A classification of code changes and test types dependencies for improving machine learning based test selection

Downloaded from: https://research.chalmers.se, 2026-04-05 20:49 UTC

Citation for the original published paper (version of record):
Al Sabbagh, K., Staron, M., Hebig, R. et al (2021). A classification of code changes and test types dependencies for improving machine learning
based test selection. SIGPLAN Notices (ACM Special Interest Group on Programming Languages): 40-49. http://dx.doi.org/10.1145/3475960.3475987

N.B. When citing this work, cite the original published paper.

research.chalmers.se offers the possibility of retrieving research publications produced at Chalmers University of Technology. It covers all kind of research output: articles, dissertations, conference papers, reports etc. since 2004. research.chalmers.se is administrated and maintained by Chalmers Library

(article starts on next page)

A Classification of Code Changes and Test Types Dependencies for Improving Machine Learning Based Test Selection

Khaled Al-Sabbagh*
Miroslaw Staron
Regina Hebig
Francisco Gomes
khaled.al-sabbagh,miroslaw.staron,regina.hebig,francisco.de.oliveira.neto@gu.se
Chalmers | University of Gothenburg, Computer Science and Engineering Department
Gothenburg, Sweden

ABSTRACT

Machine learning has been increasingly used to solve various software engineering tasks. One example of their usage is in regression testing, where a classifier is built using historical code commits to predict which test cases require execution. In this paper, we address the problem of how to link specific code commits to test types to improve the predictive performance of learning models in improving regression testing. We design a dependency taxonomy of the content of committed code and the type of a test case. The taxonomy focuses on two types of code commits: changing memory management and algorithm complexity. We reviewed the literature, surveyed experienced testers from three Swedish-based software companies, and conducted a workshop to develop the taxonomy. The derived taxonomy shows that memory management code should be tested with tests related to performance, load, soak, stress, volume, and capacity; the complexity changes should be tested with the same dedicated tests and maintainability tests. We conclude that this taxonomy can improve the effectiveness of building learning models for regression testing.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

KEYWORDS

taxonomy, testing, continuous integration

ACM Reference Format:
Khaled Al-Sabbagh, Miroslaw Staron, Regina Hebig, and Francisco Gomes. 2021. A Classification of Code Changes and Test Types Dependencies for Improving Machine Learning Based Test Selection. In *Proceedings of the 17th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE '21), August 19–20, 2021, Athens, Greece.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3475960.3475987

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PROMISE '21, August 19–20, 2021, Athens, Greece
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8680-7/21/08...$15.00
https://doi.org/10.1145/3475960.3475987

1 INTRODUCTION

Software testing has evolved to successfully accommodate for the growing demand of higher product quality and faster delivery of releases [4]. Nevertheless, testing has been notoriously costly for its massive resource consumption - accounting for more than 50% of the development life cycle. Therefore, optimizing testing processes becomes pivotal for companies of all sizes to reduce the cost overhead and increase the velocity of software development.

An essential yet costly activity in any testing process is to perform regression testing, which ensures that no new faults in the system arise due to making new changes to the code base. However, performing regression testing demands a large amount of resources and a long execution time, which makes it infeasible to run all impacted test cases on each committed code change.

To address this problem of regression testing, a number of test case selection approaches have been proposed in the literature [15],[2], and [30]. These approaches seek to improve the effectiveness of test case selection by inferring statistical models that can potentially predict affected test cases given changes in the code base. However, a mutual drawback among these approaches is that they omit to take into account the dependencies between specific types of code changes (e.g., memory and algorithmic changes) and test case types (e.g., performance and security tests) when training predictive models. For example, Al-Sabbagh et al. [2] proposed building a machine learning (ML) model for test selection by mapping history executions of test cases and their relevant code changes without considering what types of test cases are sensitive to the changes in the source code. Similarly, Knauss et al. [15] proposed an automatic recommender that analyzes the frequency in which test cases fail on a particular day given code changes made to software modules irrespective of the types of changes made in the code and their dependencies with specific test case types.

Therefore, in this paper, we set off to fill this gap by developing a facet-based taxonomy of dependencies between code changes and test cases of specific types. We define a *dependency* as a relation where a change in the source code of a given type that triggers a failure in one or more test cases of different types. The contribution of this work is two-fold. First, it gears the testing efforts at software companies by allowing the execution of test cases that are in relation with the submitted code changes to the development repositories - thereby potentially reduce the time for testing. Second, it lays down the foundation for researchers to investigate, expand, and refine the identified dependencies. The addressed research question is:

RQ: To which degree do software testers perceive content of a code commit and test case types as dependent?

To address this research question, we constructed a taxonomy, linking the test case types and the categories of source code that can trigger these test cases. First, we began the taxonomy building by identifying and extracting data from the literature to find the test types and categories of code changes and to identify potential synergies between them. Then, we surveyed testers from software companies to construct and design the faceted taxonomy [16] Finally, for two categories, where the survey results were inconclusive, we conducted a workshop with the testers to find the strength of these dependencies.

2 RELATED WORK

Our work is related to studies on defect and testing taxonomies.

2.1 Defect Taxonomies

A widely applicable taxonomy in the software testing literature is Orthogonal Defect Classification (ODC), which was designed by Chillarege et al [7]. The ODC taxonomy defines attributes for the classification of failures. Its main purpose was to identify the root cause of defects and to provide quick feedback to developers about defects' cause in the software process. The ODC can also be used for early detection of faults in static analysis. Several defect taxonomies have been built on the ODC as a starting point to develop different domain-specific taxonomies. For example, Li et al. [18] presented an extended taxonomy of ODC and named it Orthogonal Defect taxonomy for Black-box Defects (ODC-BD). The taxonomy was designed by the motive of increasing testing efficiency and improving the analysis of black-box defects. Evaluated on the analysis of 1860 black-box defects that belong to 40 software projects, the results showed that using ODC-BD reduced the testing effort by 15% in one month compared to the testing efficiency when not using the ODC-BD. Another work conducted by Li et al. [20] adopted ODC to classify web errors for an improved reliability. Their taxonomy classified web errors according to their response code, file type, referrer type, agent type, and observation time.

The primary focus of all related work described above is to improve the quality of the code base by identifying the root cause of defects and to gain insights into the types of commits that developers commit. However, our work aims to improve the testing process by providing a taxonomy of code changes and test cases that can be used to build classifiers for test case selection.

2.2 Taxonomies in Software Testing

Software testing has often been confronted with the challenge of unveiling software defects under sever time pressure and limited hardware resources. Due to its importance and practical relevance, several software testing taxonomies have been proposed in the literature. In a systematic literature review study [6], Britto identified a number of studies that present taxonomies in the area of software testing. The majority of these taxonomies, however, provides a classification of the suitability of testing techniques in different contexts. For example, Novak et al. [23] developed a tree-based classification of features that are attributed to existing static code analysis tools. The taxonomy offers a classification of existing static analyzers based on the technology, availability of rules, and the programming languages that each tool supports. Similarly, Vegas et al. [29] classified a set of unit testing techniques and mapped their characteristics with project characteristics to aid the selection of suitable testing approaches based on the project's characteristics. The presented taxonomy comprised a number of criteria such as when to use the testing approach, who to use it, and where it can be used. Felderer and Schieferdecker [11] presented a classification for supporting the categorization of risk-based testing approaches and tailoring their usages depending on the context and purpose. The taxonomy classifies different risk drivers, risk assessments, risk-based test processes. All of these taxonomies provide a generic classification of the applicability of testing techniques in different software engineering projects. However, no taxonomy discusses the dimension of dependencies between code commits and test case types. Classifying these dependencies can potentially aid in the identification and execution of tests that are relevant to the committed code and hence counteract exhaustive testing efforts. The taxonomy presented in this study aims at filling this gap by identifying facets of dependency connections from the viewpoints of software testers.

3 RESEARCH METHOD

In this study, we follow the method proposed by Usman et al.[28] to guide the construction of the taxonomy. The method comprises of four phases: i) planning, ii) identification and extraction, iii) design and construction, and iv) validation.

3.1 Planning

The first phase in the adopted method involves six activities for planning the context of the taxonomy and defining its initial settings. Table 1 illustrates the outcome of each planning activity. Since the ultimate goal of this study is to gear the testing efforts by improving the selection of test cases, then the the knowledge area associated to the taxonomy is in the domain of software testing (A1). The second activity (A2) defines the objective of the taxonomy, which in our case is to identify the degree at which testers perceive dependency patterns between code changes and test case types. The subject matters (units of classifications) are categories of code changes and test case types (A3). A faceted-based approach is devised for creating the taxonomy (A4). The procedure for classifying the subject matters are qualitative and quantitative - literature review, survey, and discussions with testers in a workshop setting (A5). Finally, the basis of the taxonomy consists of categories of code changes and test case types drawn from the literature (A6).

3.2 Identification and Extraction

The identification and extraction phase involves identifying the main categories and terms used in the taxonomy. We begin the implementation of this phase by reviewing the literature in search for knowledge about the subject matters. For this purpose, we account for two inclusion criteria in our literature search. First, we wanted to include papers that discuss the impact of specific changes in the code on the quality of the system. Second, we were only interested in papers that were written in English and accessible. The challenge in this phase was to extract terms that are consistent and not interchangeably used in different research studies. Therefore, to

Table 1: Planning Activities

Id	Planning Activity
A1	The software engineering knowledge associated to the designed taxonomy is software testing.
A2	The main objective of the proposed taxonomy is to identify dependency patterns between code changes and test case types from the perspective of testers.
A3	The subject matters of the designed taxonomy are categories of code changes and test case types.
A4	The taxonomy was designed using a facet-based structure.
A5	The procedure used for classifying the subject matters was qualitative and quantitative.
A6	The basis of the taxonomy consists of code change categories and test case types drawn from the literature.

overcome this challenge we based our literature search on the set of recognized test case types defined in the international standard ISO/IEC/IEEE CD 2911901:2020(E) document[1] (presented in Section 4.1). That is, for each test case type in the ISO document, we searched for relevant papers that empirically investigate or theoretically discuss types of code changes that trigger a reaction among the test cases. The outcome of this phase was a list of six categories of code changes and 18 test case types. Further, and based on our literature search, we identified synergy links between the six code categories and the 18 test types (as depicted in Fig 2).

3.3 Design and Construction

This phase presents the relationships between the identified categories and describes how they were connected. Since the goal of the taxonomy is to answer the question of *To which degree do software testers perceive content of a code commit and a test case types as dependent?*, we decided to open up for the community of testers to seek their opinions about potential dependency patterns between the categories of code changes and test case types and to identify the strengths of the identified dependencies.

3.3.1 Survey. We began this phase by creating a survey and distributing an invitation email to software development companies that are affiliated to a Swedish consortium called 'Software Center'. The consortium comprises a total of fifteen companies and five universities that collaborate together to advance knowledge in seven different software engineering themes.

To mitigate the risk of receiving responses from different domain perspectives (e.g., web development), we decided to focus on surveying testers that specialize in the same domain area. Therefore, we sent the invitation email to five companies that are active in the development of embedded systems. The survey comprised two column lists. The first list included definitions of the test case types (see Section 4.1), whereas the second list included the categories of code changes (see Section 4.2). As a first task, all invitees were asked to provide a mapping between each test case type and category of code changes, where a mapping corresponds to a dependency between a single test case type and a category of code change.

The second task was for testers to propose and map additional test case types with categories of code changes that were not provided in the survey. The purpose was to mitigate the risk of missing out dependency patterns that testers perceive as important.

Finally, to achieve a better understanding of our target group of testers, all invitees were asked to mark the test case types that they exercise in their workplaces. Overall, we received a total of nine responses from nine testers working at the three software development companies. A general overview of the number of experienced testers for each test case type is provided in Fig 1.

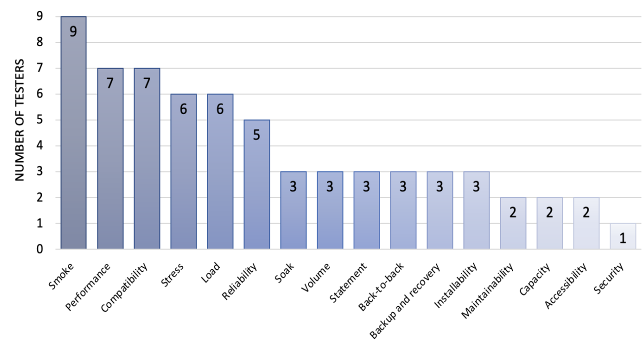

Figure 1: Number of Experienced Testers Per Each Test Type.

3.3.2 Workshop with Testers. The data from the survey provided us with the understanding of the dependencies. However, these dependencies could be of different strength and therefore we organized a workshop with the respondents to assess the strengths of dependencies for each test type to code changes. Three out of the nine respondents, who participated in the survey, and three other testers from another software company attended the workshop. Our analysis of the survey responses showed that the strongest dependencies were concentrated around the memory management and complexity categories of code changes. Therefore, we decided to focus on assessing the dependency strengths between these two categories of code changes and test case types in the workshop.

During the workshop, the entire group of testers discussed how sensitive each test type to the change of source code that affects 1) memory management or 2) complexity. The goal of the discussion was to gain an understanding of the dependency strengths from the viewpoint of testers, in the following scale:

(1) Not sensitive at all. This level was used when the testers judged that such a change would not trigger the test case to fail.
(2) Not very sensitive. This level was used when the testers judged that triggering a failure would be coincidental.
(3) Somewhat sensitive. This level was used when the testers judged that triggering would be under specific conditions.
(4) Sensitive. This level was used by the testers to indicate that a change under most conditions triggers a test case failure.
(5) Very sensitive. This level was used when the change should trigger the failure of the test case.

After discussing the sensitivity strengths, using the above scale, we asked the testers to justify their views about the sensitivity of each dependency by providing explanations for their ranking.

3.4 Validation

This phase ensures that the selected subject matters are clear and thoroughly classified Usman et al. [28]. This can be achieved using three distinct methods: Orthogonality demonstration, benchmarking and utility demonstration. Most of the taxonomies proposed in Software Engineering are evaluated via an utility demonstration, i.e., authors apply their taxonomy to an example Usman et al. [28]. In turn, benchmarking is used to compare the classification capabilities of different taxonomies. In both cases, the taxonomy needs to be applied in actual software artefacts. For this study, we cannot perform those types of validation because we do not have access to test cases or code changes from our industry partners. Therefore, we validate our taxonomy using an orthogonality demonstration. That is, we demonstrate and discuss the orthogonality between strongly dependent categories from the viewpoints of testers. The goal is to illustrate the unique classifications offered by our taxonomy. Based on this demonstration, we aim to highlight which types of tests map to unique types of code changes, as well as those dependencies that cover multiple types of tests.

4 RESULTS

This section presents the findings for the research question *To which degree do software testers perceive content of a code commit and a test case types as dependent?*

4.1 Test Case Types

In this paper, we decided to base our literature search for extracting code change categories on the list of test case types defined in this ISO/IEC/IEEE CD 2911901:2020(E) document[1]. This was done to overcome the challenge of encountering different terms of test case types that are used interchangeably in published articles. For example, the terms 'back to back'and 'differential'testing can be found and used interchangeably in the literature. Table 2 lists the definitions of all test case types that we used in our literature search. We used each test case type in the Table to search for relevant papers that empirically investigate or theoretically discuss the dependency between the relevant test case type and code changes.

4.2 Code Change Categories and Dependencies with Test Case Types

Our literature search returned a set of 16 relevant papers from which we could extract six different categories of code changes. These categories were: 1) Memory Management, 2) Complexity, 3) Design, 4) Dependency, 5) Conditional, 6) Data. Based on the literature search, we identified 21 dependency links between the six drawn categories of code and eight out of the 18 test case types defined in the ISO document, as shown in Fig 2. Each dependency corresponds to a relation where a change in one of the code category results in a failure of a test case of specific type.

We now define the identified categories of code changes and illustrate the effect of each on test case types by means of code examples written in the C++ language.

Table 2: Definitions of Test Case Types

Test Type	Definition
Smoke	Initial testing of the main functionality of a test item to determine whether subsequent testing is worthwhile.
Soak	Testing performed over extended periods to check the effect on the test item of operating for such long periods.
Stress	Testing performed to evaluate a test item's behaviour under conditions of loading above anticipated requirements.
Volume	Testing performed to evaluate the capability of the test item to process specified volumes of data in terms of capacity.
Load	Testing performed to evaluate the behaviour of a test item under anticipated conditions of varying loads.
Statement	Test design technique in which test cases are constructed to force execution of individual statements in a test item.
Maintainability	Evaluate the degree of effectiveness and efficiency with which a test item may be modified.
Security	Evaluate the degree to which a test item, and associated data, are protected against unauthorized access.
Performance	Evaluate the degree to which a test item accomplishes its designated functions within given time.
Capacity	Evaluate the level at which increasing load affects a test item's ability to sustain required performance.
Portability	Evaluate the ease with which a test item can be transferred from one environment to another.
Installability	Testing conducted to evaluate whether a set of test items can be installed as required in all specified environments.
Compatibility	Measure the degree to which a test item can function alongside other independent products.
Reliability	Evaluate the ability of a test item to perform its required functions under stated conditions for a period of time.
Accessibility	Determine the ease by which users with disabilities can use a test item.
Back-to-back	An alternative version of the system is used as an oracle to generate expected results for comparison from the same inputs.
Backup and recovery	Measures the degree to which a system state can be restored from backup within specified time in the event of failure.
Procedure	Evaluate whether procedural instructions for interacting with a test item to meet user requirements.

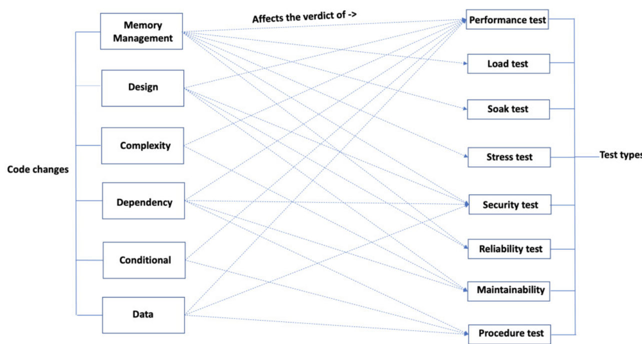

Figure 2: Extracted Categories of Code Changes and Their Dependency with Test Case Types.

Revision 1

```
int main()
{
    int* pListElementNext = new int();
    *pListElementNext = 100;
    std::cout << pListElementNext << endl;
    delete pListElementNext;
    return 0;
}
```

deleted

Revision 2

```
int main()
{
    int* pListElementNext = new int();
    *pListElementNext = 100;
    std::cout << pListElementNext << endl;
    return 0;
}
```

Figure 3: Code Example For Memory Management Change.

Memory management: This category of change involves groups that are concerned with the management of memory occupied by the system during run-time. Such changes include introducing/fixing memory leaks, buffer overflow, dangling pointers, and resource interference with multi-threading. The following test types would react to this category of change: performance[19], load[13], security[8][27], soak[14], stress[33], reliability[9] tests. A common memory leak scenario occurs when a developer allocates memory space using the *new* or *malloc* keywords, and misses freeing memory space after they were used. As the program grows in size, less memory becomes available and thereby a performance degradation is encountered. The code example in Fig 3 shows how the memory space allocated for pointer ’pListElementNext’ was unfreed from the memory after being used in revision 2.

Complexity: This category represents changes that add/reduce the time complexity of the program. It includes changes such as adding or removing loops, conditional statements, nesting blocks and/or recursions. The following test types have been identified to react to this category of change: performance [22][25], maintainability[12] [5] tests. Fig 4 shows a code example for finding the maximum integer element in an array. The function in the first revision takes a one dimensional array as input, whereas the second revision is modified to accept two-dimensional arrays. The nested loop added to the function in revision 2 would result in an increased time complexity order. Similar changes can potentially trigger performance degradation and thereby performance test failures.

Revision 1

```
int GetElementMax(int arr[], int size)
{
    int max = 0;
    for (int i = 0; i < size; i++)
    {
        if (arr[i] > max){
            max = arr[i];
        }
    }
    return max;
}

int main()
{
    int arr1D[5] = { 42, 44, 7, 12, 66 };
    std::cout << GetElementMax(arr1D, 2) << endl;
    return 0;
}
```

Revision 2

```
int GetElementMax(int arr[][1], int sizeofrows)
{
    int max = 0;
    for (int i = 0; i < sizeofrows; i++)
    {
        for (int j = 0; j < 2; j++)
        {
            if (arr[i][j] > max)
                max = arr[i][j];
        }
    }
    return max;
}
int main()
{
    int arr2D[1][1];
    arr2D[0][0] = 23;
    arr2D[1][0] = 12;
    std::cout << GetElementMax(arr2D, 2) <<endl;
    return 0;
}
```

modified, added, modified, added

Figure 4: Code Example For Complexity Change.

Design: This category involves changes that include code refactoring, adding or removing methods, classes, interfaces, and enumerators, and code smells. The following test types have been identified to react to this category of change: maintainability[12], performance[12], security[3], and reliability[17]. The code example in in Fig 5 illustrates a design change in a program that computes the sum of an array elements. The function ’CalculateRank’ was added in the modified revision to handle the task of summing up the array elements. Such design decisions reduce the amount of code lines in the program and thus improves its maintainability.

Revision 1

```
int main()
{
    int myArr[3] = {4, 5, 7};
    int sum = 0;

    sum += myArr[0];
    sum += myArr[1];
    sum += myArr[2];

    cout << sum << endl;
    return 0;
}
```

deleted

Revision 2

```
void CalculateRank(int ranks[3])
{
    int sum = 0;
    for (int i = 0; i < 3; i++)
    {
        sum += ranks[i];
    }
    std::cout << sum << endl;
}
int main()
{
    int myArr[3] = { 4, 5, 7 };
    CalculateRank(myArr);
    return 0;
}
```

added, added

Figure 5: A Code Example For Design Change.

Dependency: This category describes a code change that involves adding/ removing/ modifying a dependency to another module/ library. It can be importing/ removing/ modifying a new library, a new namespace, or a new class. Changes in the dependencies between software artefacts can trigger the following tests: maintainability[24], security[8], procedure[26], and performance[25].

Conditional: This category of change occurs when a logical operator or a comparative value in a condition is modified. A misuse in the logical expressions might result in generating the wrong outputs. Performance and procedure tests [25][26] were identified as dependent to this category of change.

Data change: This category involves 1) changing functions’ parameters, 2) passing parameters of incompatible types to modules/ functions, and 3) adding/ fixing assignments of incompatible types to

Test Case Types	Memory Management	Complexity	Design	Dependency	Data	Conditional	Total
	Code Change Categories						
Smoke Test	3	4	8	7	6	5	33
Performance Test	8	6	2	2	2	1	21
Soak Test	6	5	2	1	3	2	19
Load Test	8	5	1	0	2	2	18
Statement Test	1	1	2	4	4	6	18
Volume Test	6	5	1	0	1	2	15
Back-to-back Test	1	1	3	4	3	3	15
Stress Test	6	4	1	0	1	2	14
Maintainability Test	1	1	4	3	3	2	14
Reliability Test	3	3	2	3	2	1	14
Security Test	3	1	3	2	2	2	13
Capacity Test	6	4	1	0	1	0	12
Backup and recovery Test	1	1	3	3	2	2	12
Compatibility Test	0	0	2	3	1	1	7
Installability Test	0	1	1	2	1	1	6
Portability Test	0	0	1	2	2	1	6
Procedure testing	1	1	1	1	1	1	6
Accessibility Test	0	0	2	1	1	1	5
Functional tests	0	0	1	1	1	0	3
Regression tests	0	0	1	1	1	0	3

Figure 6: Testers' classifications of code changes and test case types. Each cell indicates the number of testers that perceive a relationship between the corresponding type of code changes and tests. Darker cells indicate stronger level of agreement between testers.

variables, casting statements, and array size allocations, and 4) modifying variable declarations. The following tests would react to such code changes: security[32], performance[25], and procedure[26].

4.3 Dependency Patterns and Strengths

4.3.1 Survey. Based on the types of tests and code changes extracted in the previous step, we created the survey. We sent our survey to 15 industry practitioners and received responses from nine participating testers (i.e., 60% response rate). Our analysis focuses on 1) examining whether testers had proposed additional types of test cases or categories of code change, and 2) examining the level of agreement and disagreement between the testers' perceived connections of types of tests and code changes. For instance, whether testers expect a connection between design changes and maintainability tests, as reported in literature. Fig 6 is a contingency table that depicts the testers' opinions about potential dependencies. Our analysis of the responses revealed the following observations:

- The strongest dependency patterns were mostly concentrated around the memory management and complexity categories of code changes.
- There was a general consensus between the testers about the mappings between performance, soak, load, stress, capacity, and volume tests and the six types of code change categories.
- Most of the discrepancies in the responses were in the classification of the design, dependency, and data categories.
- Two additional test types, i.e., not found in our literature extraction, were proposed by the testers: Regression and functional tests. The ISO/IEC/IEEE CD 2911901:2020(E) considers these two types of tests as testing activities, since these can be applied at any point in time irrespective of the testing level (unit, integration, system, and user acceptance) [1].

Due to the agreement between most testers about the connection between the complexity and memory management categories of code changes, we decided to focus the workshop on exploring the deeper connections between these two types of code changes and all types of tests. Focusing on only those two categories allowed us to capture the details of practitioners' perception about the connections between code changes and many types of tests such as process or human factors related to identifying those changes, or code constructs used in industry to classify those changes.

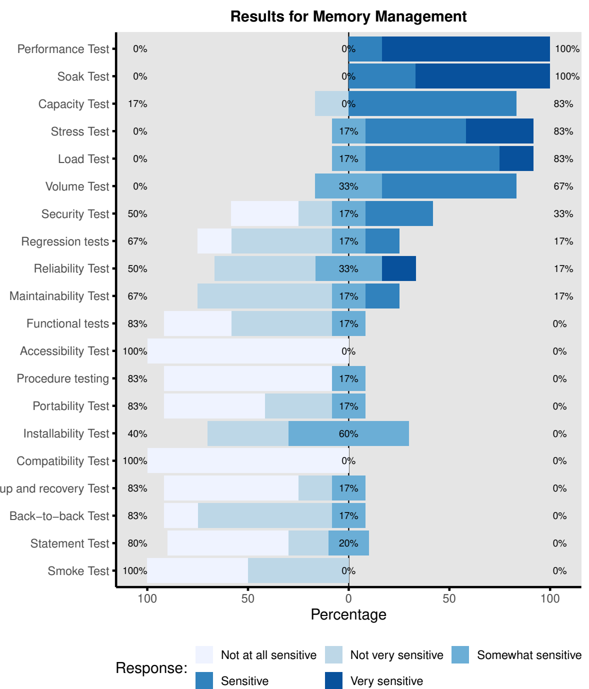

Figure 7: Diverging plot showing the strength of perceived connections between each test type and memory management changes. The percentages to the right indicate the proportion of testers that see a stronger relationship, in contrast to those that see a weaker relationship. Testers with a neutral view are shown as the percentage in the middle.

4.3.2 Workshop: We now present the results of the dependency scores given by the testers during the workshop. Figs 7 and 8 are diverging plots that show the sensitivity strengths of each test type to the memory management and complexity categories. By examining the sensitivity strength scores, of each test case type in Fig 7, we observe that the majority of the testers perceived six tests types to be mostly sensitive to memory management changes. Namely, performance, load, soak, stress, volume and capacity tests. Similarly, Fig 8 shows that performance, soak, load, statement, stress, volume, and maintainability tests were perceived as mostly sensitive to complexity related changes. In the remainder of this subsection, we present the main results of the discussions with the testers that explain their perspective on those connections.

4.3.3 Memory Management. Smoke, back-to-back, and statement tests: The respondents justified the low sensitivity strengths of these

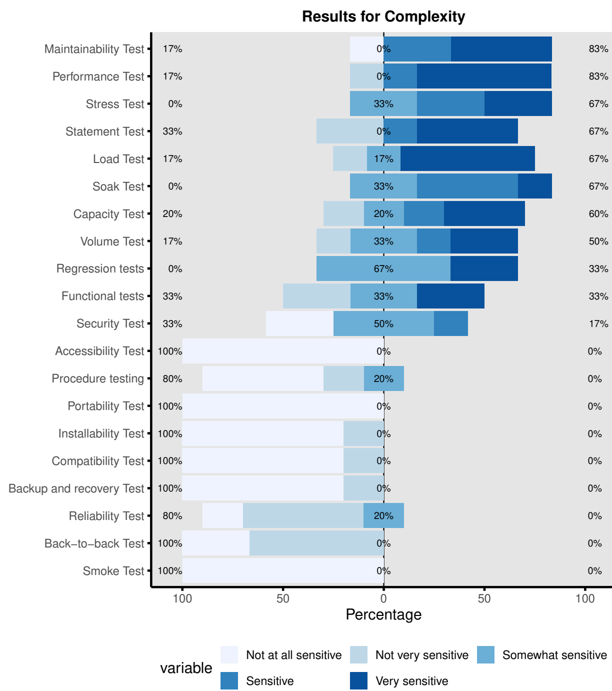

Figure 8: Diverging plot showing the strength of perceived connections between each test type and complexity changes. The percentages to the right indicate the proportion of testers that see a stronger relationship, in contrast to those that see a weaker relationship. Testers with a neutral view are shown as the percentage in the middle.

three test types to the fact that they focus on the functionality of the software system, rather than its qualities. One respondent linked the sensitivity of smoke tests to memory management changes to two specific scenarios: 1) when changing from one programming language to another, or 2) when doing major code refactoring.

> "*It's not that often that the smoke tests will break due to memory management changes but one possible scenario for this to happen is when we switch from C to C++ first we changed the compiler, then we started modernizing the code to use smart pointers. Another scenario is when we do major refactoring to optimize the code base.*" - Participant 1

Compatibility and portability tests: All testers agreed that these two types of tests are not sensitive at all to memory changes. The testers explained that these tests may only be triggered in the event of hardware failure in the environment. One opposing viewpoint considered memory management changes to have an effect on the stability of APIs used for information exchange in a shared environment, and thereby can trigger a failure in the two tests.

> "*Failure in these two types of tests can be explained by a device failure or in the way the APIs in the shared environments are handling concurrent requests, which often requires memory management changes.*" - Participant 1

Load, stress, soak, capacity, and volume tests: The majority of testers considered these test types to be very similar to performance tests. As a result, most of the justifications given about the sensitivity strengths of the five tests are somewhat similar. The testers explained that, in general, failure in one of the five test types can be triggered by memory related changes when expanding the functionality of existing classes.

> "*if you allocate more memory to expand an existing class then failure among performance tests might be triggered.*" - Participant 2

In addition, one tester emphasized that failure in any of these tests depends on the amount of changes made between releases and the information specified in the test oracle. That is, failures can only be captured when the amount of code changes made between releases is large.

> "*Failure in these tests depends on the oracle. If you just use the performance test to compare performance from the latest release then there might be no issues because the changes are too small, but if you do big changes then you might spot memory problems.*" - Participant 2

Installability tests: The sensitivity of this test type was perceived as moderate (somewhat sensitive) by 50% of the testers. These testers argued that installability testing is sensitive to memory management changes in situations where the development team decides to change from one operating system to another.

> "*When porting from a Windows environment to a Linux environment, we should make some memory changes, which trigger installability tests to fail.*" - Participant 3

Security tests: There was a disparity in the views of testers regarding the sensitivity of this test type. 33% of the testers perceived this test to be sensitive to memory changes, 17% perceived it to be somewhat sensitive, whereas 50% of testers perceive a low sensitivity to this type of test. Testers who considered this test type to be sensitive argued that memory changes lead to memory leaks which, if not properly managed, might expose the system to security breaches.

> "*I think that memory management changes could lead to things being exposed that should not be. For example exposing kernels space memory to be violated.*" - Participant 1

Disagreeing participants argued that resource leaks result in performance issues rather than security breeches. Further, they linked the sensitivity of security tests to the program domain.

> "*In specific domains, memory management is mostly handled on the cloud side providing the service. Internally, memory is not something that will trigger security tests to fail.*" - Participant 4

4.3.4 Complexity code changes. Performance, soak, load, volume, and stress tests: The majority of the testers ranked these types of tests to be either sensitive or very sensitive to complexity changes. As an argument for their ranking, the testers discussed that adding complexity changes such as nested loops will increase the cyclomatic complexity size in the system, which would in turn affects the system's response time.

> "*As the cyclomatic complexity increases, the response time of the system will also get impacted.*" - Participant 2

The remaining minority of the testers argued that developers are aware of the impact of adding complexity changes on performance. As such, it is highly unlikely that developers will commit complexity code changes without optimizing their code before testing.

> "*If developers are adding complexity consciously then there will be performance issues, but often the times, developers will address these complexity before even pushing their code for testing.*" - Participant 3

Maintainability test: All of the participants perceived this test type to be either sensitive or very sensitive to complexity changes in the code. One of the participants argued that adding more control paths in the system, such as loops and case blocks, leads to the development of larger and poorly structured software, which makes it more difficult and less efficient to maintain.

> "*Adding things like loops or method calls into the program increases its size and makes the task of debugging more difficult as the program evolves over time.*" - Participant 5

Security test: 50% of the participants indicated that security tests are somewhat sensitive to complexity changes. This was explained by the fact that adding recursion calls and loops to the code can potentially increase the size and modularity of the system under test, thus it will increase risk of missing security vulnerabilities. Conversely, around 30% of the participants believed that security tests are not sensitive at all to complexity changes. This contrasting view indicates that the links between security threats and increasing/decreasing code complexity are not clear for testers.

> "*I think it's not really a good thing to add complexity for security aware purposes. It is very important to understand what's going on in the code to be able to deal with things like security.*" - Participant 2

> "*adding loops will in no way expose the system to external threats and therefore no security tests will break if more loops are added - adding loops will not cause any vulnerabilities in the system.*" - Participant 6

The remaining 20% of the participants considered security tests to be sensitive to complexity changes, but did not provide any justification for this rank.

4.4 Resulting Taxonomy

The constructed taxonomy is based on the analysis of the overall agreement between testers who participated in the workshop and their justifications about each dependency. A test case type whose overall sensitivity to a code change was ranked as either sensitive or very sensitive by the majority of the testers was added to the taxonomy - provided that a justification for the dependency was made by one or more of the agreeing testers. Our analysis results of the workshop discussions show that testers have an aligned viewpoint with the classifications drawn from the literature in six of the dependency connections. Namely between: 1) memory management code and performance, load, soak, and stress tests, 2) complexity code and performance and maintainability tests. Beside these aligned dependencies, testers perceive six other dependencies to be in a strong causality relationship with the two categories of code. Those dependencies were between 1) memory management code changes and volume and capacity tests, 2) complexity code changes and load, soak, stress, and volume tests. Fig 9 shows the constructed taxonomy. We identify the strong and weak relationships mentioned by practitioners. Overall, the results show that the memory management code should be tested with tests related to performance, load, soak, stress, volume and capacity; the complexity changes should be tested with the same and additionally with the dedicated maintainability tests.

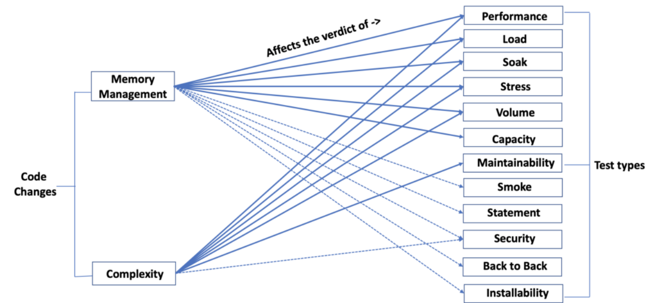

Figure 9: The final taxonomy of code changes and test case types. The solid connectors represent strong dependencies perceived by practitioners, whereas the dashed connectors correspond to those dependencies perceived as weak.

5 TAXONOMY VALIDATION

We evaluate our taxonomy by discussing the orthogonality of its classification. In other words, we illustrate how the chosen facets can support the prediction of connections between types of tests and code changes. Particularly, we emphasize the unique combinations found in our facets for supporting testers to classify the tests in connection with the code changes made. We frame the applicability of our taxonomy in relation to automated prediction of relationships between code and tests to support effective test orchestration.

5.1 Orthogonality of the Taxonomy's Facets

The majority of relationships are connected to the memory management code changes (11/18). That is not surprising as most of the types of tests found in literature cover system qualities. In fact, during workshops, practitioners rarely mention updates in functionalities (e.g., system requirements), except when discussing complexity changes. Memory management is exclusively connected with 5 test types, such that only 1 of those connections is strong (capacity tests). Consequently, those weak connections can be used to avoid overhead in test executions when focusing the verification of changes in memory management of software systems. Changes in complexity have fewer connections and most of them are actually shared with memory management (6/7), hence indicating a confounding factor between verifying changes in complexity to their impact on verifying memory management. Maintainability is only associated with complexity which is not surprising, since the complexity of a source code has impact on core aspects of maintainability such as testability and debugging [10]. The results shows one weak connection shared between both types of code changes, which is related to security testing. Still, practitioners did not seem

to have a consensus on how to handle security tests. Note that on Figs 3 and 4, security is ranked in the middle between the more explicit agreement and disagreements for both code categories. These contrasting views from practitioners on the purpose of security tests align with the findings drawn by Morrison et al. [21], where the authors highlighted a number of factors that impede the construction of effective vulnerability ML models.

5.2 Instrumenting Prediction of Dependencies

Table 3 breaks down memory management and complexity changes into specific types and their connection to specific code constructs. We choose C++ constructs because our study encompasses the domain of embedded systems. Future work aims at expanding the constructs to other programming languages such as Java or Python. Associating these code changes to specific code constructs enables automatic extraction and identification of code changes by using information from control version systems, such as git. The process of identifying and classifying code lines into their relevant categories can be instrumented using, for example, a tokenizer and a lexicon of vocabulary that contains a mapping between code tokens (constructs) and their relevant categories of code. For example, a code line that appears with a combination of the tokens *'delete, free, new, and malloc'* can be used to classify a code line as memory management related, since these tokens are used during objects' creation/destruction (Table 3). In contrast, automatically identifying and extracting types of tests is more challenging because those tests are used across different levels (e.g., unit or system) such that keyword extraction is inaccurate, particularly for higher levels of testing where tests are written in natural language (e.g., acceptance tests). Therefore, for this study, we assume that practitioners have access to the types of their tests, as part of their test process.

RQ. To which degree do software testers perceive content of a code commit and test case types as dependent?

The measured degree of perception among software testers suggests a strong dependency between performance, load, soak, stress, and volume tests and memory management related code changes. On the other hand, testers believe that soak, statement, back to back, security and installability tests are in weak dependencies with memory management code. Similarly, the majority of testers perceive the same set of strongly dependent test types with memory management changes to be dependent on complexity changes; in addition to maintainability tests and excluding capacity tests.

Based on these findings, test orchestrators that are keen on using ML models for test selection are encouraged to build their ML models on data that reflects the dependency patterns depicted in the presented taxonomy (Fig 9). Particularly, by mapping memory management and algorithmic complexity related code changes to the verdict of the strongly dependent test case types.

6 THREATS TO VALIDITY

In this section, we briefly discuss the limitations of our paper using the framework recommended by Wohlin et al. [31].

Conclusion Validity: Since this paper does not aim to provide a systemic survey, we did not use a formal protocol for conducting the literature review. Therefore, we cannot ensure that the selection of the code categories and test case types was unbiased. However, we minimize this risk by inviting testers to propose other types of code changes and test cases that are not provided in the survey invitation email. Moreover, there is a likelihood that we missed adding valid dependencies in the taxonomy as a result of 1) not discussing the sensitivity of all test types with testers, and 2) lack of experience among testers in some test case types. However, since the goal of this work is to study the dependency between code changes and test types, we accept this risk.

External Validity: The sample size of testers who participated in the survey and the workshop was small. Therefore, we acknowledge that the generalization of our findings might be delimited. However, the survey data and the workshop discussion provided some valuable insights into understanding the dependencies and sensitivity strengths of different test case types and code changes.

Internal Validity: The time span between the distribution of the survey and the the workshop was almost two months. This poses a threat with respect to the testers' comprehension of the terms and definitions that were used during the workshop (e.g., test case types). We mitigated this threat by providing definitions for all the terms used in the workshop. Another internal threat to validity is the likelihood that testers were influenced by the opinions of each other. However, since we construct our taxonomy based on a triangulated approach, we minimize the likelihood of this risk.

Construct Validity: This study builds on the assumption that there exists a dependency between code changes and test types. Nevertheless, there is a chance that such a dependency does not exist and that what we found was coincidental. We minimize this risk by constructing the taxonomy from the viewpoints of practitioners.

7 CONCLUSION AND FUTURE WORK

The taxonomy presented in this paper aims at classifying dependencies between categories of code changes and test case types. Exploring these dependencies can potentially contribute to the improvement of ML based test case selection approaches that use code analysis and test execution results. In this paper, we have observed strong dependencies between two categories of code changes and seven test case types. This knowledge can gear the test orchestration efforts by pinpointing and executing test cases that are in relation with the relevant changes in the source code. The strongest dependencies were captured between performance, load, stress, soak, volume and the two categories of code changes: memory management and complexity. On the opposite end of the spectrum, the weakest dependencies were found between smoke, back-to-back, installability, accessibility, portability, compatibility, and backup and recovery tests, and the two categories of code changes. Those test cases can be excluded from the suite when the tested code contains memory management and complexity changes only. As a future work, we plan to continue working on refining the presented taxonomy by investigating additional dependency patterns between other test case types and categories of code changes. Another important future work is to investigate potential dependency links between test script constructs and test execution outcomes of different types. Finally, we aim at evaluating the taxonomy presented in this study by using utility demonstrations on different software projects and programming languages.

Table 3: Types and Constructs Related to Memory Management and Complexity Code Changes.

Memory Management		
Subcategories	**Description**	**Code Constructs**
Dangling/Wild pointers	occurs when deleting an object from memory without altering the pointer that points to the object's location.	&variable, *variable, NULL, free
Memory leaks	occur when memory space is allocated but not freed. If such incidents occur, leaks will happen and could eventually cause the program to run out of memory resulting in a program halt.	delete, free, new, malloc
Buffer overflow	occurs when the data gets written past the boundaries of the buffer allocated in memory.	malloc, strcpy, gets, strcmp

Complexity		
Subcategories	**Description**	**Code Constructs**
Loops and conditions	repeating a sequence of instructions for n times until one or more conditions are satisfied. The repetition can occur in the form of multiple nested loops.	for, while, do, if, switch, case, break
recursion	Occurs when a function calls itself until an exit condition is satisfied.	

REFERENCES

[1] 2020. *ISO/IEC/IEEE International Standard - Software and systems engineering –Software testing–Part 1: Concepts and definitions.* Technical Report. 1–50 pages.

[2] Khaled Walid Al-Sabbagh, Miroslaw Staron, Regina Hebig, and Wilhelm Meding. 2019. Predicting Test Case Verdicts Using Textual Analysis of Committed Code Churns. In *(IWSM Mensura 2019)*, Vol. 2476. 138–153.

[3] Taimur Aslam. 1995. A taxonomy of security faults in the unix operating system. *Master's thesis, Purdue University* 199, 5 (1995).

[4] Arnon Axelrod. 2018. *Complete Guide to Test Automation.* Springer.

[5] Rajiv D Banker, Srikant M Datar, and Dani Zweig. 1989. Software complexity and maintainability. *Age* 11, 5.6 (1989), 3.

[6] Ricardo Britto. 2015. *Knowledge classification for supporting effort estimation in global software engineering projects.* Ph.D. Dissertation. Blekinge Tekniska Högskola.

[7] Ram Chillarege, Inderpal S Bhandari, Jarir K Chaar, Michael J Halliday, Diane S Moebus, Bonnie K Ray, and Man-Yuen Wong. 1992. Orthogonal defect classification-a concept for in-process measurements. *IEEE Transactions on software Engineering* 18, 11 (1992), 943–956.

[8] Fred Cohen. 1997. Information system attacks: A preliminary classification scheme. *Computers & Security* 16, 1 (1997), 29–46.

[9] Domenico Cotroneo, Roberto Pietrantuono, Leonardo Mariani, and Fabrizio Pastore. 2007. Investigation of failure causes in workload-driven reliability testing. In *Fourth international workshop on Software quality assurance: in conjunction with the 6th ESEC/FSE joint meeting.* 78–85.

[10] Michael Felderer, Bogdan Marculescu, Francisco Gomes de Oliveira Neto, Robert Feldt, and Richard Torkar. 2018. A Testability Analysis Framework for Non-functional Properties. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW).* IEEE, 54–58.

[11] Michael Felderer and Ina Schieferdecker. 2019. A taxonomy of risk-based testing. *arXiv preprint arXiv:1912.11519* (2019).

[12] Lingxiao Jiang, Zhendong Su, and Edwin Chiu. 2007. Context-based detection of clone-related bugs. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering.* 55–64.

[13] Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. 2008. Automatic identification of load testing problems. In *2008 IEEE International Conference on Software Maintenance.* IEEE, 307–316.

[14] Timo Karttunen. 2009. *Implementing Soak Testing for an Access Network Solution.* Ph.D. Dissertation. HELSINKI UNIVERSITY OF TECHNOLOGY.

[15] Eric Knauss, Miroslaw Staron, Wilhelm Meding, Ola Söder, Agneta Nilsson, and Magnus Castell. 2015. Supporting continuous integration by code-churn based test selection. In *2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering.* IEEE, 19–25.

[16] Barbara H Kwasnik. 1999. The role of classification in knowledge representation and discovery. (1999).

[17] Y Levendel. 1989. Defects and reliability analysis of large software systems: field experience. In *1989 The Nineteenth International Symposium on Fault-Tolerant Computing. Digest of Papers.* IEEE Computer Society, 238–239.

[18] Ning Li, Zhanhuai Li, and Xiling Sun. 2010. Classification of software defect detected by black-box testing: An empirical study. In *2010 Second World Congress on Software Engineering*, Vol. 2. IEEE, 234–240.

[19] Yepang Liu, Chang Xu, and Shing-Chi Cheung. 2014. Characterizing and detecting performance bugs for smartphone applications. In *Proceedings of the 36th international conference on software engineering.* 1013–1024.

[20] Li Ma and Jeff Tian. 2007. Web error classification and analysis for reliability improvement. *Journal of Systems and Software* 80, 6 (2007), 795–804.

[21] Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. 2015. Challenges with applying vulnerability prediction models. In *Proceedings of the 2015 Symposium and Bootcamp on the Science of Security.* 1–9.

[22] Adrian Nistor, Po-Chun Chang, Cosmin Radoi, and Shan Lu. 2015. Caramel: Detecting and fixing performance problems that have non-intrusive fixes. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 902–912.

[23] Jernej Novak, Andrej Krajnc, et al. 2010. Taxonomy of static code analysis tools. In *The 33rd International Convention MIPRO.* IEEE, 418–422.

[24] Ekaterina Razina and David S Janzen. 2007. Effects of dependency injection on maintainability. In *Proceedings of the 11th IASTED International Conference on Software Engineering and Applications: Cambridge, MA.* 7.

[25] Juan Pablo Sandoval Alcocer, Alexandre Bergel, and Marco Tulio Valente. 2016. Learning from source code history to identify performance failures. In *Proceedings of the 7th ACM/SPEC on International Conference on Performance Engineering.* 37–48.

[26] A. Sawant, Pranit H Bari, and P. Chawan. 2012. Software Testing Techniques and Strategies.

[27] Robert C Seacord and Allen D Householder. 2005. *A structured approach to classifying security vulnerabilities.* Technical Report. CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST.

[28] Muhammad Usman, Ricardo Britto, Jürgen Börstler, and Emilia Mendes. 2017. Taxonomies in software engineering: A systematic mapping study and a revised taxonomy development method. *Information and Software Technology* 85 (2017), 43–59.

[29] Sira Vegas, Natalia Juristo, and Victor R Basili. 2009. Maturing software engineering knowledge through classifications: A case study on unit testing techniques. *IEEE Transactions on Software Engineering* 35, 4 (2009), 551–565.

[30] Kaiyuan Wang, Chenguang Zhu, Ahmet Celik, Jongwook Kim, Don Batory, and Milos Gligoric. 2018. Towards refactoring-aware regression test selection. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE).* IEEE, 233–244.

[31] Claes Wohlin, Per Runeson, Martin Höst, Magnus C Ohlsson, Björn Regnell, and Anders Wesslén. 2012. *Experimentation in software engineering.* Springer Science & Business Media.

[32] Zhang Yan, Dong Guowei, Guo Tao, and Yang Jianyu. 2013. Taxonomy of Source Code Security Defects Based on Three-Dimension-Tree. In *International Conference on Computer and Computing Technologies in Agriculture.* Springer, 232–241.

[33] Jian Zhang, Shing-Chi Cheung, and Samuel T Chanson. 1999. Stress testing of distributed multimedia software systems. In *Formal Methods for Protocol Engineering and Distributed Systems.* Springer, 119–133.