UNLOCK**: A synchronization fault in which a previously held mutex is not released.
- **NOBREAK**: A fault where a package of a specific type is processed as if it were from a different type.

Montes et al. showed that, despite some false positives, **Tricorder** detected all inserted mutants and achieved an F1-score of 91.4% in the experiment [14]. This is an important result for the methodology, as the **YATClient** and **YATServer** applications share characteristics of low-testability software [10], including processing data streams as their primary input.

In another study, Braga explored the effectiveness of the **Tricorder** methodology for different applications in the context of artificial intelligence and machine learning [4]. For these experiments, four distinct applications were selected as SUTs for the methodology:

- **BERT**[1]: Project that brings together different natural language processing (NLP) models, presenting a new and pre-trained model for language representation.
- **Chess-Alpha Zero**[2]: Project that uses reinforcement learning to reproduce a chess game, based on the **AlphaGo Zero** algorithm [22].

[1] https://github.com/google-research/bert.
[2] https://github.com/Zeta36/chess-alpha-zero.

- **Real-Time Voice Cloning**[3]: Project of a deep learning application that uses a text input and a five-second voice sample to generate audio.
- **Decision Tree**[4]: Standard machine learning project that uses the *scikit-learn* library to test different datasets [19].

In this multi-application scenario, Braga standardized the types of faults to be artificially inserted into the code. The generated mutants carried the following types of faults:

- **MODEL**: Machine learning configuration issues.
- **API**: Missing API call or calling an API at the wrong place.
- **TRAIN**: Missing training data.
- **PROCESS**: Issues when initializing an object or using a method.
- **LOGIC**: Logic errors in the code.
- **MEMORY**: Issues when using memory.
- **CONC**: Synchronization issues.

The **Tricorder** methodology was also effective in detecting the inserted anomalies in machine learning applications, as shown by Braga [4], where it correctly identified all mutants in **BERT** and **Chess-Alpha Zero** applications, while also detecting most of the faults in **Real-Time Voice Cloning** and **Decision Tree**, despite both applications presenting adverse conditions for the use of the methodology, where similar consumption behaviors were seen even for distinct workloads.

Durães also explored the effectiveness of the **Tricorder** methodology in detecting anomalies for different application domains [8]. Based on the same mutants defined by Montes et al. [14], Durães explored the following applications:

- **Crypto-Example**[5]: Cryptographic application, implemented as a proof of concept for RSA and AES methods, using the *OpenSSL* library.
- **JavaSort**[6]: Java project that implements the sorting algorithms *MergeSort* and *QuickSort*, in their parallel variants.
- **OBS Studio**[7]: Open-source video capture and transmitting tool, widely known by the community and popular among internet video creators.
- **Jet framework**[8]: A fluid simulation engine for computer graphic applications.

In their results, Durães highlighted the effectiveness of the **Tricorder** methodology in detecting anomalies for different domains, with an overall mean detection rate of 82% across all four applications [8], reiterating that the use

[3] https://github.com/CorentinJ/Real-Time-Voice-Cloning.
[4] https://doi.org/10.5281/zenodo.4723922.
[5] https://github.com/shanet/Crypto-Example.
[6] https://github.com/shaowdy/Multithreaded-Algorithms.
[7] https://obsproject.com/.
[8] https://github.com/doyubkim/fluid-engine-dev.

of **Tricorder** is not restricted to a particular class of applications. Moreover, a sample of the analyzed faults in the **OBS Studio** application came from real faults reported by developers, indicating that the methodology could also be effective in detecting real software anomalies rather than only artificial ones. These results serve as one of the motivations for this paper.

Finally, Fernandes et al. also explored the effectiveness of **Tricorder** for the **JavaSort** application, using parallel strategies to improve the methodology's performance [9]. As a result of this study, the authors showed that applying parallel strategies to the methodology can provide significant efficiency gains without compromising the quality of the results.

Despite all the studies providing concrete and positive evidence for the effectiveness of the **Tricorder** methodology in detecting software faults through resource consumption anomalies, there is still a need to explore the methodology's ability to detect anomalies in a real scenario, where faults are not artificially inserted into the experiment but are rather collected from the evolution of the software itself.

Beyond the **Tricorder** methodology, machine learning approaches have been applied to fault detection using the **Defects4j** dataset. For instance, Sekenov [20] evaluated multiple classifiers trained on static code metrics, achieving accuracy and F1-scores above 93% with Random Forest, and recall up to 96% with XGBoost. Similarly, Arab, Magel and Akour [2] framed fault localization as a multi-class classification problem, benchmarking deep learning, Random Forest, XGBoost, and LightGBM models, finding that test suite metrics consistently outperform static and dynamic metrics as fault predictors.

While these results are compelling, both approaches depend on access to the application's source code, limiting their applicability in scenarios where it is unavailable or proprietary. In contrast, the **Tricorder** methodology operates exclusively on runtime resource consumption, treating the application as a black box. This distinction makes **Tricorder** applicable in production environments where code inspection is not feasible—a trade-off that justifies a lower detection rate in exchange for broader applicability.

4 A Reliability Tool Based on the Tricorder Methodology

To expose **Tricorder** to real faults and facilitate its adoption, we implemented the methodology as a software reliability tool. The **TriAD** (Tricorder-based Anomaly Detection) tool [11] was developed based on the *RefTEST* architecture [15], enabling wider use of **Tricorder**. The adaptation of *RefTEST* to this new context is detailed in the work of Herrera et al. [10].

TriAD was developed as a web application divided into four modules: a front-end interface built with *Angular*; a back-end built in *Python 3*, *Flask* and *SQLite*; and two *Python 3* modules that replicate the original **monitoring** and **analysis** scripts. These technologies were selected for their simplicity, rapid prototyping

capabilities, and ability to reuse parts of the original code. This architecture is shown in Fig. 3, and the tool's source code is available on GitHub.[9].

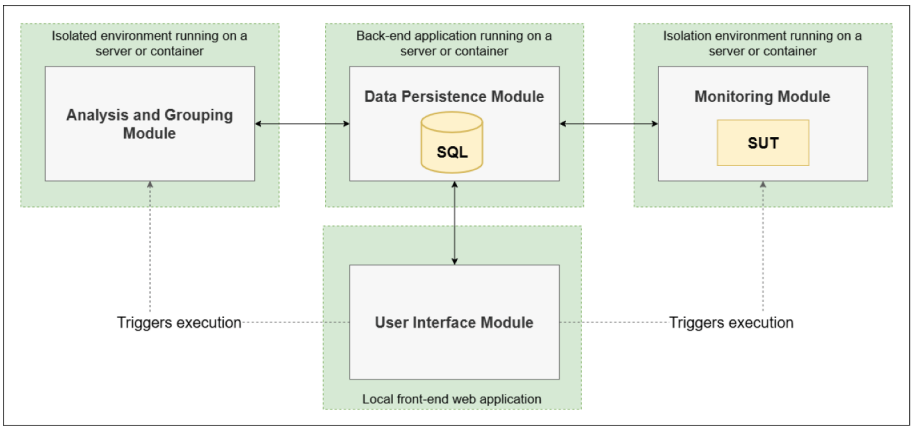

Fig. 3. Basic architecture used by the *TriAD* tool.

5 Experiment Design

To expose **Tricorder** (via **TriAD**) to real faults reported by users and developers, we selected **Defects4j** [12], a well-known dataset that summarizes faults, their descriptions, test cases that capture them, and the commits with fixes for several widely used open-source Java applications, including the main *Closure* compiler, *Apache* libraries *commons-lang* and *commons-math*, as well as *Mockito*, a framework for unit testing.

Spuy and Fischer classify 488 faults from **Defects4j** according to control and data flow representations [23]. This classification provides eight distinct classes of faults, listed below:

- **order**: Control flow fault, in which the order of instructions is incorrect;
- **jump**: Control flow fault, where an instruction A is followed by an instruction B when it should have been followed by an instruction C;
- **call**: Control flow fault, where a method is called when it should not be;
- **pred**: Control flow fault, in which a predicate condition (as an operator in a conditional or loop) is incorrect;
- **guard**: Control flow fault where a conditional or loop is missing to encapsulate a code block;
- **block**: Control flow fault, where there is a missing conditional statement or loop, and the code block that it encapsulates;

[9] https://github.com/caioherrera/triad-tricorder.

- **def**: Data flow fault, in which an incorrect value is written in a variable or the correct value is written in the incorrect variable;
- **use**: Data flow fault, where an incorrect variable is used for reading a value or an incorrect object is used for a method call.

Due to the large number of applications and faults in the dataset, a subset of **Defects4j** was chosen as the scope of this experiment. Seven out of seventeen applications were chosen; for each one, faults from each of the classes defined by Spuy and Fischer [23] were selected, with the exception of the **order** class for **Mockito**, since no fault of this class exists in that application. The selection of faults was conducted randomly, without prior knowledge of each one, with the only prioritization criteria being to choose faults with the least number of classes, in order to isolate the impact of each class as much as possible. The IDs for the selected faults can be seen in Table 1.

Table 1. IDs for the selected faults in the **Defects4j** dataset.

Project	order	jump	call	pred	guard	block	def	use
Chart	22	18	12	1	4	5	3	7
Closure	13	21	28	20	1	44	52	14
Jsoup	49	79	1	43	26	7	9	57
Lang	53	35	38	16	3	52	4	8
Math	86	1	35	15	28	44	2	34
Mockito	–	1	32	34	8	2	26	25
Time	7	15	26	19	3	24	4	16

To evaluate the effectiveness of the **Tricorder** methodology in this scenario, the following metrics were considered: anomaly detection rate and average number of iterations to detect an anomaly. We executed each application-fault combination 10 times and collected 30 profiles per execution. The reference version for each scenario was the application version with the fix to each fault, in order to provide the least number of changes between baseline and the test version.

To ensure that a high anomaly detection rate is reliable, it is also necessary to investigate the performance of **Tricorder** when evaluating the control group, i.e., a new execution of the reference version. The purpose of this evaluation is to check for false positives in the anomaly classification. It is expected that real anomalies require fewer iterations to be detected, while control group runs either produce no anomalies or require more iterations to produce one. For this analysis, we applied the **one-tailed Wilcoxon signed-rank** test to assess whether the median number of iterations differs significantly between fault detections and control group executions, as formalized in the hypotheses below:

- H_0: The number of iterations used in a control group execution is less than or equal to the number of necessary iterations to detect an anomaly.

– H_1: The number of iterations used in a control group execution is greater than the number of necessary iterations to detect an anomaly.

Despite the opportunity to explore distinct applications and fault classes in the context of the **Tricorder** methodology, using the **Defects4j** dataset poses additional challenges. Namely, it is not trivial to instantiate and execute end-to-end routines in the application. Instead, the dataset provides an API for executing individual test cases and sets of test cases for different commits. To work with these restrictions in the proposed scenario, we assume that the test cases provided by the developers exhibit behavior analogous to a conventional execution of the application and can therefore be used as an approximation of a regular application run. Still, a single unit test execution typically completes within milliseconds, well below the sampling interval established by **Tricorder**, which would yield profiles with too few data points to be meaningful. To address this, a script was created to repeatedly execute the test cases that capture each fault, collecting enough samples to generate a meaningful profile. While this makes the experiment viable, it introduces a threat to the validity of our results, that needs to be properly treated in future works.

6 Results and Discussion

The experiment was conducted in a virtual machine, hosted on a physical machine with the following specifications: a processor *Intel(R) Core(TM) i7-4790 CPU @ 3.60GHz*, with 8 cores and 32GB DDR3 RAM, running *Ubuntu 24.04.2 LTS Server*. The VM also had 8 cores, 32GB RAM, and the same OS. During the experiment, only a single user accessed the machine at a time, providing an additional level of isolation.

Table 2 presents the anomaly detection rate for each combination of fault class and application, as listed in Table 1. Overall, the **Tricorder** methodology, implemented as the **TriAD** tool, detected 33% of the anomalies. Although the detection rate is lower when compared to previous work, this result is significant: it shows **Tricorder**'s potential to identify real faults by detecting anomalies in resource consumption. Another interpretation of this result is that a tester could have caught 33% of the anomalies before they reach end users.

An important factor affecting this result is the adaptation of **TriAD** to the **Defects4j** API. As described in Sect. 5, the dataset allows for the execution of test cases that capture the fault, not necessarily an end-to-end application flow. In a real scenario where **Tricorder** would be used alongside an application under development, this barrier would naturally be resolved, suggesting a more effective use of **TriAD** in detecting real faults.

Table 3 presents, within the same conditions, the performance of the **TriAD** tool when running against the control group. Overall, **TriAD** achieved a 71% correctness rate in this scenario. This result, similar to those obtained in previous studies, reiterates the method's expected behavior and indicates that the lower anomaly detection rate may be due to subtle anomalies that have a limited

Table 2. Anomaly detection rate for **TriAD**, when executed against the **Defects4j** dataset.

Project	order	jump	call	pred	guard	block	def	use	Average
Chart	30%	60%	40%	30%	50%	20%	40%	50%	40%
Closure	40%	40%	10%	50%	40%	50%	20%	10%	33%
Jsoup	20%	20%	30%	50%	30%	10%	20%	30%	26%
Lang	20%	40%	30%	30%	50%	50%	30%	30%	35%
Math	30%	30%	40%	50%	30%	40%	20%	30%	34%
Mockito	–	20%	30%	30%	30%	30%	30%	60%	33%
Time	40%	60%	20%	30%	40%	40%	20%	20%	34%
Average	30%	39%	29%	39%	39%	34%	26%	33%	**33%**

impact on resource consumption and are hard to detect. This suggests that under more favorable conditions, the effectiveness of **Tricorder** could be increased.

Table 3. Correctness rate of the control group for **TriAD**, when executed against the **Defects4j** dataset.

Project	order	jump	call	pred	guard	block	def	use	Average
Chart	70%	90%	70%	70%	50%	90%	40%	50%	66%
Closure	90%	60%	80%	60%	80%	90%	80%	80%	78%
Jsoup	100%	70%	90%	70%	60%	60%	80%	80%	76%
Lang	50%	60%	80%	40%	70%	80%	80%	60%	65%
Math	80%	70%	90%	60%	60%	80%	90%	80%	76%
Mockito	-	70%	70%	50%	80%	70%	70%	40%	64%
Time	80%	70%	90%	60%	90%	50%	70%	70%	73%
Average	78%	70%	81%	59%	70%	74%	73%	66%	**71%**

Table 4 shows the average number of iterations needed to detect an actual fault. In contrast, Table 5 shows the average number of iterations used for control group executions. In an ideal scenario, we expect the former to be as few as possible, with the latter as close as possible to the maximum number of iterations. In our experiment, fault detection required an average of 26.5 iterations, while control group runs averaged 27.3 iterations.

Considering $\alpha = 0.05$, we can use the **one-tailed Wilcoxon signed-rank** on the data presented in Tables 4 and 5, obtaining an effective sample size of $N_{effective} = 53$. Furthermore, we get the value of $W_{crit} = 902$, with $W+ = 998.5$. From a normal approximation to a binomial distribution ($N_{effective} > 25$), we obtain $p \approx 0.006 < \alpha$, allowing us to reject the null hypothesis (H_0) in favor of the alternative hypothesis (H_1) and to assert that there is statistically

Table 4. Average number of iterations needed to detect a fault through **TriAD**, when executed against the **Defects4j** dataset.

Project	order	jump	call	pred	guard	block	def	use	Average
Chart	26.0	26.1	25.6	25.0	22.9	27.8	23.2	23.8	25.1
Closure	26.3	29.0	29.5	25.0	27.4	23.0	28.1	29.7	27.3
Jsoup	26.0	26.7	28.3	22.7	26.6	29.5	29.7	25.2	26.8
Lang	26.1	23.6	28.2	29.5	24.8	24.6	26.9	26.3	26.3
Math	28.1	27.0	26.0	22.6	29.6	26.2	27.5	26.4	26.7
Mockito	-	28.7	29.0	25.8	27.5	25.6	25.9	23.1	26.5
Time	28.2	24.3	27.2	29.1	25.6	26.9	27.0	29.3	27.2
Average	26.8	26.5	27.7	25.7	26.3	26.2	26.9	26.3	**26.5**

Table 5. Average number of iterations used for control group executions for **TriAD**, when executed against the **Defects4j** dataset.

Project	order	jump	call	pred	guard	block	def	use	Average
Chart	28.3	28.9	26.1	26.1	27.2	29.4	24.4	25.6	27.0
Closure	29.1	27.8	28.2	26.2	27.2	30.0	25.9	28.3	27.8
Jsoup	30.0	26.8	29.4	28.3	26.3	28.1	29.1	28.2	28.3
Lang	23.8	25.7	27.7	25.3	25.1	27.1	26.9	28.3	26.2
Math	27.2	27.0	28.9	26.4	26.5	28.7	29.7	28.1	27.8
Mockito	-	27.9	26.3	26.5	27.7	27.1	27.4	22.0	26.4
Time	27.4	26.3	29.7	27.7	29.2	26.1	27.4	27.8	27.7
Average	27.6	27.2	28.0	26.6	27.0	28.1	27.3	26.9	**27.3**

significant evidence suggesting that the median of iterations used in control group executions is greater than the median of iterations needed to detect actual faults. This is an important result, pointing to the effectiveness of the **Tricorder** methodology, implemented in the **TriAD** tool, in detecting real software faults.

It should be noted, however, that the practical gap between the two groups is narrow. Repeated invocations of short tests tend to produce resource profiles with low variance, reducing the separation between anomalous and normal behavior. In an end-to-end execution scenario, a wider and more reliable gap would be expected. This limitation is further reflected by the confusion matrix for **TriAD**, shown in Table 6. When evaluated against the selected faults, the **Tricorder** implementation achieved a precision of 52.44%, a recall of 31.27%, and an F1-score of 39.18%, indicating clear opportunities for improvement in future works.

Table 6. Confusion matrix for **TriAD**, when executed against the **Defects4j** dataset.

	Predicted Anomalous	Predicted Normal
Actual Anomalous	172 (True Positive)	378 (False Negative)
Actual Normal	156 (False Positive)	394 (True Negative)

7 Conclusion and Future Work

This work demonstrated the effectiveness of the **Tricorder** methodology, implemented as the **TriAD** reliability tool, when applied to real faults from well-known applications in the **Defects4j** dataset. We selected 55 faults from eight classes and seven distinct applications for investigation using the methodology. In this scenario, despite the need for adjustments and adaptations to the tool to work with the dataset-provided API, **Tricorder** achieved a fault detection rate of 33% while maintaining a correctness rate of 71% in control group executions. These results indicate the effectiveness of the **Tricorder** methodology, even though it still needs improvement, as it would be able to detect 33% of anomalies before the software reaches its final users in a production environment.

Beyond that, the **TriAD** tool needed, on average, 26.5 iterations to successfully detect an anomaly, whereas it used 27.3 iterations on average for control group executions. Through the **one-tailed Wilcoxon signed-rank** statistical test, we were able to confirm that there is a statistically significant difference between these two groups, allowing us to state that the median of iterations needed to detect a fault is smaller than the median of iterations used by the control groups, despite the small difference between them.

We obtained an F1-score of 39.18%, indicating room for improvement. This result reflects both the concessions made to enable the experiment and the inherent difficulty of detecting real faults, which are not designed to maximally impact resource consumption, unlike artificially inserted mutants. It is worth noting that **Tricorder** operates under fundamentally different constraints than white-box approaches, since it does not rely on access to the application's source code. Even so, a 33% detection rate means one in three real faults could be caught before reaching production, with no code instrumentation required—a meaningful result for scenarios where source-level analysis is infeasible.

Some limitations and threats to validity were identified during the experiment. Mainly, the necessary adaptations that were conducted to enable the use of the **Defects4j** dataset. While it provided a diverse set of applications and fault classes, it constrained the experiment to unit test execution - an approximation that, as discussed in Sect. 5, might have reduced the fault's impact on resource consumption. End-to-end runs could amplify anomalous behavior across the full execution path, potentially yielding better results, which requires further investigation.

Future work could address these challenges, while expanding the scope of the experiment to the full **Defects4j** dataset, enabling deeper analysis, including the investigation of fault classes that are more or less susceptible to resource consumption anomalies, and whether detection rates scale with application size or fault complexity. Additionally, the **TriAD** tool could be used in an end-to-end application development scenario, monitoring actual application runs rather than individual test executions.

Acknowledgments. The authors would like to thank CAPES (PROEX), CNPq (306719/2025-8, 311207/2025-1, 406941/2025-4), and FAPESP (2013/07375-0, 2019/06937-0, 2019/26702-8) for their financial support.

Disclosure of Interest. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Ammann, P., Offutt, J.: Introduction to Software Testing. Cambridge University Press, Cambridge (2016)
2. Arab, I., Magel, K., Akour, M.: Evaluating the predictive power of software metrics for fault localization. Computers **14**(6) (2025). https://doi.org/10.3390/computers14060222
3. Baah, G.K., Gray, A., Harrold, M.J.: On-line anomaly detection of deployed software: a statistical machine learning approach. In: Proceedings of the 3rd International Workshop on Software Quality Assurance, pp. 70–77. Association for Computing Machinery (2006). https://doi.org/10.1145/1188895.1188911
4. Braga, D.: (in portuguese) Avaliação do uso de agrupamento de dados de desempenho para apoiar o teste de software no domínio de aprendizagem de máquina. Master's thesis, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos (2022). https://doi.org/10.11606/D.55.2022.tde-09112022-160513
5. Cesar, B.K.M.: (in portuguese) Estudo e extensão da metodologia DAMICORE para tarefas de classificação. Master's thesis, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos, SP, Brasil (2016). https://doi.org/10.11606/D.55.2016.tde-16112016-170837
6. Cherkasova, L., Ozonat, K., Mi, N., Symons, J., Smirni, E.: Automated anomaly detection and performance modeling of enterprise applications. ACM Trans. Comput. Syst. **27**(3) (2009). https://doi.org/10.1145/1629087.1629089
7. Dean, D.J., Nguyen, H., Gu, X.: UBL: unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems. In: Proceedings of the 9th International Conference on Autonomic Computing. ICAC '12, New York, NY, USA, pp. 191–200. Association for Computing Machinery (2012). https://doi.org/10.1145/2371536.2371572

8. Durães, T.d.J.O.: (in portuguese) Avaliação do uso de perfis de desempenho aplicados no teste e software em diferentes domínios de aplicação. Master's thesis, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos (2022). https://doi.org/10.11606/D.55.2022.tde-05042023-144747
9. Fernandes, M.L., Souza, P.S.L.D.: Parallel implementation for anomaly detection in methodologies based on consumption profiles. In: 51st Latin American Informatics Conference (CLEI) (2025)
10. Guimaraes Herrera, C., Silva Montes, V., S. Souza, S.R., Ribeiro dos Santos, R., Lopes de Souza, P.S.: An architecture for a reliability tool applied to distributed systems. In: Computational Science and Its Applications – ICCSA 2025, pp. 332–348 (2025)
11. Herrera, C.G.a., Souza, S.d.R.S.d., Santos, R., Souza, P.S.L.d.: TriAD: a Tricorder-based tool for detecting anomalies in software. In: Proceedings of the 19th IEEE International Conference on Software Testing, Verification and Validation (ICST) – Testing Tools and Data Showcase Track (2026, to appear)
12. Just, R., Jalali, D., Ernst, M.D.: Defects4j: a database of existing faults to enable controlled testing studies for java programs. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. ISSTA 2014, New York, NY, USA, pp. 437–440. Association for Computing Machinery (2014). https://doi.org/10.1145/2610384.2628055
13. Liao, L., et al.: Using black-box performance models to detect performance regressions under varying workloads: an empirical study. Empirical Softw. Engg. **25**(5), 4130–4160 (2020). https://doi.org/10.1007/s10664-020-09866-z
14. Montes, V.S., Herrera, C.G., Durães, T.J.O., Delbem, A.C.B., de Souza, S.R.S.L., Souza, P.S.L.d.: Software reliability forecasting using anomalies in profiles of resource usage. In: 2025 IEEE 36th International Symposium on Software Reliability Engineering Workshops (ISSREW), pp. 189–196 (2025). https://doi.org/10.1109/ISSREW67781.2025.00071
15. Nakagawa, E., Simão, A., Ferrari, F., Maldonado, J.: Towards a reference architecture for software testing tools, pp. 157–162 (2007)
16. Patel, K., Hierons, R.: A mapping study on testing non-testable systems. Software Qual. J. **26** (2018). https://doi.org/10.1007/s11219-017-9392-4
17. Sadowski, C., Van Gogh, J., Jaspan, C., Soderberg, E., Winter, C.: Tricorder: Building a program analysis ecosystem. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, pp. 598–608 (2015). https://doi.org/10.1109/ICSE.2015.76
18. Sanches, A., Cardoso, J.M., Delbem, A.C.: Identifying merge-beneficial software kernels for hardware implementation. In: 2011 International Conference on Reconfigurable Computing and FPGAs, pp. 74–79 (2011). https://doi.org/10.1109/ReConFig.2011.51
19. Santos, S.A., Silveira, B., Durelli, V., Durelli, R., Souza, S., Delamaro, M.: On using decision tree coverage criteria fortesting machine learning models. In: Proceedings of the 6th Brazilian Symposium on Systematic and Automated Software Testing. SAST '21, New York, NY, USA, pp. 1–9. Association for Computing Machinery (2021). https://doi.org/10.1145/3482909.3482911
20. Sekenov, G.: Machine learning approaches for predicting defective java code using software metrics. Int. J. Digit. **1**, 1–9 (2026). https://doi.org/10.20508/9j0jxz49
21. Shieh, S.W., et al.: Reliability engineering in a time of rapidly converging technologies. IEEE Trans. Reliab. **73**(1), 73–82 (2024). https://doi.org/10.1109/TR.2024.3355905

22. Silver, D., et al.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm (2017)
23. van der Spuy, A., Fischer, B.: An anatomy of 488 faults from defects4j based on the control- and data-flow graph representations of programs. In: Proceedings of the 29th International Conference on Evaluation and Assessment in Software Engineering. EASE '25, New York, NY, USA, pp. 622–627. Association for Computing Machinery (2025). https://doi.org/10.1145/3756681.3757025