Noname manuscript No.
(will be inserted by the editor)

Fuzzing-based Mutation Testing of C/C++ CPS

**Jaekwon Lee · Fabrizio Pastore ·
Lionel Briand**

Received: date / Accepted: date

Abstract Mutation testing can help minimize the delivery of faulty software. Therefore, it is a recommended practice for developing embedded software in safety-critical cyber-physical systems (CPS). However, state-of-the-art mutation testing techniques for C and C++ software, which are common languages for CPS, depend on symbolic execution. Unfortunately, symbolic execution's limitations hinder its applicability (e.g., systems with black-box components).

We propose relying on fuzz testing, which has demonstrated its effectiveness for C and C++ software. Fuzz testing tools automatically create test inputs that explore program branches in various ways, exercising statements in different program states, and thus enabling the detection of mutants, which is our objective.

We empirically evaluated our approach using software components from operational satellite systems. Our assessment shows that our approach can detect between 40% and 90% of the mutants not detected by developers' test suites. Further, we empirically determined that the best results are obtained

Jaekwon Lee
Kangwon National University, Chun-cheon, KR
E-mail: jaekwon.lee@kangwon.ac.kr

Fabrizio Pastore
SnT, University of Luxembourg, Luxembourg, LU
E-mail: fabrizio.pastore@uni.lu

Lionel C. Briand
Lero, University of Limerick, Limerick, IE
University of Ottawa, Ottawa, CA
E-mail: lbriand@uottawa.ca

This work was partially done while J. Lee was affiliated with the University of Ottawa and the University of Luxembourg.

by integrating the Clang compiler, a memory address sanitizer, and relying on laf-intel instrumentation to collect coverage and guide fuzzing. Our approach detects a significantly higher percentage of live mutants compared to symbolic execution, with an increase of up to 50 percentage points; further, we observed that although the combination of fuzzing and symbolic execution leads to additional mutants being killed, the benefits are minimal (a gain of less than one percentage point).

Keywords Mutation testing, Fuzzing, Test data generation

1 Introduction

Ensuring high-quality test suites is essential for quality assurance of embedded software in cyber-physical systems (CPS); indeed, this is the reason why independent software validation and verification activities are required by safety standards [1]. Further, although embedded software for CPS can be developed using dedicated visual languages [2], our experience in the space context shows that most software is directly implemented using C and C++, which suggests that approaches supporting validation and verification shall target those languages.

Mutation analysis is an effective method to evaluate the quality of a test suite. It involves measuring the mutation score, which is the proportion of programs with artificially injected faults (mutants) detected by the test suite [3]. There is a strong association between a high mutation score and a high fault-revealing capability for test suites [4, 5]. Additionally, recent studies have demonstrated that mutation analysis can be applied cost-effectively to large CPS software [6].

In practice, mutation analysis warrants the selection of inputs for mutation testing, as test cases should ideally detect all or at least a significant proportion of the generated mutants. A mutant detected by a test suite is considered killed. However, due to the typically high number of mutants generated in large CPS projects, it is challenging for engineers to perform mutation testing manually.

Unfortunately, we lack automated test data generation techniques (*automated mutation testing* techniques) suitable for the mutation testing of CPS. Indeed, most existing techniques do not target C and C++, which are prevalent in CPS. Further, the state-of-the-art solution for automated mutation testing of C software, *SEMu* [7], is based on the KLEE symbolic execution engine [8]. While effective for command line utilities, it inherits the limitations of symbolic execution. Specifically, it requires environmental modeling (e.g., network communication) and cannot be applied to programs needing complex analyses for input generation (e.g., programs with floating point instructions). Additionally, it generates test inputs for command line utilities, which are rarely used in CPS, and does not produce unit test cases or target other CPS interfaces.

Search-based techniques developed for other programming languages (e.g., Java [9]) are impractical for C and C++ software due to the difficulty of instrumenting the software to compute dedicated fitness functions (e.g., branch distance). For instance, computing branch distance at runtime necessitates modifying all conditional statements in the software under test (SUT), requiring the source code to be processed with static analysis tools that load all dependencies. Configuring these tools to process the multiple source files in large systems is often impractical unless the tool is well integrated with the compiler used for the SUT. Moreover, CPS source files often rely on architecture-specific C constructs (e.g., for the RTEMS compiler [10]) that are not successfully parsed by static analysis frameworks [6].

Different from search-based testing techniques, grey-box fuzz testing techniques [11] (hereafter, *fuzzers*) can generate test data without relying on complex code instrumentation and thus can be easily applied to C and C++ software. However, fuzzers target console software, while CPS software is usually tested either with system-level test scripts interacting with a hardware emulator or through unit and integration test cases implemented with the same language as the SUT. In this paper, we focus on the automated generation of unit test cases because fuzzing large systems is an open research problem [12]. Approaches for fuzzing function calls at unit or API level exist [13–17], although in the case of large CPS software, their applicability might be limited by static analysis' scalability issues and difficulties in processing embedded libraries. Further, approaches to generate fuzzing drivers for mutation testing do not exist.

We propose *MutatiOn TestIng with Fuzzing* (*MOTIF*), an approach that automatically generates fuzz drivers, leverage the data produced by fuzzers, and exercise a mutated function and the corresponding original function, looking for diverging outputs to detect mutants. *MOTIF* automatically generates seed files for fuzzers and integrates strategies to process mutant-killing inputs to eliminate false positives due to nondeterminism.

Instead of designing a dedicated fuzzing algorithm, *MOTIF* leverages state-of-the-art (SOTA) fuzzers because we believe that the fuzzers' bucket-based coverage strategy can effectively select test inputs that eliminate mutants. This strategy not only selects inputs that lead to different software states but also tracks and is guided by the behavioral differences between the original and mutated functions. Although *MOTIF* can target any C/C++ program, we designed it taking into account characteristics that are prominent in embedded software for CPS. Specifically, CPS software is not executed from the textual console but directly within the operating system, largely relies on floating point instructions, and leverages specific OS and libraries that complicates its analysis or symbolic interpretation. For these reasons, we present *MOTIF* as a tool for C/C++ embedded software for CPS.

We introduced *MOTIF* in a paper presented at the 2023 International Conference on Automated Software Engineering [18] where we reported on its effectiveness on three case study subjects in the space domain developed using the C language and demonstrated that it outperforms mutation testing based on symbolic execution. In this paper, we extend our previous work by:

– Performing a large empirical study to investigate (RQ1) the best fuzzing configurations (compiler, sanitizers, and coverage metric) for mutation testing. Note that the identification of the best fuzzing configuration for different application contexts remains an open problem [19]. Our experiments show that the best results are obtained when relying (1) on the `Clang` compiler, which leads to the quickest fuzz drivers, (2) the address sanitizer (`ASAN`), which prevents false positives due to violation of function preconditions, and (3) the *LAF-Intel* [20] optimization for coverage, which enables computing a better fitness score for mathematical functions, which are, in turn, among the most critical software components. Specifically, the best fuzzing configuration, compared to a standard fuzzing configuration (i.e., AFL++ [21] with the GCC [22] compiler), leads to an improvement varying between 1.53 and 4.34 percentage points. Although limited, such improvements may still enable the detection of critical faults; for example, when reusing the generated test cases for regression testing. Further, our findings help engineers make informed decisions on projects setups. For example, mutation testing improvements may not justify a large development endeavor to enable a project to be recompiled with Clang. However, when compiling projects with Clang, ASAN, and LAF-Intel is feasible (e.g., it is sufficient to change a few configuration parameters), the best configuration is a better choice (e.g., enables testing additional boundary cases) and might be considered also for other fuzzing applications beyond mutation testing.
– Assessing (RQ4) the performance gain achieved by relying on hybrid-fuzzing, which combines fuzzing and symbolic execution. Our results confirm the findings of related work that demonstrate that hybrid fuzzing leads to improved test effectiveness; however, in our experiments, its applicability is limited as it helps significantly improve the mutation score only one in four subjects. The key advantage observed with hybrid-fuzzing is the reduction of killing time (up to 32% faster), which suggests using hybrid-fuzzing for large projects leading to a large number of mutants.
– Evaluating (RQ5) the cost-effectiveness of reusing mutant-killing inputs to test live mutants; our results suggest that reusing mutant-killing inputs to kill other mutants is recommended when there is a sufficient test budget of at least 5000 seconds per mutant.
– Replicating, with the best fuzzing configuration, our original [18] assessments of the complementarity between fuzzing and symbolic execution for mutation testing (RQ2), and the contribution of *MOTIF*'s seeding strategy to its results (RQ3). With our subjects, *MOTIF* kills between 39.5% and 88.7% of mutants, on average, with most of the mutants (52% to 95%) being killed by the search process, not the seed inputs. Further, *MOTIF* outperforms, by 13.18 to 50.33 percentage points, the results obtained with the mutation testing process based on symbolic execution implemented by

the state-of-the-art tool *SEMu*, thus suggesting that *MOTIF* is the most appropriate solution for the mutation testing of CPS embedded software. This is the first work comparing fuzzing and symbolic execution for mutation testing.
- Contributing to the software engineering literature with the first study of mutation testing on industrial software for CPS. In this paper, we assess *MOTIF* on the control software of ESAIL [23], a satellite currently on orbit, and *Sentinel-5 UVNS L1b Prototype Processor*, a ground software processing radiation data collected by the Sentinel mission of the European Space Agency (ESA) [24], in addition to the subjects considered in our conference paper: a mathematical library qualified for flight systems, a commercial utility library for nanosatellites, and a serialization/deserialization library, all made available in the context of a project with ESA [25].
- Providing a technical solution enabling the application of *MOTIF* to object-oriented C++ programs. Since state-of-the-art mutation testing tools do not target C++ software, our solution enables mutation testing assessment on a broader set of case study subjects.

MOTIF is available online [26]; also, we provide a *replication package* with our open-source subjects and all our empirical data [27].

The paper proceeds as follows. Section 2 provides background (symbolic execution and fuzzing) and related work (automated mutation testing). Section 3 describes *MOTIF*. Section 4 presents our extensions to deal with object-oriented programs in C++. Section 5 presents our empirical evaluation. Section 6 concludes the paper.

2 Background and Related Work

This paper relates to the enormous body of work on automated mutation testing and fuzzing; selected, relevant work is discussed below.

2.1 Symbolic execution

Symbolic execution (SE) is a program analysis technique that relies on an interpreter to process the source code of the SUT and automatically generate test inputs [28]. Inputs are represented through symbolic values; during the symbolic execution, the state of the SUT includes the symbolic values of program variables at that execution point, a path constraint on the symbolic values to reach that point, and a program counter. The path constraint is a boolean formula that captures the conditions that the inputs must satisfy to follow that path. Constraint solving [29] is then used to identify assignments for the symbolic inputs that satisfy the path constraint.

SE presents several limitations, including (1) the need for abstract representations for the external environment and any black-box components used by the SUT—otherwise, the SE engine cannot know what outputs to expect from the environment, (2) path explosion—the SE engine may need to process a large number of paths before satisfying a target predicate, (3) path divergence—abstract representations do not behave like the real systems, (4) the handling of complex constraints, e.g., solving constraints with floating point variables.

A recent solution to partially address the above-mentioned limitations is dynamic symbolic execution (DSE), which consists of treating only a portion of the program state symbolically. Concrete program states help dealing with complex constraints or path explosion (e.g., SE is used after a certain branch has already been reached using a concrete input). However, most frameworks with DSE capabilities, e.g., Angr [30], KLEE [8], and S2E [31], require some degree of environment modeling (e.g., libc library modeling in KLEE), which limits their practical applicability [32].

Compilation-based approaches like QSYM [33], SYMCC [32], SymQEMU [34], and SYMSAN [35] augment the original program with instructions to populate and solve symbolic expressions while the original software is executed; such characteristic eliminates some limitations of interpretation-based approaches, thus being applicable to a broader set of software systems. For example, since the symbolic execution interacts with the actual environment there is no need to emulate it within the interpretation layer. SYMCC requires the source code of the SUT, while QSYM relies on dynamic binary instrumentation. SymQEMU, instead, extends the applicability of SYMCC to binary programs by relying on the QEMU emulator for code instrumentation, while SYMSAN relies on dynamic data-flow analysis to reduce the cost of symbolic state management. Since the above-mentioned symbolic execution approaches have shown to provide their best results when combined with fuzzing, a solution referred to as *hybrid fuzzing* (see Section 2.2), we considered them when assessing the integration of *MOTIF* with hybrid fuzzing (see Section 5).

2.2 Fuzzing

Fuzzing (or fuzz testing) is an automated testing technique that generates test inputs by repeatedly modifying[1] existing inputs; the selection of the inputs to modify is usually driven by metrics collected during the execution of the SUT. Depending on the information collected during program execution, fuzzing techniques (i.e., fuzzers) are classified as black-box, white-box, or gray-box.

In this paper, we focus on *grey-box fuzzers* because they have demonstrated to effectively maximize code coverage [36] and discover faults [37] (mainly crashes and memory errors), two objectives that relate to the problem studied in this paper; indeed, to kill a mutant it is necessary to (1) exercise a mutated statement, which can be achieved by maximizing code coverage, and (2) exercise the mutated statement with many different inputs (i.e., in different states), a common practice in fuzzers to discover crashes and memory errors.

[1] To avoid confusion, we avoid the term 'mutation' when describing fuzzing techniques.

Most fuzzers generate input files to be used for system-level testing of console applications and engineers are therefore required to implement driver programs (hereafter, *fuzzing drivers*) that rely on the data generated by the fuzzer to test other software interfaces (e.g., APIs, see Section 3.1). Most fuzzers keep a pool of input files and rely on the following evolutionary search process: (1) select an input file from the pool, (2) modify the input file to generate new input files, (3) provide the new input files to the SUT and monitor its execution, (4) report crashes or problems detected through code sanitizers (hereafter, sanitizers [38]), (5) add to the pool all the input files that contribute to improve code coverage.

What facilitates the adoption of fuzzers is that they rely on simple dynamic analysis strategies to trace branch coverage of C/C++ programs. A common strategy consists of dynamically identifying branches by applying a hashing function to the identifiers assigned to code blocks by compile-time instrumentation; it is implemented as an extension of popular C/C++ compilers [39]. Further, instead of relying on traditional branch coverage [40], most fuzzers adopt a bucketing approach to track the number of times each branch has been covered by each input file: only once, twice, three times, between four and seven, between 8 and 15, between 16 and 31, etc.; the fuzzers add to the pool those files that cover a bucket not observed before for at least one branch. Such bucketing strategy helps reach software states that are not reachable by simply relying on branch coverage.

Fuzzers mainly differ with respect to the strategy adopted to (1) select what operations to apply in order to modify input files and obtain new ones (e.g., MOpt [41] relies on a particle swarm optimization algorithm) and (2) select the inputs from the input pool (e.g., AFLfast [42] and AFL++ [43] rely on a simulated annealing algorithm and prioritize new paths and paths exercised less frequently). Also, fuzzers differ in the strategy adopted to determine interesting inputs. For example, *directed grey-box fuzzers* [44], instead of maximizing code coverage, aim to reach specific targets, usually a subset of program locations (e.g., modified code) or invalid sequences of operations (e.g., use-after-free).

Hybrid fuzzers [33,45,46], instead, rely on grey-box fuzzing to explore most of the execution paths of a program and leverage DSE to explore branches that are guarded by narrow-ranged constraints when the fuzzer does not improve coverage further. Well-known hybrid fuzzing solutions consist of combining AFL [47] with QSYM [33] and SYMCC [32], which have been found to outperform earlier approaches such as Driller [46] and hybrid testing [45]. The SOTA approach is Fuzzolic [48], which relies on QEMU to generate symbolic queries and solves them with a fuzzing-based technique [49].

Some researchers have addressed the problem of generating test drivers to fuzz test program functions as in unit testing [13–17]; however, except for Hopper [17], they make the assumption that the function under test are already integrated into consumer programs (i.e., programs using the library API) or unit tests [16], and all of them rely on complex static and dynamic analyses which are infeasible with large CPS software. Nevertheless, although existing tools do not target the generation of drivers for mutation testing, studying their integration into *MOTIF* is part of future work.

To avoid relying on program analysis, building on the potential shown by LLM-based test case generation [50–52], researchers and developers are investigating LLM-based fuzz driver generation [53–55]. For example, a recent study has demonstrated the feasibility of relying on large language models (LLMs) to automatically generate fuzz driver [53]; specifically they could generate effective drivers (compile and cover the code) for 91% of the 86 APIs under test with, however, some limitations such as violating API protocol (e.g., parameter initialization) in 39% of the cases. LLM-based generation of mutation testing drivers in *MOTIF* is part of future investigations.

Other techniques address the problem of generating highly structured input files [56,57]. TensileFuzz generates structured inputs (e.g., image or zip files) by probing random executions to derive constraints for potential input fields, and then relying on string constraint solving to derive inputs [56]. SkyFire, instead, learns a probabilistic context-sensitive grammar to generate JSON and XML files [57]. Such techniques can generate input files with a complex structure but they do not generate unit test cases, which is necessary in our context; however, leveraging those approaches to populate complex data structures may also help with unit-level fuzz testing.

2.3 Automated mutation testing

To kill a mutant, a test case should satisfy three conditions: *reachability* (i.e., the test case should execute the mutated statement), *necessity* (i.e., the test case should cause an incorrect intermediate state if it reaches the mutated statement), and *sufficiency* (i.e., the observable state of the mutated program should differ from that of the original program) [58]. Automated mutation testing approaches differ regarding the strategy adopted to satisfy these conditions.

There exist two families of automated mutation testing techniques based respectively on: *constraint solving* and *meta-heuristic search*. Only one of them relies on fuzzing [59], as further described below.

In this Section, we mainly focus on techniques targeting C and C++ programs because these languages are used in many CPS; unfortunately, the C and C++ languages are more complex to process for static and dynamic analysis techniques than the higher-level languages targeted by most of the techniques in the literature (e.g., Java).

2.3.1 Techniques based on constraint solving

Inspired by the earlier work of Offut et al. [58], Holling et al. execute symbolically the original and mutated functions with input data leading them to generate different outputs [60]. A similar technique from Riener et al. [61] relies on a bounded model checker (BMC) to select the input values that kill the mutant. Unfortunately, no prototype tools for the above-mentioned approaches are available.

The SOTA tool for automated mutation testing is *SEMu* [7, 62], which relies on KLEE to generate test inputs based on SE. To speed up mutation testing, *SEMu* relies on meta-mutants (i.e., it compiles mutated statements and the original statements together). First, *SEMu* relies on SE to reach mutated statements (reachability condition). Then, for each mutant, it relies on constraint solving to determine if inputs that weakly kill the mutant exist (necessity condition). For killable mutants, it symbolically runs the mutated and the original program in parallel; when an output statement is reached (e.g., a `printf` or the `return` statements of the main function), it relies on constraint solving to identify input values that satisfy the sufficiency condition.

2.3.2 Techniques based on meta-heuristic search

Most of the work on automated mutation testing with meta-heuristic search targets Java software; we report the most relevant techniques below. Ayari et al. [63] rely on an Ant Colony Optimization algorithm [64] driven by a fitness function that focuses on the reachability condition. Precisely, their fitness measures the distance (number of basic blocks in the program's control flow graph) between the mutated statement and the closest statement reached by a test case. Fraser and Zeller [9], instead, extended the EvoSuite tool [65] with a fitness function considering the reachability and the necessity conditions (number of statements that are covered a different number of times by the original and the mutated program). The integration of mutation testing into EvoSuite has been further improved with branch distance metrics tailored to the operator used to generate the mutants [66]. Recently, EvoSuite has been further extended by Almulla et al. with adaptive fitness function selection (AFFS), a hyperheuristic approach that relies on reinforcement learning (RL) algorithms to determine which composition of fitness functions to use [67]. Unfortunately, when applied to mutation testing, AFFS does not perform better than SOTA solutions [66].

Concerning C software, we should note the work of Souza et al. [68], who rely on the Hill Climbing AVM algorithm [69]. They combine three fitness functions that rely on branch distance to measure how far an input is from satisfying each of the three killing conditions. The mutation score obtained with simple C programs ranges between 52% and 93%. The approach has been implemented on top of AUSTIN, a search-based test generation tool for C [70–72]; however, this implementation is not available. A recent search-based testing tool prototype for C is Ocelot [73]; however, it has not been extended for automated mutation testing. Another key limitation of both Ocelot and AUSTIN is that they implement preprocessing steps that do not work with complex program structures (e.g., we couldn't apply them to the subject programs considered in our empirical evaluation because of preprocessing errors).

A recent mutation testing technique targeting C software is that of Dang et al. [74], who propose a co-evolutionary algorithm that reduces the search domain at each iteration (the original search domain is replaced by the joint domain of the best solutions found); unfortunately, their prototype is not available.

2.4 Techniques based on fuzzing

The work of Bingham [59] is the only one to rely on fuzzing to automate mutation testing for C software. For input generation, it relies on TOFU [75], a grey-box, grammar-aware fuzzer that generates grammar-valid inputs by modifying existing ones. Similar to Ayari's work, TOFU's input generation strategy is guided by the distance between the mutated statement and the closest statement reached by a test case; however, instead of generating unit test cases, it generates input files matching a given grammar. Unfortunately, the results obtained by Bingham are preliminary (they targeted only the Space benchmark [76]) and a prototype tool is not available.

DifFuzz [77], instead, executes two distinct versions of a program and compares their coverage and execution cost (e.g., time) to identify inputs that trigger side channel attacks; although this procedure might be leveraged to perform mutation testing (i.e., execute the original program and the mutant), its implementation targets Java systems. Further it is worth noting that DifFuzz does not compute any difference in terms of code coverage, as well as, the difference in execution time is delegated to a driver that needs to be manually implemented by the software engineer, thus limiting the usability of the approach.

Mu2 [78], which has been developed in parallel with *MOTIF*, is a fuzzer that integrates the findings of search-based unit test generation [79] to generate test input files with fuzzing: it relies on the mutation score to drive the generation of test inputs. Different from *MOTIF*, which tests each mutant independently from the others, potentially with different inputs, Mu2 tests every live mutant with each generated input and, in the input pool, prioritizes those inputs that increase the mutation score. Results show that Mu2 kills more mutants than the inputs generated by a traditional fuzzer to test the original function. However, it is unclear whether Mu2's approach (i.e., testing all the live mutants together) is more effective than that of *MOTIF* (i.e., executing a fuzzer to test the original and mutated function in sequence). Unfortunately, a direct comparison of Mu2 and *MOTIF* is not feasible because the scalability of Mu2 is enabled by dynamic classloading and instrumentation, two options that are feasible for Java programs but not for the C/C++ programs targeted by *MOTIF*. Further, by targeting Java, Mu2 can easily determine if mutants are killed by relying on the method 'equals', which is implemented by every class to determine if two instances are equal; the method 'equals' is not available in C and C++ software. Mu2's results follow previous work showing that, in Java benchmarks, prioritizing inputs that increase the mutation score may lead to higher branch coverage and mutation score than traditional prioritization strategies based on branch coverage [80].

2.5 Industrial applications of mutation testing

Most mutation testing approaches target Java software; the few mutation testing approaches targeting C/C++ software have been assessed with open source console utilities [7] or simple algorithm implementations (e.g., merge sort) [60, 61], but there is no work assessing mutation testing tools on C/C++ industrial software. Last, most empirical studies on large industrial software systems concern mutation analysis [81], not mutation testing.

2.6 Summary

To summarize, our research is motivated by the lack of support for automated mutation testing of C/C++ software. The SOTA approach for the automated mutation testing of C/C++ software (i.e., *SEMu*) relies on KLEE and inherits its limitations, making it inapplicable to most CPS software; further, it does not generate unit test cases but selects inputs for console programs. Other SE tools (QSYM and SYMCC) also present technical limitations preventing their application to CPS software. Search-based approaches for the mutation testing of C/C++ software present acute feasibility challenges due to static analysis, which is needed for branch distance fitness but does not scale in large software projects. Though fuzzing appears to be a feasible input generation strategy for mutation testing, existing fuzzers do not generate test drivers for unit testing. The only fuzzer proposed for mutation testing is not available for download and its results are very preliminary.

Last, our paper addresses the lack of empirical assessments of mutation testing on industrial software by targeting libraries and control software used in satellite systems as case studies.

3 Proposed approach: MOTIF

Inspired by the work of Holling et al. [60], *MOTIF* aims to identify a set of test inputs that lead to different outputs when given to both the original and mutated functions. To achieve such objective with fuzzing, *MOTIF* generates a fuzzing driver for each mutated function. The fuzzing driver reads the input data generated by the fuzzer and supplies it as arguments to both the original and mutated functions. Finally, the fuzzing driver compares the outputs from both functions. If the outputs differ, the mutant is considered killed.

Our intuition is that fuzzers might be effective at killing mutants because, by invoking both the original and the mutated functions within the same fuzzing driver, we can leverage the bucket-based fuzzing strategy to cover the

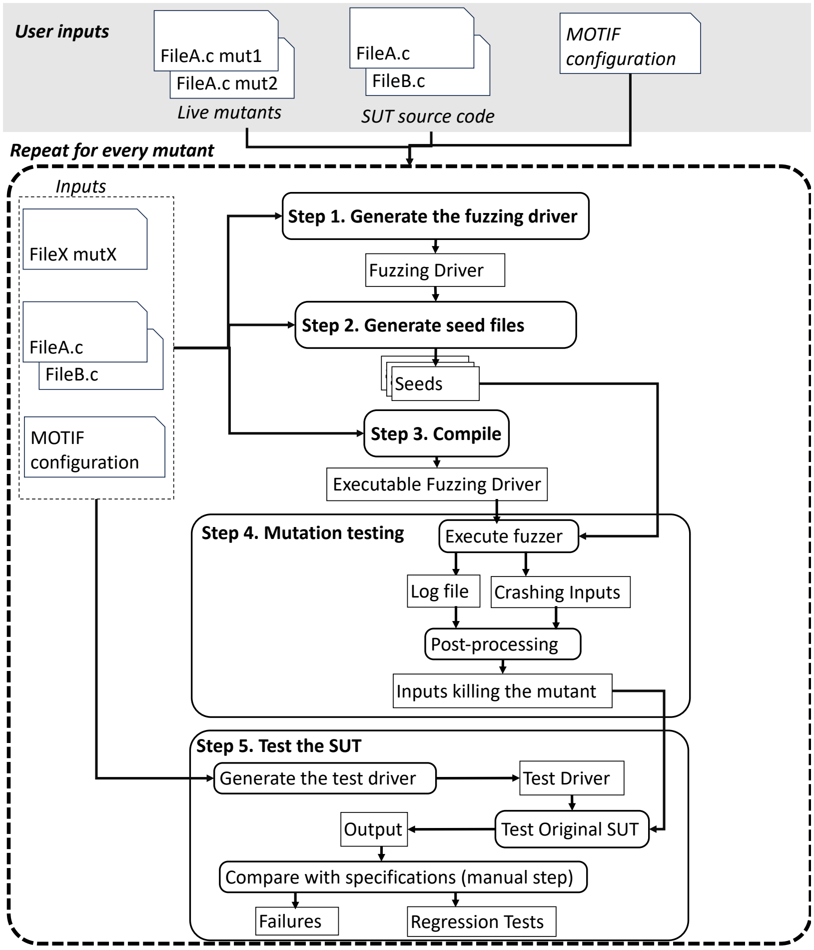


Fig. 1: The *MOTIF* process.

different branches in the two functions in diverse ways, thus reaching those program states that enable killing mutants. Essentially, the bucket-based fuzzing strategy may help kill mutants by preserving, during test generation, inputs that lead to incorrect intermediate states but do not kill the mutant (i.e., they do not meet the sufficiency condition). Subsequent iterations of the fuzzer's evolutionary search process (see Section 2.2) may modify these inputs so that they not only reach an incorrect intermediate state but also meet the sufficiency condition. Indeed, if differences in coverage between the original and mutated functions are observed, it indicates that the functions behave differently, resulting in different outputs and the mutant being killed [6, 82, 83]. Additionally, significant differences in coverage lead to new buckets being covered, and since fuzzing favors inputs that cover new buckets, it indirectly leads to inputs that kill mutants. We leave to future work the extension of fuzzers with dedicated strategies; for example, instead of measuring the coverage of the mutated function, the fuzzer could measure the difference in coverage between the original and the mutated function, and use this information to prioritize the inputs in the fuzzer queue (e.g., testing inputs that leads to larger differences first).

MOTIF creates all the necessary scaffolding to test both the original and mutated functions, and to compare their outputs. Specifically, *MOTIF* follows the workflow illustrated in Figure 1, which comprises the five steps detailed below.

MOTIF takes as input a set of mutants (source files) to be killed; each mutant matches the original source file except for the statements modified by a mutation operator. The *MOTIF* Steps in Figure 1 are repeated for each mutant. However, Steps 1 and 2 can be executed only once for all the mutants belonging to the same function; indeed, the structure of the input and output data of a function is not changed by mutation—we do not target interface mutation [84].

3.1 Step 1 – Generate the fuzzing driver

In Step 1, MOTIF relies on the *clang* static analysis library [85] to analyse the SUT and determine the types of the parameters required by the function under test. This information is then used to generate a fuzzing driver for mutation testing; an example fuzzing driver for the function `T_POS_IsConstraintValid` belonging to our *ASN1lib* case study subject is shown in Listing 1. The fuzzing driver renames the mutated function by adding the prefix *mut_*.

The fuzzing driver contains two sets of variables (Lines 5-7 and 8-10) whose types match the parameters of the function under test and are provided as input to both the original and the mutated function. In our example, it declares a `struct T_POS` and an `int` variable. These two sets of variables are then assigned by performing a byte-by-byte copy of a same portion of input file provided by the fuzzer (Lines 16-17 and 23-24, achieved by the function `get_value`); *MOTIF* ensures to copy a number of bytes to match the size of the assigned variable. If the input file provided by the fuzzer is shorter than required (the file modifications performed by fuzzers include shortening files), *MOTIF* extends it with random data (Line 2). Additionally, the fuzzing driver declares the variables required to store the functions' return values (Lines 11 to 13).

The original and the mutated functions are then invoked (Lines 19 and 26). The fuzzing driver then compares the outputs generated by both functions (Lines 28-31). Unfortunately, in C and C++, distinguishing between input and output parameters is complicated by the presence of pointer and reference arguments. Additionally, determining input parameters through data-flow analysis is impractical, as it requires preprocessing the SUT with a static analysis framework (e.g., LLVM [86]), which is often not feasible for CPS software [6].

```c
int main(int argc, char** argv){
  load_file(argv[1]);  // load the input file and
  // extends the input with random data if needed

  /* Variables for the original function */
  T_POS origin_pVal;   // for the first parameter
  int origin_pErrCode; // for the second parameter
  /* Variables for the mutated function */
  T_POS mut_pVal;      // for the first parameter
  int mut_pErrCode;    // for the second parameter
  /* Variables for the return values */
  flag origin_return;  // for the original
  flag mut_return;     // for the mutant

  /* Copy the input data to the variables for the original function */
  get_value(&origin_pVal, sizeof(origin_pVal), 0);
  get_value(&origin_pErrCode,sizeof(origin_pErrCode),0);
  log("Calling the original function");
  origin_return = T_POS_IsConstraintValid(&origin_pVal, &origin_pErrCode);

  /* Copy the same input data to the variables for the mutated function */
  seek_data_index(0); //reset the input data pointer
  get_value(&mut_pVal, sizeof(mut_pVal), 0);
  get_value(&mut_pErrCode, sizeof(mut_pErrCode), 0);
  log("Calling the mutated function");
  mut_return = mut_T_POS_IsConstraintValid(&mut_pVal, &mut_pErrCode);

  log("Comparing result values: ");
  ret += compare_value(&origin_pVal, &mut_pVal, sizeof(origin_pVal));
  ret += compare_value(&origin_pErrCode,&mut_pErrCode, sizeof(origin_pErrCode));
  ret += compare_value(&origin_return, &mut_return, sizeof(origin_return));

  if (ret != 0){
    log("Mutant killed");
    safe_abort();
  }
  log("Mutant alive");
  return 0;
}
```

Listing 1: Example fuzzing driver for the *ASN1lib* subject.

Therefore, we use a straightforward approach to compare outputs which consists of comparing all parameters and return values of the original and mutated functions; such approach does not lead to incorrect mutant killing because input parameters remain unmodified. For pointers, we compare the data they point to (e.g., an `int` instance for `int*`). If the pointer is used as an array,

Table 1: Seeds assigned to types

Type	Seed 1	Seed 2	Seed 3
int	-1	0	1
Bool	False	True	
float	-3230283776.0	0.0	1072693248.0
double	13826050856027422720.0	0.0	4602891378046628864.0
char	0xFF	0x00	0x41
byte	0xFF	0x00	0x41
ISO8601	2145916800.999999999	1970-01-01T00:00:00Z	2038-01-01T00:00:00Z

the end-user can specify the expected length of the array, so the array data can be compared. When arrays are inputs to the function under test, the end-user may not need to provide the length, as *MOTIF* automatically generates arrays with a default length of 100. If the function under test dynamically allocates arrays, the end-user should specify the minimal possible length (e.g., an array of length one) to avoid false positives from out-of-bounds readings. For data structures with pointer fields, the pointed data length and initialization procedure can be specified. If the outputs differ, the fuzzing driver halts execution with an abort signal (Line 35 in Listing 1), allowing the fuzzer to detect the aborted execution and store the input file. MOTIF then stops the fuzzer because the mutant has been killed.

3.2 Step 2 – Generate seed files

In Step 2, *MOTIF* creates seed files based on the input parameter types for the function under test. These seed files are used by the fuzzer to initiate the testing process. Typically, fuzzers are executed with seed files that correspond to typical inputs for the SUT. In our approach, we automatically generate seed files that contain enough bytes to populate all input parameters with values that cover basic cases. Specifically, for each primitive type, we have identified three representative seed values for typical input partitions, as shown in Table 1. For instance, for numeric values, we provide zero, a negative number, and a positive number. Using these seed values, *MOTIF* generates up to three seed files for each fuzzing driver, ensuring that each seed value is covered at least once for every input parameter.

Example seed files for function `T_POS_IsConstraintValid` are provided in Figure 2 (type definitions in Listing 2). *MOTIF* can also generate seed files for complex input types. For instance, the `struct T_POS` received as input by function `T_POS_IsConstraintValid` consists of an `enum` (named *kind*), which specifies the type of data stored inside the rest of the struct, and a union (named *u*), which is sufficiently large to contain the data for all the data types selectable with the variable *kind*. *MOTIF* treats such struct as an `int` array

	Seed 1	Seed 2	Seed 3
	FFFF FFFF FFFF FFFF	0000 0000 0000 0000	0001 0000 0001 0000
8048 bytes	FFFF FFFF FFFF FFFF	0000 0000 0000 0000	0001 0000 0001 0000
	*	*	*
	FFFF FFFF FFFF FFFF	0000 0000 0000 0000	0001 0000 0001 0000
	FFFF FFFF	0000 0000	0001 0000

Fig. 2: Seed files generated for the fuzzing driver in Listing 1.

```
typedef enum { T_POS_NONE,            longitude_PRESENT,
               latitude_PRESENT,      height_PRESENT,
               subTypeArray_PRESENT,  label_PRESENT,
               intArray_PRESENT,      myIntSet_PRESENT,
               myIntSetOf_PRESENT,    anInt_PRESENT
} T_POS_selection;

typedef struct {
    T_POS_selection kind;
    union { asn1Real longitude; asn1Real latitude;
            asn1Real height;    My2ndInt anInt;
            T_POS_label label;  T_ARR intArray;
            T_SET myIntSet;     T_SETOF myIntSetOf;
            T_POS_subTypeArray subTypeArray;
    } u;
} T_POS;
```

Listing 2: Definition of `struct T_POS`

thus filling it with the seeds *0xFFFFFFFF*, *0x00000000*, and *0x00000001*. The first four bytes in the seed files (see Figure 2) belong to the `enum` item *kind*, and are filled with the seed values of the `int` type. The same happens for the `union` field *u* but, since the `union` has a size of 8,052 bytes (size of *subTypeArray* with 4 bytes padding[2]), *MOTIF* repeats the same set of four bytes 2,013 times. The last four bytes belong to the second parameter of `T_POS_IsConstraintValid`, the `int` **_pErrCode*.

3.3 Step 3 – Compile the SUT

In Step 3, *MOTIF* produces an executable fuzzing driver by compiling the source code of the fuzzing driver, the mutated function, and the SUT using the fuzzer compiler, which is essential for gathering the code coverage information needed by the fuzzer. This step also compiles a false positive driver (see Section 3.4 below).

[2] https://research.nccgroup.com/2019/10/30/padding-the-struct-how-a-compiler-optimization-can-disclose-stack-memory/

3.4 Step 4 – Perform mutation testing

In Step 4, *MOTIF* runs the fuzzer to generate inputs for the executable fuzzing driver; in our experiments we leverage the AFL++ fuzzer but the approach is generic and can work with any grey-box fuzzer for C/C++. The fuzzer keeps generating input files until it reports one or more crashes, after which MOTIF halts the fuzzer. This process leads to the generation of fuzzing driver logs and crashing inputs (i.e., input files that caused a crash during the execution of the fuzzing driver). Since fuzzers generate several input files from each input taken from the file pool, and since all of them are executed by the fuzzer, more than one crashing input may be reported.

Fuzzing driver logs include checkpoints indicating the progress of testing (see Lines 18, 25, 28, 34, 37 in Listing 1). For each crashing input, *MOTIF* processes the corresponding logs to distinguish between:

- Crashes occurring during the execution of the original function. They indicate either the presence of a fault in the original function or the violation of preconditions. We ignore these inputs because they do not correspond to inputs killing a mutant.
- Crashes occurring during the execution of the mutated function. Since the mutated function is executed after the original one, we can safely conclude that the test inputs do not cause any crash in the original function. Therefore, the observed crashes indicate that the mutant introduced a fault that was exercised by the input. Thus, we can conclude such inputs kill the mutant.
- Aborted executions due to the fuzzing driver determining that the mutant has been killed (see Line 35 in Listing 1).

MOTIF retains all test inputs that kill a mutant. However, the function under test may produce non-deterministic outputs meaning that, despite observed differences in outputs, the inputs may not have actually killed the mutant. For instance, two consecutive invocations of a function that reads and writes global variables may lead to different outputs even if the mutated statement is not exercised; consequently, the input suggested by the fuzzer would be a false positive. To minimize false positives, *MOTIF* automatically re-executes every test input killing a mutant using a modified version of the fuzzing driver that invokes the original function instead of the mutated function. If this false positive driver, as we refer to it, reports a difference in outputs of the two function calls, it implies that the function under test is non-deterministic and thus that the input may not kill the mutant. *MOTIF* considers mutants exclusively killed by false positive inputs to be live. To kill mutants in functions that modify global state variables, the user must manually add, in the fuzzing driver between the two function calls, instructions to reset each state variable, similar to other fuzzing approaches for unit and library testing (e.g., LibFuzzer [87]).

```
1  int main(int argc, char** argv){
2    load_file(argv[1]); /* load the input file */
3    // Declaration of variables and assignment with input file
       data missing to save space...
4    /* Invoke the original function*/
5    _return = T_POS_IsConstraintValid(&pVal, &pErrCode);
6    /* Print output values of the original function */
7    printf_struct("pVal (T_POS)=", &pVal, sizeof(pVal));
8    printf("pErrCode (int) = %
9    printf("return (flag) = %
10   return 0;
11 }
```

Listing 3: Example test driver for the *ASN1lib* subject.

3.5 Step 5 – Test the SUT

In this Step, *MOTIF* generates a test driver for the SUT. An example test driver for function `T_POS_IsConstraintValid` is shown in Listing 3. The test driver is identical to the fuzzing driver except that (1) it only invokes the original function (Line 5 in Listing 3), (2) instead of comparing outputs from two function invocations, it prints the output data generated by the original function (Lines 7 to 9), and (3) it includes assertions that compare the execution results of the original function with the expected results (expected results are the ones observed during fuzzing for the original function). The test driver is used to test the original SUT with the inputs that kill the mutant and the outputs should then be verified by a software engineer based on the SUT specifications. If the observed output values are correct, they can be used as oracles for future regression testing, which is the reason why the test driver already includes automatically generated test assertions (i.e., *MOTIF* generates a complete regression test case). Otherwise, a fault has been identified in the SUT. This scenario highlights one of the key advantages of mutation testing: When testing the SUT with inputs that detect simulated human mistakes (mutants), actual faults in the SUT are more likely to be discovered than with randomly selected inputs (e.g., in our experiments we discovered five faults in our subjects).

In our test driver, the generation of print statements for structs and pointers is determined by the configuration of the fuzzing drivers. By default, all bytes within a struct are printed. When pointers are involved, if the end-user specifies the size of the data to which the pointers refer, the test driver prints the data pointed to, rather than the pointer value.

4 MOTIF C++ extensions

In this Section, we describe how we extended *MOTIF* to test C++ programs. This extension is necessary to accommodate the four key object-oriented programming (OOP) properties – abstraction, encapsulation, inheritance, and polymorphism – and their impact on mutation testing with *MOTIF*. Note that these problems are specific to C++ because mutation testing approaches for higher-level programming languages such as Java [9, 78] can leverage reflection to overcome them, as described below. For the sake of clarity, in this Section, we call *MOTIF-C* the version of *MOTIF* that handles C code and was presented in Section 3, and *MOTIF-C++* the version of *MOTIF* extended to deal with C++ features. Listing 4 provides a running example.

Abstraction is a key property of grouping variables and operations that are highly related. *Classes* are the means to declare those variables (i.e., attributes) and operations (i.e., methods) as new data types. It reduces the complexity of developing software by letting users use the classes without knowing how its features are implemented. Instead, for testing, instantiating a class is necessary to exercise the class instance methods. But *MOTIF-C* does not instantiate classes since, in C, operations are implemented by functions, which do not require any object to be instantiated. In C++, static methods play the same role as C functions, however they are rare. Consequently, *MOTIF-C* cannot be used to test most C++ methods.

```
class SatelliteModule {
    private:
    /** Charge in watts per hour **/
    double getCharge(){ return power; }

    protected:
    /** Average consumption in watts per hour **/
    double getAverageConsumption(){
        return consumedWatts()/elapsedTime();
    }

    public:
    /** Hours before being out of charge **/
    double getAutonomy(){
        return getCharge()/getAverageConsumption();
    }
};

class ChargeableModule : public SatelliteModule {
    private:
    /** Average watts charged per hour **/
    double getAverageCharging(){
        return chargedWatts()/elapsedTime();
    }

    protected:
    double getAverageConsumption(){
        return super::getAverageConsumption()-
    getAverageCharging();
    }

    public:
    ChargeableModule(double consumptionBoundary, double
    chargeBoundary){
```

```
        this.consumptionBoundary = consumptionBoundary;
        this.chargeBoundary = chargeBoundary;
    }

    /** True if the mission lasts less than the autonomy of the module**/
    bool isAutonomous( Mission mission){
        return mission.duration() <= getAutonomy();
    }
};

/** Absract class for missions **/
class Mission {
    public:
    /** Duration in hours **/
    virtual double duration();
}

class StaticMission {
    private:
    double presetDuration;

    public:
    StaticMission(double presetDuration){
        this->presetDuration=presetDuration;
    }

    double duration(){
        return presetDuration-elapsedTime();
    }
}

template <class ObjectiveT> class DynamicMission {
    public:
    void objectiveAchieved(ObjectiveT T){
        //remove from the list of objectives to achieve
        this->objectives.remove(T);
    }

    double duration(){
        return objectivesToAchieve()/objectivesAchievedPerHour();
    }
}
```

Listing 4: C++ example classes. For simplicity we show only in-class method declarations and we hide some method declarations.

Encapsulation is realized by hiding (i.e., making *private* or *protected*) a subset of the instance variables and methods implemented by a class. Consequently, a subset of class methods (i.e., private and protected methods) cannot be exercised by *MOTIF-C*; indeed, functions that do not belong to a class (e.g., the main function of the *MOTIF* fuzzing driver), cannot access the class private and protected members.

Inheritance enables a class to reuse (i.e., inherit) the methods declared by its superclass, if any. Inheritance complicates the unit testing automated by

```
class mut_ChargeableModule : public SatelliteModule {

    //... same code as in ChargeableModule

    bool isAutonomous( Mission mission){
        //MUTANT 1: replaced <= with <
        return
        mission.duration() < getAutonomy();
    }
};
```

Listing 5: Mutant class for a mutant of method `isAutonomous`.

MOTIF-C because it leads to dependencies between superclasses and subclasses. Indeed, not only a subclass may execute superclass methods but the execution of a method declared by a superclass may trigger the execution of a method in a subclass. For example, the execution of method `ChargeableModule.isAutonomous()` in Listing 4 triggers the execution of the superclass method `SatelliteModule.getAutonomy()`, which, in turn, triggers the execution of `ChargeableModule.getAverageConsumption()` and `ChargeableModule.getAverageCharging()`. As a consequence, the solution adopted by *MOTIF-C* to test mutants (i.e., creating a copy of the original method and renaming it) would prevent the testing of methods following such pattern; for example, renaming a mutant of method `getAverageConsumption()` in `ChargeableModule` as `mut_getAverageConsumption()` would prevent its call from method `SatelliteModule.getAutonomy()`, thus rendering testing ineffective.

Polymorphism refers to multiple language features. *Ad hoc polymorphism* (i.e., method overloading) indicates that methods with a same name can process arguments of different types. Since these methods have distinct implementations, they do not affect *MOTIF-C* because it treats them as distinct functions. *Parametric polymorphism* (i.e., class templates) implies that certain methods can be exercised only if the object they belong to has been instantiated with appropriate parameters (i.e., a specific class name). This is the case for class `DynamicMission`, which requires the specification of the data type used to model mission objectives as in "`mission = new DynamicMission<Landmark>();`" *Subtype polymorphism* occurs when the code contains a call to a method belonging to a certain object, but the concrete method to be invoked is decided at runtime, depending on the object type. This is the case for method `ChargeableModule.isAutonomous()`, which invokes `Mission.duration()` and, at runtime, may execute the implementation of `duration()` provided by either `StaticMission` or `DynamicMission`. To test those methods, it is necessary to determine the type of the object to be instantiated and passed as argument, which is not supported by *MOTIF-C*.

To address the issues above, *MOTIF-C++* implements a set of extensions that are summarized in Table 2 and described below.

Table 2: *MOTIF-C++* extensions to deal with C++ language features.

Language feature	***MOTIF-C++* extension**
Inheritance	Copy mutant class.
Abstraction	Automatically generate test utility class.
Ad hoc polymorphism	Separate testing for each method.
Encapsulation	
Parametric polymorphism	Manual editing of test utility class.
Subtype polymorphism	

```cpp
class mut_ChargeableModule : public SatelliteModule {

    //... same code as in ChargeableModule

    protected:
    double getAverageConsumption(){
        //MUTANT 2: removed '- getAverageCharging()'
        return super::getAverageConsumption();
    }

    //... same code as in ChargeableModule
};
```

Listing 6: Mutant class for a mutant of method `getAverageConsumption`.

To deal with *inheritance*, *MOTIF-C++* injects the mutated method into a copy of the original class. Examples are shown in Listing 5 and 6, which present two distinct mutants of class `ChargeableModule`. Note that in the case of Listing 6, the creation of a renamed class copy, instead of a renamed method copy as in *MOTIF-C*, ensures that the mutated method is executed when testing *mut_ChargeableModule.isAutonomous()*. In Java, this issue is addressed in a similar but more efficient way [78], by leveraging the classloader mechanism to load in memory both the hierarchy of the original class and its mutated version (one for each mutant), which is not feasible in C++ because of the lack of a classloader managed by a virtual machine.

To deal with *abstraction*, *MOTIF-C++* creates a fuzz driver that, instead of invoking the mutated method, creates an instance of a test utility class to which it delegates the instantiation of the class under test and the execution of the method under test. An example fuzz driver is shown in Listing 7; except for the instantiation of the utility class, it matches the fuzz driver created by *MOTIF-C*. A test utility class (i.e., `MOTIF::TestUtil`) is shown in Listing 8. Test utility classes are automatically generated by *MOTIF-C++* in Step 1, but often need to be manually modified as described in Section 4.1. A utility class includes state variables to hold the values to be passed as arguments to the constructor of the class under test and to the method under test (Lines 4-9 in Listing 8). Still in `MOTIF::TestUtil`, two additional state variables are used to store references to the value returned by the method under test (Line 11) and the instance under test (Line 13). The solutions implemented by approaches targeting Java never need manual intervention because they can leverage reflection to determine what are the constructors and, recursively the constructors for their parameters, and just rely on automated trial and error to identify appropriate initializations [9, 78].

In `MOTIF::TestUtil`, methods `call` and `compare` take care of the execution of the method under test and the comparison of the results obtained by the original and the mutated method, respectively. Method `call` (Lines 13-25, Listing 8), which executes the method under test, includes automatically generated instructions to populate parameters with fuzzed data (Lines 17 and 18). If the constructor of a function under test receives as input an abstract type, the instructions needed to construct the concrete type to be used shall be manually added (Lines 22-24). Last, method `call` invokes the method under test and stores the returned value (Line 26). Method `compare`, instead, is invoked by the fuzzing driver (Line 13, Listing 7) to compare the results obtained by the original and the mutated class after the execution of the mutant; its logic follows the one of the fuzzing driver generated by *MOTIF-C*. Approaches for Java [9, 78] can leverage programming conventions and reflection to compare object states by leveraging the equals method, which is normally overridden to compare two instances of a class, or inspector methods, which can be identified through naming conventions or purity analysis (simpler to achieve on Java bytecode [88]).

To deal with *protected* and *private* methods (i.e., *encapsulation*), instead of modifying the class under test to specify the test utility class as a *friend class*, we suggest the end-user to exercise, in the fuzzing driver, a public method that invokes the private/protected method under test. For example, to test mutant 2 in Listing 6, the end-user may still rely on the fuzzing driver shown in Listing 7, because it tests method `isAutonomous`, which, in turn, executes method `getAverageConsumption` (i.e., the mutated method). In Java, protected and private methods can be invoked through reflection [89].

4.1 Manual tuning of *MOTIF*'s test utility class

Like *MOTIF-C*, *MOTIF-C++* automatically identifies the variables to create in the test utility class based on method signatures. However, *MOTIF-C++* requires the manual specification of the class to instantiate to deal with *parametric* and *subtype polymorphism.* For example, if a method declares parameters of an abstract type (e.g., `isAutonomous` receives one parameter of the abstract type `Mission`), the end-user needs to specify the additional parame-

```
1  int main(int argc, char** argv){
2    load_file(argv[1]);  // load the input file and
3    // extends the input with random data if needed
4
5    log("Calling the original function")
6    auto origin = new MOTIF::TestUtil<ChargeableModule>();
7    origin->call();
8
9    log("Calling the mutated function")
10   auto mutant = new MOTIF::TestUtil<mut_ChargeableModule>();
11   mutant->call();
12
13   if (orig->compare(mut) != 0){
14     log("Mutant killed");
15     safe_abort();
16   }
17   log("Mutant alive")
18 }
```

Listing 7: Example *MOTIF-C++* fuzz driver.

ters required to instantiate the concrete class to be used to exercise the method under test (Line 9).

Note that implementing such additional instructions is much less labor intensive than manually identifying the inputs that kill the mutant. Indeed, the proper initialization of the objects required for testing simply consists of the invocation of constructor methods. In contrast, the identification of the inputs that kill a mutant require a comparison between the mutated and the original functions to determine which inputs makes their output differ. Further, we expect that end-users can easily update *MOTIF*'s test utility classes by copying the code used in unit test cases, except for assignments to primitive variables, which shall rely on the data provided by the fuzzer. Indeed, in unit test cases, like in *MOTIF*'s test utility class, all the objects required by the method under test shall be instantiated; therefore, the instructions to be manually-specified in the test utility class can be a copy of the ones used in the manually implemented test cases, except for assignments to primitive variables.

Last, we consider the need for the manual modification of test utility classes a limitation of the *MOTIF* implementation rather than the approach itself. Indeed, it is possible to integrate an additional lightweight static analysis step into *MOTIF* to automate the generation of test drivers. Indeed, as demonstrated by approaches for the generation of fuzz drivers for API fuzzing [13], it is generally feasible to implement a lightweight static analysis parser that automatically generates sequences of function calls by copying the content of existing programs (test cases, in our context); however, implementing such a parser goes beyond the objective of this paper which, for the C++ part, is concerned with proposing and assessing a solution to determine when a mutated

```
namespace MOTIF{
template<typename TypeA>
class TestUtil{
  //parameters for the constructor
  double constructor_1;
  double constructor_2;
  //parameters for the method under test
  Mission parameter_1;
  int manual_1;
  //return value
  bool _return;
  //instance
  TypeA object;

  void call(){
    //populate variables with fuzz data
    get_value(&constructor_1, sizeof(constructor_1) );
    get_value(&constructor_2, sizeof(constructor_2) );

    //instantiate the class under test
    this->object = new TypeA(param1, param2);
    //instructions for parameters of an abstract type
    get_value(&manual_1,sizeof(manual_1));//added
    this->parameter_1 = new StaticMission(manual_1);
    //invoke the method under test
    this->_return = object->isAutonomous(this->parameter_1);
  }

  template<typename TypeB>
  int compare(TestUtil<TypeB> *rhs){
    int ret = 0;
    ret += compare_value(this->parameter_1,rhs->parameter_1);
    ret += compare_value(this->_return,this->_return);
    return ret;
  }
}
}
```

Listing 8: Example utility class.

method is killed (i.e., through a test utility class and a copy of the mutated class) rather than delivering a production-ready tool.

Concluding, although *MOTIF-C++* delegates the handling of several C++ features to the manual tuning of the automatically generated test utility class, the automated generation of both the fuzzing driver and the test utility class still significantly decreases manual effort when compared to manual mutation testing.

5 Empirical Evaluation

We address the following research questions:

RQ1. What fuzzer configuration leads to best results in MOTIF? Fuzzing results can vary depending on the configurations applied to the fuzzers, which include the selection of compilers, sanitizers, and coverage metrics. Executables' speed depend on compilers' code optimization capability and affect fuzzing effectiveness; indeed, quicker executions lead to more inputs being tested and, likely, to more mutants being killed. Sanitizers are helpful in detecting invalid behaviors, such as memory overflow and arithmetic overflow during software execution; consequently, they may facilitate mutants detection (e.g., mutants leading to such invalid behaviours) although their overhead may reduce the number of generated test inputs and, in turn, mutation testing effectiveness. Coverage metrics are used to determine when an input should be kept in the fuzzing queue, and can affect input generation and mutants killing; for example, specific metrics may facilitate the discovery of unexplored paths, which may include the mutated instructions. Through this research question, we aim to determine what configurations would be optimal for *MOTIF*.

RQ2. How does mutation testing based on fuzzing compare to mutation testing based on symbolic execution, for software where the latter is applicable? Since certain CPS units may still satisfy the assumptions of SE approaches (e.g., absence of floating-point instructions and black-box components), we aim to assess what approach performs better in such cases.

RQ3. How does MOTIF *'s seeding strategy contribute to its results? MOTIF* kills mutants either through the generated seeds or through fuzzed inputs; we therefore aim to assess how the two strategies individually contribute to *MOTIF* results in order to determine if fuzzing is indeed useful.

RQ4. Does hybrid-fuzzing improve the effectiveness of MOTIF? We aim to assess hybrid-fuzzing approaches because they demonstrated to be effective in increasing the code coverage obtained through fuzzing and thus may improve mutants detection. Fuzzing can be an advantage to exercise conditions that are easy to satisfy (e.g., branch conditions controlled by a relational operator in a basic clause, such as `x>0`) whereas symbolic execution helps satisfy narrow branch conditions (e.g., joining multiple clauses, such as `x>0 && x<3`) [33].

RQ5. Is it cost-effective to reuse mutant-killing inputs? Since mutants are often redundant [3], multiple mutants can be killed by the same inputs. Consequently, inputs generated by MOTIF that successfully kill some mutated versions of a function might be used to test other mutated versions that remained live. However, the execution of additional inputs increases mutation testing time. This research question aims to assess the tradeoff between mutation score improvement and increased testing time due to reusing mutant-killing inputs.

Table 3: Subject artifacts.

Subject	**Open-source**	**LOC**	**# Test cases**	**Statement coverage**	**Mutation score (MS)**
MLFS	Yes	5,402	4,042	100.00%	81.80%
LIBU	No	10,576	201	83.20%	71.20%
ASN1lib	Yes	7,260	139	95.80%	58.31%
ESAIL	No	2,235	384	95.36%	65.36%
S5	No	54,696	36	62.23%	64.13%

5.1 Subjects of the study

To address our research questions, we considered software deployed on space CPS (satellites) currently in orbit. This included (a) *MLFS*, the Mathematical Library for Flight Software [91], which complies with the ECSS criticality category B [1,92], (b) *LIBU*, which is a utility library developed by GomSpace and used in NanoSatellites, (c) *ASN1lib*, a serialization/deserialization library, (d) *ESAIL*, a subset of the control software of a micro-satellite developed by LuxSpace to track ships worldwide, and (e) Sentinel-5 UVNS L1b Prototype Processor (*S5*, for brevity), which is ground software developed by Huld to process radiation data received from the Sentinel 5 satellite instruments. *ASN1lib* has been generated with ASN1SCC from a test grammar provided by ESA. ASN1SCC is a compiler that generates C/C++ code suitable for low resource environments [93,94]. All the subjects are implemented in C except *S5*, which is implemented in C++.

Our software subjects are provided with test suites whose code coverage is reported in Table 3. Most test suites do not achieve 100% statement coverage because they include components that are tested with specific hardware not available to us; therefore, we generated mutants only for the covered statements. We generated mutants with MASS [6, 95]; specifically, we rely on all the mutation operators supported by MASS (i.e., the sufficient set [96] and the deletion set [97,98]), which proved effective in previous experiments on similar subjects. We excluded mutants that are identified as equivalent or duplicate according to trivial compiler equivalence methods [6]. The last column in Table 3 provides the mutation score (MS) for our case study subjects; it corresponds to the proportion of mutants detected by the test suite. The highest mutation score is observed with MLFS, whose test suite achieves MC/DC adequacy [99]. The lowest mutation score is observed with *ASN1lib*, which is automatically generated by ASN1SCC using a grammar-based approach [94]. Our subjects' mutation scores are in line with empirical investigations reporting mutation scores ranging from 55% to 95% [100, 101], for CPS software.

[3] A mutant is considered redundant if it can be killed by the same input that kills another mutant, even though they may differ syntactically [90].

To perform test data generation, we rely on the mutants not killed by the original test suites. We assume that the live mutants are not equivalent (i.e., produce the same outputs for every input) to the original software. Although this could be an under-approximation, it does not introduce bias in the comparison between the different approaches considered (i.e., *SEMuP* and *MOTIF* configured with different fuzzers and options) because all of them cannot kill equivalent mutants. Further, two mutants m_a and m_b can also be duplicates (i.e., they lead to the same outputs for every input) or subsumed (i.e., m_a is killed by a superset of the test cases killing m_b). However, the identification of test inputs that kill mutants is a precondition to determine if mutants are duplicate or subsumed [102]; for this reason, including duplicate and subsumed mutants should not introduce bias in the comparison of the approaches. In other words, a mutation testing approach should easily kill mutants that are either duplicates or subsume other killed mutants; if it does not happen, it is correct to penalize such an approach in the empirical evaluation.

The number of live mutants for each subject is: 443 for *LIBU*, 1,347 for *ASN1lib*, 3,891 for *MLFS*, 581 for *ESAIL*, and 99 for *S5*. For *LIBU*, *MLFS*, and *ESAIL*, we configured *MOTIF* to generate arrays of a specific size and, for void pointers (i.e., `void *`), to create variables or arrays considering the expected data type. Regarding *S5*, the relatively limited number of mutants considered is due to the need for the manual editing of most of the fuzz drivers generated by *MOTIF* (see Section 4). Indeed, advanced C++ features, including abstract classes, templates, friend functions, and unnamed namespaces, as well as the standard library implementations, render the drivers automatically generated by *MOTIF* ineffective. For example, `vector<>`, one of the standard template libraries (STL), uses pointers to trace the stored data, which renders the memcopy-based solution adopted by *MOTIF* to instantiate objects inappropriate (pointer addresses cannot be filled with data generated by the fuzzer). To address such limitation, we manually modified the fuzzing drivers automatically generated by *MOTIF*. Precisely, in total, to kill 99 mutants, *MOTIF* generated 25 drivers, 21 of which had to be manually modified. However, 14 drivers required a simple modification such as instantiating a template class (e.g., `vector<int>`), while seven drivers required an initialization step for the class under test, which we copied from the unit test cases provided by S5 developers. In addition, recall that, as discussed in Section 4.1, the need for such manual modifications is a limitation of the *MOTIF* implementation, not the approach, and can be overcome through an improved static analysis step. Further, compared to manual mutation testing, the automated generation of both the fuzzing driver and the test utility class still significantly decreases manual effort.

5.2 Experimental setup

We performed our experiments using a prototype implementation of *MOTIF* [18, 103].

```
1  int main(int argc, char** argv){
2      // Declare variable to hold function returned value
3      _Bool result;
4      // Declare arguments and make input ones symbolic
5      T_POS pVal;
6      int pErrCode;
7
8      klee_make_symbolic(&pVal, sizeof(pVal), "pVal");
9      // Call function under test
10     result = T_POS_IsConstraintValid(&pVal, &pErrCode);
11     // Print output data
12     printf("pErrCode = %
13     printf("result = %
14     return (int)result;
15 }
```

Listing 9: Example *SEMu* driver corresponding to the fuzzing driver in Listing 1.

To address all our RQs, as fuzzer for *MOTIF*, we selected `AFL++` because it is the fuzzer that performed better in terms of code coverage, according to recent benchmarks in the literature [36, 104, 105]; moreover, along with HonggFuzz [106], it is the fuzzer that maximizes fault coverage in another recent benchmark [37].

To address RQ2, and compare *MOTIF* with a mutation testing approach relying on symbolic execution, we modified the *MOTIF* pipeline to enable test generation with *SEMu*; we call such pipeline *SEMuP*. At a high level, *SEMuP* follows the same steps of *MOTIF*, with differences concerning how input and output variables are defined.

For *SEMuP*, in Step 1, we generate *SEMu* drivers enabling symbolic execution. An example *SEMu* driver generated for function `T_POS_IsConstraintValid` is shown in Listing 9. These drivers must specify what are the input parameters to be treated symbolically (see Line 8 in Listing 9), a task performed by the end-user. *SEMu* drivers do not include explicit comparisons between the outputs of the mutated and the original function because such comparison is handled by *SEMu* when symbolically executing the original and the mutated functions in parallel (see Section 2.3.1). Precisely, the *SEMu* driver invokes only the function under test and prints to standard output the data values that should be considered to determine if a mutant has been killed. Similar to *MOTIF*, *SEMu* also requires end-users to manually specify how to process data values belonging to data structures referenced with pointers. *SEMuP* does not include a Step for the generation of seed input (i.e., *MOTIF*'s Step 2). It includes a step (corresponding to *MOTIF*'s Step 3) to compile the mutated function and the *SEMu* drivers with LLVM, followed by a step, corresponding to *MOTIF*'s Step 4, for the execution of *SEMu* and the processing of its logs to determine killed mutants.

To address RQ3, to perform hybrid fuzzing, we selected `SymCC` [32], whose performance is similar to that of the more recent tools Fuzzolic, SymSAN, and SymQEMU [35]. We excluded SymSAN and its recent extension Marco [107] because SymSAN requires 64-bit compilation, which is not feasible with some of our subjects. We excluded Fuzzolic and SymQEMU since they rely on a version of QEMU that cannot run on some of our subjects (e.g., *ESAIL*). Therefore, in our response to RQ3, the benefits of hybrid fuzzing can be considered to be a lower bound.

To leverage hybrid fuzzing, we extended *MOTIF*'s Step 3 to also compile the fuzzing driver and SUT with `SymCC`'s compiler, in addition to `AFL++`'s. In Step 4, *MOTIF* executes the two compiled executable drivers together, following `AFL++` procedures for hybrid fuzzing with `SymCC`[4]. When `AFL++` discovers new interesting inputs according to its coverage metrics, `SymCC` is triggered to perform symbolic execution. The inputs found by `SymCC` are then merged into the queue of `AFL++`. Once either tool finds inputs killing the mutant, fuzzing stops and *MOTIF* proceeds with Step 4's post-processing activity.

In our experiments, we test mutants in parallel, by leveraging the multiple nodes of an HPC infrastructure [108]. This decision depends on the observation that leveraging parallelism (e.g., by relying on Cloud solutions) is a cost-effective choice in the case of complex CPS with many live mutants. Parallel testing of mutants is enabled by the fact that, in contrast to from MU2 [78], *MOTIF* does not test live mutants with the same inputs. Therefore, *MOTIF* can simply be deployed on different nodes and executed. However, we assess (RQ4) how MOTIF's effectiveness can be improved by reusing the mutant-killing inputs obtained in the different, parallel executions.

To account for randomness, we executed each approach (i.e., different *MOTIF* setups and *SEMuP*) ten times for each subject. For each mutant, we executed each approach for 10,000 seconds, which we determined, in a preliminary study, to be sufficient for *SEMuP* to maximize the percentage of killed mutants. Precisely, for fuzzing and hybrid fuzzing, we allocate 10,000 seconds of time budget and stop fuzzing once this is reached. But for *SEMuP*, we allocate 10,000 seconds to the symbolic execution process, which means that, after the timeout, if the mutant has not been killed yet, *SEMuP* still tries to generate test inputs using the path conditions traversed so far, which leads to an execution time for *SEMuP* that is slightly higher than others (around 650 seconds more).

5.3 RQ1 - Fuzzer configurations

5.3.1 Design

Since we measure effectiveness in terms of percentage of killed mutants, to address RQ1, we compare the percentage of mutants killed when different

[4] Guideline for SymCC: https://github.com/AFLplusplus/AFLplusplus/tree/ stable/custom_mutators/symcc

fuzzer configurations are selected; specifically, we compare the configuration parameters belonging to three distinct dimensions: compilers, sanitizers, and coverage metrics. For each configuration considered in our experiment, we execute MOTIF for at most 10,000 seconds for each mutant and track the time required to kill each mutant. We compare distinct approaches in terms of number of mutants killed (i.e., mutation score, *MS*) over time and evaluate the significance of the difference by relying on the Mann–Whitney U-test, a non-parametric test.

The first comparison focuses on the impact of the compiler choice on *MOTIF*'s effectiveness. We consider two compilers: `GCC` and `Clang`. `GCC` is the default compiler for most software projects in several domains (e.g., space). `Clang` is a recent compiler that is gaining popularity across industries and is particularly appealing for fuzzing because it integrates several optimization techniques leading to faster programs[5][6]. We select `Clang` version 14, since it supports all sanitizers and coverage options in AFL++, and `GCC` version 11.

The second comparison is among sanitizers. We consider four sanitizers supported by AFL++: `ASAN` [109] which detects array out-of-bound and invalid memory access, `UBSAN` [110] which detects undefined behavior such as arithmetic overflow, `LSAN` [111] which focuses on heap memory leaks, and `MSAN` [112] which focuses on uninitialized memory reads. We do not consider the two other sanitizers supported by AFL++, Control Flow Integrity SANitizer (CFISAN) and Thread SANitizer (TSAN), because they focus on problems specific for object-oriented and concurrent programs that are unlikely caused by code mutations (indeed, the mutation operators considered for our experiments don't aim at introducing multi-threading faults nor breaking control flows by altering pointer to functions). To perform our experiment, we compile our subjects (fuzzing drivers and SUT) multiple times, one for each sanitizer, with `Clang` (the best compiler based on our results, as discussed below), and execute *MOTIF* on each version. We compare the observed results (MS) with the ones obtained with a baseline consisting of the use of `Clang` without any sanitization options.

In the third comparison, we considered three *coverage metric options* supported by AFL++, which is motivated by the fact that, since fuzzers are driven by code coverage, coverage metrics affect the fitness of inputs. They are: (1) `LAF` [20], which splits complex comparison operations into multiple conditions[7], thus preserving inputs that exercise boundary cases differently than previous inputs; (2) `NGRAM` [113] which tracks the coverage of sequences of N edges, thus preserving inputs that exercise different sub-paths in a program, and context-

[5] AFL++ with LLVM: https://github.com/AFLplusplus/AFLplusplus/ blob/stable/instrumentation/README.llvm.md

[6] Clang-Features and Goals: https://clang.llvm.org/features.html

[7] LAF splits conditions with non-strict relational operators such as `if (a <= b)doA` or `if (a >= b)doB`) into compound conditions such as `if (a == b){ doA } else if (a > b){ doA }`). It also splits invocations of `strcmp`, to compare strings, into an appropriate set of nested if conditions. Last it splits `switch` blocks into nested if conditions performing comparisons by-by-byte.

Table 4: Results obtained by *MOTIF* with different fuzzer configurations. Each cell reports the average percentage of mutants killed over ten runs by each configuration after 10,000 seconds and p-values based on U-test against the baseline.

Experiment group	Options	*LIBU*	*ASN1lib*	*MLFS*	*ESAIL*	*S5*
Compilers (baseline: GCC)	`GCC`	48.31%	85.44%	35.52%	38.38%	83.03%
	`Clang`	49.68% (0.0009)	86.57% (0.0002)	38.43% (0.0002)	39.04% (0.0001)	82.73% (0.2641)
Sanitizers (baseline: Clang)	`ASAN`	49.22% (0.1081)	86.82% (0.0200)	38.29% (0.0019)	39.36% (0.0048)	81.52% (0.0007)
	`UBSAN`	51.35% (0.0003)	84.31% (0.0002)	23.78% (0.0002)	38.67% (0.0001)	82.42% (0.2781)
	`LSAN`	47.39% (0.0002)	85.78% (0.0003)	37.60% (0.0002)	38.62% (0.0002)	80.81% (0.0000)
	`MSAN`	45.56% (0.0002)	83.39% (0.0002)	34.33% (0.0002)	38.30% (0.0001)	6.06% (0.0000)
Coverage metrics (baseline: Clang)	`LAF`	49.79% (0.6158)	88.84% (0.0002)	40.09% (0.0002)	38.73% (0.0004)	82.53% (0.3006)
	`NGRAM`	50.85% (0.0016)	86.27% (0.0023)	38.27% (0.0045)	38.97% (0.3446)	83.64% (0.0005)
	`CTX`	50.89% (0.0030)	86.49% (0.3989)	38.29% (0.0070)	38.98% (0.5518)	83.74% (0.0001)

n.nn% and n.nn% : significantly outperformed the baseline with $p \leq 0.01$ and $p \leq 0.05$, respectively

n.nn% and n.nn% : significantly underperformed, compared to the baseline, with $p \leq 0.01$ and $p \leq 0.05$, respectively

sensitive coverage (`CTX`) [113] which combines edge coverage with information on call points, thus enabling the fuzzer to distinguish between the execution of the same code from different calling contexts. For `LAF`, we enabled the splitting of all the types of compound expressions supported by `LAF`, and for `NGRAM`, we set the parameter N to 2. To compile our subjects, we rely on `Clang`.

5.3.2 Results

Table 4 reports *MOTIF*'s performance, for each subject, when different fuzzer configurations are selected. Each row shows the average percentage of live mutants killed by *MOTIF* over ten runs after 10,000 seconds, for a given fuzzer configuration. p-values are shown in brackets, capturing the statistical significance of the difference between the selected configuration and the `Clang` baseline; for the compiler selection assessment, p-values simply capture the significance of the difference between the results obtained using the two available options. A green background highlights cases where the selected configuration significantly outperformed the baseline; a red background highlights the opposite. Cases not leading to any significant difference have a white background.

The compilers' comparison results show that `Clang` outperforms `GCC` by 1.37 percentage points (pp) in *LIBU*, 1.13 pp in *ASN1lib*, 2.91 pp in *MLFS*, and 0.65 pp in *ESAIL*. For each subject of these four subjects, the results obtained with the two compilers are significantly different at every minute. The better performance of `Clang` is likely due to the fact that the number of inputs generated by AFL++ with `Clang` is 4 to 20 times higher than the one observed with `GCC`. Indeed, search algorithms (i.e., AFL++) are more likely to explore a larger portion of the input space when they generate more inputs. The larger number of inputs generated with `Clang` can be explained by `Clang` generating executable programs (in our case, fuzz drivers) that are quicker to execute than `GCC` ones and therefore AFL++ can try more inputs with `Clang` than with `GCC`, for the same test budget. In contrast to the above, for *S5*, `Clang` performed worse than `GCC` by 0.30 pp but the difference is not statistically significant, enabling us to conclude that overall **`Clang` is the best choice for *MOTIF*.**

The sanitizer results in Table 4 show that *MOTIF* equipped with sanitizers tends to perform worse than the baseline, *MOTIF* without any sanitizer. `ASAN` led to the highest number of cases with improvements; indeed, it slightly outperformed the baseline by 0.25 pp ($p < 0.05$) in *ASN1lib* and by 0.33 pp ($p < 0.01$) in *ESAIL*. `UBSAN`, instead, led to the largest improvement, with a proportion of killed mutants higher than the baseline by 1.67 pp ($p < 0.01$) in *LIBU*; however, it performs worse than the baseline for most of the subjects.

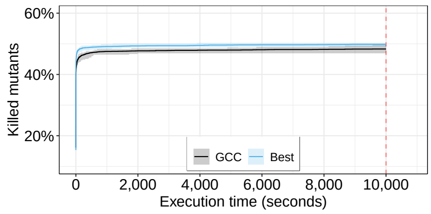
The primary reason why *MOTIF* with sanitizers cannot outperform the baseline is the overhead of the instrumented code introduced by sanitizers to detect invalid behaviors. Such overhead reduces the number of inputs generated by AFL++, consequently decreasing the number of mutants killed, even though sanitizers may kill some mutants not killed without them. In our investigation, the worst performance is obtained with `MSAN`, which led to 96.77% fewer inputs than the baseline, on average, and showed significantly lower performance. The worst `MSAN` result is obtained with *S5*, likely because of poor support for `C++` libraries (e.g., `MSAN` led to crashes whenever it encounters a standard library function such as `std::map` and `std::set`). `ASAN`, instead, led to 61.5% fewer inputs, on average, but enabled the detection of a large number of additional mutants thus compensating the loss due to execution cost; consequently, `ASAN` achieved results that are comparable to those obtained without sanitizer. `ASAN` effectiveness in killing mutants mainly depends on its capability to detect out-of-bound memory accesses caused by mutants; such mutants are harder to detect otherwise because the *sufficiency condition* (see Section 2.3) is hard to meet without a sanitizer (the effect of the out-of-bound access shall affect the values assigned to an output variable). Further, `ASAN` prevents the generation of test inputs that kill mutants but are not valid; those are inputs that violate function preconditions and cause out-of-bound accesses in the original function under test, and are thus discarded by *MOTIF*. Different from `ASAN`, `UBSAN` prevents the testing of several mutants because it detects invalid behaviors in the original function (on average, 653.8 fuzzing drivers failed when executing the original function), which need to be fixed before proceeding with mutation testing. A representative example is provided by *MLFS*, where `UBSAN` detects several arithmetic overflows caused by shift operators applied to variables that may have negative values. Although `UBSAN` may have detected potential bugs in the SUT, by terminating the execution of the original function, `UBSAN` prevents mutation testing. In summary, **`ASAN` is the only sanitizer that is beneficial to *MOTIF* for two of the subjects** (i.e., prevents false positives without largely affecting effectiveness).

Coverage metrics results in Table 4 show that their effectiveness vary across subjects. Specifically, *MOTIF* with `LAF` outperformed the baseline (i.e., *MOTIF* with `AFL++` using the default coverage options) by 2.27 pp and 1.66 pp, with p-value ≤ 0.01, in *ASN1lib* and *MLFS*, respectively. With these two subjects, `LAF` likely improves reachability (i.e., generating inputs that reach the mutated statement) because they present several conditional statements that can be split into simpler conditions; indeed, *ASN1lib* contains compound conditions that check input validity, while *MLFS*, relies on several signed and floating point comparisons. In *ESAIL*, `LAF` performed significantly worse than the baseline by 0.31 pp; however, the practical impact of such decrease in performance is very limited (2 fewer mutants killed, in the worst case).

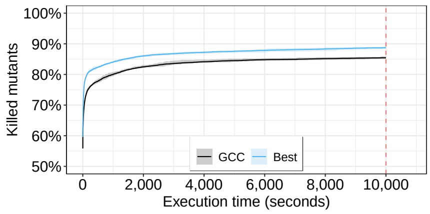
MOTIF with `NGRAM` and `CTX` outperformed the baseline in both *LIBU* and *S5*. In *LIBU*, we observe an improvement of 1.17 pp and 1.21 pp for `NGRAM` and `CTX`, respectively (p-value $\leq$ 0.01). In *S5*, the improvements are 0.91 pp and 1.01 pp, respectively (p-value $\leq$ 0.01). In *LIBU*, the improvements are likely due to *LIBU* including longer call sequences whose results depend on state variables. In *S5*, the improved performance is likely due to `C++` language features (i.e., overriding and polymorphism), resulting in method invocations that behave differently depending on the call context. In the worst case, `NGRAM` and `CTX`, decrease *MOTIF*'s performance by 0.30 pp (`NGRAM` on *ASN1lib*) and 0.14 pp (`CTX` on *MLFS*), with an average of 4 to 6 fewer mutants being killed, a small impact in practice. To summarize, **it is not possible to identify a coverage metric that works best with all the subjects; however, since the differences are limited, engineers may select the option that improves testing results in critical code units** (e.g., `LAF` because it improves MS in mathematical functions).

Although our results did not help identifying a sanitizer and coverage metric configuration that provides effectiveness improvements across all subjects, for the sake of simplifying the remaining experiments, we identify a configuration for *MOTIF* that is likely to provide improvements in most subjects. It consists of combining `Clang`, `ASAN`, and `LAF` and we call such configuration `Best`, . We selected `ASAN` because it performs similarly to the baseline in terms of percentage of killed mutants but also prevents the generation of inputs that are invalid and cause memory access errors. We selected `LAF` because it leads to the largest number of killed mutants, across all subjects (3283 versus 3186 for NGRAM and 3190 for CTX), and further leads to the largest effectiveness improvement (+2.27 pp), and also improves the subject with the most effective test suite (i.e., *MLFS*, which achieves MC/DC and 81.80% MS).

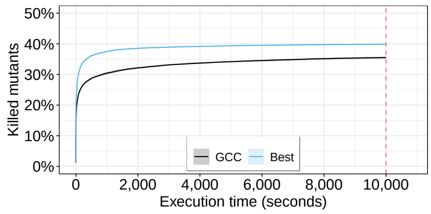
To ensure that the selected configuration options do not interfere, we performed ten additional executions of *MOTIF* configured with the `Best` settings, with each subject, and compare the observed results with *MOTIF* relying on the default `AFL++` options (i.e., relying on `GCC` without additional sanitizers or coverage metrics). Figure 3 shows the percentage of mutants killed by *MOTIF* after each second, when using `GCC` and `Best`, for each subject. Each line captures the average percentage observed in ten runs, with the shaded area capturing the upper and lower bounds across those runs. The vertical dashed line shows the time budget for the experiment. With `Best`, the percentages of killed mutants are 49.84% for *LIBU*, 88.74% for *ASN1lib*, 39.85% for *MLFS*, 39.50% for *ESAIL*, and 82.53% for *S5*. These performance results are significantly better than those of our baseline (i.e., *MOTIF* with `GCC`) for four subjects: it increases the percentage of killed mutants by 1.53 pp in *LIBU*,



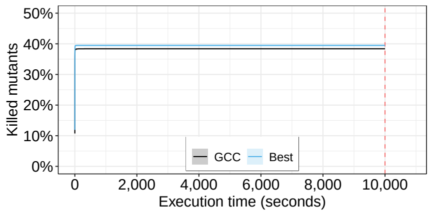
(a) *LIBU*



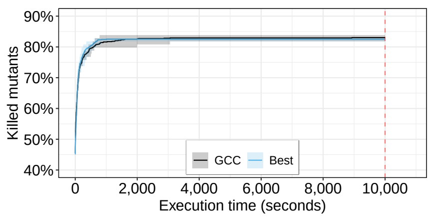
(b) *ASN1lib*



(c) *MLFS*



(d) *ESAIL*

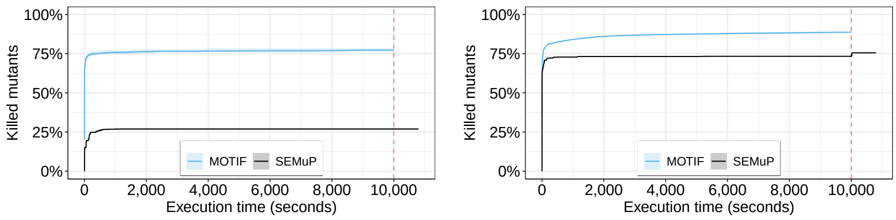


(e) *S5*

Fig. 3: Comparison between *MOTIF* with `GCC` and *MOTIF* with the `Best` fuzzer options.

3.30 pp in *ASN1lib*, 4.34 pp in *MLFS*, and 1.12 pp in *ESAIL*. For *S5*, the performance is similar to that of the baseline. Further, we manually inspected the mutants killed only by the `Best` configuration and observed that 70% of the generated test cases enable exercising boundary cases missed by the original test suite, which include access to first/last items of arrays and use of boundary values in conditional statement. We can thus conclude that `Best` helps improve the quality of test suites because in safety-critical systems it is desirable to exercise such situations. We can thus conclude that **combining `Clang`, `ASAN`, and `LAF` leads to the best results in *MOTIF***.

We leave the identification of solutions to overcome the mutation score plateau (see Figure 3) to future work; however, it could also be partially due to the presence of equivalent mutants.



(a) *LIBU* (b) *ASN1lib*

Fig. 4: Percentage of mutants killed by *MOTIF* and *SEMuP*.

5.4 RQ2 - Fuzzing vs Symbolic Execution

5.4.1 Design

We compare fuzzing and symbolic execution in terms of cost-effectiveness. The effectiveness of an automated mutation testing tool can be measured in terms of the proportion of live mutants killed. Its cost is determined by the time required to kill the mutants; indeed, lengthy test data generation may delay the testing process and increase the usage of computing resources. Though cost is also driven by the time required to manually inspect test outputs, *MOTIF* and *SEMuP* should require the same manual inspection time because they invoke the same functions under test and print out the same output values. Therefore, regarding cost, we focus on execution time and thus compare cost-effectiveness in terms of live mutants killed for different time budgets.

We could not consider *MLFS* to address RQ2 because it works mainly with floating point arguments, which are not supported by KLEE. An old version of KLEE addresses floating point variables but it is not integrated into *SEMu*. We could not consider *ESAIL* and *S5* because of the lack of KLEE support for multi-threading libraries and C++. We therefore focus on *LIBU* and *ASN1lib*; however, for *LIBU* we considered only four out of 27 source files, because all the other source files included I/O operations, which are not supported by KLEE/*SEMu*, or cannot be compiled into LLVM bitcode. This leads to 1,347 live mutants for *ASN1lib* and 153 for *LIBU*. For *MOTIF*, we considered the `Best` configuration identified in RQ1.

5.4.2 Results

Figure 4 depicts the percentage of live mutants killed by *MOTIF* and *SEMuP* for *LIBU* (4a) and *ASN1lib* (4b), respectively. Each line represents the average percentage from ten runs, with the shaded area around each line indicating the upper and lower bounds across those runs. The vertical dashed line means the time budget for each experiment. At that point, *SEMuP* stops exploring paths and generates inputs that satisfy the current path condition, which, sometimes, is sufficient to identify inputs that kill mutants. We observe a rapid increase in the number of mutants killed by *SEMuP* for the *ASN1lib*, which includes paths with several nested conditions.

The plots show that *MOTIF* outperforms *SEMuP*. After 10,000 seconds, *MOTIF* kills between 117 (76.47%) and 120 (78.43%) mutants for *LIBU* (avg. is 118.20, 77.25%) and between 1,190 (88.34%) and 1,200 (89.09%) for *ASN1lib* (avg. is 1,195.30, 88.74%). In contrast, *SEMuP* kills 41 (26.80%) to 42 (27.45%) mutants for *LIBU* (avg. is 41.2, 26.93%) and 1,017 (75.50%) to 1,018 (75.58%) for *ASN1lib* (avg. is 1,017.8, 75.56%). On average, across the ten runs, *MOTIF* kills a percentage of mutants that is 50.33 percentage points (pp) and 13.18 pp higher than *SEMuP*'s, for *LIBU* and *ASN1lib*, respectively.

The difference between *MOTIF* and *SEMuP* is significant at every timestamp, based on a U-test ($p < 0.01$). For example, after one minute, *MOTIF* kills, on average, 110.5 (*LIBU*) and 1,045.00 (*ASN1lib*) mutants, while *SEMuP* kills 29 (*LIBU*) and 924.6 (*ASN1lib*) mutants. For both *LIBU* and *ASN1lib*, *MOTIF* quickly reaches a near plateau; in *LIBU*, *MOTIF* reaches the plateau quicker because of *LIBU*'s simple control logic.

Though *MOTIF* outperforms *SEMuP*, they show some degree of complementarity, which justifies the integration of hybrid fuzzers in *MOTIF* (see Section 2.2). If we consider the best run of each approach, in the case of *ASN1lib*, *MOTIF* kills 257 (19.08%) mutants not killed by *SEMuP*, while *SEMuP* kills 75 (5.57%) mutants not killed by *MOTIF*. In the case of *LIBU*, *MOTIF* kills an additional 78 (50.98%) mutants on top of the mutants killed by *SEMuP*. We manually inspected some of the mutants and noticed that *SEMuP* is sometimes better at generating inputs that satisfy narrow, simple constraints. However, such a characteristic is more useful for *ASN1lib*, which mainly performs boundary checks for nested data structures, rather than the utility library. On the other hand, *MOTIF* is better when *SEMuP* fails to solve complex constraints. For example, for *LIBU*, *SEMuP* could not kill 52 mutants affecting a conditional statement with 24 bitwise operations, 44 mutants affecting a conditional statement with 13 conditions expressed using inequalities, and 5 mutants affecting the size of the buffer used in `snprintf` statements. Finally, *MOTIF* enabled the discovery of four bugs that were confirmed by developers: two concern missing checks for out-of-domain numerical inputs, the other two are an integer overflow affecting operations on a numerical data structure with multiple fields. We detected the missing checks by observing that the generated test cases include an out-of-domain input but do not result in an execution error flag being set (likely they were not detected by test cases because the valid domain is underspecified in specifications). The integer overflow was discovered by observing that the result of a sum of two large numeric items led to a lower number. The integer underflow can be noticed by observing that the difference between a zero-filled data instance and a data instance with small positive numbers result in large positive numbers. *SEMuP* discovered only three of them.

5.5 RQ3 - Seeding effectiveness

5.5.1 Design

To discuss how *MOTIF*'s seeds contribute to mutation testing results, we focus on the proportion of mutants killed with seed inputs in the experiments performed to address RQ1 with the *MOTIF*'s `Best` configuration.

5.5.2 Results

MOTIF's seed inputs contributed to killing mutants as follows: 22.2 (5.08%) for *LIBU*, 304 (22.57%) for *ASN1lib*, 81 (2.08%) for *MLFS*, 69 (11.88%) for *ESAIL*, and 39 (39.39%) for *S5*. However, although in all the subjects except *LIBU* seed inputs killed the same mutants across the ten runs, in *LIBU* the number of killed mutants varied between 22 and 23 because of a non-deterministic function that computes the time difference before and after invoking the `sleep()` function.

The percentage of mutants killed by seed inputs largely depends on the nature of the functions under test. For mutants in *MLFS* and *LIBU*, such percentages are low because they mainly alter mathematical operations whose mutants are killed with inputs satisfying complex constraints. For *ASN1lib*, *ESAIL*, and *S5*, the proportion of mutants killed is higher because the mutants modified lines that affect the output observed with any input value. Please note that seed inputs do not introduce bias in RQ2 results since *SEMuP* kills most of the mutants killed by seed inputs (267/304 for *ASN1lib* and 1/1 for *LIBU*).

Concluding, although the selected seed inputs help kill mutants, the contribution of the fuzzing process is significant with, at the very minimum (*S5*), 52.27% (i.e., $100\% - \frac{39.39\%/}{82.53\%}$) of the killed mutants being killed by fuzzing.

5.6 RQ4 - Applying Hybrid-fuzzing

5.6.1 Design

We compare *MOTIF* with *MOTIF-Hybrid*, our *MOTIF* extension integrating `AFL++` with `SymCC` to leverage hybrid fuzzing for mutation testing (see Section 5.2). To ensure a fair comparison, we applied the `Best` fuzzing configuration identified in RQ1 to both approaches. As for RQ2, we compare the two approaches in terms of cost-effectiveness, by reporting on the MS and the time taken to kill mutants.

We considered four subjects: *LIBU*, *ASN1lib*, *MLFS*, and *ESAIL*. We excluded *S5* because `SymCC` cannot successfully compile C++ code relying on the `STL` library. Also, since `SymCC` fails to correctly compile code chunks with increment and decrement operators for `boolean`, we excluded two *ESAIL* mutants. Finally, we executed the two approaches ten times on each subject for 10,000 seconds.

Table 5: Complementarity between *MOTIF-Hybrid*, *MOTIF*, and *SEMuP* in *ASN1lib*: number of mutants killed and not killed by the selected approaches, for their best execution.

		MOTIF-Hybrid		*SEMuP*	
		KILLED	LIVE	KILLED	LIVE
SEMuP	KILLED	955	63	-	-
	LIVE	267	62	-	-
MOTIF	KILLED	1188	12	943	257
	LIVE	34	113	75	72

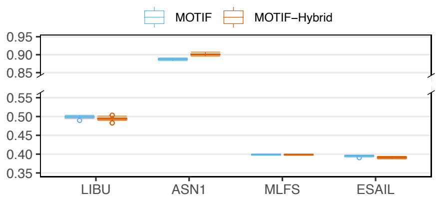
5.6.2 Results

Figure 5 presents boxplots providing the distributions of MS (5a) after 10,000 seconds and the time taken to kill mutants (5b), across the ten experimental runs.

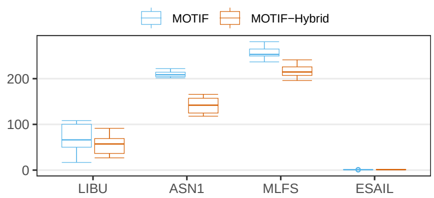
MOTIF-Hybrid increases the percentage of killed mutants in *ASN1lib* by up to 1.37 pp (18.5 mutants) on average, while the performance differences for the other subjects are not statistically significant.

As for the time taken to kill mutants, we report that the mutants in *ASN1lib* and *MLFS* were killed significantly faster with *MOTIF-Hybrid*. The median per mutant for *ASN1lib* and for *MLFS* is 142.00 vs. 208.66 seconds and 214.60 vs. 253.08 seconds, respectively, when compared with *MOTIF*. In *LIBU* and *ESAIL*, the difference is not significant, however, with the median observed for *MOTIF-Hybrid* with *LIBU* being lower (57.13 seconds with *MOTIF-Hybrid* vs. 66.13 seconds with *MOTIF*).

Additionally, we manually investigated the complementarity between *MOTIF-Hybrid*, *MOTIF*, and *SEMuP* in *ASN1lib*, which, in RQ2, showed that *MOTIF* and *SEMuP* are complementary. Table 5 shows the number of killed and live mutants observed in the best run of each approach. *MOTIF-Hybrid* killed 267 mutants (19.82%) that were not killed by *SEMuP*. However, *MO-*



(a) Mutation score



(b) Kill time (seconds)

Fig. 5: Results observed when applying *MOTIF* with `AFL++` (MOTIF) and *MOTIF* with `SymCC` (MOTIF-*Hybrid*).

Table 6: Comparison *MOTIF* with `Best` set of fuzzing options and results after reusing inputs for five selected subjects

Subject	Target mutants	# of killed mutants (Avg. of 10 runs)			# of inputs for functions (10 runs)			Reuse time (50 runs, seconds)			Kill time (50 runs, seconds)		
		FUZZED	REUSED	DIFF	Mean	Min	Max	Mean	Min	Max	Mean	Min	Max
LIBU	325	61.78%	62.71%	0.92 pp (0.0048)	299.49	1	3166	5.72	0.01	264.84	0.21	0.02	5.02
ASN1lib	762	80.09%	83.88%	3.79 pp (0.0002)	143.69	4	779	9.13	0.10	854.79	0.12	0.01	1.06
MLFS	3623	42.50%	43.57%	1.08 pp (0.0002)	125.65	1	1267	12.51	0.01	353.51	0.11	0.01	4.14
ESAIL	418	54.90%	55.02%	0.12 pp (0.2515)	246.78	1	1398	484.40	0.23	835.10	0.24	0.14	1.06
S5	28	88.21%	89.29%	1.07 pp (0.0767)	11.33	7	28	1.76	0.00	6.74	0.10	0.08	0.15

n.nn pp% : outperformed the baseline with $p \leq 0.01$,

TIF-Hybrid killed only 12 of the 75 mutants killed by *SEMuP* but not by *MOTIF*; the fact that `SymCC` cannot kill all the mutants killed by *SEMuP* shows that `SymCC` presents some limitations compared to *SEMuP* (e.g., it may need to augment the set of inputs executed symbolically). However, if we subtract the mutants missed by *MOTIF-Hybrid* from the additional mutants killed by *MOTIF-Hybrid* compared to *SEMuP* and *MOTIF*, we observe that *MOTIF-Hybrid* still kills 204 more mutants than *SEMuP* and 22 more mutants than *MOTIF*, thus remaining the most effective choice.

Concluding, ***MOTIF-Hybrid*** **is the best approach for mutation testing since it is faster at killing mutants and has a higher MS in half of the subjects, while getting an equivalent MS with the others, when technical limitations do not prevent its adoption (e.g., in `C++` projects like *S5*).** ***MOTIF-Hybrid*** **could therefore be a practical solution for large projects resulting in a significant number of mutants.** However, the setup required for *MOTIF-Hybrid* is more complex than the one required for *MOTIF*; indeed, it is necessary to set up additional configurations for the AFL fuzzer and to compile the fuzzing drivers twice, with the AFL compiler and with SYM-CC. Consequently, *MOTIF-Hybrid* is less likely to be adopted in practice, in favor of *MOTIF*.

5.7 RQ5 - Reusing inputs

5.7.1 Design

RQ5 aims to assess the tradeoff between mutation score improvement and increased testing time observed when reusing inputs killing mutants to kill other mutants not successfully killed by *MOTIF*. For each mutated function with at least one mutant killed by *MOTIF*, we rely on the mutant-killing inputs generated by *MOTIF* to test the mutants not successfully killed by *MOTIF*. Precisely, we collect the fuzzed files that kill mutants in a function and, after excluding duplicates, provide them as inputs to the fuzzing driver for a live mutant of the same function. We also apply the *MOTIF* post-processing step to avoid false positives (see Step 4 in Section 3.4).

We considered all the five subjects used for RQ1, and the mutant-killing inputs obtained with the `Best` configuration. To account for randomness, we apply the approach to the results of all the ten runs considered for RQ1.

To discuss effectiveness, we compare the MS obtained by reusing inputs with the MS obtained by *MOTIF* after 10,000 seconds.

As for cost, we measure, for each mutant, the time taken to execute fuzzing drivers with all inputs and the time taken until the mutant is killed. Since the order of inputs may affect execution time, we repeat our experiment 50 times, after shuffling each input set.

5.7.2 Results

Table 6 presents our results. Column `target mutants` reports the number of mutants that belong to the functions that have at least one killed mutant. Column `FUZZED` reports the MS obtained by *MOTIF* on average over 10 runs. Column `REUSED` reports the MS obtained after reusing inputs to kill additional mutants. Column `DIFF` shows the differences between `FUZZED` and `REUSED`, and reports on the significance of the difference (we highlight significant improvements). The next three sets of columns report statistics (mean, min, and max) on the number of inputs available for reuse across target functions, the time required to test each mutant with the reused inputs, and the time taken to kill a mutant.

The experiment results indicate that MS can be improved by up to 3.79 pp when reusing inputs, with significant improvements in most of the subjects, which indicates that input reuse is beneficial for *MOTIF*. The time taken for reusing inputs varies, taking up to 15 minutes (900 seconds) depending on several factors, including the number of inputs, software size, and the type of mutants. For instance, *ESAIL* has approximately 70% of its mutants within a single function, which leads to a large number of additional executions when reusing inputs. In the case of *S5*, the larger size causes a longer execution time for fuzzing drivers compared to other subjects, but the maximum execution time is lower due to the number of inputs collected. The execution time is also affected by the type of mutants. Specifically, some mutants took longer than others because they modified stopping conditions in program loops.

Since, in the worst case, input reuse takes 855 seconds, and given that Figure 3 shows that after 9000 seconds *MOTIF* has already reached the plateau, we can conclude that with a test budget of 10,000 seconds per mutant, it is convenient to rely on input reuse. For lower test budgets input reuse may provide benefits only for a subset of subjects. However, based on our results, up to 5000 seconds of test budget, input reuse is likely to be beneficial; indeed, in all our subjects, after 4150 seconds, *MOTIF* has killed at most 1 pp mutants less than at 10,000 seconds. Concluding, **we suggest executing *MOTIF* with a budget of 5000 seconds per mutant, and dedicate the last 900 seconds to input reuse**.

5.8 Threats to validity

To address threats to internal validity, we manually verified that *MOTIF*, *SEMuP*, and *MOTIF-Hybrid* correctly execute; and further, we manually inspected a large subset of the generated test cases and mutants killed in our experiments. Further, our false positive driver ensures that *MOTIF* results are not affected by the presence of global variables or, more generally, non-determinism. Although we do not reset global state variables in fuzzing drivers, note that across all experiment runs, out of 32,875 mutants reported as killed by the fuzzing driver, only 125 were false positives (0.38%), thus showing that non-determinism does not undermine the applicability of *MOTIF*.

Though our results may depend on the specific fuzzer used in our experiments, AFL++ is one of the best performing grey-box fuzzers according to recent benchmarks (see Section 5.2). Further, we assessed *MOTIF* with a state-of-the art hybrid fuzzing solution. Although alternative hybrid fuzzers were not applicable to some of our subjects (see Section 5.2), they might have led to different and potentially better results; we leave their investigation to future work.

To address generalizability issues, we selected diverse software subjects that are installed and running on space CPS, including satellites currently in orbit: a mathematical library, a utility library, a data serialization component, onboard control software, and ground software. Since they implement a diverse set of features (mathematical operations, serialization, string and time utilities), they strengthen the generalizability of our results. Further, these types of software components are typical in many CPS systems including avionics, robotics, and automotive, thus suggesting the proposed approach may be useful in many sectors other than space.

6 Conclusion

We propose *MOTIF*, an approach that leverages fuzzing to automatically generate test data for mutation testing of embedded software deployed in cyber-physical systems (CPS). It aims to overcome the limitations of SOTA approaches, which rely on symbolic execution and cannot easily be applied in many contexts, especially CPS ones.

MOTIF is implemented through a pipeline that generates a test driver that processes the input data generated by the fuzzer, provides such input data to the original and mutated versions of a function under test, and determines when the outputs generated by the two functions differ (i.e., the mutant is killed). By monitoring the coverage achieved when executing the original and mutated functions, the fuzzer identifies inputs leading to different behaviors across these functions and, consequently, is driven towards the identification of inputs that kill the mutant.

We performed an empirical evaluation with embedded software deployed on satellites currently in orbit. We empirically determined the fuzzer configurations leading to the best mutation testing results, which consists of relying on the Clang compiler with address sanitization and LAF coverage optimization. In our subjects, such configuration enables killing between 40% and 83% of the mutants. Further, although our seeding strategy contributes to quickly killing mutants, most of the mutants (between 60% and 97%) are killed thanks to the fuzz testing process. We compared *MOTIF* with a SOTA approach based on symbolic execution, which showed that the percentage of mutants killed by *MOTIF* is higher than the SOTA approach by 13 and 50 percentage points in our two case studies where symbolic execution is applicable. Our results therefore clearly show that fuzzing should be adopted as the preferred method to use to perform mutation testing. However, we also demonstrated that hybrid-fuzzing, which integrates fuzzing and symbolic execution, leads to slightly increasing the percentage of killed mutants (up to 1.37 pp).

Acknowledgment

The experiments presented in this paper were carried out using the HPC facilities of the University of Luxembourg (see `http://hpc.uni.lu`). This research was supported by ESA via a GSTP element contract (RFQ/3-17554/21/NL/AS/kkIMPROVE) and by the NSERC Discovery and Canada Research Chair programs. The authors would like to thank Oscar Cornejo and Enrico Viganò for having contributed to the preliminary versions of *MOTIF*.

References

1. ESA, "ECSS-E-ST-40C - Software general requirements." 2009. [Online]. Available: http://ecss.nl/standard/ecss-e-st-40c-software-general-requirements/
2. L. Buffoni, L. Ochel, A. Pop, P. Fritzson, N. Fors, G. Hedin, W. Taha, and M. Sjölund, "Open source languages and methods for cyber-physical system development: Overview and case studies," *Electronics*, vol. 10, no. 8, p. 902, 2021. [Online]. Available: https://www.mdpi.com/2079-9292/10/8/902
3. 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*, 2019.
4. 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.
5. 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.
6. O. E. Cornejo Olivares, F. Pastore, and L. Briand, "Mutation Analysis for Cyber-Physical Systems: Scalable Solutions and Results in the Space Domain," *IEEE Transactions on Software Engineering*, vol. 48, no. 10, pp. 3913–3939, 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3107680
7. T. T. Chekam, M. Papadakis, M. Cordy, and Y. L. Traon, "Killing stubborn mutants with symbolic execution," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 30, no. 2, Jan. 2021.

8. C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08, vol. 8. USA: USENIX Association, 2008, p. 209–224.
9. G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 278–292, 2011.
10. Cobham Gaisler, "RTEMS Cross Compilation System," https://www.gaisler.com/index.php/products/operating-systems/rtems, 2021.
11. V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.
12. M. Kim, Q. Xin, S. Sinha, and A. Orso, "Automated test generation for REST APIs: no time to rest yet," in *ISSTA '22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, S. Ryu and Y. Smaragdakis, Eds. ACM, 2022, pp. 289–301. [Online]. Available: https://doi.org/10.1145/3533767.3534401
13. K. Ispoglou, D. Austin, V. Mohan, and M. Payer, "FuzzGen: Automatic fuzzer generation," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2271–2287. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/ispoglou
14. D. Babić, S. Bucur, Y. Chen, F. Ivančić, T. King, M. Kusano, C. Lemieux, L. Szekeres, and W. Wang, "Fudge: Fuzz driver generation at scale," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 975–985. [Online]. Available: https://doi.org/10.1145/3338906.3340456
15. C. Zhang, X. Lin, Y. Li, Y. Xue, J. Xie, H. Chen, X. Ying, J. Wang, and Y. Liu, "APICraft: Fuzz driver generation for closed-source SDK libraries," in *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 2811–2828. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/zhang-cen
16. B. Jeong, J. Jang, H. Yi, J. Moon, J. Kim, I. Jeon, T. Kim, W. Shim, and Y. H. Hwang, "UTopia: Automatic Generation of Fuzz Driver using Unit Tests," in *Proceedings - IEEE Symposium on Security and Privacy*, vol. 2023-May, 2023, pp. 2676–2692.
17. P. Chen, Y. Xie, Y. Lyu, Y. Wang, and H. Chen, "Hopper: Interpretative fuzzing for libraries," in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '23. New York, NY, USA: Association for Computing Machinery, 2023, p. 1600–1614. [Online]. Available: https://doi-org.proxy.bnl.lu/10.1145/3576915.3616610
18. J. Lee, E. Vigano, O. Cornejo, F. Pastore, and L. Briand, "Fuzzing for cps mutation testing," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Los Alamitos, CA, USA: IEEE Computer Society, sep 2023, pp. 1377–1389. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ASE56229.2023.00079
19. D. Zhang, A. Fioraldi, and D. Balzarotti, "On understanding and forecasting fuzzers performance with static analysis," in *CCS 2024, 31st ACM conference on Computer and Communications Security, 14-18 October 2024, Salt Lake City, UT, USA*, ACM, Ed., Salt Lake City, 2024.
20. "Laf-intel," https://lafintel.wordpress.com/, accessed: 2024-02-28.
21. X. Zhu and M. Böhme, "Regression greybox fuzzing," in *Proceedings of the 28th ACM Conference on Computer and Communications Security*, ser. CCS, 2021.
22. F. Free Software Foundation, "GCC, the GNU Compiler Collection," https://gcc.gnu.org/, 2021.
23. European Space Agency, "ESAIL microsatellite," 2021. [Online]. Available: https://www.esa.int/Applications/Telecommunications_Integrated_Applications/ESAIL_maritime_satellite_ready_for_launch
24. G. Soltana, M. Sabetzadeh, and L. C. Briand, "ESA sentinel missions," 2021. [Online]. Available: https://sentinel.esa.int/web/sentinel/home

25. "FAQAS project," https://faqas.uni.lu, 2023.
26. J. Lee, E. Viganò, O. Cornejo, F. Pastore, and L. Briand, "MOTIF toolset," https://github.com/SNTSVV/MOTIF, 2024.
27. J. Lee, F. Pastore, and L. Briand, "Replication package," https://figshare.com/s/5a9a1fa723c374f5d0fd, 2024.
28. S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, P. McMinn, A. Bertolino, J. Jenny Li, and H. Zhu, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software*, vol. 86, no. 8, pp. 1978–2001, 2013.
29. L. Bordeaux, Y. Hamadi, and L. Zhang, "Propositional satisfiability and constraint programming: A comparative survey," *ACM Comput. Surv.*, vol. 38, no. 4, Dec. 2006. [Online]. Available: https://doi.org/10.1145/1177352.1177354
30. Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," in *IEEE Symposium on Security and Privacy*, 2016.
31. V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: A platform for in-vivo multi-path analysis of software systems," in *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVI. New York, NY, USA: Association for Computing Machinery, 2011, p. 265–278. [Online]. Available: https://doi.org/10.1145/1950365.1950396
32. S. Poeplau and A. Francillon, "Symbolic execution with SymCC: Don't interpret, compile!" in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 181–198.
33. I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in *27th {USENIX} Security Symposium*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761.
34. S. Poeplau and A. Francillon, "SymQEMU: Compilation-based symbolic execution for binaries," in *Network and Distributed System Security Symposium*. Network & Distributed System Security Symposium, February 2021.
35. J. Chen, W. Han, M. Yin, H. Zeng, C. Song, B. Lee, H. Yin, and I. Shin, "SYMSAN: Time and space efficient concolic execution via dynamic dataflow analysis," in *31st USENIX Security Symposium (USENIX Security 22)*. Boston, MA: USENIX Association, Aug. 2022, pp. 2531–2548. [Online]. Available: https://www.usenix.org/conference/usenixsecurity22/presentation/chen-ju
36. J. Metzman, L. Szekeres, L. Maurice Romain Simon, R. Trevelin Sprabery, and A. Arya, "FuzzBench: An Open Fuzzer Benchmarking Platform and Service," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1393–1403. [Online]. Available: https://doi.org/10.1145/3468264.3473932
37. D. Asprone, J. Metzman, A. Arya, G. Guizzo, and F. Sarro, "Comparing fuzzers on a level playing field with fuzzbench," in *2022 IEEE Conference on Software Testing, Verification and Validation (ICST)*. Los Alamitos, CA, USA: IEEE Computer Society, apr 2022, pp. 302–311. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/ICST53961.2022.00039
38. CLANG, "Undefined Behavior Sanitizer," http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#ubsan-checks, 2020.
39. M. Zalewski, "How AFL works," 2022. [Online]. Available: https://afl-1.readthedocs.io/en/latest/about_afl.html#how-afl-works
40. P. Ammann and J. Offutt, *Introduction to software testing.* Cambridge University Press, 2016.
41. C. Lyu, S. Ji, C. Zhang, Y. Li, W.-H. Lee, Y. Song, and R. Beyah, "MOPT: Optimized mutation scheduling for fuzzers," in *28th USENIX Security Symposium (USENIX Security 19)*. Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1949–1966. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/lyu
42. M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-Based Greybox Fuzzing as Markov Chain," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer

and Communications Security, ser. CCS '16. New York, NY, USA: Association for Computing Machinery, 2016, p. 1032–1043.
43. A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "AFL++ : Combining incremental steps of fuzzing research," in *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.
44. P. Wang, X. Zhou, T. Yue, P. Lin, Y. Liu, and K. Lu, "The progress, challenges, and perspectives of directed greybox fuzzing," *Software Testing, Verification and Reliability*, vol. 34, no. 2, p. e1869, 2024. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1869
45. R. Majumdar and K. Sen, "Hybrid concolic testing," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 416–426.
46. N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting Fuzzing Through Selective Symbolic Execution Nick," *Network and Distributed System Security Symposium*, no. February, pp. 21–24, 2016.
47. M. Zalewski, "American Fuzzy Lop: a security-oriented fuzzer," 2020. [Online]. Available: http://lcamtuf.coredump.cx/afl/
48. L. Borzacchiello, E. Coppa, and C. Demetrescu, "FUZZOLIC: mixing fuzzing and concolic execution," *Computers & Security*, 2021.
49. ——, "Fuzzing symbolic expressions," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 711–722.
50. J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, "Software testing with large language models: Survey, landscape, and vision," *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, 2024.
51. S. Y. Shin, F. Pastore, D. Bianculli, and A. Baicoianu, "Towards generating executable metamorphic relations using large language models," 2024. [Online]. Available: https://arxiv.org/abs/2401.17019
52. A. Fan, B. Gokkaya, M. Harman, M. Lyubarskiy, S. Sengupta, S. Yoo, and J. M. Zhang, "Large language models for software engineering: Survey and open problems," in *2023 IEEE/ACM International Conference on Software Engineering: Future of Software Engineering (ICSE-FoSE)*, 2023, pp. 31–53.
53. C. Zhang, M. Bai, Y. Zheng, Y. Li, X. Xie, Y. Li, W. Ma, L. Sun, and Y. Liu, "Understanding Large Language Model Based Fuzz Driver Generation," in *Proceedings of the 33rd International Symposium on Software Testing and Analysis (ISSTA)*. Association for Computing Machinery, 2024. [Online]. Available: http://arxiv.org/abs/2307.12469
54. G. O.-F. Team, "Oss-fuzz-gen: Llm powered fuzzing via oss-fuzz," https://github.com/google/oss-fuzz-gen, 2024, accessed: 2024-08-28.
55. Y. Lyu, Y. Xie, P. Chen, and H. Chen, "Prompt Fuzzing for Fuzz Driver Generation," in *Proceedings of the 31st ACM Conference on Computer and Communications Security (CCS)*. Association for Computing Machinery, 2024. [Online]. Available: http://arxiv.org/abs/2312.17677
56. X. Liu, W. You, Z. Zhang, and X. Zhang, "Tensilefuzz: Facilitating seed input generation in fuzzing via string constraint solving," in *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 391–403. [Online]. Available: https://doi.org/10.1145/3533767.3534403
57. J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *2017 IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 579–594.
58. 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.
59. D. B. Brown, *Mutation Testing: Algorithms and Applications.* The University of Wisconsin-Madison, 2020.
60. 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.

61. 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.
62. T. T. Chekam, "SEMu: Symbolic Execution-based Mutant Analysis Framework," https://github.com/thierry-tct/KLEE-SEMu, 2023.
63. K. Ayari, S. Bouktif, and G. Antoniol, "Automatic mutation test input data generation via ant colony," in *Proceedings of the 9th annual conference on Genetic and evolutionary computation*, 2007, pp. 1074–1081.
64. M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE computational intelligence magazine*, vol. 1, no. 4, pp. 28–39, 2006.
65. G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ser. ESEC/FSE '11. New York, NY, USA: ACM, 2011, pp. 416–419. [Online]. Available: http://doi.acm.org/10.1145/2025113.2025179
66. ——, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, no. 3, pp. 783–812, 2015.
67. H. Almulla and G. Gay, "Learning how to search: generating effective test cases through adaptive fitness function selection," *Empirical Software Engineering*, vol. 27, no. 2, p. 38, 2022. [Online]. Available: https://doi.org/10.1007/s10664-021-10048-8
68. F. C. M. Souza, M. Papadakis, Y. Le Traon, and M. E. Delamaro, "Strong mutation-based test data generation using hill climbing," in *Proceedings of the 9th International Workshop on Search-Based Software Testing*, 2016, pp. 45–54.
69. B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.
70. K. Lakhotia, M. Harman, and H. Gross, "Austin: A tool for search based software testing for the c language and its evaluation on deployed automotive systems," in *2nd International symposium on search based software engineering*. IEEE, 2010, pp. 101–110.
71. ——, "Austin: An open source tool for search based software testing of c programs," *Information and Software Technology*, vol. 55, no. 1, pp. 112–125, 2013, special section: Best papers from the 2nd International Symposium on Search Based Software Engineering 2010.
72. K. Lakhotia, "Honggfuzz," https://github.com/kiranlak/austin-sbst, 2022.
73. S. Scalabrino, G. Grano, D. Di Nucci, M. Guerra, A. De Lucia, H. C. Gall, and R. Oliveto, "Ocelot: A search-based test-data generation tool for c," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 868–871. [Online]. Available: https://doi.org/10.1145/3238147.3240477
74. X. Dang, X. Yao, D. Gong, and T. Tian, "Efficiently generating test data to kill stubborn mutants by dynamically reducing the search domain," *IEEE Transactions on Reliability*, vol. 69, no. 1, pp. 334–348, 2019.
75. Z. Wang, B. Liblit, and T. Reps, "Tofu: Target-oriented fuzzer," *arXiv preprint arXiv:2004.14375*, 2020.
76. European Space Agency, "Space," 2021. [Online]. Available: https://sir.csc.ncsu.edu/portal/bios/space.php
77. S. Nilizadeh, Y. Noller, and C. S. Pasareanu, "Diffuzz: Differential fuzzing for side-channel analysis," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, 2019, pp. 176–187.
78. V. Vikram, I. Laybourn, A. Li, N. Nair, K. OBrien, R. Sanna, and R. Padhye, "Guiding greybox fuzzing with mutation testing," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. New York, NY, USA: ACM, 2023, p. 929–941. [Online]. Available: https://doi.org/10.1145/3597926.3598107
79. G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Softw. Engg.*, vol. 20, no. 3, p. 783–812, jun 2015.
80. R. Qian, Q. Zhang, C. Fang, and L. Guo, "Investigating coverage guided fuzzing with mutation testing," in *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, ser. Internetware '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 272–281. [Online]. Available: https://doi.org/10.1145/3545258.3545285

81. R. Just, G. Fraser, M. Ivanković, and G. Petrovic, "Practical mutation testing at scale: A view from google," *IEEE Transactions on Software Engineering*, vol. 47, pp. 2780–2795, 2021.
82. D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *STVR*, 2013.
83. 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 (ISSTA'18)*, 2009.
84. M. E. Delamaro, J. Maidonado, and A. P. Mathur, "Interface mutation: An approach for integration testing," *IEEE transactions on software engineering*, vol. 27, no. 3, pp. 228–247, 2001.
85. LLVM project, "Clang library," https://clang.llvm.org/, 2023.
86. ——, "The LLVM compiler infrastructure project," https://llvm.org/, 2023.
87. LLVM, "LLVM documentation - libfuzzer – a library for coverage-guided fuzz testing." https://llvm.org/docs/LibFuzzer.html, 2022.
88. A. Sălcianu and M. Rinard, "Purity and side effect analysis for java programs," in *Verification, Model Checking, and Abstract Interpretation (VMCAI 2005)*, ser. Lecture Notes in Computer Science, vol. 3385. Springer, 2005, pp. 199–215. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-30579-8_14
89. A. Arcuri, G. Fraser, and R. Just, "Private api access and functional mocking in automated unit test generation," in *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, March 13–17 2017, pp. 126–137. [Online]. Available: https://homes.cs.washington.edu/~rjust/publ/mocking_reflection_testing_icst_2017.pdf
90. D. Shin, S. Yoo, and D.-H. Bae, "A theoretical and empirical study of diversity-aware mutation adequacy criterion," *IEEE TSE*, 2017.
91. European Space Agency, "MLFS - mathematical library for space software," 2021. [Online]. Available: https://essr.esa.int/project/mlfs-mathematical-library-for-flight-software
92. ESA, "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/
93. Semantix and Neuropublic, "ASN.1 certified compiler," https://github.com/ttsiodras/asn1scc, 2021.
94. G. Mamais, T. Tsiodras, D. Lesens, and M. Perrotin, "An ASN.1 compiler for embedded/space systems," in *Embedded Real Time Software and Systems (ERTS2012)*, Toulouse, France, Feb. 2012. [Online]. Available: https://hal.science/hal-02263447
95. O. Cornejo, F. Pastore, and L. Briand, "Mass: A tool for mutation analysis of space cps," in *2022 IEEE/ACM 44th International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, 2022, pp. 134–138.
96. 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.
97. 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.
98. 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.
99. 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.
100. 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.
101. 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.

102. 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.
103. J. Lee, E. Vigano, F. Pastore, and L. Briand, "Motif: A tool for mutation testing with fuzzing," in *17th IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2024. [Online]. Available: https://orbilu.uni.lu/handle/10993/61840
104. Fuzzbench Team, "FuzzBench: 2024-08-03-test report," https://www.fuzzbench.com/reports/2024-08-10-test/index.html, 2024.
105. J. Ounjai, V. Wüstholz, and M. Christakis, "Green fuzzer benchmarking," in *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ser. ISSTA 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 1396–1406. [Online]. Available: https://doi.org/10.1145/3597926.3598144
106. Google, "Honggfuzz," https://github.com/google/honggfuzz, 2022.
107. J. Hu, Y. Duan, and H. Yin, "Marco: A stochastic asynchronous concolic explorer," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623301
108. S. Varrette, P. Bouvry, H. Cartiaux, and F. Georgatos, "Management of an academic HPC cluster: The UL experience," in *Proceedings of the 2014 International Conference on High Performance Computing & Simulation (HPCS'14)*. IEEE, 2014, pp. 959–967.
109. K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov, "Addresssanitizer: A fast address sanity checker," in *Proceedings of the USENIX ATC 2012*, 2012. [Online]. Available: https://www.usenix.org/conference/usenixfederatedconferencesweek/addresssanitizer-fast-address-sanity-checker
110. "Clang 19.0.0git documentation: Undefined bahavior sanitizer," https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html, accessed: 2024-02-28.
111. "Clang 19.0.0git documentation: Leak sanitizer," https://clang.llvm.org/docs/LeakSanitizer.html, accessed: 2024-02-28.
112. E. Stepanov and K. Serebryany, "Memorysanitizer: Fast detector of uninitialized memory use in c++," in *Proceedings of the 2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2015, pp. 46–55.
113. J. Wang, Y. Duan, W. Song, H. Yin, and C. Song, "Be sensitive and collaborative: Analyzing impact of coverage metrics in greybox fuzzing," *Proceedings of the 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID2019)*, pp. 1–15, 2019.