Mutation Analysis for Cyber-Physical Systems: Scalable Solutions and Results in the Space Domain

Oscar Cornejo, Fabrizio Pastore, *Member, IEEE,* and Lionel C. Briand, *Fellow, IEEE*

Abstract—On-board embedded software developed for spaceflight systems (*space software*) must adhere to stringent software quality assurance procedures. For example, verification and validation activities are typically performed and assessed by third party organizations. To further minimize the risk of human mistakes, space agencies, such as the European Space Agency (ESA), are looking for automated solutions for the assessment of software testing activities, which play a crucial role in this context. Though space software is our focus here, it should be noted that such software shares the above considerations, to a large extent, with embedded software in many other types of cyber-physical systems.

Over the years, mutation analysis has shown to be a promising solution for the automated assessment of test suites; it consists of measuring the quality of a test suite in terms of the percentage of injected faults leading to a test failure. A number of optimization techniques, addressing scalability and accuracy problems, have been proposed to facilitate the industrial adoption of mutation analysis. However, to date, two major problems prevent space agencies from enforcing mutation analysis in space software development. First, there is uncertainty regarding the feasibility of applying mutation analysis optimization techniques in their context. Second, most of the existing techniques either can break the real-time requirements common in embedded software or cannot be applied when the software is tested in Software Validation Facilities, including CPU emulators and sensor simulators.

In this paper, we enhance mutation analysis optimization techniques to enable their applicability to embedded software and propose a pipeline that successfully integrates them to address scalability and accuracy issues in this context, as described above. Further, we report on the largest study involving embedded software systems in the mutation analysis literature. Our research is part of a research project funded by ESA ESTEC involving private companies (GomSpace Luxembourg and LuxSpace) in the space sector. These industry partners provided the case studies reported in this paper; they include an on-board software system managing a microsatellite currently on-orbit, a set of libraries used in deployed cubesats, and a mathematical library certified by ESA.

Index Terms—Mutation analysis, Mutation testing, Space software, embedded software, Cyber-physical systems

1 Introduction

From spacecrafts to ground stations, software has a prominent role in space systems; for this reason, the success of space missions depends on the quality of the system hardware as much on the dependability of its software. Mission failures due to insufficient software sanity checks [1] are unfortunate examples, pointing to the necessity for systematic and predictable quality assurance procedures in space software.

Existing standards for the development of space software regulate software quality assurance and emphasize its importance. The most stringent regulations are the ones that concern flight software, i.e., embedded software installed on spacecrafts, our target in this paper. In general, software testing plays a prominent role among quality assurance activities for space software, and standards put a strong emphasis on the quality of test suites. For example, the European Cooperation for Space Standardization (ECSS) provides detailed guidelines for the definition and assessment of test suites [2], [3].

Test suites assessment is typically based on code inspections performed by space authorities and independent software validation and verification (ISVV) activities, which include the verification of test procedures and data (e.g., ensure that all the requirements have been tested and that representative input partitions have been covered [4]). Though performed by specialized teams, such assessment is manual and thus error prone and time-consuming. **Automated and effective methods to evaluate the quality of the test suites are thus necessary.**

Since one of the primary objectives of software testing is to identify the presence of software faults, an effective way to assess the quality of a test suite consists of artificially injecting faults in the software under test and verifying the extent to which the test suite can detect them. This approach is known as *mutation analysis* [5]. In mutation analysis, faults are automatically injected in the program through automated procedures referred to as mutation operators. Mutation operators enable the generation of faulty software versions that are referred to as *mutants*. Mutation analysis helps evaluate the effectiveness of a test suite, for a specific software system, based on its mutation score, which is the

- *O. Cornejo, F. Pastore, and L. Briand are affiliated with SnT Centre for Security, Reliability and Trust, University of Luxembourg, Luxembourg. E-mail:{oscar.cornejo,fabrizio.pastore,lionel.briand}@uni.lu*
- *L. Briand also holds a faculty appointment with school of EECS, University of Ottawa.*

Manuscript received December 20, 2020; revised June 16, 2021.

percentage of mutants leading to test failures.

Despite its potential, mutation analysis is not widely adopted by industry in general and space system development in particular. The main reasons include its limited scalability and the pertinence of the mutation score as an adequacy criterion [6]. Indeed, for a large software system, the number of generated mutants might prevent the execution of the test suite against all the mutated versions. Also, the generated mutants might be either semantically equivalent to the original software [7] or redundant with each other [8]. Equivalent and redundant mutants may bias the mutation score as an adequacy criterion.

The mutation analysis literature has proposed several optimizations to address problems related to scalability and mutation score pertinence. For example, scalability problems are addressed by approaches that sample mutants [9] [10] or solutions that prioritize and select the test cases to be executed for each mutant [11]. Equivalent and redundant mutants can be detected by comparing the code coverage of the original program and its mutants [12], [13], [14], [15]. However, these approaches have not been evaluated on industrial, embedded systems and there are no feasibility studies concerning the integration of such optimizations and their resulting, combined benefits. For example, we lack mutants sampling solutions that accurately estimate the mutation score in the presence of reduced test suites; indeed, mutant sampling, which comes with a certain degree of inaccuracy, may lead to inaccurate results when applied to reduced test suites that do not have the same effectiveness of the original test suite.

Finally, there is no work on identifying and assessing approaches that are feasible and effective for embedded software in general and space software in particular. Such software is very different than other types of software systems (e.g., Java graphical libraries or Unix utility programs) in several ways. More precisely, space software – like many other types of embedded software within cyber-physical systems (CPS) [16], [17] – presents a combination of characteristics that altogether substantially limit the applicability of existing mutation testing optimizations. First, this type of software normally contains many functions to deal with signals and data transformation, which may diminish the effectiveness of approaches—both compiler-based and coverage-based—to identify equivalent and redundant mutants. Indeed, the proportion of equivalent and redundant mutants detected by compiler optimization approaches may vary for different software systems [18], [19]. In addition, coverage-based approaches [12], [13], [14] may not be effective with systems performing a large number of mathematical operations. Second, embedded software for CPS is thoroughly tested with test suites (e.g., to satisfy functional safety standards) that may take hours to execute[1], thus exacerbating scalability problems. Finally, it requires dedicated hardware, software emulators[2], or simulators[3], which affect the applicability of optimizations that make use of multi-threading or other OS functions [20]. The reliance on dedicated hardware, emulators, and simulators also prevents the use of static program analysis to detect equivalent mutants [21], [22], [23]. Such characteristics are common to embedded software in other types of CPS domains including avionics, automotive, and industry 4.0 (e.g., robotics systems).

In this paper, *we define and evaluate a mutation analysis pipeline* to assess the quality of test suites targeting embedded software developed for spaceflight systems or the many other CPS with similar characteristics. We call our mutation analysis pipeline: *Mutation Analysis for Space Software* (*MASS*). To account for the above-mentioned limitations, we propose and sometimes adapt a set of mutation analysis optimizations. More precisely, *MASS* combines (i) trivial compiler optimizations, to remove equivalent and redundant mutants after their generation, (ii) mutation sampling [9] [10], to reduce the number of mutants to execute, (iii) code coverage information regarding the original software [11], to prioritize and select test cases, and (iv) code coverage information regarding the mutated files [14], to further detect equivalent and redundant mutants. *MASS* is based on analyses that are feasible with large, real-time embedded software, including an incremental compilation pipeline to scale up the generation of mutants and the collection of coverage data focused on mutated files only. To provide statistical guarantees about the accuracy of mutants sampling, we propose to sample mutants by relying on sequential analysis based on the fixed size confidence interval approach (FSCI) [24]. Also, we extend FSCI to provide accurate results in the presence of reduced test suites. Finally, to effectively use code coverage information for test suite prioritization and equivalent/redundant mutants detection, we apply four different distance metrics to compare coverage results between test cases: Jaccard, Ochiai, Euclidean distance, and Cosine similarity.

To evaluate the effectiveness of the proposed approach, we rely on a space software benchmark provided by our industry partners, which are the European Space Agency [25], GomSpace Luxembourg (GSL), a manufacturer and supplier of nanosatellites [26], and LuxSpace (LXS), a developer of infrastructure products (e.g., microsatellites) and solutions for space [27]. Our benchmark consists of (1) the on-board embedded software system (service layer, high-level drivers, application layer) for *ESAIL* [28], a maritime microsatellite launched into space on September 2020 [29], (2) a set of libraries designed for Cubesats, including a network-layer delivery library, a utility library, and a satellite configuration library, and (3) a mathematical library for flight software [30]. To the best of our knowledge, **this is the first study proposing and assessing a mutation analysis optimization pipeline in the context of testing embedded software in cyber-physical systems**. Though our benchmark does not include embedded software for other CPS

1. Although this might be also true for other types of software, in the context of CPS it is due to a large input space including inputs in the continuous domain (e.g., signals) combined with long test execution times (e.g., to observe a specific signal shape or to verify diverse combinations of inputs).

2. Software emulators are used to test software compiled for specific hardware architectures.

3. In this paper, we use the term simulator to indicate software that model complex environments, including physical phenomena and hardware.

domains, we believe that its characteristics, as discussed above, make it representative of embedded software across many CPS domains, which has never been considered in mutation analysis studies (see Section 5).

In our empirical evaluation, we assess the validity, in the space software context, of the reported scientific findings concerning the state-of-the-art mutation analysis optimizations integrated into *MASS*. Further, we evaluate the feasibility of their integration into the *MASS* pipeline by reporting on the accuracy of the estimated mutation score and the execution time savings obtained with *MASS*. Our results show that (1) different compiler optimization options are complementary for the detection of trivially equivalent or duplicate mutants; we confirm related work findings about the possibility of discarding 30% of the mutants that way. (2) The proposed FSCI-based mutant sampling strategy outperforms state-of-the-art strategies; indeed, it is the only approach that minimizes the number of mutants selected while providing accuracy guarantees both when a full test suite or a prioritized subset of it are executed. (3) The proposed test suite selection and prioritization approach enables an impressive reduction of mutation analysis time (above 70%), thus making mutation analysis feasible also for large and complex space software. (4) Small differences in code coverage enable the identification of nonequivalent mutants.

To summarize, our contributions include the following:

- A **mutation analysis pipeline** targeting **embedded software for CPS** that innovatively combines (a) state-of-the-art techniques (i.e., equivalent mutants detection with trivial compiler optimizations), (b) improvements to techniques proposed in related work (i.e., coverage-based prioritization and selection of test cases [11], along with coverage-based detection of equivalent and redundant mutants [14]), and (c) new techniques for mutant sampling.
- To **address scalability problems**, which are acute in the case of embedded software for CPS, we propose FSCI-based mutant sampling. In addition, we also apply a test suite prioritization and selection strategy [11] adapted to work in the CPS context (i.e., to deal with test cases with long execution times and in the presence of mutants sampling). .
- To **address** problems related to the **pertinence of mutation scores**—which is an open problem in the case of CPS since it is not possible to rely on static program analysis to detect equivalent mutants, we adapt a strategy based on code coverage [14] to work with embedded software while avoiding intrusive monitoring (e.g., collecting return values).
- To evaluate the effectiveness of *MASS*, which integrates strategies that may interfere with each other and had never been evaluated with embedded software for CPS, we rely on an **industrial benchmark** including space software currently on orbit.

The paper proceeds as follows. Section 2 presents state-of-the-art mutation analysis techniques and assesses their applicability to space software. Section 3 presents our proposed pipeline for the mutation analysis of embedded software. Section 4 presents our empirical evaluation in the space domain. Section 5 presents related work. Section 6 concludes the paper.

2 Background and Applicability of State-of-the-art Mutation Analysis Techniques to Space Software

In this section, we discuss the applicability of state-of-the-art mutation analysis optimizations in the context of space software. Mutation analysis can drive the generation of test cases, which is referred to as *mutation testing* in the literature. A detailed overview of mutation testing and analysis solutions and optimizations can be found in recent surveys [31], [32].

2.1 Mutation Adequacy and Mutation Score computation

A mutant is said to be killed if at least one test case in the test suite fails when exercising the mutant. Mutants that do not lead to the failure of any test case are said to be live. Three conditions should hold for a test case to kill a mutant: *reachability* (i.e, the test case should execute the mutated statement), *necessity* (i.e., the test case should reach an incorrect intermediate state after executing the mutated statement), and *sufficiency* (i.e., the final state of the mutated program should differ from that of the original program) [33].

The mutation score, i.e., the percentage of killed mutants, is a quantitative measure of the quality of a test suite. Recent studies have shown that achieving a high mutation score improves significantly the fault detection capability of a test suite [34], a result which contrasts with that of structural coverage measures [35]. However, a very high mutation score (e.g., above 75%) is required to achieve a higher fault detection rate than the one obtained with other coverage criteria, such as statement and branch coverage [35]. In other words, there exists a strong association between a high mutation score and a high fault revelation capability for test suites.

The capability of a test case to kill a mutant also depends on the observability of the program state. To overcome the limitations due to observability, different strategies to identify killed mutants can be adopted; they are known as strong, weak, firm, and flexible mutation coverage [36]. With strong mutation, to kill a mutant, there shall be an observable difference between the outputs of the original and mutated programs. With weak mutation, the state (i.e., the valuations of the program variables in scope) of the mutant shall differ from the state of the original program, after the execution of the mutated statement [37]. With firm mutation, the state of the mutant shall differ from the state of the original program at execution points between the first execution of the mutated statement and the termination of the program [38]. Flexible mutation coverage consists of checking if the mutated code leads to object corruption [39]. For space software, we suggest to rely on strong mutation because it is the only criterion that truly assesses the fault detection capability of the test suite; indeed, it relies on a mutation score that reflects the percentage of mutants leading to test failures. With the other mutation coverage

criteria, a mutant is killed if the state of the mutant after execution of the mutated statement differs from the one observed with the original code, without any guarantee that either the erroneous values in state variables will propagate or the test oracles will detect them.

2.2 Mutation Operators

Mutation analysis introduces small syntactical changes into the code (source code or machine code) of a program through a set of mutation operators that simulate programming mistakes.

The *sufficient set of operators* is widely used for conducting empirical evaluations [40], [41], [42], [43]. The original sufficient set, defined by Offutt et al., is composed of the following operators: Absolute Value Insertion (ABS), Arithmetic Operator Replacement (AOR), Integer Constraint Replacement (ICR), Logical Connector Replacement (LCR), Relational Operator Replacement (ROR), and Unary Operator Insertion (UOI) [40]. Andrews et al. [42] have also included the *statement deletion operator* (SDL) [44], which ensures that every pointer-manipulation and field-assignment statement is tested.

The sufficient set of operators enables an accurate estimation of the mutation score of a test suite [45]; furthermore, the mutation score computed with the sufficient set is a good estimate of the fault detection rate (i.e., the portion of real faults discovered) of a test suite [42], [46].

However, empirical work has shown that, to maximize the detection of real faults, a set of operators should be used in addition to the sufficient set: Conditional Operator Replacement (COR), Literal Value Replacement (LVR), and Arithmetic Operator Deletion (AOD) [47].

The SDL operator has inspired the definition of mutation operators (e.g., *OODL operators*) that delete portions of program statements, with the objective of replacing the sufficient set with a simpler set of mutation operators. The OODL mutation operators include the delete Arithmetic (AOD), Bitwise (BOD), Logical (LOD), Relational (ROD), and Shift (SOD) operators. Empirical results show that deletion operators produce significantly fewer equivalent mutants[4] [44], [49] and, furthermore, test suites that kill mutants generated with both SDL and OODL operators kill a very high percentage of all mutants (i.e., 97%) [49].

Another alternative to the sufficient set of operators is the generation of *higher order mutants*, which result from the application of multiple mutation operators for each mutation [50], [51], [52], [53]. However, higher order mutants are easier to kill than the first order ones (i.e., less effective to assess test suites limitations) [32], [54], and there is limited empirical evidence regarding which mutation operators should be combined to resemble real faults and minimize the number of redundant mutants [32].

2.3 Compile-time Scalability

The potentially large size of the software under test, combined with the large number of available mutation operators, may make the compilation of all mutants infeasible.

4. For example, statement deletion can lead to equivalent mutants only if statements are redundant, which is unlikely [48].

To reduce the number of invocations to the compiler to one, *mutant schemata* include all the mutations into a single executable [20]. With mutant schemata, the mutations to be tested are selected at run-time through configuration parameters. This may lead to a compilation speed-up of 300% [55].

Another solution to address compile-time scalability issues consists of *mutating machine code* (e.g., binary code [56], assembly language [57], Java bytecode [58], and .NET bytecode [59]), thus avoiding the execution of the compilation process after creating a mutant. A common solution consists of mutating the LLVM Intermediate Representation (IR) [60], which enables the development of mutants that work with multiple programming languages [61] and facilitates the integration of optimizations based on dynamic program analysis [62].

Unfortunately, the mutation of machine code may lead to mutants that are not representative of real faults (i.e., faults caused by human mistakes at development time) because they are impossible to generate from the source code [62]. For instance, a function invocation in the source code may lead to hundreds of machine code instructions (e.g., the function call *std::vector::push_back* leads to 200 LLVM IR instructions) and, consequently, some of the mutants derived from such instructions cannot be derived by mutating the source code. In the case of IR mutation, some of these impossible mutants can be automatically identified [62]; however, the number of generated mutants tend to be higher at the IR level than at the source code level, which may reduce scalability [61]. In addition, we have encountered three problems that prevented the application of mutation analysis tools based on LLVM IR to our case study systems. First, space software relies on compiler pipelines (e.g., RTEMS [63]) that include architecture-specific optimizations not supported by LLVM. Second, there is no guarantee that the executables generated by LLVM are equivalent to those produced by the original compiler. Third, efficient toolsets based on LLVM often perform mutations dynamically [62], which is infeasible when the software under test needs to be executed within a dedicated simulator, a common situation with space software and many other types of embedded software in cyber-physical systems.

2.4 Runtime Scalability

A straightforward mutation analysis process consists of executing the full test suite against every mutant; however, it may lead to scalability problems in the case of a large software under test (SUT) with expensive test executions. *Simple optimizations* that can be applied to space software consist of (S1) stopping the execution of the test suite when the mutant has been killed, (S2) executing only those test cases that cover the mutated statements [64], and (S3) rely on timeouts to automatically detect infinite loops introduced by mutation [32].

Split-stream execution consists of generating a modified version of the SUT that creates multiple processes (one for each mutant) only when the mutated code is reached [65], [66], thus saving time and resources. Unfortunately, it cannot be applied in the case of space software that needs to run with simulators because, in general, the hosting simulator cannot be forked by the hosted SUT.

Another feasible solution consists of *randomly selecting a subset of the generated mutants* [9], [10], [67]. Zhang et al. [9] empirically demonstrated that a random selection of 5% of the mutants is sufficient for estimating, with high confidence, the mutation score obtained with the complete mutants set. Further, they show that sampling mutants uniformly across different program elements (e.g., functions) leads to a more accurate mutation score prediction than sampling mutants globally in a random fashion. For large software systems that lead to thousands of mutants, random mutation analysis is the only viable solution. However, for very large systems such as the ones commonly found in industry, randomly selecting 5% of the mutants may still be too expensive.

Gopinath et al. estimate the number of mutants required for an accurate mutation score [10]. They rely on the intuition that, under the assumption of independence between mutants, mutation analysis can be seen as a Bernoulli experiment in which the outcome of the test for a single mutant is a Bernoulli trial (i.e., mutant successfully killed or not) and, consequently, the mutation score should follow a binomial distribution. They rely on Tchebysheff's inequality [68] to find a theoretical lower bound on the number of mutants required for an accurate mutation score. More precisely, they suggest that, with 1,000 mutants, the estimated mutation score differs from the real mutation score at most by 7 percentage points. However, empirical results show that the binomial distribution provides a conservative estimate of the population variance and, consequently, 1,000 mutants enable in practice a more accurate estimate ($> 97\%$) of the mutation score than expected.

In the statistics literature, the correlated binomial model [69], and related models [70], [71], [72] are used when Bernoulli trials are not independent [73]. In our work, based on the results achieved by Gopinath et al., we assume that the degree of correlation between mutants is limited and the binomial distribution can be used to accurately estimate the mutation score, which is supported by our empirical results (see Section 4). In Appendix B, we verify the correctness of our assumptions by reporting the degree of association between trials and by comparing the probability mass function for the binomial and the correlated binomial distributions, for all our subjects.

The statistics literature also provides a number of approaches for the computation of a sample size (i.e., the number of mutants, in our context) that enables estimates with a given degree of accuracy [74], [75], [76], [77]. For binomial distributions, the most recent work is that of Gonçalves et al. [78], that determines the sample size by relying on heuristics for the computation of confidence intervals for binomial proportions. A confidence interval has a probability p_c (the confidence level) of including the estimated parameter (e.g., the mutation score). Results show that the largest number of samples required to compute a 95% confidence interval is 1,568.

If used to drive the selection of mutants, both the approaches of Gopinath et al. and Gonçalves et al., which suggest sampling at least 1,000 mutants, may be impractical when mutants are tested with large system test suites.

An alternative to computing the sample size before performing an experiment is provided by sequential analysis approaches, which determine the sample size while conducting a statistical test [79]. Such approaches do not perform worst-case estimates and may thus lead to smaller sample sizes. For example, the sequential probability ratio test, which can be used to test hypotheses, has been used in mutation analysis as a condition to determine when to stop test case generation (i.e., when the mutation score is above a given threshold) [80]. In our context, we are interested in point estimation, not hypothesis testing; in this case, the sample size can be determined through a fixed-width sequential confidence interval (FSCI), i.e., by computing the confidence interval after every new sample and then stop sampling when the interval is within a desired bound [24], [81], [82]. Concerning the method used to compute the confidence interval in FSCI, the statistics literature [24] reports that the Wald method [83] minimizes the sample size but requires an accurate variance estimate. We will therefore resort to a non-parametric alternative, which is Clopper-Pearson [84]. Note that FSCI has never been applied to determine the number of mutants to consider in mutation analysis.

Other solutions to address *runtime scalability problems* in mutation analysis aim to *prioritize test cases* to maximize the likelihood of executing first those that kill the mutants [11], [85], [86]. The main goal is to save time by preventing the execution of a large subset of the test suite, for each mutant. Previous work aimed at prioritizing faster test cases [85] but this may not be adequate with system-level test suites whose test cases have similar, long execution times. Approaches that rely on data-flow analysis to identify and prioritize the test cases that likely satisfy the killing conditions [86] are prohibitively expensive and are unlikely to scale to large systems. Other work [11] combines three coverage criteria: (1) the number of times the mutated statement is exercised by the test case, (2) the proximity of the mutated statement to the end of the test case (closer ones have higher chances of satisfying the sufficiency condition) , and (3) the percentage of mutants belonging to the same class file of the mutated statement that were already killed by the test case. Criterion (3) is also used to reduce the test suite size, by only selecting the test cases above a given percentage threshold. Unfortunately, only criterion (1) seems applicable in our context; indeed, criterion (2) is ineffective with system test cases whose results are checked after long executions, while criterion (3) may be inaccurate when only a random, small subset of mutants is executed, as discussed above.

2.5 Detection of Equivalent Mutants

A mutant is equivalent to the original program when they both generate the same outputs for the same inputs. Although identifying equivalent mutants is an undecidable problem [7], [87], several heuristics have been developed to address it.

The simplest solution consists of relying on *trivial compiler optimisations* [18], [32], [43], i.e., compile both the mutants and the original program with compiler optimisations enabled and then determine whether their executables match. In C programs, compiler optimisations can reduce the total number of mutants by 28% [43].

Solutions that identify equivalent mutants based on *static program analysis* (e.g., concolic execution [21], [23] and

bounded model checking [22]) show promising results (e.g., to automatically identify non-equivalent mutants for batch programs [23]) but they rely on static analysis solutions that cannot work with system-level test cases that execute with hardware and environment simulators in the loop. Indeed, (1) simulation results cannot be predicted by pure static analysis, (2) concolic execution tools, which rely on LLVM, cannot be run if the SUT executable should be generated with a specific compiler (see Section 2.3), (3) there are no solutions supporting the concolic execution of large software systems within simulation environments (state-of-the-art techniques work with small embedded software [88]), and (4) communication among components not based on direct method invocations (e.g., through network or databases) is not supported by existing toolsets.

Alternative solutions rely on *dynamic analysis* and compare data collected when testing the original software and the mutants [12], [13], [14], [15]. The most extensive empirical study on the topic shows that nonequivalent mutants can be detected by counting the number of methods (excluding the mutated method) that, for at least one test case, either (1) have statements that are executed at a different frequency with the mutant, (2) generate at least one different return value, or (3) are invoked at a different frequency [14]. To determine if a mutant is non-equivalent, it is possible to define a threshold indicating the smallest number of methods with such characteristics. A threshold of one identifies non-equivalent mutants with an average precision above 70% and an average recall above 60%. This solution outperforms more sophisticated methods relying on dynamic invariants [15]. Also, coverage frequency alone leads to results close to the ones achieved by including all three criteria above [14]. However, such approaches require some tailoring because collecting all required data (i.e., coverage frequency for every program statement, return values of every method, frequency of invocation of every method) has a computational and memory cost that may break real-time constraints.

2.6 Detection of Redundant Mutants

Redundant mutants are either *duplicates*, i.e., mutants that are equivalent with each other but not equivalent to the original program, or *subsumed*, i.e., mutants that are not equivalent with each other but are killed by the same test cases.

Duplicate mutants can be detected by relying on the same approaches adopted for equivalent mutants.

According to Shin et al., subsumed mutants should not be discarded but analyzed to augment the test suite with additional test cases that fail with one mutant only [8]. The augmented test suite has a higher fault detection rate than a test suite that simply satisfies mutation coverage; however, with large software systems the approach becomes infeasible because of the lack of scalable test input generation approaches.

2.7 Summary

We aim to rely on the sufficient set of operators since it has been successfully used to generate a mutation score that accurately estimates the fault detection rate for software written in C and C++, languages commonly used in embedded software. Further, since recent results have reported on the usefulness of both LVR and OODL operators to support the generation of test suites with high fault revealing power [47], the sufficient set may be extended to include these two operators as well.

To speed up mutation analysis by reducing the number of mutants, we should consider the SDL operator alone or in combination with the OODL operators. However, such heuristic should be carefully evaluated to determine the level of confidence we can expect.

Among compile time optimizations, only mutant schemata appear to be feasible with space software. Concerning scalability, simple optimizations (i.e., S1, S2, and S3 in Section 2.4) are feasible. Alternative solutions are the ones relying on mutant sampling and coverage metrics. However, to be applied in a safety or mission critical context, mutant sampling approaches should provide guarantees about the level of confidence one may expect. Currently, this can only be achieved with approaches requiring a large number of sampled mutants (e.g., $1,000$). Therefore, sequential analysis based on FSCI, which minimizes the number of samples and provides accuracy guarantees, appears to be the most appropriate solution in our context. Further, test suite selection and prioritization strategies based on code coverage require some tailoring to cope with real time constraints.

Equivalent mutants can be identified through trivial compiler optimizations and the analysis of coverage differences; however, it is necessary to define and evaluate appropriate coverage metrics. The same approach can be adopted to identify duplicate mutants. The generation of test cases that distinguish subsumed mutants is out of the scope of this work.

A high-level description of a possible mutation testing pipeline was proposed in a recent survey[5] [32]. It consists of the following sequence of activities: select (sample) mutants, compile mutants, remove equivalent and redundant mutants, generate test inputs that kill mutants, execute mutants, compute mutation score, reduce test suites and prioritize test cases. Unfortunately, such pipeline does not enable the integration of many optimizations proposed above, which further motivates our work. For example, it cannot support FSCI-based sampling, which requires mutants sampling to be coupled with mutants execution. Also, it does not envision the detection of equivalent and redundant mutants based on code coverage. Moreover, it only partially addresses scalability issues since test suite reduction and prioritization are performed after mutation analysis. Further, it includes a test input generation step that is not feasible in the context of CPS. Finally, it has never been implemented and therefore its feasibility has not been evaluated.

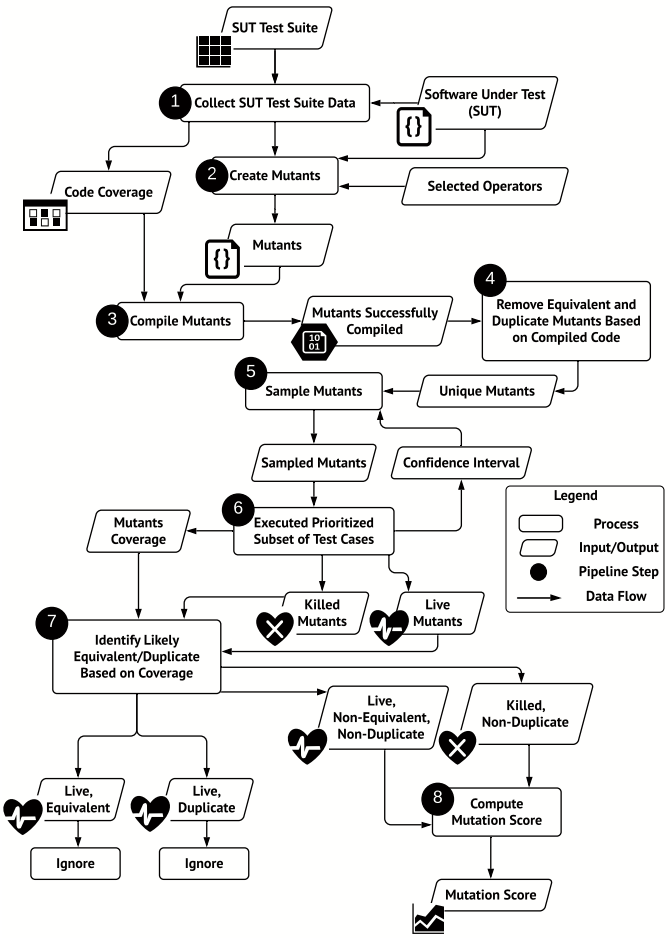


Fig. 1: Overview of the proposed Mutation Analysis Pipeline

3 Space Software Mutation Analysis Pipeline

Figure 1 provides an overview of the mutation analysis process that we propose, *MASS*, based on the discussions and decisions in the previous section. Its goal is to propose a comprehensive solution for making mutation analysis applicable to embedded software in industrial cyber-physical systems. The ultimate goal of *MASS* is to assess the effectiveness of test suites with respect to detecting violations of functional requirements.

Different from the mutation analysis pipeline presented in related work [32], *MASS* enables the integration of all mutation analysis optimization techniques that are feasible in our context to address scalability and pertinence problems (see Section 2.7). *MASS* consists of eight steps: (Step 1) Collect SUT Test Suite Data, (Step 2) Create Mutants, (Step 3) Compile Mutants, (Step 4) Remove Equivalent and Duplicate Mutants Based on Compiled Code, (Step 5) Sample Mutants, (Step 6) Execute Prioritized Subset of Test Cases, (Step 7) Identify Likely Equivalent / Duplicate mutants Based on Coverage, and (Step 8) Compute the Mutation Score. Different from related work, *MASS* enables FSCI-based sampling by iterating between mutants sampling (Step 5) and test cases execution (Step 6). Also, it integrates test suite prioritization and reduction (Step 6) before the computation of the mutation score. Finally, it includes methods to identify likely equivalent and duplicate mutants based on code coverage (Step 7). We describe each step in the following paragraphs.

5. The main objective of such pipeline was to walk the reader through the survey, not to propose a precise and feasible solution.

TABLE 1: Implemented set of mutation operators.

	Operator	Description*
Sufficient Set	ABS	$\{(v, -v)\}$
	AOR	$\{(op_1, op_2) \mid op_1, op_2 \in \{+, -, \star, /, \%\} \wedge op_1 \neq op_2\}$ $\{(op_1, op_2) \mid op_1, op_2 \in \{+=, -=, \star=, /=, \%=\} \wedge op_1 \neq op_2\}$
	ICR	$\{i, x) \mid x \in \{1, -1, 0, i+1, i-1, -i\}\}$
	LCR	$\{(op_1, op_2) \mid op_1, op_2 \in \{\&\&, \|\|\} \wedge op_1 \neq op_2\}$ $\{(op_1, op_2) \mid op_1, op_2 \in \{\&=, \|=, \&=\} \wedge op_1 \neq op_2\}$ $\{(op_1, op_2) \mid op_1, op_2 \in \{\&, \|, \&\&\} \wedge op_1 \neq op_2\}$
	ROR	$\{(op_1, op_2) \mid op_1, op_2 \in \{>, >=, <, <=, ==, !=\}\}$ $\{(e, !(e)) \mid e \in \{\texttt{if(e)}, \texttt{while(e)}\}\}$
	SDL	$\{(s, \texttt{remove}(s))\}$
	UOI	$\{(v, --v), (v, v--), (v, ++v), (v, v++)\}$
OODL	AOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{+, -, \star, /, \%\}\}$
	LOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{\&\&, \|\|\}\}$
	ROD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{>, >=, <, <=, ==, !=\}\}$
	BOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{\&, \|, \wedge\}\}$
	SOD	$\{((t_1\ op\ t_2), t_1), ((t_1\ op\ t_2), t_2) \mid op \in \{>>, <<\}\}$
Other	LVR	$\{(l_1, l_2) \mid (l_1, l_2) \in \{(0, -1), (l_1, -l_1), (l_1, 0), (true, false), (false, true)\}\}$

*Each pair in parenthesis shows how a program element is modified by the mutation operator on the left; we follow standard syntax [47]. Program elements are literals (l), integer literals (i), boolean expressions (e), operators (op), statements (s), variables (v), and terms (t_i, which might be either variables or literals).

3.1 Step 1: Collect SUT Test Data

In Step 1, the test suite is executed against the SUT and code coverage information is collected. More precisely, we rely on the combination of gcov [89] and GDB [90], enabling the collection of coverage information for embedded systems without a file system [91].

3.2 Step 2: Create Mutants

In Step 2, we automatically generate mutants for the SUT by relying on a set of selected mutation operators. In *MASS*, based on the considerations provided in Section 2.2, we rely on an extended sufficient set of mutation operators, which are listed in Table 1. In addition, in our experiments, we also evaluate the feasibility of relying only on the SDL operator, combined or not with OODL operators, instead of the entire sufficient set of operators.

To automatically generate mutants, we have extended SRCIRor [92] to include all the operators in Table 1. After mutating the original source file, our extension saves the mutated source file and keeps track of the mutation applied. Our toolset is available under the ESA Software Community Licence Permissive [93] at the following URL **https://faqas.uni.lu/**.

3.3 Step 3: Compile Mutants

In Step 3, we compile mutants by relying on an optimized compilation procedure that leverages the build system of the SUT. To this end, we have developed a toolset that, for each mutated source file: (1) backs-up the original source file, (2) renames the mutated source file as the original source file, (3) runs the build system (e.g., executes the command

`make`), (4) copies the generated executable mutant in a dedicated folder, (5) restores the original source file.

Build systems (e.g., GNU make [94] driving the GCC [95] compiler) create one object file for each source file to be compiled and then link these object files together into the final executable. After the first build, in subsequent builds, build systems recompile only the modified files and link them to the rest. For this reason, our optimized compilation procedure, which modifies at most two source files for each mutant (i.e., the mutated file and the file restored to eliminate the previous mutation), can reuse almost all the compiled object files in subsequent compilation runs, thus speeding up the compilation of multiple mutants. The experiments conducted with our subjects (Section 4) have shown that our optimization is sufficient to make the compilation of mutants feasible for large projects. Other state-of-the-art solutions introduce additional complexity (e.g., change the structure of the software under test [20]) that does not appear to be justified by scalability needs.

3.4 Step 4: Remove Equivalents and Duplicates

In Step 4, we rely on trivial compiler optimizations to identify and remove equivalent and duplicate mutants. More precisely, for every available compiler optimization level (e.g., O0, O1, O2, O3, O4, Os, and Ofast for GCC), or a subset selected by engineers, *MASS* re-executes Step 3 and stores the SHA-512 hash summaries of all the mutant and original executables [96]. To detect equivalent mutants, *MASS* compares the hash summaries of the mutants with that of the original executable. To detect duplicate mutants but avoid combinatorial explosion, *MASS* focuses its comparison of hash summaries on pairs of mutants belonging to the same source file (restricting the scope of the comparison is common practice [43]). Hash comparison allows us to (1) determine the presence of equivalent mutants (i.e., mutants having the same hash as the original executable), and (2) identify duplicate mutants (i.e., mutants with the same hash). Equivalent and duplicate mutants are then discarded. We compare hash summaries rather than executable files because it is much faster, an important consideration when dealing with a large number of mutants. The outcome of Step 4 is a set of *unique mutants*, i.e., mutants with compiled code that differs from the original software and any other mutant.

3.5 Step 5: Sample Mutants

In Step 5, *MASS* samples the mutants to be executed to compute the mutation score. *MASS* does not selectively generate mutants but samples them from the whole set of successfully compiled, nonequivalent, and nonduplicated mutants (result of Steps 2 to 4). This choice aims to avoid sampling bias which may result from the presence of such mutants; indeed, there is no guarantee that these mutants, if they were discarded after being sampled, would be uniformly distributed across program statements. Our choice does not affect the feasibility of *MASS* since Steps 2 to 4 have negligible cost (see Section 4).

Our pipeline supports different sampling strategies: *proportional uniform sampling*, *proportional method-based sampling*, *uniform fixed-size sampling*, and *uniform FSCI sampling*.

The strategies *proportional uniform sampling* and *proportional method-based sampling* were selected based on the results of Zhang et al. [9], who compared eight strategies for sampling mutants. The former was the best performing strategy and consists of sampling mutants evenly across all functions of the SUT, i.e., sampling $r\%$ mutants from each set of mutants generated inside the same function. The latter consists of randomly selecting $r\%$ mutants from the complete mutants set. This is included in our study because it is simpler to implement and showed to be equivalent to stratified sampling strategies, based on recent work [10].

The *uniform fixed-size sampling* strategy stems from the work of Gopinath et al. [10] and consists of selecting a fixed number N_M of mutants for the computation of the mutation score. Based their work, with 1,000 mutants, one can guarantee an accurate estimation of the mutation score.

In this paper, we introduce the *uniform FSCI sampling* strategy that determines the sample size dynamically, while exercising mutants, based on a fixed-width sequential confidence interval approach. With *uniform FSCI sampling*, we introduce a cycle between Step 6 and Step 5, such that a new mutant is sampled only if deemed necessary. More precisely, *MASS* iteratively selects a random mutant from the set of unique mutants and exercises it using the SUT test suite. Based on related work, we assume that the mutation score computed with a sample of mutants follows a binomial distribution (see Section 2.4). For this reason, to compute the confidence interval for the FSCI analysis, we rely on the Clopper-Pearson method since it is reported to provide the best results (see Section 2.4). Mutation analysis (i.e., sampling and testing a mutant) stops when the confidence interval is below a given threshold T_{CI} (we use $T_{CI} = 0.10$ in our experiments). More formally, given a confidence interval $[L_S; U_S]$, with L_S and U_S indicating the lower and upper bound of the interval, mutation analysis stops when the following condition holds:

$$(U_S - L_S) < T_{CI}. \tag{1}$$

Unfortunately, the assumption about the estimated mutation score following a binomial distribution may not hold when a subset of the test suite is executed for every mutant (which could happen in Step 6). Without going into the details behind the implementation of Step 6, which is described in Section 3.6, we can expect that a reduced test suite may not be able to kill all the mutants killed by the entire test suite, i.e., the estimated mutation score may be affected by negative bias. Consequently, over multiple runs, the mean of the estimated mutation score may not be close to the *actual mutation score* (i.e., the mutation score computed with the entire test suite exercising all the mutants for the SUT) but may converge to a lower value. To compute a correct confidence interval that includes the actual mutation score of the SUT, we thus need to take into account this negative bias.

To study the effect of negative bias on the confidence interval, we address first the relation between the actual mutation score and the mutation score computed with the reduced test suite when the entire set of mutants for the SUT is executed. A mutant killed by the entire test suite has a probability P_{KErr} of not being killed by the reduced test

suite. The probability P_{KErr} can be estimated as the proportion of mutants (erroneously) not killed by the reduced test suite

$$P_{KErr} = \frac{|E_R|}{|M|} \tag{2}$$

with E_R being the subset of mutants that are killed by the entire test suite but not by the reduced test suite, and M being the full set of mutants for the SUT.

The mutation score for the reduced test suite (MS_R) can be computed as

$$MS_R = \frac{|K| - |E_R|}{|M|} = \frac{|K|}{|M|} - \frac{|E_R|}{|M|} = MS - \frac{|E_R|}{|M|} = MS - P_{KErr} \tag{3}$$

where K is the set of mutants killed by the whole test suite, M is the set of all the mutants of the SUT, and MS is the actual mutation score. Consequently, the actual mutation score can be computed as

$$MS = MS_R + P_{Err_R} \tag{4}$$

We now discuss the effect of a reduced test suite on the confidence interval for a mutation score estimated with mutants sampling. When mutants are sampled and tested with the entire test suite, the actual mutation score is expected to lie in the confidence interval $[L_S; U_S]$. In the presence of a reduced test suite, we can still rely on the Clopper-Pearson method to compute the confidence interval $CI_R = [L_R; U_R]$. However, we have to take into account the probability of an error in the computation of the mutation score MS_R; MS_R can be lower than MS and, based on Equation 4, we expect the actual mutation score to lie in an interval that is shifted with respect to the interval for MS_R:

$$CI = [L_R + P_{KErr}; U_R + P_{KErr}] \tag{5}$$

We can only estimate P_{KErr} since computing it would require the execution of all the mutants with the complete test suite, thus undermining our objective of reducing test executions. To do so, we can randomly select a subset M_R of mutants, on which to execute the entire test suite and identify the mutants killed by the reduced test suite. The size of the set M_R should be lower than the number of mutants we expect FSCI sampling to return, otherwise sampling would not provide any cost reduction benefit. Since, for every mutant in M_R, we can determine if it is erroneously reported as not killed by the reduced test suite R, we can estimate the probability P_{KErr} as the percentage of such mutants. As for the case of the mutation score, we assume that the binomial distribution provides a conservative estimate of the variance for P_{KErr}.

We can estimate the confidence interval for P_{KErr} using one of the methods for binomial distributions. We rely on the Wilson score method because it is known to perform well with small samples [97]. The value of P_{KErr} will thus lie within $CI_E = [L_E; U_E]$, with L_E and U_E indicating the lower and upper bounds of the interval.

Based on Equation 5, the confidence interval to be used with FSCI sampling in the presence of a reduced test suite should thus be

$$CI = [L_R + L_E; U_R + U_E] \tag{6}$$

The estimated mutation score is the value lying in the middle of the interval.

Since the width of the confidence interval CI (hereafter, $|CI|$) results from the sum of $|CI_R|$ and $|CI_E|$, mutation sampling with a reduced test suite may lead to the execution of a larger set of mutants.

Based on Equations 1 and 6, $|CI_R| \leq T_{CI} - |CI_E|$. Consequently, when $|CI_E| > T_{CI}$, the reduced test suite cannot lead to sufficiently accurate results. Also, a large $|CI_E|$ may prevent the identification of accurate results with a feasible number of mutants. For example, Clopper-pearson may require up to 1568 samples for a confidence interval below 0.05 [78]. We shall thus identify a threshold (T_{CE}) for the confidence interval $|CI_E|$ that enables accurate estimates with a small sample size (e.g., in the worst case, with less than 1000 samples, the sample size for related work). For this reason, starting from a minimal number of samples to estimate P_{KErr} (150 in our experiments), *MASS* keeps estimating P_{KErr} until it yields $|CI_E| \leq T_{CE}$. In our experiments we set $T_{CE} = 0.035$. To select T_{CE}, we have identified a reasonable minimal mutation score to be expected in space software (i.e., 65%) and identified, based on confidence interval estimation methods with finite population correction factor [98], the minimal value for $|CI_E|$ that requires a number of samples below 850 (i.e., $1000 - 150$).

When it is not possible to estimate $|CI_E| \leq T_{CE}$ or when the number of samples required to estimate $|CI_E| \leq T_{CE}$ is sufficient to accurately estimate the mutation score, the test suite can be prioritized but not reduced and the confidence interval is computed using the traditional Clopper-Pearson method, i.e., $[L_S; U_S]$.

3.6 Step 6: Test Prioritization

In Step 6, we execute a prioritized subset of test cases. We select only the test cases that satisfy the reachability condition (i.e., cover the mutated statement) and execute them in sequence. Similarly to the approach of Zhang et al. [11], we define the order of execution of test cases based on their estimated likelihood of killing a mutant. However, in our work, this likelihood is estimated differently since, as discussed above, the measurements they rely on are not applicable in the context of system-level testing and complex cyber-physical systems (see Section 2.4). In contrast, to minimize the impact of measurements on real-time constraints, we only collect code coverage information for a small part of the system.

To reduce the number of test cases to be executed with a mutant, we should first execute the ones that more likely satisfy the necessity condition. This might be achieved by executing a test case that exercises the mutated statement with variable values not observed before. Unfortunately, in our context, the size of the SUT and its real-time constraints prevent us from recording all the variable values processed during testing.

Therefore, we rely on code coverage to determine if two test case executions exercise the mutated statement with diverse variable values. Such coverage is collected by efficient procedures provided by compilers, thus having lower impact on execution performance than other types of dynamic analysis solutions (e.g., tracing variable values).

Since, because of control- and data-flow dependencies, a different set of input values may lead to differences in code coverage, the latter helps determine if two or more test cases likely exercise a mutated statement with different variable values. To increase the likelihood that the observed differences in code coverage are due to the use of different variable values to exercise the mutated statement, we restrict the scope of code coverage analysis to the functions belonging to the component (i.e., the source file) that contains the mutated statement. Indeed, such functions typically present several control- and data-flow dependencies, thus augmenting the likelihood that a coverage difference is due to the execution of the mutated statement with a diverse set of values. Also, collecting code coverage for a small part of the system further reduces the impact of our analysis on system performance.

Based on related work, we have identified two possible strategies to characterize test case executions based on code coverage:

- S1 Compare the sets of source code statements that have been covered by test cases [12].
- S2 Compare the number of times each statement has been covered by test cases [14].

To determine how dissimilar two test cases are and, consequently, how likely they are to exercise the mutated statement with different values, we rely on widely adopted distance metrics. In the case of S1, we rely on the Jaccard and Ochiai indices, which are two similarity indices for binary data which have been successfully used to compare program executions based on code coverage [99], [100], [101]. Given two test cases T_A and T_B, the Jaccard (D_J) and Ochiai (D_O) distances are computed as follows:

$D_J(T_a, T_b) = 1 - \frac{|C_a \cap C_b|}{|C_a \cup C_b|}$ $D_O(T_a, T_b) = 1 - \frac{|C_a \cap C_b|}{\sqrt{|C_a| * |C_b|}}$,

where C_a and C_b are the set of covered statements exercised by T_a and T_b, respectively.

In the case of S2, we compute the distance between two test cases by relying on the euclidean distance (D_E) and the cosine similarity distance (D_C), two popular distance metrics used in machine learning. Given two vectors V_A and V_B, whose elements capture the number of times a statement has been covered by test cases T_A and T_B, the distances D_E and D_C can be computed as follows:

$D_E = \sqrt{\sum_{i=1}^{n}(A_i - B_i)^2}$

$D_C = 1 - \frac{\sum_{i=1}^{n} A_i * B_i}{\sqrt{\sum_{i=1}^{n} {A_i}^2} * \sqrt{\sum_{i=1}^{n} {B_i}^2}}$, where A_i and B_i refer to the number of times the i-th statement had been covered by T_A and T_B, respectively.

Figure 2 shows the pseudocode of our algorithm for selecting and prioritizing test cases. It generates as output a prioritized test suite (*PTS*). Based on the findings of Zhang et al. [11], we first select the test case that exercises the mutated statement the highest number of times (Line 3) and add it to the prioritized test suite (Line 4). Then, in the next iterations, the test case selected is the one with the largest distance from the closest test case already selected (Lines 10 to 13). When two or more test cases have the same distance, we select randomly among the test cases that exercise the mutated statement the most.

Require: *TS*, the test suite of the software under test
Require: *Cov*, coverage information, for each test case
Require: *ms*, the mutated statement
Ensure: *PTS*, a list of test cases to be executed, sorted by priority
1: $TS_m \leftarrow$ subset of TS that cover the mutated statement ms, based on *Cov*
2: $PTS \leftarrow newlist$ //this list is initially empty
3: $PTS \leftarrow$ based on *Cov* select from TS_m the test case t that exercises ms more times
4: $PTS \leftarrow PTS \cup t$ //include first the test case selected above
5: **repeat**
6: **for each** n in (TS_m - PTS) , i.e., is the set of test cases not already added to PTS
7: **for each** t in PTS
8: compute the distance between t and n
9: identify t_n i.e., the test case t with the minimal d
10: among all the t_n identified, select the one with the highest distance d
11: **if** $d > 0$ //there is at least a test case with a different coverage
12: //note: n is the test case in the set (TS_m - PTS) closer to t_n
13: $PTS \leftarrow PTS \cup n$
14: **until** $d > 0$

Fig. 2: PrioritizeAndReduce: Algorithm for prioritizing test cases

The algorithm iterates as long as it identifies a test case showing a difference in code coverage from the already selected test cases (Line 14).

Test cases are then executed in the selected order. During execution, we collect code coverage information and identify killed and live mutants.

3.7 Step 7: Discard Mutants

In this step, we identify likely nonequivalent and likely nonduplicate mutants by relying on code coverage information collected in the previous step.

Similarly to related work [14], we identify nonequivalent and nonduplicate mutants based on a threshold, which we will empirically investigate in Section 4.8.

In our case, consistently with previous steps of *MASS*, we compute normalized distances based on the distance metrics D_J, D_O, D_E, and D_C. A mutant is considered nonequivalent when the distance from the original program is above the threshold T_E, for at least one test case. Similarly, a mutant is considered nonduplicate when the distance from every other mutant is above the threshold T_D, for at least one test case. For the identification of nonequivalent mutants, we consider live mutants only. To identify nonduplicate mutants, we consider both live and killed mutants; however, to avoid combinatorial explosion, we compare only mutants belonging to the same source file (indeed, mutants belonging to different files are unlikely to be redundant). Killed mutants that lead to the failure of different test cases are not duplicate, regardless of their distance.

Thresholds T_E and T_D should enable the identification of mutants that are guaranteed to be nonequivalent and nonduplicate. In particular, we are interested in the set of *live, nonequivalent, nonduplicate mutants* (hereafter, *LNEND*) and the set of *killed, nonduplicate mutants* (hereafter, *KND*). With such guarantees, the mutation score can be adopted as an adequacy criterion in safety certification processes. For example, certification agencies may require safety-critical software to reach a mutation score of 100%, which is feasible in the presence of nonequivalent mutants.

3.8 Step 8: Compute Mutation Score

The mutation score (MS) is computed as the percentage of killed nonduplicate mutants over the number of nonequivalent, nonduplicate mutants identified in Step 7):

$$MS = \frac{|KND|}{|LNEND| + |KND|} \quad (7)$$

4 Empirical Evaluation

Our empirical evaluation aims to assess the effectiveness of the techniques integrated into *MASS* to address scalability and pertinence problems (i.e., Steps 2, 4, 5, 6, 7, and 8, in Figure 1). Our objectives include (RQ1) confirming, in our context, trivial compiler optimization results observed in related work (Step 4), (RQ2) identifying the most effective solution for mutants sampling (Step 5), (RQ3) comparing mutants generation strategies implemented by *MASS* (Step 2), evaluating the (RQ4) accuracy and (RQ5) effectiveness of the strategies proposed for test suite prioritization (Step 6), and (RQ6) evaluating the accuracy of the strategy for the identification of likely equivalent/duplicate mutants (Step 7). Finally, we aim to (RQ7) compare the mutation score computed by *MASS* (Step 8) with the mutation score computed without *MASS* optimizations. In the following, we provide our research questions and describe the *MASS* steps they aim to evaluate in more detail.

RQ1 (Step 4) What are the cost savings provided by compiler optimization techniques detecting equivalent and duplicate mutants? We wish to determine what is the percentage of mutants reported as being equivalent and duplicate by compiler optimization techniques. After accounting for the additional compilation time entailed by such techniques, we want to identify the optimal subset of compilation options to be used in Step 4 of *MASS*.

RQ2 (Step 5) Can a randomly selected subset of mutants be used to accurately estimate the mutation score obtained from the entire set of mutants? We attempt to evaluate four mutants sampling strategies: *proportional uniform sampling*, *proportional method-based sampling*, *uniform fixed-size sampling*, and *uniform FSCI sampling*. More precisely, we aim to determine the best configuration for each sampling strategy (i.e., sampling ratio, sample size, and confidence interval). Furthermore, we need to identify which strategy offers the best trade-off between the number of mutants to be tested and accuracy.

RQ3 (Step 2) Do mutants generated with deletion operators (i.e., SDL and OODL) lead to a mutation score that accurately estimates the mutation score of the entire set of mutants? We want to determine if we can minimize the number of selected mutants by only relying on deletion operators. To do so, we compare the mutation score generated with SDL and OODL operators with the mutation score based on all available mutation operators.

RQ4 (Step 6) Can a prioritized subset of test cases that maximizes test suite diversity be used to accurately estimate the mutation score of the entire test suite? We investigate how the various distance metrics used in the PrioritizeAndReduce algorithm implemented by *MASS* (Step 6) compare in terms of accuracy.

RQ5 (Step 6) To what extent different test suite prioritization strategies can speed up the mutation analysis process? We investigate the execution time reduction achieved by different distance metrics used in the PrioritizeAndReduce algorithm.

RQ6 (Step 7) Is it possible to identify thresholds, based on code coverage information, that enable the detection of nonequivalent and nonduplicate mutants? We investigate the accuracy of our strategy based on threshold values for the best distance metric (*MASS* Step 7).

RQ7 (Step 8) How does the mutation score computed by *MASS* relate to the mutation score of the original test suite based on the complete set of mutants? In other words, is there any tradeoff between the gains in scalability due to *MASS* and mutation score accuracy? We therefore analyze the difference between the *MASS* mutation score, which is obtained with a subset of the test suite and excludes likely equivalent and duplicate mutants, and the mutation score obtained with the entire set of mutants tested with the full test suite.

4.1 Subjects of the study

To perform our experiments, we considered five software artifacts (hereafter, *subjects*), each one developed by one of the aforementioned industry partners for different satellites: *ESAIL-CSW* (central software), *LIBU*, *LIBN*, *LIBP*, and *MLFS*. They are representative of common types of flight software— that are also typically present in other cyber-physical systems—including on-board controllers (*ESAIL-CSW*), libraries providing features related to the application layer (*LIBP*), as well as networking (*LIBN*), utility (*LIBU*), and mathematical functions (*MLFS*).

ESAIL is a microsatellite developed by LXS in a Public-Private-Partnership with ESA and ExactEarth. The Payload is an AIS Receiver for ship and vessel detection from space. For our empirical evaluation, we considered the onboard central control software of *ESAIL* (hereafter, simply *ESAIL-CSW*), which consists of 924 source files with a total size of 187,116 LOC. *ESAIL-CSW* is verified by unit test suites and system test suites that run in different facilities (e.g., Software Validation Facility [102], FlatSat [103], Protoflight Model [104]). Except for the test suite running in the Software Validation Facility (SVF), which is a simulator for the onboard hardware [102], the other test suites require dedicated hardware. The SVF simulates both the target hardware and the satellite units (e.g., a magnetometer connected to the Attitude Determination And Control Subsystem unit). For this study, we considered the SVF test suite, which consists of a total of 384 carefully selected test cases targeting mainly functional and interface requirements of the system. Other requirements (e.g., timing, robustness, and performance requirements) are verified by the other system test suites. Unit test suites are used for preliminary development stages and later to ensure higher level of code coverage for critical modules on the target hardware. For this study, we could not consider all the available test suites because of hardware

availability; also, our evaluation required repeated executions of the provided test suites, which would not have been practically feasible with dedicated hardware devices in the loop (see Section 4.2). The SVF test suite already takes 10 hours to execute.

LIBN, *LIBP*, and *LIBU* are utility libraries developed by GSL[6]. LIBN is a network protocol library including low-level drivers (e.g., CAN, I2C). *LIBP* is a light-weight parameter system designed for GSL satellite subsystems. *LIBU* is a utility library providing cross-platform APIs for use in both embedded systems and Linux development environments.

The Mathematical Library for Flight Software (*MLFS*) implements mathematical functions qualified for flight software (it complies with ECSS criticality category B).

The first four columns of Table 2 provide additional details. These software components differ in size and complexity; they range from 3,179 (*LIBP*) to 74,155 (*ESAIL-CSW*) LOC (see column *LOC* in Table 2). We also provide information concerning a subset of *ESAIL-CSW* (i.e., $ESAIL_S$) that is introduced in the following paragraphs.

All the test suites considered in our study are characterized by high statement coverage as required by space software standards (e.g., category C software requires statement adequacy according to ECSS standards [2]). However, in our study, we do not consider dedicated test suites that require the target hardware to be executed because of scalability issues, costs, and hardware safety. Indeed, our experiments imply the execution of a large number of test cases (see Section 4.2) that cannot be parallelized when real hardware is required, as only one or few hardware components are available because of their high cost. Also, hardware often needs to be manually set-up, which would make our experiments prohibitively expensive. Finally, the automatically generated mutants may damage the hardware. In the case of *LIBN*, *LIBP*, and *LIBU*, we considered unit and integration test suites that exercise the SUT in the development environment (a Linux-based system). For *MLFS*, we considered a unit test suite achieving modified condition/decision coverage (MC/DC) coverage [105]. Since we exclude test cases that must be executed with hardware in the loop, the test suites considered in our study do not achieve 100% statement coverage, except for *MLFS*. Working with test suites that do not achieve statement adequacy should not affect the validity of our findings because we apply mutation only to statements that are covered by the test suite.

The test suites considered in our experiments differ regarding the distribution of test cases exercising each statement (see Figure 3). For unit and integration test suites, test cases focus on a subset of functionalities and input ranges, as a result, the number of test cases exercising a same statement is expectedly low, between one and 18. For the $ESAIL_S$ system test suite, whose test cases exercise multiple functionalities (e.g., periodic tasks), the number of test cases exercising a same statement is much higher, between one and 121, with a median equal to 58. These numbers further highlight the diversity across our subjects.

To address some of our research questions, all the mutants must be executed against the test suite, which is not feasible for the case of *ESAIL-CSW* due to its large size and test suite. For this reason, we have identified a subsystem of *ESAIL-CSW* (hereafter, $ESAIL_S$) that consists of a set of files, selected by LXS engineers, that are representative of the different functionalities in *ESAIL-CSW*: service/protocol layer functions, critical functions of the satellite implemented in high-level drivers, application layer functions. Details about $ESAIL_S$ are reported in Table 2.

Except for *ESAIL-CSW*, all subjects are compiled to generate executables for the development environment OS (Linux); we rely on the Gnu Compiler Collection (GCC) for Linux X86 [95] versions 5.3 and 6.3 for *MLFS* and ONE, respectively. *ESAIL-CSW* is compiled with the LEON/ERC32 RTEMS Cross Compilation System, which includes the GCC C/C++ compiler version 4.4.6 for RTEMS-4.8 (Sparc architecture) [63].

It is important to note that the technical and test suite characteristics described above are very common in embedded software across many industry domains and cyber-physical systems, thus suggesting our results can be generalizable beyond space software.

6. We use anonymized acronyms according to GSL policy.

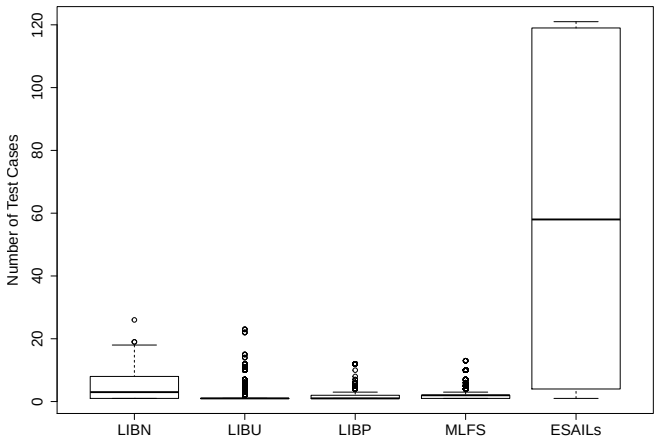


Fig. 3: Distribution of test cases exercising each statement.

TABLE 2: Descriptions of subject artifacts.

Subject	LOC	Test suite type	# Test cases	Statements coverage
ESAIL-CSW	74,155	System	384	90.38%
$ESAIL_S$	2,235	System	384	95.36%
LIBN	9,836	Integration	89	63.10%
LIBP	3,179	Integration	170	77.60%
LIBU	10,576	Unit	201	83.20%
MLFS	5,402	Unit	4042	100.00%

TABLE 3: Generated and compiled mutants per subject.

Subject	Mutants generated	MGT (sec)	Mutants compiled	% of compiled mutants	MCT (sec)
ESAIL-CSW	142,763	182	121,848	85.35%	151,234
$ESAIL_S$	7,212	9	5,347	74.14%	7,640
LIBN	8,666	12	7,878	90.91%	11,425
LIBP	7,252	7	6,440	88.80%	9,392
LIBU	22,295	28	20,268	90.91%	30,624
MLFS	31,526	20	28,069	89.03%	3,157
Total*	212,502	249	184,503	86.82%	205,832

*We ignore $ESAIL_S$ from the total counting because it is a subset of *ESAIL-CSW*.
Legend: MGT = mutants generation time, MCT = mutants compilation time.

4.2 Experimental Setup

To perform the empirical evaluation, we have implemented the *MASS* pipeline in a toolset that is available under the ESA Software Community Licence Permissive [93] at the following URL **https://faqas.uni.lu/** [7]. For the implementation of mutation operators, we extended the SRCiror toolset [92]. In our analysis, we consider all the operators reported in Table 1.

Related studies [9], [67] are performed by relying on mutation adequate test suites (i.e., test suites that kill all non-equivalent mutants). Such test suites are typically automatically generated using static analysis [106]. Since we cannot leverage static analysis to generate mutation adequate test suites (see Section 2.5), we rely on the original test suites provided with the subjects. As a result, to perform our study, we mutate only the statements that are covered by the considered test suites.

For every subject, we generated mutants by executing the *MASS* toolset on Linux OS running on a MacBook Pro with 2,3 GHz 8-Core Intel Core i9. Table 3 reports, for every subject, the total number of mutants that were generated, their generation time, the number of mutants successfully compiled, the proportion of compiled mutants with respect to the overall number of mutants generated, and the time required to compile mutants using one compiler optimization level only (i.e., the one originally selected by engineers for each subject). The generation of mutants is fast, it takes at most 182 seconds on the largest subject (*ESAIL-CSW*). On average, across subjects, it takes 11 milliseconds to generate a single mutant. The proportion of successfully compiled mutants is large (i.e., 86.82% overall), though it varies from 85.35% for *ESAIL-CSW* to 90.91% for *LIBN* and *LIBU*. This proportion is in line with the ones reported in related work, though there is variation. For example, industrial systems have shown lower success rates (e.g., 81.13% for safety-critical software components [107]) than open source, batch utilities (e.g., 96.30% for Coreutils [60]).

As shown in Table 3, the time required to compile all the mutants ranges from 3,157 to 151,234 seconds, for an average of 0.96 seconds required to compile a single mutant, across subjects. Because of our selective compilation strategy (see Section 3.3), the time required by our pipeline to compile a single mutant is significantly lower than the one required by state-of-the-art approaches, which is around 4.6 seconds per mutant, even when software components have a lower number of LOC than our subjects [43].

To collect the data required to address research questions RQ2 to RQ8, for every subject and unique mutant generated by Step 3, we have executed all the test cases covering the mutated statement. Table 4 provides, for every subject, the overall number of executed test cases and the total execution time required for our experiments. In total, the entire experiment took 1,912,662 minutes (31,878 hours or more than 1300 days). Test execution time depends on the number of mutants, the number of executed test cases, and the test suite level (e.g., system test suites exercise more complex scenarios than unit or integration test suites).

7. MASS can be retrieved also using the following DOI: https://doi.org/10.5281/zenodo.5235941

TABLE 4: Scale of experiments.

Subject	Total test cases executed	Total execution time (minutes)
$ESAIL_S$	302,158	1,624,336
LIBN	771,274	33,756
LIBP	1,094,800	7,205
LIBU	4,481,295	57,732
MLFS	170,303,452	189,633
Total	176,952,979	1,912,662

To be able to execute test cases for 31,878 hours, we performed our experiments using the HPC cluster of the University of Luxembourg [108]. The HPC cluster consists of Intel Xeon E5-2680 v4 (2.4 GHz) nodes. To perform our experiments, we tested 100 mutants in parallel, each one on a dedicated node.

In the following sections, we discuss statistical significance using a non-parametric Mann Whitney U-test (with α = 0.05) [109]. We discuss effect size based on Vargha and Delaney's A_{12} statistics, a non-parametric effect size measure [109], [110]. Based on A_{12}, effect size is considered small when $0.56 \leq A_{12} < 0.64$, medium when $0.64 \leq A_{12} < 0.71$, large when $A_{12} \geq 0.71$. Otherwise the compared samples are considered equivalent, that is to be drawn from the same population [110].

4.3 RQ1

Design and measurements

RQ1 aims to determine the cost savings provided by trivial compiler optimization techniques [18], [43]. To do so, we assess the number of equivalent and duplicate mutants discarded by these techniques and the additional costs introduced by the augmented compilation process. Since the mutants detected by these techniques are a subset of the overall set of equivalent and duplicate mutants, we refer to them as *trivially equivalent* and *trivially duplicate* mutants.

For every subject, we compile every mutant six times, each time with a different optimization level enabled. We consider all the available optimization levels for the GCC compiler, which are *-O0*, *-O1*, *-O2*, *-O3*, *-Os*, and *-Ofast* [111]. Level *-O0* indicates that no optimization is applied. Levels *-O1*, *-O2*, *-O3*, and *-Ofast*, in this order, enable an increasing number of optimization options (e.g., level *-Ofast* includes all the optimizations of level *-O3* plus two additional ones, which are *-ffast-math* and *-fallow-store-data-races*). Level *-Os* enables all *-O2* optimizations except those that increase code size. This is the first reported experiment including options *-Os* and *-Ofast* in such an analysis.

To identify the most effective compiler optimization level, we consider the percentage of trivially equivalent and trivially duplicate mutants they detect. Also, to discuss how complementary different optimization levels are and, therefore, whether they should be combined, we report the number of mutants identified as equivalent and duplicate by each compiler optimization only (hereafter called *univocal-trivially-equivalent mutants* and *univocal-trivially-duplicate mutants*).

To further assess the different optimization levels, we analyze the distribution of trivially equivalent and duplicate mutants across the different mutation operators considered in our study. Last, we compare our results with the ones reported in related work [18].

TABLE 5: RQ1. Proportion (%) of Trivially-Equivalent and Trivially-Redundant Mutants Detected by Compiler Optimizations.

Subject	Equivalent									Duplicate								
	All	Overall%	-O0-3	-O0	-O1	-O2	-O3	-Os	-Of	All	Overall%	-O0-3	-O0	-O1	-O2	-O3	-Os	-Of
ESAIL-CSW	8,861	7.27	7.13	34.18	95.25	96.15	95.61	97.44	-	35,133	28.83	27.74	39.94	59.80	61.96	61.52	62.46	-
LIBN	701	8.90	8.90	25.11	97.43	72.90	74.04	44.94	47.93	2,655	33.70	27.63	37.55	66.14	50.17	57.55	63.65	62.98
LIBP	450	6.99	6.89	33.56	95.11	94.00	94.44	96.89	94.44	2,076	32.24	31.44	43.79	63.10	64.84	64.84	64.93	64.84
LIBU	1,366	6.74	6.65	28.84	93.48	90.26	91.87	91.29	91.87	4,392	21.67	21.26	47.65	70.40	75.59	76.71	77.03	76.71
MLFS	361	1.29	1.04	31.86	63.71	77.84	78.12	85.04	81.16	6,356	22.64	20.13	37.54	51.04	56.95	57.08	58.72	61.28
Total	11,739	6.36	6.22	32.14	89.95	93.39	89.93	95.50	17.86	50,612	27.43	25.99	40.46	60.09	62.37	62.83	62.96	20.66

TABLE 6: RQ1. Proportion (%) of Univocal-Trivially-Equivalent and Univocal-Trivially-Redundant Mutants Detected.

Subject	Univocal-Equivalent								Univocal-Duplicate							
	All	% of Equivalent	-O0	-O1	-O2	-O3	-Os	-Of	All	% of Duplicate	-O0	-O1	-O2	-O3	-Os	-Of
ESAIL-CSW	237	2.67	0.00	4.64	0.84	19.83	**74.68**	0.00	827	2.35	7.26	11.61	3.26	28.78	**49.09**	0.00
LIBN	112	15.98	0.00	**97.32**	0.00	2.68	0.00	0.00	202	7.61	0.99	3.47	0.00	0.00	**48.02**	47.52
LIBP	11	2.44	0.00	45.45	0.00	0.00	**54.55**	0.00	44	2.12	0.00	38.64	6.82	0.00	**54.55**	0.00
LIBU	96	7.03	1.04	77.08	2.08	0.00	**19.79**	0.00	148	3.37	4.05	34.46	4.05	0.00	**54.73**	2.70
MLFS	65	18.01	0.00	9.23	0.00	0.00	60.00	**30.77**	471	7.41	0.85	2.55	0.85	0.00	37.37	**58.39**
Total	521	4.44	0.19	18.43	2.11	9.02	**50.67**	3.84	1,692	3.34	4.26	10.82	2.36	14.07	**46.34**	22.16

TABLE 7: RQ1. Compilation Time Required by Compiler Optimization Techniques.

Subject	Time required to compile all the mutants(sec)						
	-O0	-O1	-O2	-O3	-Os	-Of	All O* (hours)
ESAIL-CSW	135,455	132,528	149,088	145,620	151,234	N/A	198
LIBN	11,053	11,256	11,079	11,425	10,299	11,424	18
LIBP	9,036	9,246	9,352	9,392	8,794	9,442	16
LIBU	25,228	26,914	28,564	30,624	26,845	29,583	47
MLFS	2,176	2,509	3157	3,167	3,052	3,164	5
Total	182,948	182,453	201,240	200,228	200,224	53,613	284

By default, subjects are compiled with different compiler optimization options, *-Os* for *ESAIL-CSW*, *-O2* for *MLFS*, *-O3* for *LIBN*, *LIBP*, and *LIBU*. To estimate the costs entailed by the augmented compilation process, we collect, for every subject, the time required for compiling all their mutants with each of the five optimization levels enabled. To compile mutants, we rely on the optimized compilation process implemented in Step 3 of *MASS* (see Section 3.3).

Results

Table 5 provides the results concerning the detection of trivially equivalent and trivially duplicate mutants. We report the total number of such mutants detected for each subject (column *All*), their percentage with respect to the set of mutants successfully compiled (column *Overall %*), and the percentages obtained with the options included in related work (i.e., by discarding the equivalent and duplicate mutants detected by *-O0*, *-O1*, *-O2*, and *-O3*, reported in column *-O0-3*). The proportion of trivially equivalent mutants detected with optimization levels O0-O3 (6.22%) is in line with related work [18] (7%) and the range observed for the different subjects (i.e., 1.04% to 8.90%) largely overlaps with the range observed in related work (2%-10%). For trivially duplicate mutants, instead, we observe a slightly larger set of duplicate mutants when compared to related work. Optimization levels O0-O3 determine that 25.99% of the mutants are trivially duplicate, while in related work the average is around 21%. Finally, optimization levels *-Os* and *-Of* enable the detection of additional trivially equivalent and duplicate mutants, thus leading to an average of 6.36% and 27.43% of the mutants being discarded, respectively (see column *Overall %*). In particular, **we observe that the optimization level *-Os*, not evaluated by related work, is the most effective**. Our results confirm the effectiveness of compiler optimizations for removing a significant percentage of equivalent and duplicate mutants.

Table 6 provides additional details about the trivially equivalent and duplicate mutants univocally detected by the different optimization options. In total, 4.44% and 3.34% of these mutants are univocally detected by one optimization level (see columns *% of Equivalent* and *% of Duplicate*); moreover, since all optimization options contribute to the univocal detection of equivalent and redundant mutants, **it is preferable to rely on all the available compiler optimization options**. Overall, the most effective optimization option is *-Os*, which detects 50.67% and 46.34% of univocal-equivalent and univocal-duplicate mutants, respectively. It is followed by *-O1*, detecting 18.43% and 10.82% of such mutants, respectively. These results suggest that, when the number of compilation runs must be limited, then *-Os* and *-O1* should be prioritized over the other options. This is interesting since **stronger optimization levels such as *-Ofast* and *-O3* do not contribute more than *-Os* to the detection trivially equivalent and duplicate mutants**. Surprisingly, the optimization level *-O0*, which does not enable any compiler optimization option, can detect trivially equivalent and trivially redundant mutants not detected by other optimization levels. However, this seems to highly depend on the code surrounding the mutated statement. For example, in one *LIBU* mutant the optimization level *-O0* detected that `if ( ptr )` is equivalent to `if( ptr > NULL )`, with `ptr` being a pointer, which was not detected with other optimization levels; however, the same instructions are detected as equivalent by other optimization levels when they belong to other functions (i.e., when the code surrounding them is different than the one appearing in the *LIBU* mutant).

Details about the distribution of trivially equivalent and duplicate mutants per mutation operator are reported in Appendix A.

Table 7 provides the time required to compile the artifacts with the different optimization levels. Different from related work, which reports that optimization levels, in the worst case, lead an increase in compilation time by a factor

of 5, **we do not observe a large difference in compilation time among the different optimization levels**. Indeed, in the worst case (i.e., option -*Os*) this factor is 1.1, an increase of 10%. This directly results from the *MASS* compilation pipeline, which minimizes the number of source files that need to be compiled. If developers can accept a compilation time increased by a factor of 5, as suggested in related work, all the compilation optimization levels can be applied, thus maximizing the number of equivalent and duplicate mutants being detected. In three out of five subjects, it takes less than a day to compile all the mutants with all the available optimization levels, which is acceptable, given the cost saved in subsequent steps. For the cases in which it may take multiple days, our practical solution consists in executing the compilation of the various mutants in parallel (e.g., on Cloud systems); for example, our toolset includes scripts to parallelize mutants compilation on HPC and cloud platforms. In the case of *ESAIL-CSW*, the parallel compilation of 142,763 mutants, with the four available compilation options, can be performed in 90 minutes using 100 nodes.

4.4 RQ2 - Accuracy of Mutant Sampling Methods

Design and measurements

RQ2 aims to investigate to what extent the mutation score computed from a sample of mutants (hereafter, *estimated mutation score*) accurately estimates the mutation score of the complete set of mutants (hereafter, *actual mutation score*).

In our study, we consider the sampling strategies which are part of *MASS*, as justified earlier: *proportional uniform sampling*, *proportional method-based sampling*, *uniform fixed-size sampling*, and *uniform FSCI sampling*.

Because of the complexity and size of space software, combined with its high test execution cost, we are interested in selecting a very small subset of mutants. For this reason, to evaluate *proportional uniform sampling* and *proportional method-based sampling*, we consider sampling ratios ranging from 1% to 10%, in steps of 1%. Further, we also cover the range 10% to 100%, in steps of 10%. To evaluate *uniform fixed-size sampling*, consistent with our earlier discussion, we consider a number of mutants in the range 100 to 1000, in steps of 100. Finally, to evaluate *proportional method-based sampling*, we consider a threshold for the confidence interval (i.e., T_{CI}) that ranges from 0.05 to 0.10, in steps of 0.01, with a confidence level of 95%, which is a common choice. The experiments conducted to address RQ2 entail the execution of the entire test suite for every sampled mutant. Executions with a prioritized and reduced test suite are addressed in Section 4.6. The evaluation of different values for T_{CI} enable us to determine the costs associated with a more accurate estimation of the mutation score, in order to better understand the trade-offs.

We compute the actual mutation score of each system by executing the test suite against all the mutants that were successfully compiled, excluding mutants detected as being equivalent or duplicate by simple compiler optimization techniques (see RQ1). For each sampling ratio, to account for randomness, we repeat the analysis 100 times, i.e., we compute the mutation score 100 times, based on 100 randomly selected subsets of mutants. Since it is not feasible to test all the mutants generated for *ESAIL-CSW*, as discussed above, we focus on $ESAIL_S$.

TABLE 8: Actual mutation scores across subjects.

Subject	Mutants	Killed	Live	Mutation Score (%)
$ESAIL_S$	3,536	2,311	1,225	65.36
LIBN	4,982	3,270	1,712	65.64
LIBP	3,931	2,717	1,214	69.12
LIBU	14,574	10,376	4,198	71.20
MLFS	21,375	17,484	3,981	81.80

TABLE 9: RQ2. Accuracy of proportional uniform sampling.

	LIBN		*LIBP*		*LIBU*		*MLFS*		$ESAIL_S$	
r=	**#M**	δ_{acc}	**#M**	δ_{acc}	**#M**	δ_{acc}	**#M**	δ_{acc}	**#M**	δ_{acc}
0.01	50	13.64	40	12.19	146	7.54	214	**4.90**	36	14.04
0.02	100	10.64	79	10.03	292	6.15	428	**3.19**	71	11.84
0.03	150	10.36	118	7.70	438	**4.20**	642	**3.15**	107	8.84
0.04	200	6.40	158	6.46	583	**3.28**	855	**2.53**	142	7.92
0.05	250	7.07	197	6.98	729	**3.11**	1,069	**2.58**	177	6.39
0.06	299	5.95	236	5.78	875	**2.92**	1,283	**2.24**	213	6.25
0.07	349	5.01	276	5.73	1,021	**2.84**	1,497	**2.24**	248	5.20
0.08	399	5.12	315	5.01	1,166	**3.24**	1,710	**1.71**	283	5.68
0.09	449	**4.53**	354	**3.48**	1,312	**2.15**	1,924	**1.73**	319	**4.55**
0.10	499	**4.61**	394	**4.36**	1,458	**2.10**	2,138	**1.55**	354	**5.26**
0.20	997	**2.81**	787	**3.24**	2,915	**1.57**	4,275	**1.09**	708	**3.52**
0.30	1,495	**2.32**	1,180	**2.24**	4,373	**1.00**	6,413	**0.80**	1,061	**2.56**
0.40	1,993	**1.60**	1,573	**1.64**	5,830	**0.91**	8,550	**0.74**	1,415	**2.04**
0.50	2,491	**1.58**	1,965	**1.49**	7,287	**0.67**	10,688	**0.48**	1,768	**1.50**
0.60	2,990	**1.18**	2,358	**1.25**	8,745	**0.66**	12,825	**0.45**	2,122	**1.28**
0.70	3,488	**1.05**	2,750	**1.07**	10,202	**0.43**	14,963	**0.34**	2,476	**1.16**
0.80	3,986	**0.61**	3,143	**0.88**	11,660	**0.38**	17,100	**0.25**	2,829	**0.92**
0.90	4,484	**0.51**	3,534	**0.43**	13,117	**0.26**	19,238	**0.17**	3,183	**0.55**

Note: #M, number of mutants. Accurate results (i.e., $\delta_{acc} \leq 5\%$) are in bold.

Our goal is to determine if the estimated mutation score is an accurate estimate of the actual mutation score. This happens when the estimated mutation score differs from the actual mutation score for less than a small delta (hereafter, accuracy delta, δ_{acc}) for a large percentage of the runs (e.g., 95%). We thus study the distribution of the difference between the estimated and actual mutation scores across all runs. More precisely, we estimate the 2.5% and 97.5% quantiles[8]. Since these two quantiles delimit 95% of the population, we consider the mutation scores to be accurately estimated when they are within a pre-defined small range of the actual score $[-\delta_{acc};+\delta_{acc}]$. In other words, we consider the estimated mutation score to be accurate when the absolute value of the largest difference between quantiles and the actual score is below δ_{acc}. Since the range of acceptable mutation score values is small (75%-100%, see Section 2.1), we decided to use a threshold of 5%, which is more conservative than that reported in related work [10].

Below, we analyze δ_{acc} for varying sampling rates. To improve readability, we discuss the results concerning the different sampling strategies separately.

Results - proportional uniform sampling

Table 8 reports on the mutation scores obtained with the entire test suite for all subjects. As expected, the best mutation score is obtained for *MLFS*, whose test suite achieves MC/DC coverage.

Table 9 provides accuracy results (column δ_{acc}) for proportional uniform sampling for a range of sampling rates (r). To enable comparisons across sampling methods, Column *#M* reports the number of mutants sampled for

8. We rely on linear interpolation using the type 8 algorithm suggested in Hyndman and Fan [112]. It does not make assumptions about the underlying distribution.

TABLE 10: RQ2. Accuracy of proportional method-based sampling.

	LIBN		*LIBP*		*LIBU*		*MLFS*		*ESAIL*$_S$	
r=	**#M**	δ_{acc}	**#M**	δ_{acc}	**#M**	δ_{acc}	**#M**	δ_{acc}	**#M**	δ_{acc}
0.01	19	23.53	15	22.45	111	9.51	232	5.50	33	13.84
0.02	75	11.67	77	11.36	250	**4.82**	447	**4.29**	64	16.18
0.03	131	6.88	120	8.82	422	**4.76**	661	**2.53**	104	8.63
0.04	194	6.52	165	6.73	564	**3.98**	881	**2.91**	137	9.93
0.05	258	**4.90**	208	6.36	731	**3.97**	1,094	**1.96**	178	6.84
0.06	312	**3.78**	254	5.54	905	**2.62**	1,306	**1.86**	223	6.18
0.07	368	**4.21**	290	5.72	1,045	**2.11**	1,517	**1.72**	254	5.72
0.08	417	**4.51**	335	5.22	1,197	**2.36**	1,733	**1.59**	287	**4.67**
0.09	466	**4.61**	378	**4.72**	1,353	**2.51**	1,942	**1.37**	331	**4.59**
0.1	515	**3.03**	413	**4.49**	1,512	**2.20**	2,159	**1.23**	364	**3.87**
0.2	1,030	**2.78**	811	**2.95**	2,963	**1.75**	4,295	**1.26**	721	**2.95**
0.3	1,523	**1.95**	1,210	**2.15**	4,446	**1.38**	6,430	**0.75**	1,079	**2.76**
0.4	2,045	**1.35**	1,605	**1.52**	5,942	**0.76**	8,576	**0.72**	1,432	**1.81**
0.5	2,518	**1.10**	1,985	**1.46**	7,361	**0.79**	10,702	**0.49**	1,779	**1.42**
0.6	3,035	**0.84**	2,395	**1.19**	8,853	**0.60**	12,863	**0.46**	2,139	**1.22**
0.7	3,550	**0.82**	2,791	**0.70**	10,334	**0.46**	15,007	**0.40**	2,494	**1.03**
0.8	4,027	**0.48**	3,178	**0.70**	11,779	**0.34**	17,134	**0.23**	2,849	**0.86**
0.9	4,527	**0.39**	3,574	**0.42**	13,228	**0.24**	19,272	**0.16**	3,202	**0.62**

Note: #M, number of mutants. Accurate results (i.e., $\delta_{acc} \leq 5\%$) are in bold.

each sampling rate. As expected, a larger sampling rate leads to more accurate results (i.e., low δ_{acc}). We notice that for test suites that ensure MC/DC coverage (i.e., *MLFS*), even a very small sampling ratio (i.e., 0.01) guarantees a δ_{acc} below 5%. However, to achieve an accurate mutation score estimate across all subjects, a minimum sampling rate of 0.09 is required.

In addition, we observe that, for $r = 0.09$, the worst results (highest deltas) are observed for smaller projects, which indicates that **the estimation accuracy may not depend on the percentage of sampled mutants but on the size of the sample**; indeed, for most of the subjects, accurate results (i.e., $\delta_{acc} < 5\%$) are obtained with a number of mutants between 350 and 450. This aspect is further studied when considering *uniform fixed-size sampling* and *uniform FSCI sampling*.

Results - proportional method-based sampling

Table 10 shows the accuracy results for proportional method-based sampling. Interestingly, for two subjects (i.e., *LIBN* and *LIBU*), proportional method-based sampling leads to accurate estimates of the mutation score with a lower number of mutants than proportional uniform sampling (i.e., around 250). However, to achieve accurate results with all subjects, we need a minimal sampling rate of $r = 0.09$, as for proportional uniform sampling, which, in the case of method-based sampling, leads to a slightly higher number of mutants. For this reason, we do not see any benefit in using method-based sampling.

Results - uniform fixed-size sampling, uniform FSCI sampling

Table 11 shows the accuracy results for uniform fixed-size sampling and uniform FSCI sampling. For each subject, we sort results according to the number of mutants. For FSCI sampling, we report the confidence interval threshold T_{CI}.

The best results (i.e., lowest number of mutants with $\delta_{acc} \leq 5\%$) are obtained using FSCI sampling with $T_{CI} = 0.10$. Predictably, FSCI sampling with $T_{CI} = 0.10$ guarantees $\delta_{acc} \leq 5\%$ (half of T_{CI}); indeed, by construction, if our assumptions on the limited correlation between mutants and the mutation score following a binomial distribution hold (see Section 2.4), FSCI sampling with $T_{CI} = 0.10$ is expected to guarantee $\delta_{acc} \leq 5\%$ (see Appendix B for further details on the distribution of the mutation score across subjects).

In addition, our results suggest that a limited number of mutants (between 300 and 400) is required to achieve the desired δ_{acc}. This sample size is much lower than the (worst case) sample size proposed by Gopinath et al., which is 1,000 [10]. Also, our sample size is smaller than the one estimated, for a mutation score between 60% and 80%, by approaches based on confidence-interval estimation, which is still around 1,000 [78]. However, we confirm the finding of Gopinath et al., who demonstrated that the binomial distribution accurately estimates the mutation score [10].

To summarize, this is the first study demonstrating that **FSCI sampling is the best approach for obtaining the smallest sample size while providing guarantees on the accuracy of mutation score estimates.** We therefore propose a better solution than that of Gopinath et al., who provide an upper bound for the number of mutants to be considered in uniform fixed-size sampling, since we have evidence suggesting that FSCI sampling helps to select a significantly smaller sample size for a desired confidence interval.

4.5 RQ3 - SDL accuracy

Design and measurements

RQ3 assesses if mutants generated using only deletion operators can accurately estimate the mutation score of the complete mutants set.

To this end, we study the difference between the mutation score obtained by executing the entire test suite on the mutants generated with all the operators (i.e., the actual mutation score) and the mutation score obtained with either (1) the mutants generated with the SDL operator only, or (2) the mutants generated with both the SDL and OODL operators. As for RQ2, to be accurate, the mutation score obtained with a subset of operators should differ by at most 5%.

Results

In Table 12, column # *Mutants* shows, for each subject, the number of mutants generated with either the SDL operator or both the SDL and OODL operators. Column *Mutation score* shows the mutation score obtained when using the entire test suite to exercise the mutants generated with either all the operators, the SDL operator only, or both the SDL and OODL operators. Between parentheses, we also report the difference between the mutation score obtained with all the operators and that obtained with a subset of operators. Results show that, for some of our subjects, the mutation score obtained with the SDL operator does not accurately estimate the mutation score obtained with a broader set of operators. Though these results do not invalidate related work [49], whose focus is on the evaluation of the strength of SDL and OODL operators, it shows that **SDL and OODL operators should not be adopted to estimate the mutation score computed with a larger set of operators**. We leave the evaluation of the strength of SDL and OODL operators to future work.

TABLE 11: RQ2. Accuracy with uniform fixed-size sampling and uniform FSCI sampling.

LIBN			*LIBP*			*LIBU*			*MLFS*			*ESAIL$_S$*		
#Mutants	δ_{acc}	**Method**	**#Mutants**	δ_{acc}	**Method**	**#Mutants**	δ_{acc}	**Method**	**#Mutants**	δ_{acc}	**Method**	**#Mutants**	δ_{acc}	**Method**
100	9.88	FIXED	100	10.17	FIXED	100	7.32	FIXED	100	7.80	FIXED	100	8.89	FIXED
200	5.86	FIXED	200	6.12	FIXED	200	5.73	FIXED	200	5.20	FIXED	200	6.14	FIXED
300	**4.48**	FIXED	300	5.88	FIXED	300	5.37	FIXED	248	**4.56**	FSCI 0.1	300	5.53	FIXED
364	**4.42**	FSCI 0.1	346	**4.26**	FSCI 0.1	333	**4.73**	FSCI 0.1	300	**4.04**	FIXED	366	**3.92**	FSCI 0.1
400	**5.49**	FIXED	400	**4.27**	FIXED	400	**4.45**	FIXED	302	**4.64**	FSCI 0.09	400	**4.52**	FIXED
447	**3.70**	FSCI 0.09	425	**3.79**	FSCI 0.09	409	**4.08**	FSCI 0.09	379	**4.01**	FSCI 0.08	449	**3.66**	FSCI 0.09
500	**3.85**	FIXED	500	**3.63**	FIXED	500	**3.80**	FIXED	400	**3.80**	FIXED	500	**4.08**	FIXED
564	**3.53**	FSCI 0.08	536	**3.75**	FSCI 0.08	514	**3.94**	FSCI 0.08	490	**3.90**	FSCI 0.07	567	**3.03**	FSCI 0.08
600	**3.65**	FIXED	600	**3.72**	FIXED	600	**3.29**	FIXED	500	**3.11**	FIXED	600	**3.73**	FIXED
700	**3.00**	FIXED	696	**2.89**	FSCI 0.07	668	**3.99**	FSCI 0.07	600	**2.89**	FIXED	700	**3.01**	FIXED
734	**3.62**	FSCI 0.07	700	**3.27**	FIXED	700	**3.30**	FIXED	667	**2.78**	FSCI 0.06	738	**2.77**	FSCI 0.07
800	**2.90**	FIXED	800	**2.51**	FIXED	800	**3.26**	FIXED	700	**2.80**	FIXED	800	**2.55**	FIXED
900	**3.09**	FIXED	900	**2.50**	FIXED	900	**3.04**	FIXED	800	**2.44**	FIXED	900	**2.37**	FIXED
994	**3.00**	FSCI 0.06	945	**2.42**	FSCI 0.06	906	**3.28**	FSCI 0.06	900	**3.02**	FIXED	998	**2.65**	FSCI 0.06
1,000	**2.41**	FIXED	1,000	**2.72**	FIXED	1,000	**2.31**	FIXED	960	**2.31**	FSCI 0.05	1,000	**2.96**	FIXED
1,422	**2.44**	FSCI 0.05	1,352	**1.93**	FSCI 0.05	1,298	**2.72**	FSCI 0.05	1,000	**2.35**	FIXED	1,429	**1.70**	FSCI 0.05

Accurate results (i.e., $\delta_{acc} \leq 5\%$) are in bold.

TABLE 12: Comparison of mutation scores obtained with mutants generated using all operators, the SDL operator only, and the SDL + OODL operators.

Subject	**# Mutants**		**Mutation score**		
	SDL	SDL+OODL	ALL	SDL	SDL+OODL
ESAIL$_S$	701	974	65.36	61.91 (-3.45)	63.45 (-1.91)
LIBN	912	1,546	65.64	70.72 (+5.08)	71.35 (+5.71)
LIBP	731	1,324	69.12	64.84 (-4.28)	66.39 (+2.73)
LIBU	2,341	3,811	71.20	73.26 (+2.06)	72.63 (+1.43)
MLFS	1,729	5,971	81.80	85.71 (3.91)	88.03 (+6.23)

4.6 RQ4 - Mutation Score Accuracy with PrioritizeAndReduce

Design and measurements

RQ4 assesses whether the mutation score obtained with the reduced and prioritized test suite generated by *MASS* (hereafter, the *MASS-reduced* test suite) accurately estimates the actual mutation score. To this end, we compare the accuracy obtained with the four distance metrics (i.e., D_J, D_O, D_E, and D_C) used by the proposed *PrioritizeAndReduce* algorithm (Figure 2). In addition, to determine to what extent our prioritization strategy based on code coverage contributes to the selection of test cases that kill mutants, we also compare the results obtained with a simple baseline that, for each mutant, randomly selects one test case among the ones that cover the mutant.

For all subjects, we consider (a) the complete set of mutants, (b) the reduced subset of mutants providing accurate results (i.e., the one obtained with FSCI sampling with $T_{CI} = 0.10$). Based on RQ3 results, we exclude mutants generated with the SDL and SDL+OODL operators only. For FSCI sampling, since we evaluate the accuracy of a reduced test suite, we derive the confidence interval using Equation 5.

For each subject and each distance metric, and for each of the two sets of mutants considered, we generated ten different *MASS-reduced* test suites. In the case of FSCI, since it randomly selects mutants, we considered ten different sets of mutants derived with distinct executions of the FSCI algorithm. For each *MASS-reduced* test suite, we computed the mutation score obtained. Then, to determine if the mutation score of the *MASS-reduced* test suite is accurate, we follow the same procedure adopted for RQ2, i.e., we rely on the 2.5/97.5 percentile distance from the actual mutation score.

TABLE 13: RQ4. Mutation score accuracy for the different strategies implemented by *PrioritizeAndReduce*

		δ_{acc} for different prioritization strategies				
Subject	**Mutants set**	Random	D_J	D_O	D_E	D_C
LIBN	ALL	7.2055	1.3455	1.3100	0.7300	0.7300
	FSCI 0.10	>5%	3.87	3.80	4.14	4.14
LIBP	ALL	7.7927	0	0	0	0
	FSCI 0.10	>5%	3.22	3.22	3.22	3.22
LIBU	ALL	3.1400	0.0300	0.0300	0.0199	0.0199
	FSCI 0.10	>5%	1.95	1.95	1.95	1.95
MLFS	ALL	6.721	0.3299	0.3299	0.0199	0.0300
	FSCI 0.10	>5%	2.97	2.97	2.85	2.85
ESAIL$_S$	ALL	38.8885	24.1688	24.3650	4.0800	3.9833
	FSCI 0.10	>5%	2.87	2.87	2.67	2.49

Results

Table 13 provides the values of δ_{acc} obtained for the different subjects and distance metrics (i.e., the random baseline and the four distance metrics supported by *MASS*). Rows named *ALL* report the results obtained when executing the entire set of mutants, rows named *FSCI* report the results obtained with the FSCI strategy.

Unsurprisingly, for all the subjects except *ESAIL$_S$*, the mutation score computed with the entire set of mutants tested with the *MASS-reduced* test suite (i.e., row ALL) is more accurate than the mutation score computed with a subset of the mutants tested with the same test suite (i.e., row FSCI). However, the mutation score estimated with FSCI is always accurate (i.e, $\delta_{acc} < 5$). In the case of *ESAIL$_S$*, where each statement is covered by a large number of test cases (see Section 4.1), test suite reduction has a higher probability of retaining a test case that does not kill a mutant. For this reason, executing a reduced test suite with a subset of mutants selected with FSCI, which estimates the error due to test suite reduction, leads to a more accurate mutation score than executing a reduced test suite with the entire set of mutants without estimating such error (i.e., row ALL).

The only distance metric that consistently leads to inaccurate estimates of the mutation score (i.e., $\delta_{acc} > 5$) across subjects is the random baseline. Based on a non-parametric Mann Whitney test, the difference between the random baseline (i.e., Random) and the four distance metrics implemented by *MASS* (i.e., D_*) is always significant with a p-value < 0.05. This indicates that **the *MASS* distance**

metrics are necessary to accurately estimate the mutation score while reducing the number of test cases to execute.

Among the proposed distance metrics, D_J and D_O provide inaccurate results with $ESAIL_S$. We conjecture the main reason is their inability to account for the number of times a statement is exercised by a test case. We believe this is an important factor as system test cases that repeatedly exercise mutated statements, with different variable values, are more likely to kill mutants than test cases exercising such statements only once (e.g., because of the uncertainty regarding the incorrect intermediate state propagating to the state variables verified by test oracles, see Section 2.1). On the other hand, unit and integration test cases, which exercise much simpler scenarios in other subjects, are more likely to kill a mutant when the mutated statement is executed only once (e.g., because the oracle is closer to the mutated statement). This is why D_J and D_O fare similarly to the other distance metrics for these subjects. In contrast, D_C and D_E are distance metrics ensuring an accurate estimate of the mutation score and providing the lowest δ_{acc}. The differences between D_E and D_C are always statistically significant, except for *MLFS*. However, there are no practically significant differences between them. Since D_C provides a normalized score, which is required by Step 8, we select D_C as the preferred metric to be used in *MASS*.

4.7 RQ5 - Time Savings with PrioritizeAndReduce

Design and measurements

RQ5 assesses to what extent the *MASS-reduced* test suite speeds up the mutation analysis process.

For each subject considered for RQ4, we measure the execution time taken by the *MASS-reduced* test suite to execute on the mutants. We compute time saving as the ratio of the difference in execution time from the original test suite over the time it requires to execute, for the set of mutants selected. In particular, as in RQ4, we consider three scenarios (1) all the mutants are selected, (2) mutants are selected with FSCI sampling, (3) mutants are selected with FSCI sampling but we execute the entire test suite, as opposed to the *MASS-reduced* test suite. By considering scenario (3), we can estimate both the time saved thanks to mutant sampling and the additional time saved when combining it with the *MASS-reduced* test suite. For the original test suite, to emulate a realistic mutation analysis process according to state-of-the-art solutions, we measure the time required to execute test cases until the mutant is killed (for live mutants it means that we execute the entire test suite). Also, we set a test case timeout equal to three times the duration of the original test case.

Since the execution time of test cases depends on multiple factors such as the underlying test harness, the development practices in place (e.g., verifying multiple scenarios in a single test case), and the type of testing conducted (e.g., unit, integration, system), we also compute the ratio of the number of test cases not executed by the *MASS-reduced* test suite over the total number of test cases.

Results

Table 14 reports, for every subject, the time required to test all the mutants with the entire test suite, in seconds. It also reports the total number of test cases executed. We observe that mutation analysis requires a large amount of time. It takes between 13 and 73 hours for subjects tested with unit and integration test suites (which are faster to execute). When a system test suite needs to be executed (i.e., $ESAIL_S$), traditional mutation analysis becomes infeasible as shown by the 11,000 hours required to perform mutation analysis with $ESAIL_S$.

Figures 4 and 5 provide boxplots depicting the saving achieved when the *MASS-reduced* test suite is executed with all the mutants and with the FSCI-selected mutants, in terms of execution time and test cases, respectively. Each observation corresponds to the saving obtained with one of the ten executions performed on each subject, for a specific configuration (i.e., distance D_* and strategy adopted for selecting mutants). In Figures 4 and 5, horizontal dashed lines show the average across all subjects, whiskers are used to identify outliers (i.e., they are placed below/above 1.5*Inter Quartile Range of the upper quartile/lower quartile). A detailed table including min, max, mean, and median values for each subject is provided in Appendix C.

Unsurprisingly, the smallest reduction in execution time and number of executed test cases is achieved when executing the *MASS-reduced* test suite with all the mutants. Measuring such time reduction allows us to evaluate the benefits of test suite prioritization and selection when it is not combined with mutant selection. Excluding outliers, execution time reduction goes from -0.52% to 16.82% and appears to be correlated with the reduction in number of test cases to execute, which varies from 4.80% to 33.17%. The largest reductions are obtained with D_J and D_O (the median is 13.36 and 13.40, respectively), which do not consider differences in coverage frequency, thus removing the largest number of test cases; however, D_J and D_O also lead to the worst accuracy results according to RQ4. Metrics D_C and D_E, instead, lead to limited benefits in terms of time reduction.

A negative reduction indicates that the reduced and prioritized test suite increases the execution time of the mutation analysis process. This happens when (1) test cases are sorted in such a way that test cases that kill the mutants are executed later with respect to the original test suite, (2) test cases that kill the mutants but have long execution times (e.g., because they trigger a timeout) are executed before test cases with shorter execution times that kill the mutants. Negative reduction, however, affects only a few executions, thus showing that a reduced and prioritized test suite tends overall to be beneficial to the mutation analysis process.

Mutant sampling alone contributes to a high reduction in execution time; indeed, in Figure 4, the boxplot *FSCI 0.1-Full TS*, depicting the time saving for FSCI mutants tested with the entire test suite, shows minimum, median, and maximum values of 65.76%, 75.10%, and 84.72%. Indeed, by reducing the number of mutants to execute, FSCI sampling significantly reduced the total number of test cases to execute within a [49.98% - 80.54%] range (as shown by the boxplot in Figure 5).

The highest reduction in execution time is achieved when combining the *MASS-reduced* test suite with FSCI sampling. It ranges from 72.09% to 90.83%. D_C and D_E, which are the approaches that guarantee accurate results,

TABLE 14: Execution time and number of test cases executed when mutation analysis is based on the original test suite.

Subject	Execution time, seconds (hours)	# Test cases
ESAIL$_S$	39,604,457 (11,001)	155,751
LIBN	252,776 (70)	10,250
LIBP	47,949 (13)	6,629
LIBU	214,016 (59)	17,672
MLFS	171,790 (47)	28,159
Total	40,290,988 (11,191)	218,461

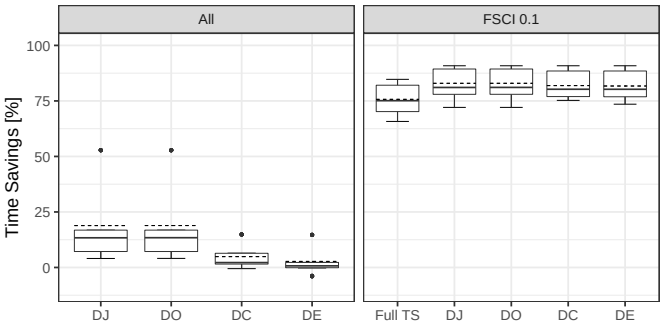


Fig. 4: Time savings for different sets of mutants, with the *MASS-reduced* test suite being generated based on different distance measures.

lead to an execution time reduction in the ranges [75.25% - 90.83%] and [73.53% - 90.83%], respectively, an impressive achievement. Test case savings, as well, are above 65%, [68.28% - 83.45%] and [68.08% - 82.70%] for D_C and D_E, respectively. Based on savings results, there is no practical difference between D_C and D_E.

To conclude, **we suggest to combine FSCI sampling with the *MASS-reduced* test suite to minimize the time required by mutation analysis**. For *ESAIL$_S$*, on average, when combining the *MASS-reduced* test suite with FSCI sampling, mutation analysis time goes from 11,000 hours to 1,531 hours, which makes mutation analysis feasible in one week with 10 computing nodes. Note that for safety or mission critical systems, such as satellites software, the cost of using computing nodes is minimal compared to the development cost of the entire system. Indeed, to test such systems, even paying for the computational power of 100 HPC nodes to make mutation analysis feasible in half a day, is economically justifiable. Otherwise, without *MASS*, mutation analysis leads to 11,000 hours of test cases execution, thus being practically infeasible since it would require more than 100 days to be completed, even with 100 HPC nodes.

Our results also show that when it is not feasible to collect coverage data for the mutants under test (a requirement to generate the *MASS-reduced* test suite), **FSCI sampling alone, without a reduced test suite, may still provide a high reduction in execution time**. In the case of *ESAIL$_S$*, this leads to 2,920 mutation analysis hours, which require less than two days with 100 HPC nodes.

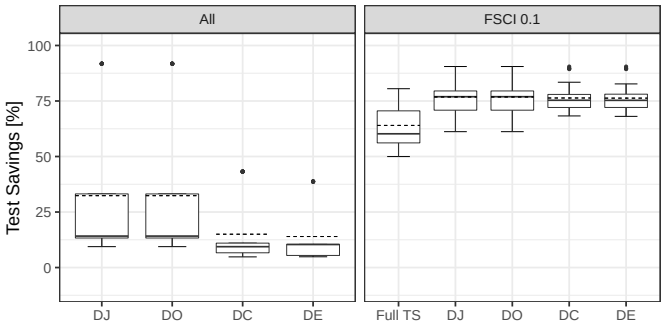


Fig. 5: Test case savings for different sets of mutants, with the *MASS-reduced* test suite being generated based on different distance measures.

4.8 RQ6 - Precise Detection of Equivalent and Duplicate Mutants

Design and measurements

RQ6 investigates if it is possible to identify thresholds that enable the accurate identification of mutants that are nonequivalent (T_E) and nonduplicate (T_D), following the procedure described in Section 3.7.

To determine T_E and T_D, we rely on the optimal distance metric identified in RQ4 (D_C). We analyze precision and recall of the results obtained for different values of T_E and T_D. To determine T_E, we measure precision as the percentage of mutants with a distance above T_E that are nonequivalent, recall as the percentage of nonequivalent mutants with a distance above T_E. To determine T_D, we measure precision as the percentage of mutant pairs with a distance above T_D that are duplicate, recall as the percentage of duplicate mutant pairs that have a distance above T_D.

Since the quality of results might be affected by both test suite reduction (i.e., less coverage data may be available) and mutants sampling (e.g., less mutants might be sampled), consistent with the finding of previous RQs, we consider the following two configurations:

- Execution of the original test suite with all the generated mutants (ALL)
- Execution of the *MASS-reduced* test suite with FSCI sampling (*MASS*)

We determine the values of T_E and T_D based on the analysis of the cumulative distribution of the distance values computed to determine equivalent and duplicate mutants, for the two configurations listed above. Figures 6 and 7 show the cumulative distribution — the Y-axis shows the percentage of mutants and mutant pairs with a distance lower or equal to the value in the X-axis. For both Figures 6 and 7 we can observe that the distribution of mutants is not uniform in the range 0-1 (otherwise we would have straight lines with 45 degree angle) but we observe a large proportion of mutants (Figures 6) and mutant pairs (Figures 7) having small distances. For example, Figure 6 shows that, across all subjects, more than 60% of the mutants have a distance below 0.05 (i.e., with x equal to 0.05, the value of y is above 60).

To evaluate precision and recall, we thus select values for T_D and T_E that either largely differ (i.e., 0.0, 0.4, and 0.8) or

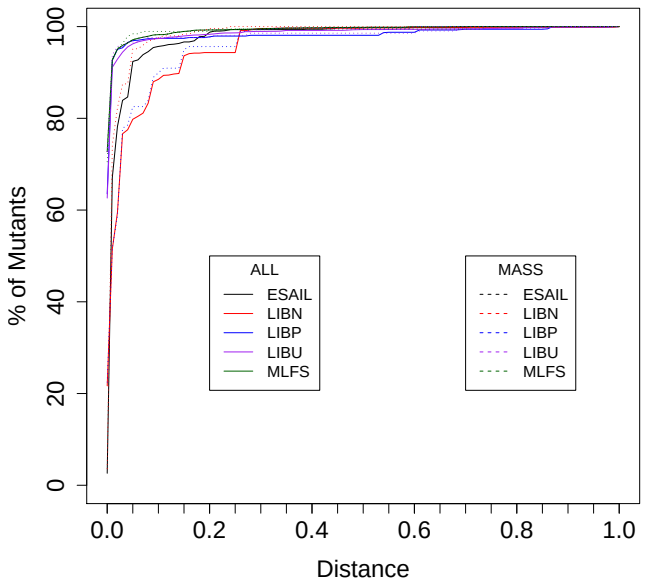


Fig. 6: Cumulative distribution of mutants over distance values computed to determine equivalent mutants.

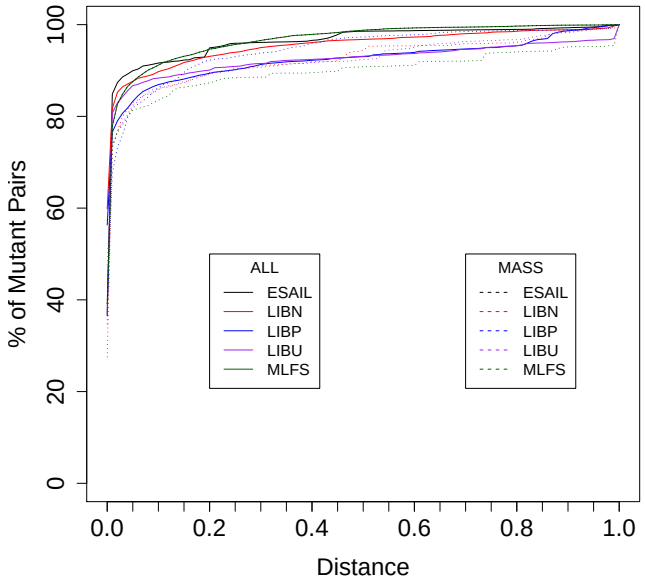


Fig. 7: Cumulative distribution of mutant pairs over distance values computed to determine duplicate mutants.

delimit ranges including a large proportion of the mutants (i.e., 0, 0.01, and 0.05). Table 15 reports the percentage of mutants and mutant pairs belonging to the ranges delimited by the selected values, for the two configurations considered in our study; the distribution of mutants in Table 15 is consistent with Figures 6 and 7.

To compute precision and recall for different values of

TABLE 15: Distribution (%) of mutants in distance ranges.

	Distance for nonequivalent						Distance for nonduplicate					
Config.	(0.00 0.00]	(0.00 0.01]	(0.01 0.05]	(0.05 0.40]	(0.4 0.8]	(0.8 1.0]	(0.00 0.00]	(0.00 0.01]	(0.01 0.05]	(0.05 0.40]	(0.4 0.8]	(0.8 1.0]
ALL	53.9	29.8	10.1	5.5	0.5	0.2	39.2	40.5	7.3	7.6	2.6	2.8
MASS	39.7	36.7	16.4	6.6	0.4	0.1	33.7	37.9	10.5	10.5	3.8	3.3

T_E, since the distribution of mutants is not uniform, we rely on stratified sampling, as follows. We divide all the live mutants into six buckets, based on their distance from the original program, according to the ranges reported in Table 15. We determine the ratio (r_R) of nonequivalent mutants in a specific range R by randomly selecting 20 mutants (four for each subject) and inspecting them with the help of the engineers who developed the software. We rely on r_R to estimate e_R, that is, the number of nonequivalent mutants in the entire set of mutants with a distance within the specific range R

$$e_R = r_R * n_R$$

with n_R being the number of mutants observed in the range R for all the subjects[9]. Based on e_R, we estimate the number of nonequivalent mutants above a threshold and, consequently, compute precision and recall. We perform the analysis for both selected configurations, ALL and *MASS*.

Our aim is to identify a threshold value above which we maximize the number of nonequivalent mutants being selected (high recall) and maximize the number of equivalent mutants being discarded (high precision); since both precision and recall are equally important, we look for a threshold value that maximizes the harmonic mean of precision and recall (F-value).

To determine T_D, we repeated the same procedure as for T_E, except that we considered both killed and live mutants according to the procedure indicated in Section 3.7.

In total, we manually inspected 410 mutants (186 mutants to detect equivalent mutants and 224 mutant pairs to detect duplicate ones), a larger number than that considered in related studies [14]. The number of inspected mutants is lower than the maximum of 480 ({20 mutants + 20 mutant pairs} x 6 buckets x 2 configurations) because the mutant distribution across ranges is not perfectly uniform (see Table 16 for the number of observations per bucket).

Results

The ratio (r_R) of nonequivalent mutants, for the distance ranges reported in Table 16, shows similar results for both configurations (ALL and *MASS*). The differences in distribution between mutants (Table 15) and equivalent mutants (Table 16), for both configurations, is indeed not statistically significant and effect size is negligible. Such similarity suggests that nonequivalent and nonduplicate mutants follow the same distribution for both configurations. This can be explained since FSCI sampling uniformly selects a subset of the mutants considered by the ALL configuration, which includes all mutants.

The 14 nonequivalent mutants leading to $d = 0$ across the two configurations (seven for ALL, seven for *MASS*)

9. n_R can be derived from Table 15

TABLE 16: RQ6. Ratio (r_R) of Nonequivalent/Nonduplicate Mutants per Distance Range.

	Distance range (nonequivalent)						Distance range (nonduplicate)					
Config.	(0.00 0.00]	(0.00 0.01]	(0.01 0.05]	(0.05 0.40]	(0.4 0.8]	(0.8 1.0]	(0.00 0.00]	(0.00 0.01]	(0.01 0.05]	(0.05 0.40]	(0.4 0.8]	(0.8 1.0]
ALL	0.35 (20)	0.85 (20)	1.00 (20)	1.00 (20)	1.00 (18)	1.00 (12)	0.95 (20)	1.00 (20)	1.00 (20)	1.00 (20)	1.00 (20)	1.00 (20)
MASS	0.35 (20)	0.95 (20)	1.00 (20)	1.00 (10)	1.00 (6)	N/A (0)	1.00 (20)	1.00 (20)	1.00 (20)	1.00 (20)	1.00 (18)	1.00 (16)

Note: We report the number of observations in parenthesis.

TABLE 17: RQ6. Precision (P), Recall (R), and their harmonic mean (F) obtained for the Coverage-Based Detection of Nonequivalent/Nonduplicate Mutants.

		Threshold (nonequivalent)						Threshold (nonduplicate)					
Conf.		d $\geq$ 0	d>0	d>0.01	d>0.05	d>0.4	d>0.8	d $\geq$ 0	d>0	d>0.01	d>0.05	d>0.4	d>0.8
ALL	P	0.62	0.90	1.00	1.00	1.00	1.00	0.98	1.00	1.00	1.00	1.00	1.00
	R	1.00	0.71	0.28	0.10	0.01	0.01	1.00	0.62	0.21	0.13	0.05	0.03
	F	0.77	0.80	0.44	0.19	0.03	0.01	0.99	0.77	0.34	0.23	0.10	0.05
MASS	P	0.81	0.97	1.00	1.00	1.00	N/A	1.00	1.00	1.00	1.00	1.00	1.00
	R	1.00	0.89	0.33	0.11	0.01	N/A	1.00	0.63	0.22	0.16	0.06	0.02
	F	0.90	0.93	0.50	0.20	0.02	N/A	1.00	0.77	0.37	0.27	0.11	0.05

have the following characteristics. Four mutants (29%) invalidate data buffers' preconditions (e.g., an array size is indicated as larger than it should be). Since such faults are typically detected through profiling (e.g., by using Valgrind [113]), not detecting such mutants cannot be considered a major weakness of the approach. Seven mutants (50%) affect variables that are not used in the mutated source file (i.e., the one for which we collect code coverage). Static analysis should, in principle, enable the identification of these mutants as nonequivalent. Three mutants (21%) concern the deletion of clauses that are not tested by our test suites; these cases might be detected by our approach after combining statement coverage with additional coverage measures (e.g., clause coverage) to compute distances, but this is left to future work. Based on the above, the percentage of nonequivalent mutants that may potentially indicate limitations of the test suite, cannot easily be detected by other means, and are ignored with T_E, when set to zero, is very low (i.e., three out of 160, or 1.88%). For this reason, **we consider the proposed T_E threshold precise enough to be used for test suite evaluation in a safety context**.

Table 17 provides precision and recall obtained for different T_E values; more precisely, we report the results obtained when all mutants are considered nonequivalent (i.e., $d \geq 0$), along with the results obtained for T_E being set to 0, 0.01, 0.05, 0.4, and 0.8.

We can observe that T_E **set to zero enables the accurate detection of nonequivalent mutants**. Indeed, for $d > 0$, we achieve the highest F-value, and the highest precision and recall, given that a value of 1.00 cannot be achieved simultaneously for precision and recall. These results are in line with related work [11] reporting that a difference in the frequency of execution of a single line of code (i.e., $d > 0$) is indicative of a mutant not being equivalent to the original software. Moreover, these results also indicate that **FSCI mutants sampling and *MASS-reduced* test suite selection enable the accurate identification of nonequivalent mutants based on T_E**.

As for duplicate mutants, based on Table 17, mutants are highly likely to be nonduplicate and thus **it is not possible to determine a threshold to identify duplicate mutants**. Indeed, among all the considered threshold values, the highest F-value is obtained when all the mutants are considered nonduplicate (i.e., $d \geq 0$). These results are in line with related work [114] showing that test suites are unlikely to distinguish nonredundant mutants (i.e., many nonduplicate and nonsubsumed mutants yield the same test results). With test suites that do not distinguish nonredundant mutants, it is very likely that nonduplicate mutants show the same coverage in addition to showing the same results. This is the reason why in Table 16, we observe a large percentage of nonduplicate mutants having the same coverage (i.e., $d = 0$). For this reason, when no methods are available to automatically generate test cases that distinguish subsumed mutants (see Section 2.6), we suggest that all mutants should be considered as nonduplicate when computing the mutation score:

$$MS = \frac{KND}{LNE + KND} \tag{8}$$

where LNE is the number of live, nonequivalent mutants.

4.9 RQ7 - *MASS* Mutation Score

Design and measurements

RQ7 investigates the extent to which the mutation score estimated by MASS with Equation 8, can accurately predict the actual mutation score of the system.

To this end, we apply *MASS* to the five subjects described in Section 4.1 and compute the mutation score according to equation 8. We compare the resulting mutation scores with those obtained with a traditional, non-optimized mutation analysis process that tests all the mutants with the entire test suite and do not discard likely equivalent mutants. Since we have already demonstrated that FSCI, applied to a reduced and prioritized test suite, accurately estimates the mutation score (see RQ4), we discuss the percentage of live mutants that are discarded by means of T_E and the effect it has on the mutation score.

Results

From Table 18, one can see that, on average, the percentage of live mutants that are discarded because considered equivalent is 42.28%, which is in line with related work (i.e., 45% [11]). Across our subjects, such percentage varies from 2.61% ($ESAIL_S$) to 69.37% (*MLFS*), because of nondeterminism. Indeed, complex embedded software, even when generating consistent functional results across multiple runs, may show nondeterminism in their nonfunctional properties (e.g., number of tasks started) when it is not possible to control the resources provisioned by the test environment. For example, in our environment, $ESAIL_S$, which is a system including real-time tasks, show different code coverage for multiple executions of a same test case. The same happens for *LIBN*, a network library, which may execute a different set of instructions based on the current network usage (e.g., ports available on the host OS). Unsurprisingly, in our experiments, the subject having the largest number of predicted equivalent mutants removed is the mathematical library *MLFS*, which should not be affected

by nondeterministic behaviour due to real-time constraints or networking.

To maximize the number of equivalent mutants detected by *MASS*, it is therefore advisable to minimize the sources of nondeterminism. It can be achieved, for example, by executing test cases in a dedicated testing environment, which is standard practice for space software. However, since our analysis concerned the execution of a large, entire set of mutants, not only a sampled subset, we relied on a shared HPC environment. This may have introduced unexpected delays in the execution of the simulator and altered the number of available ports, thus exacerbating nondeterminism.

As expected, the removal of equivalent mutants results in the *MASS* mutation score being higher than that computed with a traditional approach. On average, the score increased by 10.52 percentage points (i.e., from 70.62% to 81.14%).

To provide some additional insights about the software features that, according to *MASS* mutation analysis results, warrant to be verified with additional test cases, we report on the characteristics of manually inspected mutants having $d > 0$ in Table 16. According to our analysis, live mutants concern (1) logging functions (11%), (2) code developed by third parties (5%), (3) time operations (e.g., timeouts, 4%), (4) thread synchronisations (e.g., mutex locks, 5%), (5) memory operations (e.g., malloc and free operations, 20%), and (6) the application logic (55%). Most of these categories either do not need to be tested (cases 1 and 2), or concern operations that are difficult to test (cases 3, 4, and 5) and often verified by other means, e.g., test suites including hardware in the loop or through manual inspection. However, most of the live mutants concerning the application logic have enabled engineers to identify weaknesses in test suites (e.g., corner cases not being tested, scenarios testable with simulators but verified only by test suites with hardware in the loop), which further stresses the importance of mutation analysis in this context. Furthermore, the manual inspection of these live mutants led to the identification of one previously undetected bug since the test suite was not covering a specific combination of boolean clauses in a function, a problem that may occur even when MC/DC adequacy is achieved by test suites [115].

TABLE 18: RQ7. *MASS* Mutation Score.

Subject	Predicted Equivalent Mutants Removed (%)	Mutation Score (%)	
		Traditional	*MASS*
$ESAIL_S$	2.61	65.36	65.95
LIBN	21.67	65.64	70.92
LIBP	63.43	69.12	85.95
LIBU	54.34	71.20	84.41
MLFS	69.37	81.80	93.49
Average	42.28	70.62	81.14

4.10 Discussion

Our results show that the *MASS* pipeline helps to overcome mutation analysis limitations caused by common characteristics of embedded software, present in space systems and, more generally, in similar CPS (e.g., avionics, automotive, and industry 4.0).

Although the need for dedicated hardware and simulators prevent the applicability of optimizations that make use of multi-threading or other OS functions to minimize mutants compilation time [20], we have shown that our selective compilation strategy (see Section 3.3) is sufficient to achieve an efficient mutant compilation process (see Section 4.2).

Trivial compiler optimization approaches are useful to eliminate a large proportion of mutants that are equivalent or duplicate. Based on our results, the presence of functions to deal with signals and data transformation does not limit the effectiveness of trivial compiler optimization approaches, which, across our subjects, enable the removal of 33,38% of the mutants (62,351 out of 184,503). Our results are in line with empirical studies in the literature [18]. However, we show that for pure mathematical software (i.e., *MLFS*), their effectiveness is significantly lower (i.e., 21%, 6,717 out of 31,526).

The time required to perform a traditional mutation analysis process that does not rely on mutants sampling (Step 5 of *MASS*) is particularly high. Indeed it takes between 13 and 70 hours for unit and integration test suites and 11,001 hours for the system level test suite of $ESAIL_S$. These numbers confirm that the thorough testing required by critical CPS software, combined with long test execution times, may exacerbate the scalability problems of mutation analysis. *MASS* applies FSCI-based mutation sampling and executes a prioritized subset of the test suite to address scalability issues. Our results show that such an optimized solution helps address scalability problems to a significant extent by reducing mutation analysis time by more than 70% across subjects. In practice, for large software systems like *ESAIL-CSW*, such reduction can make mutation analysis practically feasible; indeed, with 100 HPC nodes available for computation, *MASS* can perform the mutation analysis of *ESAIL-CSW* in half a day. In contrast, a traditional mutation analysis approach would take more than 100 days, thus largely delaying the development and quality assurance processes. Last, we demonstrated that FSCI-based sampling, a contribution of this paper, outperforms state-of-the-art mutants sampling strategies [9], [10] both in terms of mutation score accuracy and size of the selected mutant set.

Finally, we confirm that a coverage-based approach (*MASS* Step 7) enables the accurate identification of equivalent mutants, thus confirming related work's results [11] in our context. In addition, we demonstrate that such an approach still provides accurate results in the presence of optimizations (i.e., test suite reduction) that may affect the code coverage achieved by mutants. Coverage-based approaches, instead, are not effective in detecting likely duplicate mutants.

This paper focused on investigating and identifying practical solutions to address the scalability of mutation analysis and the pertinence of mutation scores as an adequacy criterion in the context of embedded software for CPS. Important work remains concerning specific subjects in embedded software. For example, our work does not aim to assess if test suites can detect faults concerning the communication between heterogeneous components or the interoperability of different technologies and tools, two typical CPS problems. To address such issues, our future work includes the definition of mutation operators that alter the data exchanged by software components instead of their

implementation.

4.11 Threats to validity

In our experiments, despite their extremely large scale (test execution time over 30 thousands hours in total), the main threats to validity remains generalizability. To address this threat we have selected, as experimental subjects, software developed by our industry partners that is representative of different types of space software and, more generally, software in many cyber-physical systems. Our subjects include utility libraries (*LIBU*, *LIBN*, *LIBP*), mathematical libraries (*MLFS*), and embedded control software (*ESAIL-CSW*) including service/protocol layer functions, satellite drivers, and application layer functions. To help generalize our results to different quality assurance practices, we considered different types of test suites: unit (*MLFS* and *LIBU*), integration (*LIBP* and *LIBN*), and system test suites requiring hardware and environment simulators (*ESAIL-CSW*). Also, our research is monitored by ESA, who evaluates the applicability of the proposed solutions to a range of systems that go beyond the ones considered for our empirical evaluation.

5 Related Work

The mutation testing and analysis literature includes a number of relevant empirical studies involving large open-source systems. Table 19 provides the list of C/C++ benchmarks considered in articles appearing in top software engineering conferences and journals, between 2013 and 2020[10]. Comparisons suggest that the subjects considered in our paper are among the largest considered so far, regarding the size of the SUT and of the test suite. Further, in most of related work, only a subset of the reported subject software had been analyzed. Finally, our benchmark is the only embedded software application among the large subjects considered in the literature.

Based on our analysis, empirical studies on the applicability of mutation testing and analysis to safety-critical, industrial systems are very limited. The most recent and relevant analyses on this topic are those of Ramler et al. [116], Delgado et. al [19], and Baker et al. [107].

Baker et al. applied mutation analysis to 22 blocks (20 LOC each) of C code and 25 blocks of Ada code belonging to control and diagnostic software in aircraft engines [107]. Their main objectives were the study of compilation errors induced by mutation operators, the distribution of equivalent and live mutants across operators, and the identification of classes of test case deficiencies leading to live mutants. Delgado et al., instead, applied mutation analysis to 15 functions of a Commercial-Off-The-Shelf Component used in nuclear systems [19]. According to their findings, 30% of the live mutants are equivalent. Both Baker et al. and Delgado et al. did not investigate mutation analysis scalability which is, in contrast, addressed by our work.

The largest cyber-physical system investigated in a mutation analysis study is the one considered by Ramler et al., who qualitatively assessed mutation analysis with the application components of a safety critical industrial system (around 60,000 LOC, in total) [116]. Their main outcomes included the identification of the most frequent test suite weaknesses [107] identified by means of mutation analysis (a) imprecise and over-optimistic oracles, (b) poor selection of test data conditions, and (c) child procedures not tested within the parent. Although Ramler et al. indicate that scalability issues prevent the practical applicability of mutation analysis, they do not address this challenge. In this paper, we complement the findings of Ramler et al. with a comprehensive solution for scaling mutation testing to industrial cyber-physical systems and a large-scale empirical evaluation.

Earlier investigations of mutation analysis for safety-critical systems focus on the representativeness of mutants, an objective that is orthogonal to ours. Daran et al. conducted a study to identify if mutations are correlated with real faults [117]; the experimentation was carried out on a critical software from the civil nuclear field. Andrews et al., instead, explored the relation between hand-seeded and real faults in the software *Space* [42]. Space is a space software development utility, verifying the consistency of a specification file with a reference grammar and constraints [118], that was developed by ESA and has been used as case study subject in software engineering papers since 1998 [119].

To summarize, our work includes the largest industrial embedded software benchmark ever considered in the scientific literature on mutation testing and analysis; in addition, it complements existing findings, focusing on the applicability of mutation analysis to embedded software in cyber-physical systems, with a comprehensive solution (complete mutation testing pipeline) for scalable mutation analysis and its extensive evaluation.

6 Conclusion

In this paper, we proposed and assessed a complete mutation analysis pipeline (*MASS*) that enables the application of mutation analysis to space software. It is also expected to be widely applicable for many other types of embedded software, that is typically part of many cyber-physical systems, and that share similar characteristics. In particular, we address challenges related to scalability—probably the most acute problem regarding the application of mutation analysis—and the detection of equivalent and duplicate mutants. Our contributions include (1) the identification and integration in a complete tool pipeline of state-of-the-art mutation analysis optimizations tailored to the context of embedded software and cyber-physical systems, (2) a strategy for the prioritization and selection of test cases to speed up mutation analysis, (3) an approach for the sampling of mutants that provides accuracy guarantees, even when mutation analysis relies on a reduced test suite, (4) a strategy for the identification of nonequivalent and nonduplicate mutants based on code coverage information,

10. We considered the following venues: IEEE International Conference on Software Testing, Validation and Verification (and its Workshops), IEEE/ACM International Conference on Software Engineering, IEEE/ACM International Conference on Automated Software Engineering, International Symposium on Software Testing and Analysis, IEEE Transactions on Software Engineering, ACM Transactions on Software Engineering and Methodology, IEEE Transactions on Reliability, Elsevier Information and Software Technology Journal, Elsevier Science Computer Programming Journal, Wiley Software Testing, Verification and Reliability Journal

TABLE 19: Subject systems in the mutation testing literature.

Case Study (SUT)	SUT category	Size of SUT	# Test cases	References
LLVM Framework*	C	18 kLOC	625	[62]
OpenSSL*	GL	20 kLOC	77	[62]
Codeflaws	B	68 LOC	31	[120]
RODOS*	OS	3,510	48	[62]
Coreutils*	CLT	8-83 kLOC	1,022-18,719	[34], [61], [121]
ESAIL-CSW*	FS	74 kLOC	384	This paper.
Siemens*	RPS	1-69 kLOC	1,531-22,138	[122], [123], [124], [125], [126]
Vim*	SUI	39-42 kLOC	98	[18], [43], [123]
Make*	CLT	7-35 kLOC	691	[18], [34], [43], [121], [125]
Memory Benchmark	B	32 kLOC	503	[127]
Findutils	CLT	18 kLOC	4,931	[34], [121]
KmyMoney	SUI	13 kLOC	248	[128]
Curl	CLT	12 kLOC	N/A	[122]
LIBU	FS	10 kLOC	201	This paper.
LIBN	FS	9 kLOC	89	This paper.
Grep	CLT	9 kLOC	5,899	[34], [121]
Space*	SDU	9 kLOC	13,585	[66], [124], [125], [126]
Git*	CLT	8 kLOC	N/A	[18], [43]
MSMTP*	CLT	6 kLOC	N/A	[18], [43]
MLFS	FS	5 kLOC	4,042	This paper.
Matrix TCL Pro	GL	3 kLOC	24	[128]
LIBP	FS	3 kLOC	170	This paper.
Dolphin	SUI	3 kLOC	70	[128]
Deletion Bench.	B	2,853 LOC	814	[44], [49]
Gzip*	CLT	2,819 LOC	N/A	[18], [43]
TinyXML2	GL	2,620 LOC	62	[128], [129]
XmlRPC++	GL	2,194 LOC	34	[128], [129]
QtDom	GL	2,117 LOC	56	[128]
Yao Benchmark	B	1,208 LOC	N/A	[125]
Flex*	CLT	10-14 kLOC	567	[124], [125]
NuclearIndustrial	ES	484 LOC	302	[19]
SafetyCritical	ES	60 kLOC	N/A	[116]
Nequivack Bench.	B	403 LOC	N/A	[21]
MuVM Bench.	B	302 LOC	2,256	[66]

We use an asterisk (*) to indicate work that applied mutation analysis to a subset of the system.
SUT category Legend: C=Compiler, FS=Flight software (library or whole system running on components on-orbit), GL=Generic library, RPS=Research prototype software, SUI=Software with UI, CLT=Command line tool, OS=Operating system, B=Benchmark, ES=Embedded software, SDU=Space software development utility.

(5) an extensive empirical evaluation (experiments took 1300 days of computation) conducted on a representative space software benchmark provided by our industry partners. Our benchmark is the largest to date as far as embedded software is concerned.

Our empirical evaluation provides the following results: (1) different compiler optimization options are complementary for the detection of trivially equivalent and redundant mutants, (2) our mutants sampling strategy outperforms state-of-the-art strategies both in terms of mutation score accuracy and size of the selected mutant set, (3) the proposed test suite selection and prioritization approach enables an impressive reduction of mutation analysis time (above 70%), thus helping make mutation analysis applicable to large systems, (4) cosine distance applied to statement coverage, collected from mutated files, can be used to successfully identify nonequivalent mutants. The above results show that our mutation analysis pipeline can be integrated into the software quality assurance practices enforced by certification bodies and safety standards in space systems and many other cyber-physical systems with similar characteristics.

APPENDIX A
DISTRIBUTION OF TRIVIALLY EQUIVALENT AND DUPLICATE MUTANTS

To enable further comparison with related work [43], in Table 20, we report the proportion of trivially equivalent and duplicate mutants per mutation operator. The mutation operator causing the largest number of trivially equivalent mutants is UOI (e.g., it applies a post-increment operator to the last use of a variable), followed by ROR, ROD, and ICR. Related studies are conducted with a smaller set of operators [43]; however, the operators causing the largest numbers of trivially equivalent and duplicate mutants are the same. The main difference with related work [43] concerns

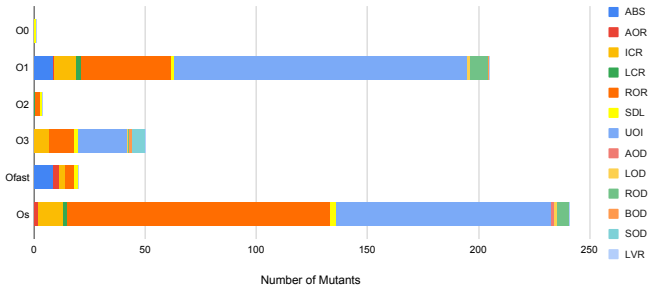


Fig. 8: Univocal, Trivially Equivalent Mutants detected by Compiler Optimizations.

the ABS operator, which leads to a small set of equivalent mutants in our case, the main reason being that we rely on a definition of the ABS operator that minimizes the number of equivalent mutants by simply inverting the sign of the value instead of using the *abs* function [47]. Indeed, in functions with a positive integer domain, the replacement of a value with its absolute value trivially leads to equivalent mutants. Except for the ABS operator, **our study confirms the ranking observed in related work** despite different showing proportions. In addition, our results show that **the nature of the software affects the distribution of equivalent and duplicate mutants across operators**. Indeed, *MLFS*, which focuses on mathematical functions, includes larger proportions of equivalent and duplicate mutants caused by ICR and AOR. In *LIBP*, which does not deal with mathematical functions, the number of equivalent and duplicate mutants caused by these operators is much smaller. Finally, we notice that the **SDL and OODL operators lead to a minimal set of trivially equivalent and duplicate mutants, except for ROD**.

Finally, to further characterize the differences across different compiler optimization levels, we provide in Figures 8 and 9, for each compiler optimization level, the distribution of univocal, trivially equivalent and duplicate mutants across mutation operators. The optimization level -*Os* is more effective in detecting trivially equivalent mutants caused by the ROR operator (a larger number of ROR mutants is associated to -*Os* as captured by the length of the orange bar), while the option -*O1* is more effective in detecting trivially equivalent mutants caused by the UOI operator. Concerning the detection of trivially duplicate mutants (Figure 9), -*Os* performs better in detecting the duplicate mutants caused by almost all the operators, except for the ones caused by operators affecting math expressions (i.e., AOR, AOD, and LVR), which are better detected by optimization level -*Ofast*, probably because it includes additional math optimization options.

APPENDIX B
ASSESSMENT OF ASSUMPTIONS ON LIMITED CORRELATION BETWEEN MUTANTS

We aim to assess the correctness of the two assumptions underlying the adoption of the binomial distribution to estimate the mutation score:

TABLE 20: RQ1. Proportion (%) of Trivially Equivalent/Duplicate Mutants Detected per Mutation Operator.

	Trivially Equivalent													Trivially Duplicate											
Subject	**ABS**	**AOR**	**ICR**	**LCR**	**ROR**	**SDL**	**UOI**	**AOD**	**LOD**	**ROD**	**BOD**	**SOD**	**LVR**	**ABS**	**AOR**	**ICR**	**LCR**	**ROR**	**SDL**	**UOI**	**AOD**	**LOD**	**ROD**	**BOD**	**SOD**	**LVR**
ESAIL-CSW	3.65	1.85	6.74	2.09	18.00	1.60	52.34	1.53	0.05	7.64	3.42	0.51	0.59	1.52	6.20	18.79	1.05	23.89	8.04	20.23	4.83	1.06	8.86	2.20	1.96	1.39
LIBN	8.84	0.00	2.28	1.14	17.40	3.14	56.35	0.00	0.14	9.84	0.86	0.00	0.00	2.11	2.37	10.77	1.24	31.45	9.04	23.92	1.69	1.36	13.48	1.58	0.79	0.19
LIBP	8.89	0.00	0.00	0.22	25.56	6.22	55.33	0.00	0.00	3.56	0.22	0.00	0.00	0.96	3.47	7.95	1.01	36.71	13.58	12.52	1.49	2.46	18.45	0.92	0.00	0.48
LIBU	6.66	0.07	3.95	1.24	15.96	7.91	58.86	0.00	0.37	3.81	0.95	0.00	0.22	0.34	7.38	15.64	0.87	30.35	12.61	13.52	1.66	2.66	12.77	0.89	0.52	0.77
MLFS	6.37	0.55	11.63	0.28	27.98	2.77	39.61	0.00	0.00	9.42	1.39	0.00	0.00	5.03	10.12	25.36	0.87	23.77	4.14	6.73	8.51	0.88	9.44	2.33	1.90	0.91
Total	4.59	1.42	6.04	1.81	18.32	2.64	**53.06**	1.16	0.09	7.22	2.79	0.38	0.47	1.87	6.48	18.47	1.02	25.36	8.23	**17.83**	4.72	1.25	9.91	2.02	1.68	1.17

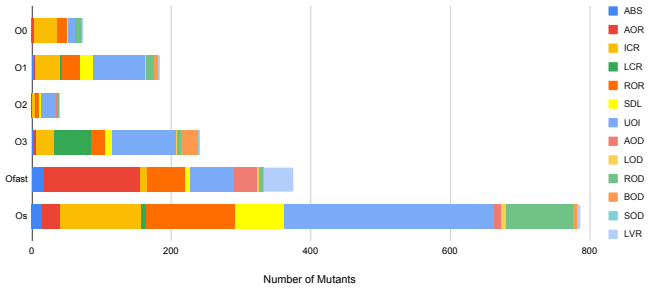


Fig. 9: Univocal, Trivially Duplicate Mutants detected by Compiler Optimizations.

A1 *The correlation between mutants is limited*. We aim to measure and discuss the degree of association between the trials (i.e., the outcome of a sampled mutant) in our experiments. The degree of association should be low.

A2 *The binomial distribution enables the estimation of the mutation score at the same level of accuracy as the correlated binomial distribution*. We compare the distribution of the mutation score observed in our experiments with the distribution of the mutation score estimated by the probability mass function (PMF) for the binomial distribution (i.e., PMF_B) and the PMF for the correlated binomial distribution (i.e., PMF_C). We aim to demonstrate that, in practice, in our context, there is no difference between the accuracy (i.e., the difference between the estimated mutation score and the mutation score of the sampled mutants) of PMF_B and PMF_C.

B.1 Assumption 1 - The correlation between mutants is limited.

B.1.1 Measurements

The association between two trials (i.e., the i^{th} and the j^{th} mutant in our case) can be measured by computing a coefficient of association from the contingency tables derived for every pair of trials [72]. More precisely, we generate a $2X2$ contingency table for every pair of mutants m_i and m_j. Each cell of the contingency table measures the frequency of occurrence, over R mutation analysis runs ($R = 100$ in our study), for the following situations: (a) mutants are both killed, (b) m_i killed with m_j live, (c) m_i live with m_j killed, (d) mutants are both live. Based on the contingency table, we derive two coefficients of association, Yule's Q [130] and the odds ratio. Yule's Q_{ij} measures the association between the i^{th} and the j^{th} mutant:

$$Q_{ij} = \frac{ad - bc}{ad + bc}$$

Q_{ij} ranges between -1 and +1, and for independent mutants it should be equal to zero.

The odds ratio is the ratio of (1) the odds of m_i being killed when m_j is killed and (2) the odds of m_i being killed when m_j is not killed:

$$OR_{ij} = \frac{ad}{bc}$$

Two mutants are independent when OR_{ij} is equal to one.

We compute Q_{ij} and OR_{ij} for every pair of mutants considering the mutation analysis results collected when sampling 300 (i.e., less than the mutants sampled by FSCI), 400 (i.e., the upper bound observed for FSCI), and 1000 mutants (i.e., the number of mutants suggested by Gopinath et al. [10]). To compute Q_{ij} and OR_{ij}, we considered the data collected in the 100 experiments carried out for RQ2 (see Section 4.4 of the main manuscript).

B.1.2 Results

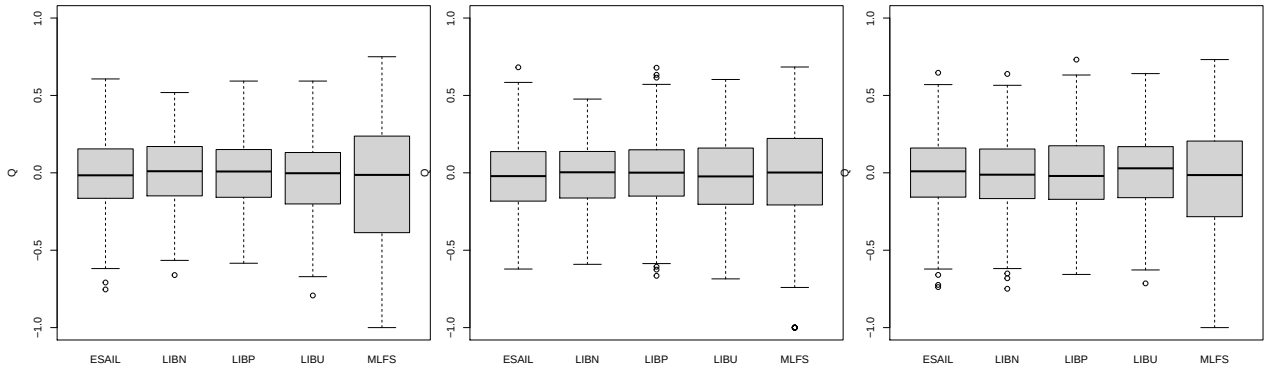
Figures 10a to 10c show boxplots of the distribution of Q for every pair of mutants (i.e., every data point reports the value of Q_{ij} computed for a specific pair of mutants). Across the three boxplots, the median is between 0.001 and 0.013, that is practically zero, thus showing that, overall, the mutants tend to be independent. Unsurprisingly, mutants are more correlated for the *MLFS*, where the first and third quartiles (i.e., half of the pairs of mutants), are equal to -0.386 and 0.237, respectively. This is due to *MLFS* having the most exhaustive test suite (it achieves MC/DC coverage); indeed, in such a situation, it is very likely that pairs of mutants fail together because the test suite will likely kill them both. For the subjects, the first and third quartiles lie between -0.203 and 0.174, thus showing limited association levels.

Figures 11a to 11c show boxplots with the distribution of OR for every pair of mutants. It once again shows that mutants tend to be independent; indeed, across the three boxplots, the median lies between 0.95 and 1.06, while the first and third quartiles lie between 0.44 and 1.62, respectively.

B.2 Assumption 2 - The binomial distribution accurately estimates the mutation score.

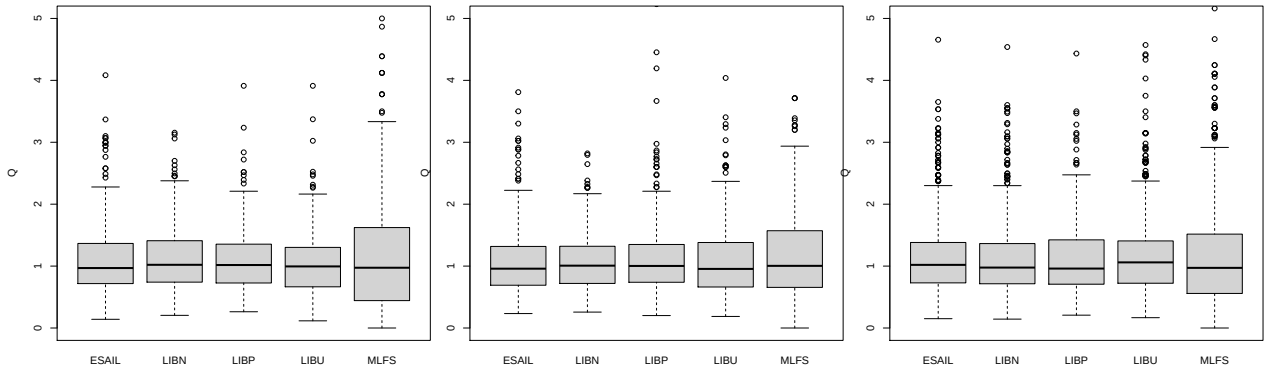
B.2.1 Measurements

As mentioned in Section 2.4, when trials are not independent, the correlated binomial distribution (also known as Bahadur-Lazarsfeld distribution) can be used to estimate the outcome of a binomial experiment (i.e., the computation of the mutation score in our context).



(a) 300 mutants (b) 400 mutants (c) 1000 mutants

Fig. 10: Yule's Q computed for experiments involving different numbers of mutants.



(a) 300 mutants (b) 400 mutants (c) 1000 mutants

Fig. 11: Odds ratio computed for experiments involving different numbers of mutants.

In the presence of second order interactions[11], the probability mass function of the correlated binomial distribution can be computed as

$$P(Y,N) = P'(Y,N,p) * \Big(1 + r_2 * g_2(Y,N,p)\Big)$$

with Y indicating the number of successful trials (i.e., the number of mutants killed, in our context) out of N trials, $P'(Y,N,p)$ being the probability mass function of the binomial distribution with probability of success p, r_2 being the correlation coefficient, and $g_2(Y,N,p)$ being a defined as

$$g_2(Y,N,p) = \frac{(Y - N*p)^2 - (1-2*p)(Y-N*p) - N*p(1-p)}{2*p(1-p)}$$

The correlated binomial distribution reduces to the binomial distribution when r_2 is zero.

We aim to compare the distribution of the mutation score observed in our experiments (hereafter, PMF_S) with the distribution of the mutation score estimated by both the PMF of the binomial (hereafter, PMF_B) and the PMF of the correlated binomial distributions (hereafter, PMF_C).

To compute PMF_C, we should identify proper values for the parameters p and r_2. To this end, we follow the approach proposed by Zhang et al. [73], which consists in relying on a non-linear least squares algorithm[12] to identify the values that minimize the error sum of squares (ESS) computed as

$$ESS_C = \sum_{y=0}^{N} \Big(h(y) - P'(y,N,\tilde{p}) * \Big(1 + \tilde{r_2} * g_2(y,N,\tilde{p})\Big)\Big)^2$$

with $\tilde{p}$ and $\tilde{r_2}$ being the parameters estimated for p and r_2. $h(y)$ is the proportion of runs in which, for our experiments, we observed Y mutants being killed.

To determine if the correlated binomial distribution is more accurate than the binomial distribution, we can compare ESS_C with the ESS computed for PMF_B

11. To simplify our discussion we ignore interactions above the second order, which is usual practice [73]; the interested reader is referred to related work [72].

12. We rely on the Gauss-Newton algorithm provided by R [131].

TABLE 21: Parameters estimated for PMF_C considering different numbers (N) of sampled mutants.

N		*ESAIL*	*LIBN*	*LIBP*	*LIBU*	*MLFS*
300	$\tilde{p}$	65.26	65.54	68.92	70.8	81.80
	$\tilde{r_2}$	-0.00029	0.00000	-0.00013	-0.00036	0.00005
	ESS	0.01037	0.01888	0.00776	0.00839	0.00884
400	$\tilde{p}$	65.06	65.73	68.97	71.02	81.90
	$\tilde{r_2}$	-0.00067	-0.00047	-0.00032	-0.00039	0.00008
	ESS	0.01002	0.00955	0.01194	0.00905	0.01017
1000	$\tilde{p}$	65.29	65.43	68.91	71.01	81.66
	$\tilde{r_2}$	-0.00080	-0.00043	-0.00045	0.00000	0.00010
	ESS	0.01337	0.00548	0.00739	0.00754	0.01088

TABLE 22: ESS obtained with PMF_B and delta with PMF_C (i.e., $\Delta = PMF_B - PMF_C$).

N		*ESAIL*	*LIBN*	*LIBP*	*LIBU*	*MLFS*
All	p	65.36	65.64	69.12	71.20	81.80
300	ESS	0.01044	0.01892	0.00784	0.008859	0.008849
	Δ	0.00007	0.00007	0.00008	0.00046	0.0
400	ESS	0.01065	0.00975	0.01211	0.00929	0.0102
	Δ	0.00063	0.00020	0.00020	0.00024	0.00003
1000	ESS	0.01565	0.00633	0.00841	0.00771	0.01103
	Δ	0.00228	0.00085	0.00102	0.00017	0.00015

$$ESS_B = \sum_{y=0}^{N} \Big(h(y) - P'(y, N, p)\Big)^2$$

The parameter p is the actual mutation score. If ESS_B is close to ESS_C, the two PMF perform similarly in our context (i.e., the effect of covariance is negligible). Also, the effect of covariance can be considered negligible if the estimated value for r_2 is close to zero.

To further facilitate the understanding of the similarity between PMF_B and PMF_C, we also discuss the shape of the curve of PMF_B and PMF_C with that of the curve that fits the distribution of the mutation score observed in our experiments (hereafter, PMF_S). To derive PMF_S, we rely on a density estimation function based on a gaussian kernel [132].

Between PMF_B and PMF_C, the curve that approximates better PMF_S is the one with the largest intersection of their Area Under Curve (AUC). We thus compute, first, $PAUC_B$ as the proportion of the area under PMF_S and PMF_B that is under both curves (i.e., intersection over union). Then we do the same for PMF_C by computing $PAUC_C$ as the proportion of the area under PMF_S and PMF_C that is under both curves. The curve that better approximates PMF_S is the one leading to the highest value for $PAUC$. The binomial distribution is an appropriate approximation of the mutation score if $PAUC_B$ is close to $PAUC_C$.

TABLE 23: Proportion (%) of AUC for PMF_S shared with PMF_B and PMF_C. Higher values are highlighted.

N	PAUC	*ESAIL*	*LIBN*	*LIBP*	*LIBU*	*MLFS*
300	$PAUC_B$	**92.49**	94.60	**93.49**	**89.52**	**92.89**
	$PAUC_C$	92.09	**94.85**	93.01	87.44	92.79
400	$PAUC_B$	92.88	**95.14**	94.22	**94.82**	**95.19**
	$PAUC_C$	**93.73**	92.44	**94.92**	91.23	94.19
1000	$PAUC_B$	**93.84**	**89.79**	**91.83**	93.89	**92.98**
	$PAUC_C$	84.13	85.59	85.73	**94.58**	92.08

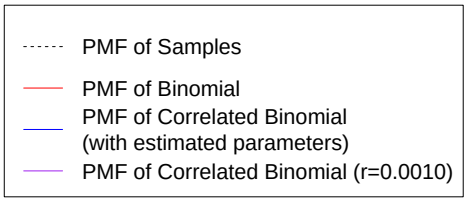


Fig. 12: Legend for Figure 13

B.2.2 Results

Table 21 provides the values estimated for p and r_2 (i.e., $\tilde{p}$ and $\tilde{r_2}$) for PMF_C, considering the different subjects in our study, for different numbers of sampled mutants. We can notice that the value of $\tilde{r_2}$ is practically zero in all the cases, which indicates that the effect of covariance is negligible. In most of the cases the covariance is slightly below zero (i.e., the detection of a mutant implies that another mutant is not detected), which we believe to be due to two reasons. First, the test suites of our subjects do not test all the implemented functions with the same degree of thoroughness, e.g., since our test suites focus on functional testing, they do not exercise well functions concerning real-time scheduling. Consequently, there is a certain likelihood that, after sampling a mutant belonging to a well tested function, we sample a mutant belonging to a function that is not thoroughly tested. To support this interpretation, we highlight the fact that *MLFS*, which is the benchmark component with the most thorough test suite, is not characterized by a negative $\tilde{r_2}$. The second reason for a negative $\tilde{r_2}$ is that the correlated binomial distribution may be suboptimal to model our data; we leave the analysis of other distributions to future work.

Table 22 provides the ESS obtained with PMF_B for different numbers of sampled mutants and the difference (Δ) with respect to ESS_C. The difference is small (i.e., Δ is always below 0.0025), which indicates that, in practice, **the binomial distribution approximates the mutation score as well as the correlated binomial distribution**.

Figures 13-a to Figures 13-q provide the plots of the curves obtained for PMF_S (dotted black), PMF_B (red), and PMF_C (blue). The plots show that for all subjects, PMF_B and PMF_C largely overlap, with PMF_C being slightly narrower and taller. The consequence is that PMF_B tends to be more conservative than PMF_C. Consequently, PMF_C leads to a computation of a larger confidence interval, which, in turn, for FSCI, leads to the sampling of more mutants. Since this negatively only affects the scalability of mutation analysis (i.e., a slightly higher number of mutants will be tested) but does not negatively affect the quality of results, we can argue that PMF_B estimates the mutation score as accurately as PMF_C.

Concerning the area under curve, we can observe that

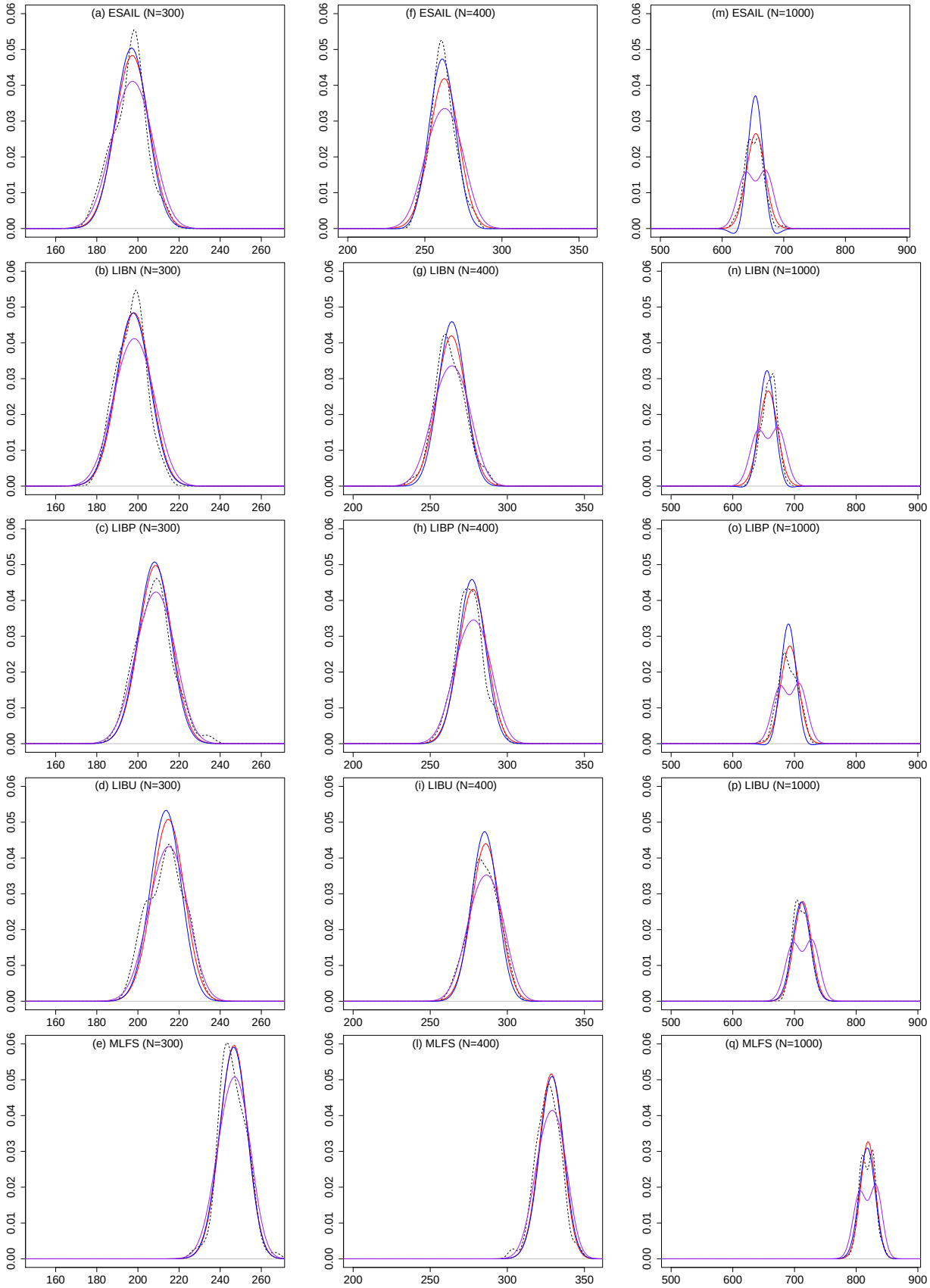


Fig. 13: Probability distribution for different PMFs for a varying number (N) of mutants being sampled. The x-axis shows the number of mutants being killed, the y-axis the probability. Legend is provided in Figure 12.

both PMF_B and PMF_C cover most of the AUC of PMF_S. Table 23 provides the values for $PAUC_B$ and $PAUC_C$; they are very close, with PMF_B performing better than PMF_C in 11 out of 15 cases, which shows that, in our context, the binomial distribution appropriately estimates the distribution of the mutation score.

To discuss what might happen when trials are affected by high covariance, in the charts of Figure 13 we also plotted the curve of PMF_C (hereafter, PMF_C^{10}) obtained with p equal to the population mean (i.e., the actual mutation score) and $r_2 = 0.0010$. We can notice that for PMF_C^{10} the distribution is slightly more spread out, which is in line with the statistics literature indicating that (1) a distribution has more variance with a positive correlation among the Bernoulli trials [133] [73] and (2) the sample size required in sequential tests should be increased proportionally to the value of the correlation coefficient [134]. In practice, in the presence of higher correlations than the ones observed in experimental subjects, the confidence interval estimated by FSCI, which assumes a binomial distribution, could become unreliable (i.e., smaller than it should be); however, such situation does not occur in our context, where subject are characterized by limited correlations.

APPENDIX C
DETAILS ABOUT THE SAVINGS OBTAINED WITH THE *MASS* TEST SUITE

Table 24 provides additional details about the data plotted in Figures 3 and 4 of the main manuscript.

TABLE 24: RQ5. Savings (Execution Time and Number of Test Cases) with the *MASS* Test Suite.

			Time Savings [%]				Test Savings [%]			
Subject	**D**	**MS**	**Min**	**Max**	**Med**	**Mn**	**Min**	**Max**	**Med**	**Mn**
LIBN	D_J	ALL	13.01	13.50	13.36	13.32	33.11	33.17	33.16	33.15
LIBP	D_J	ALL	16.81	16.82	16.82	16.81	14.17	14.17	14.17	14.17
LIBU	D_J	ALL	4.05	4.29	4.18	4.17	9.41	9.44	9.42	9.42
MLFS	D_J	ALL	7.18	7.21	7.20	7.20	13.29	13.33	13.31	13.31
$ESAIL_S$	D_J	ALL	52.78	52.85	52.82	52.82	91.79	91.81	91.79	91.80
LIBN	D_O	ALL	13.07	13.49	13.40	13.37	33.11	33.17	33.15	33.14
LIBP	D_O	ALL	16.81	16.82	16.81	16.81	14.17	14.17	14.17	14.17
LIBU	D_O	ALL	4.09	4.27	4.18	4.19	9.41	9.43	9.42	9.42
MLFS	D_O	ALL	7.18	7.20	7.19	7.19	13.29	13.33	13.30	13.31
$ESAIL_S$	D_O	ALL	52.78	52.83	52.81	52.81	91.76	91.82	91.78	91.79
LIBN	D_C	ALL	-0.53	-0.26	-0.34	-0.39	9.39	9.40	9.39	9.39
LIBP	D_C	ALL	14.86	14.88	14.87	14.87	11.00	11.00	11.00	11.00
LIBU	D_C	ALL	2.03	2.25	2.20	2.17	4.80	4.84	4.81	4.82
MLFS	D_C	ALL	1.53	1.53	1.53	1.53	6.64	6.64	6.64	6.64
$ESAIL_S$	D_C	ALL	5.97	6.44	6.24	6.27	43.05	43.27	43.20	43.20
LIBN	D_E	ALL	-0.21	-0.03	-0.08	-0.10	10.37	10.37	10.37	10.37
LIBP	D_E	ALL	14.66	14.69	14.68	14.68	10.44	10.44	10.44	10.44
LIBU	D_E	ALL	2.15	2.28	2.22	2.22	4.84	4.87	4.85	4.85
MLFS	D_E	ALL	0.71	0.73	0.72	0.72	5.41	5.46	5.45	5.45
$ESAIL_S$	D_E	ALL	-4.01	-3.74	-3.94	-3.89	38.60	38.83	38.76	38.73
LIBN	D_J	FSCI	77.65	83.20	78.91	79.90	72.54	80.31	76.93	76.40
LIBP	D_J	FSCI	75.53	79.83	77.15	77.47	68.76	71.07	69.95	70.06
LIBU	D_J	FSCI	88.54	90.33	89.50	89.40	77.38	79.53	77.87	78.13
MLFS	D_J	FSCI	72.09	82.05	79.87	78.33	61.19	75.37	72.26	69.94
$ESAIL_S$	D_J	FSCI	88.64	90.83	89.53	89.53	88.80	90.52	89.51	89.55
LIBN	D_O	FSCI	77.63	83.22	78.89	79.91	72.54	80.35	76.94	76.40
LIBP	D_O	FSCI	75.53	79.83	77.16	77.47	68.76	71.07	69.95	70.06
LIBU	D_O	FSCI	88.54	90.33	89.50	89.40	77.38	79.53	77.87	78.13
MLFS	D_O	FSCI	72.09	82.06	79.87	78.33	61.19	75.37	72.27	69.94
$ESAIL_S$	D_O	FSCI	88.64	90.83	89.53	89.57	88.80	90.52	89.51	89.55
LIBN	D_C	FSCI	76.24	82.85	77.57	78.77	70.60	78.03	73.72	74.29
LIBP	D_C	FSCI	75.25	79.63	76.94	77.22	68.28	70.69	69.53	69.65
LIBU	D_C	FSCI	88.48	90.28	89.43	89.33	77.26	79.41	77.70	77.98
MLFS	D_C	FSCI	80.00	82.28	81.45	81.16	72.36	75.40	74.14	73.86
$ESAIL_S$	D_C	FSCI	75.27	90.83	83.25	83.04	81.19	90.52	86.41	86.00
LIBN	D_E	FSCI	76.27	82.85	77.61	78.80	70.66	78.18	73.85	74.41
LIBP	D_E	FSCI	75.19	79.59	76.90	77.18	68.08	70.48	69.44	69.55
LIBU	D_E	FSCI	88.48	90.28	89.42	89.33	77.24	79.42	77.69	77.98
MLFS	D_E	FSCI	80.00	82.26	81.43	81.15	72.36	75.38	74.12	73.84
$ESAIL_S$	D_E	FSCI	73.53	90.83	82.40	82.14	80.50	90.52	86.03	85.69
LIBN	*Full*	FSCI	80.35	82.35	81.55	81.53	67.25	71.68	69.44	69.50
LIBP	*Full*	FSCI	65.76	72.03	67.86	68.40	49.98	58.91	53.39	54.31
LIBU	*Full*	FSCI	82.18	84.72	83.18	83.23	55.06	61.16	57.35	57.97
MLFS	*Full*	FSCI	66.59	76.03	70.29	70.89	52.56	65.73	57.69	58.54
$ESAIL_S$	*Full*	FSCI	73.46	75.47	74.48	74.51	78.94	80.54	79.61	79.71

Legend: D, distance; Full, no distance measure, the full test suite has been considered; MS, mutants set; Med, median; Mn, Mean.

ACKNOWLEDGMENTS

This work has been funded by the European Space Agency (ITT-1-9873/FAQAS), the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277), and NSERC Discovery and Canada Research Chair programs. Authors would like to thank the ESA ESTEC officers, the GomSpace Luxembourg team, Yago Isasi Parache and LuxSpace software engineers for their valuable support.

REFERENCES

[1] European Space Agency, "ExoMars 2016 - Schiaparelli Anomaly Inquiry," *DG-I/2017/546/TTN*, 2017. [Online]. Available: http://exploration.esa.int/mars/59176-exomars-2016-schiaparelli-anomaly-inquiry/

[2] European Cooperation for Space Standardization., "ECSS-Q-ST-80C Rev.1 - Software product assurance." 2017. [Online]. Available: http://ecss.nl/standard/ecss-q-st-80c-rev-1-software-product-assurance-15-february-2017/

[3] ——, "ECSS-E-ST-40C - Software general requirements." 2009. [Online]. Available: http://ecss.nl/standard/ecss-e-st-40c-software-general-requirements/

[4] European Space Agency, "ESA ISVV Guide, issue 2.0, 29/12/2008." 2008. [Online]. Available: ftp://ftp.estec.esa.nl/pub/wm/anonymous/wme/ecss/ESAISVVGuideIssue2.029dec2008.pdf

[5] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, April 1978.

[6] M. Papadakis, C. Henard, M. Harman, Y. Jia, and Y. Le Traon, "Threats to the validity of mutation-based test assessment," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 2016, pp. 354–365.

[7] L. Madeyski, W. Orzeszyna, R. Torkar, and M. Jozala, "Overcoming the equivalent mutant problem: A systematic literature review and a comparative experiment of second order mutation," *IEEE Transactions on Software Engineering*, vol. 40, no. 1, pp. 23–42, 2013.

[8] D. Shin, S. Yoo, and D. Bae, "A theoretical and empirical study of diversity-aware mutation adequacy criterion," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 914–931, Oct 2018.

[9] L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid, "Operator-based and random mutant selection: Better together," in *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2013, pp. 92–102.

[10] R. Gopinath, A. Alipour, I. Ahmed, C. Jensen, and A. Groce, "How hard does mutation analysis have to be, anyway?" in *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2015, pp. 216–227.

[11] L. Zhang, D. Marinov, and S. Khurshid, "Faster mutation testing inspired by test prioritization and reduction," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013, pp. 235–245.

[12] B. J. Grün, D. Schuler, and A. Zeller, "The impact of equivalent mutants," in *2009 International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2009, pp. 192–199.

[13] D. Schuler and A. Zeller, "(un-) covering equivalent mutants," in *2010 Third International Conference on Software Testing, Verification and Validation*. IEEE, 2010, pp. 45–54.

[14] ——, "Covering and uncovering equivalent mutants," *Software Testing, Verification and Reliability*, vol. 23, no. 5, pp. 353–374, 2013.

[15] D. Schuler, V. Dallmeier, and A. Zeller, "Efficient mutation testing by checking invariant violations," in *Proceedings of the eighteenth international symposium on Software testing and analysis*. ACM, 2009, pp. 69–80.

[16] J. Kapinski, J. V. Deshmukh, X. Jin, H. Ito, and K. Butts, "Simulation-Based Approaches for Verification of Embedded Systems," *IEEE Control Systems Magazine*, no. November, 2016.

[17] X. Zhou, X. Gou, T. Huang, and S. Yang, "Review on Testing of Cyber Physical Systems: Methods and Testbeds," *IEEE Access*, vol. 6, pp. 52 179–52 194, 2018.

[18] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 2015, pp. 936–946.

[19] P. Delgado-Pérez, I. Habli, S. Gregory, R. Alexander, J. Clark, and I. Medina-Bulo, "Evaluation of mutation testing in a nuclear industry case study," *IEEE Transactions on Reliability*, vol. 67, no. 4, pp. 1406–1419, 2018.

[20] R. H. Untch, A. J. Offutt, and M. J. Harrold, "Mutation analysis using mutant schemata," in *ACM SIGSOFT Software Engineering Notes*, vol. 18, no. 3. ACM, 1993, pp. 139–148.

[21] D. Holling, S. Banescu, M. Probst, A. Petrovska, and A. Pretschner, "Nequivack: Assessing mutation score confidence," in *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2016, pp. 152–161.

[22] H. Riener, R. Bloem, and G. Fey, "Test case generation from mutants using model checking techniques," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 388–397.

[23] T. T. Chekam, M. Papadakis, M. Cordy, and Y. L. Traon, "Killing stubborn mutants with symbolic execution," *ACM Trans. Softw. Eng. Methodol.*, vol. 30, no. 2, Jan. 2021. [Online]. Available: https://doi.org/10.1145/3425497

[24] J. Frey, "Fixed-width sequential confidence intervals for a proportion," *The American Statistician*, vol. 64, no. 3, pp. 242–249, 2010.

[25] ESA, "European Space Agency," 2020. [Online]. Available: https://www.esa.int

[26] Gomspace, "Systems for cubesats and nanosatellites." 2020. [Online]. Available: https://gomspace.com/

[27] OHB LuxSpace, "The first provider of space systems, applications and services in Luxembourg." 2020. [Online]. Available: https://luxspace.lu/

[28] LuxSpace, "ESAIL - maritime microsatellite," 2020. [Online]. Available: https://directory.eoportal.org/web/eoportal/satellite-missions/e/esail

[29] European Space Agency, "ESA Vega launch," 2020. [Online]. Available: https://www.esa.int/Enabling_Support/Space_Transportation/Vega/

[30] ——, "MLFS - Mathematical Library for Flight Software ," https://essr.esa.int/project/mlfs-mathematical-library-for-flight-software, 2020.

[31] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE transactions on software engineering*, vol. 37, no. 5, pp. 649–678, 2010.

[32] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, and M. Harman, "Mutation testing advances: an analysis and survey," in *Advances in Computers*. Elsevier, 2019, vol. 112, pp. 275–378.

[33] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Software testing, verification and reliability*, vol. 7, no. 3, pp. 165–192, 1997.

[34] M. Papadakis, D. Shin, S. Yoo, and D.-H. Bae, "Are mutation scores correlated with real fault detection? a large scale empirical study on the relationship between mutants and real faults," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 537–548.

[35] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, 2017, pp. 597–608.

[36] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.

[37] J. A. Offut and S. D. Lee, "An Empirical Evaluation of Weak Mutation," *IEEE Transactions on Software Engineering*, vol. 20, no. 5, pp. 337–344, 1994.

[38] M. Woodward and K. Halewood, "From weak to strong, dead or alive? an analysis of some mutation testing issues," in *[1988] Proceedings. Second Workshop on Software Testing, Verification, and Analysis*, 1988, pp. 152–158.

[39] P. R. Mateo, M. P. Usaola, and J. L. F. Aleman, "Validating second-order mutation at system level," *IEEE Transactions on Software Engineering*, vol. 39, no. 4, pp. 570–587, 2012.

[40] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 5, no. 2, pp. 99–118, 1996.

[41] G. Rothermel, R. H. Untch, and C. Zapf, "An experimental determination of sufficient mutant operators a. jefferson offutt ammei lee george mason university," *ACM Transactions on software engineering methodology*, vol. 5, no. 2, pp. 99–118, 1996.

[42] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *Proceedings of the 27th international conference on Software engineering*. ACM, 2005, pp. 402–411.

[43] M. Kintis, M. Papadakis, Y. Jia, N. Malevris, Y. Le Traon, and M. Harman, "Detecting trivial mutant equivalences via compiler optimisations," *IEEE Transactions on Software Engineering*, vol. 44, no. 4, pp. 308–333, 2017.

[44] M. E. Delamaro, J. Offutt, and P. Ammann, "Designing deletion mutation operators," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 11–20.

[45] A. Siami Namin, J. H. Andrews, and D. J. Murdoch, "Sufficient mutation operators for measuring test effectiveness," in *Proceedings of the 30th international conference on Software engineering*. ACM, 2008, pp. 351–360.

[46] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014. New York, NY, USA: Association for Computing Machinery, 2014, p. 654–665.

[47] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. Le Traon, "How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults," *Empirical Software Engineering*, vol. 23, no. 4, pp. 2426–2463, aug 2018.

[48] L. Deng, J. Offutt, and N. Li, "Empirical evaluation of the statement deletion mutation operator," in *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, 2013, pp. 84–93.

[49] M. E. Delamaro, L. Deng, V. H. S. Durelli, N. Li, and J. Offutt, "Experimental evaluation of sdl and one-op mutation for c," in *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*. IEEE, 2014, pp. 203–212.

[50] Y. Jia and M. Harman, "Higher order mutation testing," *Information and Software Technology*, vol. 51, no. 10, pp. 1379–1393, 2009.

[51] M. Kintis, M. Papadakis, and N. Malevris, "Evaluating mutation testing alternatives: A collateral experiment," in *2010 Asia Pacific Software Engineering Conference*. IEEE, 2010, pp. 300–309.

[52] A. J. Offutt, "Investigations of the software testing coupling effect," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 1, no. 1, pp. 5–20, 1992.

[53] M. Papadakis and N. Malevris, "An empirical evaluation of the first and second order mutation testing strategies," in *2010 Third International Conference on Software Testing, Verification, and Validation Workshops*. IEEE, 2010, pp. 90–99.

[54] M. Papadakis, N. Malevris, and M. Kintis, "Mutation testing strategies-a collateral approach." in *ICSOFT (2)*, 2010, pp. 325–328.

[55] M. Papadakis and N. Malevris, "Automatic mutation test case generation via dynamic symbolic execution," in *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 2010, pp. 121–130.

[56] M. Becker, D. Baldin, C. Kuznik, M. M. Joy, T. Xie, and W. Mueller, "Xemu: an efficient qemu based binary mutation testing framework for embedded software," in *Proceedings of the tenth ACM international conference on Embedded software*, 2012, pp. 33–42.

[57] Y. Crouzet, H. Waeselynck, B. Lussier, and D. Powell, "The sesame experience: from assembly languages to declarative models," in *Second Workshop on Mutation Analysis (Mutation 2006-ISSRE Workshops 2006)*. IEEE, 2006, pp. 7–7.

[58] Y.-S. Ma, J. Offutt, and Y.-R. Kwon, "Mujava: a mutation system for java," in *Proceedings of the 28th international conference on Software engineering*. ACM, 2006, pp. 827–830.

[59] A. Derezinska and K. Kowalski, "Object-oriented mutation applied in common intermediate language programs originated from c," in *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*. IEEE, 2011, pp. 342–350.

[60] F. Hariri, A. Shi, H. Converse, S. Khurshid, and D. Marinov, "Evaluating the effects of compiler optimizations on mutation testing at the compiler ir level," in *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 2016, pp. 105–115.

[61] F. Hariri, A. Shi, V. Fernando, S. Mahmood, and D. Marinov, "Comparing mutation testing at the levels of source code and compiler intermediate representation," in *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2019, pp. 114–124.

[62] A. Denisov and S. Pankevich, "Mull it over: mutation testing based on llvm," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 25–31.

[63] Cobham Gaisler, "RTEMS Cross Compilation System," https://www.gaisler.com/index.php/products/operating-systems/rtems, 2020.

[64] M. E. Delamaro, J. C. Maldonado, and A. Mathur, "Proteum-a tool for the assessment of test adequacy for c programs user's guide," in *PCS*, vol. 96, 1996, pp. 79–95.

[65] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Software: Practice and Experience*, vol. 21, no. 7, pp. 685–718, 1991.

[66] S. Tokumoto, H. Yoshida, K. Sakamoto, and S. Honiden, "Muvm: Higher order mutation analysis virtual machine for c," in *2016 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2016, pp. 320–329.

[67] L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei, "Is operator-based mutant selection superior to random mutant selection?" in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 435–444.

[68] P. Tchebichef, "Des valeurs moyennes," *Journal de Mathématiques Pures et Appliquées*, vol. 2, no. 12, p. 177–184, 1867.

[69] R. R. Bahadur, *A Representation of the Joint Distribution of Responses to N Dichotomous Items*. Stanford University Press, 1961, pp. 158–168.

[70] L. L. Kupper and J. K. Haseman, "The Use of a Correlated Binomial Model for the Analysis of Certain Toxicological Experiments," *Biometrics*, vol. 34, no. 1, p. 69, 1978.

[71] T.-H. Ng, "A new class of modified binomial distributions with applications to certain toxicological experiments," *Communications in Statistics - Theory and Methods*, vol. 18, no. 9, pp. 3477–3492, 1989.

[72] P. A. G. Van Der Geest, "The binomial distribution with dependent bernoulli trials," *Journal of Statistical Computation and Simulation*, vol. 75, no. 2, pp. 141–154, 2005.

[73] N. F. Zhang, "The Use of Correlated Binomial Distribution in Estimating Error Rates for Firearm Evidence Identification," *Journal of Research (NIST JRES)*, vol. 124, no. 124026, pp. 1–16, 2019. [Online]. Available: https://www.nist.gov/publications/use-correlated-binomial-distribution-estimating-error-rates-firearm-evidence

[74] R. V. Krejcie and D. W. Morgan, "Determining sample size for research activities," *Educational and Psychological Measurement*, vol. 30, no. 3, pp. 607–610, 1970.

[75] W. G. Cochran, *Sampling Techniques*. New York, USA: John Wiley & Sons, 1977.

[76] J. E. Bartlett, J. W. Kotrlik, and C. C. Higgins, "Organizational research: Determining appropriate sample size in survey research," *Information Technology, Learning, and Performance Journal*, vol. 19, no. 1, pp. 43–50, 2001.

[77] K. Krishnamoorthy and J. Peng, "Some properties of the exact and score methods for binomial proportion and sample size calculation," *Communications in Statistics - Simulation and Computation*, vol. 36, no. 6, pp. 1171–1186, 2007.

[78] L. Gonçalves, M. R. de Oliveira, C. Pascoal, and A. Pires, "Sample size for estimating a binomial proportion: comparison of different methods," *Journal of Applied Statistics*, vol. 39, no. 11, pp. 2453–2473, nov 2012.

[79] A. Wald, "Sequential tests of statistical hypotheses," *Ann. Math. Statist.*, vol. 16, no. 2, pp. 117–186, 06 1945.

[80] E. S. William Hsu, Mehmet Sahinoglu, "An experimental approach to statistical mutation-based testing," Software Engineering Research Center, West Lafayette, IN 47907, Tech. Rep. SERC-TR-63-P, April 1990.

[81] Z. Chen and X. Chen, "Exact Group Sequential Methods for Estimating a Binomial Proportion," *Journal of Probability and Statistics*, vol. 2013, p. 603297, 2013.

[82] T. Yaacoub, G. V. Moustakides, and Y. Mei, "Optimal stopping for interval estimation in bernoulli trials," *IEEE Transactions on Information Theory*, vol. 65, no. 5, pp. 3022–3033, 2019.

[83] S. E. Vollset, "Confidence intervals for a binomial proportion," *Statistics in Medicine*, vol. 12, no. 9, pp. 809–824, 1993.

[84] C. J. Clopper and E. S. Pearson, "The use of confidence or fiducial limits illustrated in the case of the binomial," *Biometrika*, vol. 26, no. 4, pp. 404–413, 1934.

[85] R. Just, G. M. Kapfhammer, and F. Schweiggert, "Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis," in *2012 IEEE 23rd International Symposium on Software Reliability Engineering*. IEEE, 2012, pp. 11–20.

[86] M. Papadakis and N. Malevris, "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing," *Software Quality Journal*, vol. 19, no. 4, p. 691, 2011.

[87] T. A. Budd and D. Angluin, "Two notions of correctness and their relation to testing," *Acta Informatica*, vol. 18, no. 1, pp. 31–45, 1982. [Online]. Available: https://doi.org/10.1007/BF00625279

[88] V. Herdt, D. Große, H. M. Le, and R. Drechsler, "Early concolic testing of embedded binaries with virtual prototypes: A risc-v case study*," in *2019 56th ACM/IEEE Design Automation Conference (DAC)*, 2019, pp. 1–6.

[89] F. Free Software Foundation, "gcov - A Test Coverage Program," https://gcc.gnu.org/onlinedocs/gcc/Gcov.html, 2020.

[90] Free Software Foundation, "GDB: The GNU Project Debugger." 2020. [Online]. Available: https://www.gnu.org/software/gdb/

[91] T. Tsiodras, "Cover me!" 2020. [Online]. Available: https://www.thanassis.space/coverage.html

[92] F. Hariri and A. Shi, "Srciror: a toolset for mutation testing of c source code and llvm intermediate representation." in *ASE*, 2018, pp. 860–863.

[93] European Space Agency, "ESA Community Licence Permissive v2.4," 2020. [Online]. Available: https://essr.esa.int/license/european-space-agency-community-license-v2-4-permissive

[94] F. Free Software Foundation, "GNU Make," https://www.gnu.org/software/make/, 2020.

[95] ——, "GCC, the GNU Compiler Collection," https://gcc.gnu.org/, 2020.

[96] National Institute of Standards and Technology, "Announcing Approval of Federal Information Processing Standard (FIPS) 180-2, Secure Hash Standard; a Revision of FIPS 180-1," 2002. [Online]. Available: https://www.govinfo.gov/content/pkg/FR-2002-08-26/pdf/02-21599.pdf

[97] R. G. Newcombe, "Two-sided confidence intervals for the single proportion: comparison of seven methods," *Statistics in Medicine*, vol. 17, no. 8, pp. 857–872, 1998.

[98] M. L. Berenson, D. M. Levine, and K. Szabat, *Basic Business Statistics: Concepts and Applications*, 7th ed. USA: Prentice Hall PTR, 1998.

[99] D. Zou, J. Liang, Y. Xiong, M. D. Ernst, and L. Zhang, "An empirical study of fault localization families and their combinations," *IEEE Transactions on Software Engineering*, pp. 1–1, 2019.

[100] F. Keller, L. Grunske, S. Heiden, A. Filieri, A. van Hoorn, and D. Lo, "A critical evaluation of spectrum-based fault localization techniques on a large-scale software system," in *2017 IEEE International Conference on Software Quality, Reliability and Security (QRS)*, 2017, pp. 114–125.

[101] M. Golagha, A. Pretschner, and L. Briand, "Can We Predict the Quality of Spectrum-based Fault Localization?" in *2020 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. IEEE, 2020, pp. 1–10.

[102] Y. Isasi, A. Pinardell, A. Marquez, C. Molon-Noblot, A. Wagner, M. Gales, and M. Brada, "The esail multipurpose simulator," in *Onlin Proceedings of Simulation and EGSE for Space Programmes (SESP2019)*, 2019, https://atpi.eventsair.com/QuickEventWebsitePortal/sesp-2019/website/Agenda.

[103] J. Eickhoff, *Simulating spacecraft systems*. Berlin, Germany: Springer Science & Business Media, 2009.

[104] European Cooperation for Space Standardization., "ECSS-E-HB-10-02A Rev.1 - Space Engineering, Verification guidelines," 2010. [Online]. Available: https://ecss.nl/hbstms/ecss-e-10-02a-verification-guidelines/

[105] J. J. Chilenski and S. P. Miller, "Applicability of modified condition/decision coverage to software testing," *Software Engineering Journal*, vol. 9, no. 5, pp. 193–200, 1994.

[106] M. Papadakis and N. Malevris, "Mutation based test case generation via a path selection strategy," *Information and Software Technology*, vol. 54, no. 9, pp. 915–932, 2012.

[107] R. Baker and I. Habli, "An empirical evaluation of mutation testing for improving the test quality of safety-critical software," *IEEE Transactions on Software Engineering*, vol. 39, no. 6, pp. 787–805, 2013.

[108] S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, "Management of an academic hpc cluster: The ul experience," in *Proc. of the 2014 Intl. Conf. on High Performance Computing & Simulation (HPCS 2014)*. Bologna, Italy: IEEE, July 2014, pp. 959–967.

[109] A. Arcuri and L. Briand, "A practical guide for using statistical tests to assess randomized algorithms in software engineering," in *Proceedings of the 33rd International Conference on Software Engineering*, ser. ICSE '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 1–10.

[110] A. Vargha and H. D. Delaney, "A critique and improvement of the cl common language effect size statistics of mcgraw and wong," *Journal of Educational and Behavioral Statistics*, vol. 25, no. 2, pp. 101–132, 2000.

[111] Free Software Foundation, "GCC Optimizations," https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html, 2020.

[112] R. J. Hyndman and Y. Fan, "Sample quantiles in statistical packages," *The American Statistician*, vol. 50, no. 4, pp. 361–365, 1996.

[113] Valgrind, "Valgrind," 2020. [Online]. Available: https://valgrind.org/

[114] D. Shin, S. Yoo, and D.-H. Bae, "A theoretical and empirical study of diversity-aware mutation adequacy criterion," *IEEE Transactions on Software Engineering*, vol. 44, no. 10, pp. 914–931, 2017.

[115] G. Gay, A. Rajan, M. Staats, M. Whalen, and M. P. Heimdahl, "The effect of program and model structure on the effectiveness of MC/DC test adequacy coverage," *ACM Transactions on Software Engineering and Methodology*, vol. 25, no. 3, 2016.

[116] R. Ramler, T. Wetzlmaier, and C. Klammer, "An empirical study on the application of mutation testing for a safety-critical industrial software system," *Proceedings of the ACM Symposium on Applied Computing*, vol. Part F128005, no. Section 4, pp. 1401–1408, 2017.

[117] M. Daran and P. Thévenod-Fosse, "Software error analysis: A real case study involving real faults and mutations," *ACM SIGSOFT Software Engineering Notes*, vol. 21, no. 3, pp. 158–171, 1996.

[118] European Space Agency, "Space," https://sir.csc.ncsu.edu/portal/bios/space.php, 2020.

[119] P. G. Frankl and O. Iakounenko, "Further empirical studies of test effectiveness," in *Proceedings of the 6th ACM SIGSOFT international symposium on Foundations of software engineering*, 1998, pp. 153–162.

[120] M. Papadakis, T. T. Chekam, and Y. Le Traon, "Mutant quality indicators," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 32–39.

[121] T. T. Chekam, M. Papadakis, Y. Le Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 597–608.

[122] D. L. Phan, Y. Kim, and M. Kim, "Music: Mutation analysis tool with high configurability and extensibility," in *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 2018, pp. 40–46.

[123] B. Wang, Y. Xiong, Y. Shi, L. Zhang, and D. Hao, "Faster mutation analysis via equivalence modulo states," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2017, pp. 295–306.

[124] M. Papadakis, M. Delamaro, and Y. Le Traon, "Mitigating the effects of equivalent mutants with mutant classification strategies," *Science of Computer Programming*, vol. 95, pp. 298–319, 2014.

[125] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proceedings of the 36th International Conference on Software Engineering*. ACM, 2014, pp. 919–930.

[126] J. A. Clark, H. Dan, and R. M. Hierons, "Semantic mutation testing," *Science of Computer Programming*, vol. 78, no. 4, pp. 345–363, 2013.

[127] F. Wu, J. Nanavati, M. Harman, Y. Jia, and J. Krinke, "Memory mutation testing," *Information and Software Technology*, vol. 81, pp. 97–111, 2017.

[128] P. Delgado-Pérez, S. Segura, and I. Medina-Bulo, "Assessment of c++ object-oriented mutation operators: A selective mutation approach," *Software Testing, Verification and Reliability*, vol. 27, no. 4-5, p. e1630, 2017.

[129] P. Delgado-Pérez, I. Medina-Bulo, J. J. Domínguez-Jiménez, A. García-Domínguez, and F. Palomo-Lozano, "Class mutation operators for c++ object-oriented systems," *annals of telecommunications-annales des télécommunications*, vol. 70, no. 3-4, pp. 137–148, 2015.

[130] G. U. Yule, "On the methods of measuring association between two attributes," *Journal of the Royal Statistical Society*, vol. 75, no. 6, pp. 579–652, 1912.

[131] R-project, "Nonlinear least squares," 2020. [Online]. Available: https://stat.ethz.ch/R-manual/R-devel/library/stats/html/nls.html

[132] W. N. Venables and B. D. Ripley, *Modern Applied Statistics with S*. Springer Publishing Company, Incorporated, 2010.

[133] C. A. R. Diniz, M. H. Tutia, and J. G. Leite, "Bayesian analysis of a correlated binomial model," *Braz. J. Probab. Stat.*, vol. 24, no. 1, pp. 68–77, 03 2010.

[134] S. Mingoti, "A note on the sample size required in sequential tests for the generalized binomial distribution," *Journal of Applied Statistics*, vol. 30, no. 8, pp. 873–879, 2003.

Oscar Cornejo is a Research Associate at the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg. He obtained his PhD degree in Computer Science from the University of Milano - Bicocca in 2019.

His research interests are in software engineering, focusing on automated software testing and program analysis. He is currently involved in research projects with industry partners from the space domain.

Fabrizio Pastore is Chief Scientist II at the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg. He obtained his PhD in Computer Science in 2010 from the University of Milano - Bicocca.

His research interests concern automated software testing, including security testing and testing of AI-based systems; his work relies on the integrated analysis of different types of artefacts (e.g., requirements, models, source code, and execution traces). He is active in several industry partnerships and national, ESA, and EU-funded research projects.

Lionel C. Briand is professor of software engineering and has shared appointments between (1) School of Electrical Engineering and Computer Science, University of Ottawa, Canada and (2) The SnT centre for Security, Reliability, and Trust, University of Luxembourg. He is the head of the SVV department at the SnT Centre and a Canada Research Chair in Intelligent Software Dependability and Compliance (Tier 1).

He holds an ERC Advanced Grant, the most prestigious European individual research award, and has conducted applied research in collaboration with industry for more than 25 years, including projects in the automotive, aerospace, manufacturing, financial, and energy domains. He was elevated to the grades of IEEE and ACM fellow, granted the IEEE Computer Society Harlan Mills award (2012) and the IEEE Reliability Society Engineer-of-the-year award (2013) for his work on software verification and testing. His research interests include: Testing and verification, search-based software engineering, model-driven development, requirements engineering, and empirical software engineering.