

Fuzzing for CPS Mutation Testing

Jaekwon Lee^{*,†}, Enrico Viganò^{*}, Oscar Cornejo^{*}, Fabrizio Pastore^{*}, Lionel Briand^{*,†}

^{*}University of Luxembourg, [†]University of Ottawa
Luxembourg, LU; Ottawa, CA

{jaekwon.lee,enrico.vigano,oscar.cornejo,fabrizio.pastore,lionel.briand}@uni.lu

Abstract—Mutation testing can help reduce the risks of releasing faulty software. For such reason, it is a desired practice for the development of embedded software running in safety-critical cyber-physical systems (CPS). Unfortunately, state-of-the-art test data generation techniques for mutation testing of C and C++ software, two typical languages for CPS software, rely on symbolic execution, whose limitations often prevent its application (e.g., it cannot test black-box components).

We propose a mutation testing approach that leverages fuzz testing, which has proved effective with C and C++ software. Fuzz testing automatically generates diverse test inputs that exercise program branches in a varied number of ways and, therefore, exercise statements in different program states, thus maximizing the likelihood of killing mutants, our objective.

We performed an empirical assessment of our approach with software components used in satellite systems currently in orbit. Our empirical evaluation shows that mutation testing based on fuzz testing kills a significantly higher proportion of live mutants than symbolic execution (i.e., up to an additional 47 percentage points). Further, when symbolic execution cannot be applied, fuzz testing provides significant benefits (i.e., up to 41% mutants killed). Our study is the first one comparing fuzz testing and symbolic execution for mutation testing; our results provide guidance towards the development of fuzz testing tools dedicated to mutation testing.

Index Terms—Mutation testing, Fuzzing, Test data generation

I. INTRODUCTION

Software testing plays a key role in verifying and validating embedded software for cyber-physical systems (CPS). Ensuring the high-quality of test suites is therefore essential for quality assurance purposes in such contexts.

Mutation analysis is an effective approach to assess the quality of a test suite. Indeed, it entails measuring the mutation score, which is the proportion of programs with artificially injected faults (i.e., mutants) detected by the test suite [1], and there exists a strong association between a high mutation score and a high fault revealing capability for test suites [2], [3]. Further, recent work has shown that mutation analysis can be cost-effectively applied to large CPS software by combining multiple optimizations [4].

In practice, mutation analysis warrants the effective selection of inputs for mutation testing since test cases are required to detect all or a large proportion of the generated mutants. A mutant detected by a test suite is said to be killed. However, due to the typically high number of mutants generated in the context of large CPS projects [4], it is challenging for engineers to perform mutation testing manually.

Unfortunately, we lack automated test data generation techniques (*automated mutation testing* techniques) applicable to CPS; indeed, most of the existing techniques do not target the C and C++ languages, which are widely used in CPS domains.

The state-of-the-art (SOTA) solution for the automated mutation testing of C software (i.e., *SEMu* [5]) is based on the KLEE symbolic execution engine [6]. Though it has shown to be effective with command line utilities, it inherits the limitations of symbolic execution. Specifically, it requires modeling of the environment (e.g., network communication) and cannot deal with programs that require complex analyses to enable input generation (e.g., programs with floating point instructions). Further, it generates test inputs for command line utilities, which are seldom used in CPS, and does not generate unit test cases nor target other CPS interfaces. Search-based techniques developed for other programming languages (e.g., Java) [7] are impractical for C and C++ software because of the difficulty of instrumenting the software to compute dedicated fitness functions (e.g., branch distance). For example, to compute branch distance at runtime, it is necessary to modify all the conditional statements in the software under test (SUT) and, for that, the source code must be processed with static analysis tools that require loading all the dependencies. Unfortunately, for large systems, this often leads to configuring such tools to process several source files in nested directories, which is impractical except if the tool is well integrated with the compiler already in use for the SUT. Moreover, CPS source files often rely on architecture-specific C constructs (e.g., for the RTEMS compiler [8]) that are not successfully parsed by static analysis frameworks [4].

In this paper, we propose relying on gray-box fuzz testing techniques [9], also called fuzzing techniques or fuzzers, to generate test data for mutation testing. Grey-box fuzzers apply evolutionary algorithms to data (called seed data), which is either randomly generated or user-provided, to generate test input data (usually input files) that maximize code coverage and trigger failures. In contrast to other search-based testing techniques [10], they do not rely on branch distance [11] but instead make use of coarser heuristics like bucket-based branch coverage, which analyzes if an input exercises a branch for a number of times not observed before (see Section II-B); such heuristics can be implemented with simple extensions of standard C/C++ compilers, which facilitates their adoption in industry.

Because of their effectiveness and applicability to C and C++ software, gray-box fuzzers are promising alternatives

to support the automated mutation testing of CPS 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]. For simplicity, we use the term unit test cases to indicate test cases implemented with the same language as the SUT providing inputs to a function under test; however, the unit test cases generated by our approach may exercise either single units (e.g., a C function) or multiple components (e.g., if the function under test invokes other functions or interacts with remote components through the network).

Since fuzzers cannot automatically generate test drivers for unit testing, several techniques that generate test drivers to enable fuzz testing of C and C++ APIs have been developed [13]–[15]; however, they are not maintained and therefore not applicable in practice. Further, they require the availability of client programs using the API under test, which are not available in our context.

To address the limitations above, as a first contribution of this paper, we propose an approach based on fuzzing that consists of an automated pipeline supporting the generation of unit test cases for mutation testing; we call our approach *MOTIF* (MutatiOn TestIng with Fuzzing). Our pipeline includes the automated generation of seed data and the automated generation of test drivers; to enable mutation testing, test drivers automatically determine if the outputs of the mutant differ from the outputs of the original software. We do not design a dedicated fuzzing algorithm but rather propose an approach to apply SOTA fuzzers to support automated mutation testing. Our intuition is that the bucket-based coverage strategy of fuzzers can effectively drive the selection of test inputs that kill mutants because such strategy, in addition to selecting inputs leading to different software states, can also track, and be guided by, differences in the behavior of the original and the mutated function. From a practical standpoint, relying on standard fuzzing algorithms helps with the adoption of our solution by practitioners because the maintenance of well-known fuzzing tools is guaranteed by several interest groups (e.g., companies investing in reliability and security).

As a second contribution, we compare *MOTIF* with a SOTA symbolic execution approach to determine if fuzzing is more effective and can overcome the limitations of symbolic execution in practical settings. For our experiments, we considered three software components used in satellites currently in orbit: *MLFS*, a mathematical library qualified by the European Space Agency (ESA) for flight systems, *LIBU*, a utility library for nanosatellites developed by one of our industry partners in a project with ESA [16], and *ASNlib*, a serialization/deserialization library generated with the ESA ASN.1 compiler [?].

Our empirical results show that, in the two case study subjects where symbolic execution is applicable (*ASNlib* and part of *LIBU*), *MOTIF* outperforms mutation testing based on

symbolic execution by 46.86 and 10.52 percentage points, respectively. For subjects in which symbolic execution is not applicable (*MLFS* and part of *LIBU*), *MOTIF* achieves interesting results by killing 35.97% and 41.38% of the live mutants, respectively.

This paper proceeds as follows. Section II provides related work on automated mutation testing and background on symbolic execution and fuzzing. Section III describes our pipeline for automated mutation testing with fuzzing and an alternative pipeline relying on symbolic execution. Section IV presents our empirical evaluation. Section V concludes the paper.

II. BACKGROUND AND RELATED WORK

This paper relates to techniques for automated mutation testing and fuzzing; relevant work is discussed below.

A. 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 [17]. 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 [18] 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 (i.e., abstract representations do not behave like the real systems), (4) 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 like Angr [19], KLEE [6], and S2E [20] rely on binary interpretation, which, in practice, requires some degree of environment modeling (e.g., libc library modeling in KLEE) and limit their practical applicability [21].

Compilation-based approaches like QSYM [22] and SYMCC [21] 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. However, we have excluded QSYM and SYMCC from our investigation because there is no dedicated mutation testing approach for them. Further, implementing such a mutation testing approach is a significant research challenge since it entails finding solutions to integrate, within the original program, the logic to derive inputs that kill mutants. Last, these approaches have shown to provide their best results when combined with fuzzing, a solution referred to as *hybrid fuzzing* (see Section II-B). *Hybrid fuzzing* is nevertheless difficult to apply in our context because QSYM and SYMCC still present technical limitations preventing their application to CPS software; indeed, SYMCC relies on LLVM, which is not applicable to certain systems [4], [23], while QSYM is not maintained [24] and relies on an outdated version of the PIN instrumenter [25], [26].

B. Fuzzing

Fuzzing (or fuzz testing) is an automated testing technique that generates test inputs by repeatedly modifying¹ existing inputs; the selection of the inputs to modify is usually driven by metrics collected during the execution of the SUT. Based 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 [27] and discover faults [28] (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.

Most fuzzers generate input files to be used for system-level testing of console applications; however, engineers can 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 III-A1). 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 sanitizers [29], (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 [30]. Further, instead of relying on traditional branch

coverage [31], most fuzzers adopt a bucketing approach to track the number of times each branch has been covered by each input file across ranges: 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 at least one branch for a range of times (i.e., a bucket) not observed before. Such bucketing strategy help 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 [32] relies on a particle swarm optimization algorithm) and (2) select the inputs from the input pool (e.g., AFLfast [33] and AFL++ [34] 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* [35], 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* [22], [36], [37], 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. The two SOTA hybrid fuzzers combine AFL [38] with QSYM [22] and SYMCC [21].

Some researchers have addressed the problem of generating test drivers to fuzz test program functions as in unit testing [13]–[15]; however, they all target library APIs and make the assumption that such APIs have been already integrated into consumer programs (i.e., programs using the library API). FuzzGen [13] relies on the static analysis of both the API under test and its consumers to derive call graphs capturing valid sequences of function invocations and derive test drivers. Different from FuzzGen, Fudge [14] works with a single API consumer and, instead of synthesizing test drivers from a graph, it relies on code snippets (i.e., sequences of API calls and the variables in scope) extracted from the consumer. ApiCraft [15] targets APIs without source code and leverages both static and dynamic information (headers, binaries, and traces) to collect control and data dependencies for API functions. Unfortunately, consumer programs are not available when performing mutation analysis for CPS, which makes the above-mentioned approaches inapplicable; indeed, some components are often developed only for a specific product (e.g., the application layer for a satellite under development), while other components, despite being implemented for reuse (e.g., utility libraries), should be verified by mutation testing before they are integrated into consumer programs.

Other techniques address the problem of generating highly structured input files [39], [40]. 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 [39]. SkyFire, instead, learns a probabilistic context-sensitive grammar to generate JSON and XML files [40]. Such techniques

¹To avoid confusion, we avoid the term ‘mutation’ when describing fuzzing 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.

C. 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) [41]. 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 [42], 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).

1) *Techniques based on constraint solving*: Inspired by the earlier work of Offut et al. [41], Holling et al. execute symbolically the original and mutated functions with input data leading them to generate different outputs [43]. A similar technique from Riener et al. [44] 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* [5], [45], 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) *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. [46] rely on an Ant Colony Optimization algorithm [47] 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 [7], instead, extended the EvoSuite tool [10] 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 [48]. 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 [49]. Unfortunately, when applied to mutation testing, AFFS does not perform better than SOTA solutions [48].

Concerning C software, we should note the work of Souza et al. [50], who rely on the Hill Climbing AVM algorithm [51]. 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 [52]–[54]; however, this implementation is not available. A recent search-based testing tool prototype for C is Ocelot [55]; 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. [56], 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.

D. Techniques based on fuzzing

The work of Bingham [42] is the only one to rely on fuzzing to automate mutation testing for C software. For input generation, it relies on TOFU [57], 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 [58]) and a prototype tool is not available.

Mu2 [59], which has been developed in parallel with *MOTIF*, is a fuzzer that integrates the findings of search-based unit test generation [60] to generate test input files with fuzzing: it relies on the mutation score to drive the generation of test inputs. Different from *MOTIF*, Mu2 tests every live mutant with each generated input and, in the file pool, prioritizes those files that increase the mutation score; the scalability of such choice 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. Results show that Mu2 kills more mutants than the inputs generated by a traditional fuzzer; however, the question remains if Mu2 (i.e., testing all the live mutants together) is more effective than the approach used by *MOTIF* (i.e., testing the original and mutated function in sequence). 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 [61].

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 because of the difficulty of executing static analysis, which is needed for branch distance fitness, 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.

III. PROPOSED APPROACHES

In this Section, we present two approaches for automated mutation testing: (1) MutatiOn Testing wIth Fuzzing (*MOTIF*), our main contribution, which is automated through a pipeline of commands to generate unit test cases by relying on fuzzing (Section III-A). (2) *SEMuP*, which is a pipeline derived from *MOTIF* to perform unit mutation testing with *SEMu* (Section III-B). They are both used in our empirical evaluation.

A. *MOTIF*

Similar to Holling et al. [43], we aim to identify a set of test inputs that lead to different outputs when provided to the original and to the mutated function. To achieve such objective with fuzzing, for each mutated function, *MOTIF* generates a fuzzing driver that reads the input data generated by the fuzzer and then appropriately provides such data, as arguments, to both the original and the mutated function. Finally, the fuzzing driver compares the output data generated by the original and the mutated function, if they differ, the mutant has been killed.

Our intuition is that fuzzers not only help kill mutants because they can achieve high coverage [27] and reach multiple program states, including faulty ones [28], but also that, by invoking the original and the mutated functions within a same fuzzing driver, we can leverage the bucket-based

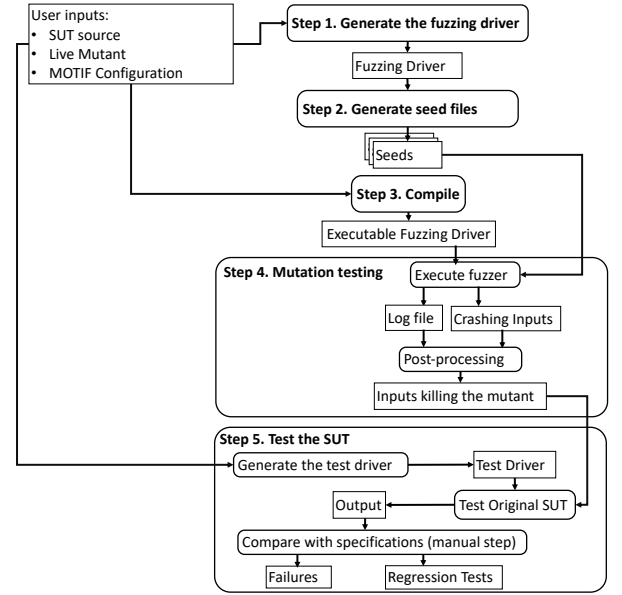


Fig. 1: The *MOTIF* process.

fuzzing strategy to drive the generation of test inputs towards killing mutants. Indeed, if differences in the coverage of the original and the mutated function are observed then the two functions behave differently and, consequently, they yield different outputs leading to the mutant being killed [4], [62], [63]. Also, large differences in coverage lead to new buckets being covered and since fuzzing favors inputs covering new buckets, it indirectly leads to inputs killing mutants. In other words, the bucket-based fuzzing strategy may help kill mutants by preserving, during test generation, those inputs that lead to incorrect intermediate states but do not kill the mutant (i.e., they do not satisfy the sufficiency condition); the following iterations of the evolutionary search process implemented by the fuzzer (see Section II-B) may modify such inputs such that, in addition to reaching an incorrect intermediate state, they also satisfy the sufficiency condition. 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 may measure the difference in coverage between the original and the mutated function, and use this information to further prioritize the inputs in the fuzzer queue (e.g., test first the inputs that leads to larger differences).

MOTIF automatically generates all the scaffolding required to test the original function, the mutated function, and compare their outputs. Specifically, *MOTIF* implements the workflow depicted in Figure 1, which consists of five Steps that we describe below.

MOTIF receives 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, 2, 3, and part of Step 5 (i.e., *Generate the test driver*) can be executed only once for all the mutants belonging to a same

function; indeed, the structure of the input and output data of a function does not change based on the mutants—we do not target interface mutation [64].

1) *Step 1 – Generate the fuzzing driver:* In Step 1, *MOTIF* relies on the *clang* static analysis library [65] to process the SUT and determine the types of the parameters required by the function under test. Such information is 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_*.

```

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   /* Variables for the original function */
6   T_POS origin_pVal; // for the first parameter
7   int origin_pErrCode; // for the second parameter
8   /* Variables for the mutated function */
9   T_POS mut_pVal; // for the first parameter
10  int mut_pErrCode; // for the second parameter
11  /* Variables for the return values */
12  flag origin_return; // for the original
13  flag mut_return; // for the mutant
14
15  /* Copy the input data to the variables for the
16   original function */
17  get_value(&origin_pVal, sizeof(origin_pVal), 0);
18  get_value(&origin_pErrCode, sizeof(origin_pErrCode), 0);
19  log("Calling the original function");
20  origin_return = T_POS_IsConstraintValid(&origin_pVal,
21  &origin_pErrCode);
22
23  /* Copy the same input data to the variables for the
24   mutated function */
25  seek_data_index(0); //reset the input data pointer
26  get_value(&mut_pVal, sizeof(mut_pVal), 0);
27  get_value(&mut_pErrCode, sizeof(mut_pErrCode), 0);
28  log("Calling the mutated function");
29  mut_return = mut_T_POS_IsConstraintValid(&mut_pVal, &
30  mut_pErrCode);
31
32  log("Comparing result values: ");
33  ret += compare_value(&origin_pVal, &mut_pVal, sizeof(
34  origin_pVal));
35  ret += compare_value(&origin_pErrCode, &mut_pErrCode,
36  sizeof(origin_pErrCode));
37  ret += compare_value(&origin_return, &mut_return,
38  sizeof(origin_return));
39
40  if (ret != 0){
41    log("Mutant killed");
42    safe_abort();
43  }
44  log("Mutant alive");
45  return 0;
46 }

```

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

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 the original and to the mutated function, respectively; in our example, it declares a struct `T_POS` and an `int` variable. The 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); *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, *MOTIF* extends it with random data (Line 2). Additionally, the fuzzing driver

TABLE I: 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

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 output generated by the original and the mutated function (Line 28-31). Unfortunately, in C and C++, the presence of pointer and reference arguments complicates distinguishing input and output parameters. Further, to determine input parameters, we cannot rely on data-flow analysis because it entails preprocessing the SUT with a static analysis framework (e.g., LLVM [66]), which is often not feasible with CPS software [4]. Therefore, we adopt a simple solution consisting of comparing all the parameters and return values of the original and mutated function; indeed, comparing input parameters does not lead to incorrect mutant killing since they are not modified. For pointers, we compare the pointed data (e.g., an `int` instance for `int*`). If the pointer is used as an array, 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 because *MOTIF* automatically generates arrays with a default length (100). If arrays are dynamically allocated by the function under test, the end-user should specify the minimal possible length (e.g., an array of length one), to avoid false positives due to readings out of the array bounds. For data structures with pointer fields, it is possible to specify the pointed data length and the initialization procedure. When the outputs differ, the fuzzing driver stops its execution with an abort signal (Line 35 in Listing 1) thus letting the fuzzer detect the aborted execution and store the input file; *MOTIF* then stops the fuzzer because the mutant has been killed.

2) *Step 2 – Generate seed files:* In Step 2, *MOTIF* generates seed files based on the types of input parameters for the function under test. Seed files are used by the fuzzer to start the testing process; usually, fuzzers are executed with seed files that correspond to typical inputs for the SUT. In our case, we automatically generate seed files that contain enough bytes to fill all the input parameters with values covering basic cases. Precisely, for each primitive type, we have identified three seed values that are representative of typical input partitions; they are reported in Table I. For example, for numeric values, we provide zero, a negative, and a positive number. Based on these seed values, for each fuzzing driver, *MOTIF* generates at most three seed files in such a way 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).

	Seed 1	Seed 2	Seed 3
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 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.

Please note that *MOTIF* can generate seed files also for complex input types, indeed the `struct T_POS` received as input by function `T_POS_IsConstraintValid` consists of an `enum` (named *kind*), which is used to specify 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 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 padding²), *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`.

```

1 typedef enum { T_POS_NONE, longitude_PRESENT,
2               latitude_PRESENT, height_PRESENT,
3               subTypeArray_PRESENT, label_PRESENT,
4               intArray_PRESENT, myIntSet_PRESENT,
5               myIntSetOf_PRESENT, anInt_PRESENT
6 } T_POS_selection;
7
8 typedef struct {
9     T_POS_selection kind;
10     union { asn1Real longitude; asn1Real latitude;
11            asn1Real height; My2ndInt anInt;
12            T_POS_label label; T_ARR intArray;
13            T_SET myIntSet; T_SETOF myIntSetOf;
14            T_POS_subTypeArray subTypeArray;
15     } u;
16 } T_POS;

```

Listing 2: Definition of `struct T_POS`

3) *Step 3 – Compile the SUT*: In Step 3, *MOTIF* compiles the fuzzing driver, the mutated function, and the SUT using the fuzzer compiler; this is necessary to collect the code coverage information required by the fuzzer.

4) *Step 4 – Perform mutation testing*: In Step 4, *MOTIF* runs the fuzzer to generate inputs for the executable fuzzing driver. The fuzzer keeps generating input files until it reports one or more crashes, after which *MOTIF* stops the fuzzer. The execution 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 crashes occur during the execution of the mutated function, which 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 consider these inputs as inputs that kill the mutant.
- Aborted executions due to the fuzzing driver determining that the mutant has been killed (see Line 35 in Listing 1).

MOTIF keeps all the test inputs killing a mutant. However, the function under test may generate non-deterministic outputs and in such situations, despite observed differences in outputs, the inputs may not have killed the mutant. For example, 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 with 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 the outputs of the two function calls, it implies that the function under test is non-deterministic and thus that the input does not kill the mutant. *MOTIF* considers mutants exclusively killed by false positive inputs to be live. To kill mutants in functions modifying global state variables, the end-user should manually introduce the instructions required to reset the state between the two function calls in the fuzzing driver, which is similar to what required by other fuzzing approaches for unit and library testing (e.g., LibFuzzer [67]).

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 matches the fuzzing driver except that (1) it invokes only the original function (Line 5 in Listing 3) and (2) instead of comparing the outputs obtained from two function invocations, it prints out the output data generated by the original function (Lines 7 to 9). 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. Otherwise, a fault has been found in the SUT; such a scenario is one of the key advantages of mutation testing: by testing the SUT with inputs that detect simulated human mistakes (mutants), actual faults in the SUT are more likely to be found than with randomly selected inputs.

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

In our test driver, the print statements for struct and pointers are generated based on the configuration of the fuzzing drivers. Precisely, by default, all the bytes belonging to a struct are printed out. In the presence of pointers, if the end-user specifies the size of the data referred to by pointers, the test driver prints the pointed data instead of the pointer value.

```

1 int main(int argc, char** argv){
2   load_file(argv[1]); /* load the input file */
3   // Declaration of variables and assignment with input
4   // file data missing to save space...
5   /* Invoke the original function */
6   _return = T_POS_IsConstraintValid(&pVal, &pErrCode);
7   /* Print output values of the original function */
8   printf_struct("pVal (T_POS)=", &pVal, sizeof(pVal));
9   printf("pErrCode (int) = %
10  printf("return (flag) = %
11  return 0;
12 }

```

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

B. SEMuP

To compare the effectiveness of fuzzing and SE when used for automated mutation testing, we have adapted the *MOTIF* pipeline to enable test generation with *SEMu*; we call the adapted pipeline *SEMuP*. At a high level, *SEMuP* follows the same steps of *MOTIF*, with differences concerning how inputs and outputs are declared to enable test generation with SE.

```

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

In Step 1, we generate *SEMu* drivers instead of fuzzing drivers; an example *SEMu* driver generated for function *T_POS_IsConstraintValid* is shown in Listing 4. In *SEMu* drivers it is necessary to specify what are the input parameters to be treated symbolically (see Line 8 in Listing 4); input parameters are provided as configuration parameters 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 taken care by *SEMu* when symbolically executing the original and the mutated functions in parallel (see Section II-C1). 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.

For *SEMu*, there is no Step 2 (i.e., we do not generate seed inputs). In Step 3, we compile the mutated function and the

TABLE II: Subject artifacts.

Subject	Open-source	LOC	# Test cases	Statements coverage	MS
<i>MLFS</i>	Yes	5,402	4,042	100.00%	81.80%
<i>LIBU</i>	No	10,576	201	83.20%	71.20%
<i>ASN1lib</i>	Yes	7,260	139	95.80%	58.31%

SEMu drivers with LLVM. Step 4 concerns the execution of *SEMu* and the processing of its logs to determine if mutants have been killed. Step 5 is conceptually the same as for *MOTIF*, except that we load the inputs generated by KLEE.

IV. EMPIRICAL EVALUATION

We address the following research questions:

RQ1. How does mutation testing based on fuzzing compare to mutation testing based on symbolic execution, for software where the latter is applicable? SE proved to be an effective mean to perform mutation testing of command line tools that do not rely on floating-point instructions nor integrate black-box components. Therefore, SE may still outperform fuzzing when applied to mutation testing of CPS units that satisfy those assumptions.

RQ2. How does mutation testing based on fuzzing perform with software that cannot be tested with symbolic execution? The motivation for our work stems from the limited applicability of SE and we therefore aim to assess if fuzzing can effectively overcome such limitations.

RQ3. How does MOTIF's seeding strategy contribute to its results? *MOTIF* kills mutants either through the generated seeds or through the inputs generated by the fuzzer; we therefore aim to assess how the two strategies individually contribute to *MOTIF* results.

A. 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 [68], which complies with the ECSS criticality category B [69], [70], (b) *LIBU*, which is a utility library developed by one of our industry partners and used in NanoSatellites, and (c) *ASN1lib*, a serialization/deserialization library generated with *ASN1SCC* from a test grammar provided by ESA. *ASN1SCC* is a compiler that generates C/C++ code suitable for low resource environments [71], [72].

Our software subjects are provided with test suites; information about their code coverage is reported in Table II. Some of our test suites do not achieve 100% statement coverage because some components need to be tested with specific hardware not available to us; therefore, we generated mutants only for the covered statements. We generated mutants with MASS [4], [73]; specifically, we rely on all the mutation operators supported by MASS and 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 [4]. Column *MS* in Table II provides the mutation score 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 [74]. The lowest mutation score is observed with *ASNlib*, which is automatically generated by the *ASN1SCC* using a grammar-based approach [?]. Our subjects' mutation score is in line with empirical investigations reporting mutation scores ranging from 55% to 95% [75], [76], for CPS software.

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 and though this could be an under-approximation, it does not introduce bias in the comparison between *MOTIF* and *SEMuP*, which both 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 [77]; for this reason, including duplicate and subsumed mutants should not introduce bias in the comparison of the two approaches. In other words, a mutation testing approach should easily kill mutants that are either duplicates and subsume other killed mutants; if it does not happen, it is correct to penalize such an approach in the empirical evaluation. Finally, for *LIBU*, we have excluded 8 mutants manually identified as equivalent after inspecting the (few) live mutants not killed by *MOTIF* for RQ1; we could not perform the same analysis for the other cases as such manual analysis would take too long.

B. Experimental setup

We performed our experiments using a prototype implementation of the *MOTIF* and *SEMuP* pipelines described in Section III.

For *MOTIF*, as fuzzer, we selected AFL++ because it is the fuzzer that performed better in terms of code coverage, according to a recent benchmark in the literature [27]; moreover, along with HonggFuzz [78], it is the fuzzer that maximizes fault coverage in another recent benchmark [28].

Since the number of live mutants is large for complex CPS, we assume that an effective setup for mutation testing consists in relying on distributed services that enable the execution of a large number of computing nodes in parallel; for example, we execute our experiments on a grid infrastructure. Although multiple mutants may be killed by similar test inputs [5], we do not test live mutants with inputs that have killed other mutants because we test mutants in parallel. In the future, we will assess how *MOTIF*'s effectiveness can be improved by reusing inputs that have killed mutants.

To account for randomness factors in *MOTIF* and *SEMuP*, we executed each approach ten times for each subject. For each mutant, we executed both *MOTIF* and *SEMuP* 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 *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 *MOTIF*'s (around 650 seconds more).

MOTIF is available online [79]; also, we provide a *repliation package* with our open-source subjects and all our empirical data [80].

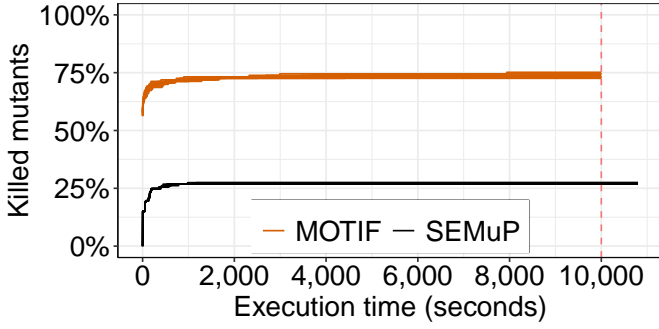
C. RQ1 - Fuzzing vs Symbolic Execution

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 depends on the time required to kill the mutants; indeed, lengthy test data generation may delay the testing process and increase the usage of computing resources. Cost is also driven by the time required to manually inspect test outputs; however, *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.

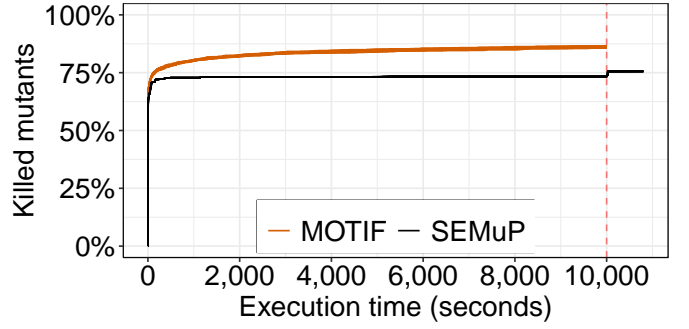
To address RQ1, we could not consider *MLFS* because it works mainly with floating point arguments, which are not supported by KLEE. An old version of KLEE addresses floating point variables but it is not integrated into *SEMu*. We therefore focus on *LIBU* and *ASNlib*; however, for *LIBU* we considered 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 *ASNlib* and 153 for *LIBU*.

2) *Results*: Figure 3 depicts the percentage of live mutants killed by *MOTIF* and *SEMuP* for *LIBU* (3a) and *ASNlib* (3b), respectively. Each plot depicts the percentage of mutants killed after each second, for each run. Separate curves are plotted to visualize dispersion across the ten runs. The vertical dashed line shows the 10,000 seconds timeout when, for *ASNlib*, which includes paths with several nested conditions, we observe a rapid increase in the number of mutants killed by *SEMuP*. 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.

The plots show that *MOTIF* outperforms *SEMuP*. After 10,000 seconds, *MOTIF* kills between 111 (72.55%) and 115 (75.16%) mutants for *LIBU* (avg. is 112.9, 73.79%) and between 1,153 (85.60%) and 1,167 (86.64%) for *ASNlib* (avg. is 1,159.5, 86.08%). 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 *ASNlib* (avg. is 1,017.8, 75.56%). On average, across the ten runs, *MOTIF* kills a percentage of mutants that is 46.86 percentage points (pp) and 10.52 pp higher than *SEMuP*'s, for *LIBU* and *ASNlib*, respectively.



(a) *LIBU*



(b) *ASNlib*

Fig. 3: Percentage of live mutants killed by *MOTIF* and *SEMuP*

The difference between *MOTIF* and *SEMuP* is significant at every timestamp, based on Fisher test³ [81] ($\alpha < 0.01$). For example, after one minute, *MOTIF* kills, on average, 101.3 (*LIBU*) and 976.2 (*ASNlib*) mutants, while *SEMuP* kills 29 (*LIBU*) and 924.6 (*ASNlib*) mutants. For *LIBU*, *MOTIF* quickly reaches a near plateau because of *LIBU*'s simple control logic.

Though *MOTIF* outperforms *SEMuP*, they show some degree of complementarity, which suggests that future work should integrate hybrid fuzzers in *MOTIF* (see Section II-B, including the limited applicability of existing hybrid fuzzers). If we consider the best run of each approach, in the case of *ASNlib*, *MOTIF* kills 252 (18.70%) mutants not killed by *SEMuP*, while *SEMuP* kills 103 (7.65%) mutants not killed by *MOTIF*. In the case of *LIBU*, *MOTIF* kills 74 (48.36%) mutants not killed by *SEMuP*, while *SEMuP* kills 1 (0.65%) mutant not killed by *MOTIF*. 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 *ASNlib*, 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 in *LIBU* that were confirmed by developers; *SEMuP* discovered three of them too.

D. RQ2 - Fuzzing effectiveness

1) *Design*: The mutants considered for RQ2 are the ones that cannot be tested with *SEMuP* because of limitations of symbolic execution, which appear to be prevalent in the context of CPS. Recall that such software often cannot be compiled with LLVM, include I/O operations, and rely on floating point variables. To determine if fuzzing is effective

³We compare the proportion of mutants killed by the two approaches across the ten experiments, which gives us high-statistical power given the large number of mutants.

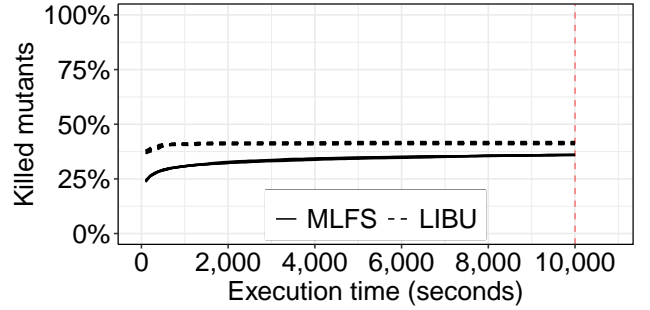


Fig. 4: RQ2 results with *MLFS* and *LIBU*.

at compensating for the limitations of symbolic execution, we applied *MOTIF* to all the mutants of *MLFS* (3,891) and a subset of the mutants derived from the *LIBU* functions excluded for RQ1 (290). Precisely, for *LIBU*, we selected all the mutants for which we can derive a complete fuzzing driver automatically (i.e., eight functions that can be tested without executing setup operations, 94 mutants) and a random subset of the other functions (32 functions, 196 mutants).

As in RQ1, we discuss cost-effectiveness to determine if fuzzing can effectively overcome the limitations of symbolic execution.

2) *Results*: Figure 4 shows the percentage of live mutants killed by *MOTIF* for *MLFS* and *LIBU*.

In the case of *MLFS*, after 10,000 seconds, *MOTIF* kills on average 1,399.4 (35.97%) mutants (min 1,391, max 1,408). The proportion of killed mutants is lower than for RQ1 because of, most probably, the mathematical nature of *MLFS*, resulting in mutants being killed only by inputs from a very small part of the input domain.

For *LIBU*, after 10,000 seconds, *MOTIF* kills, on average, 120 (41.38%) mutants (min 118, max 122). Such percentage of killed mutants is again lower than for RQ1 because some of the mutants can be killed only with inputs belonging to a narrow portion of a large input domain (e.g., an input string that matches a string stored in a global variable).

As for RQ1, for both *MLFS* and *LIBU*, after one minute, *MOTIF* kills a large proportion of the mutants killed after 10,000 seconds. On average, after one minute, *MOTIF* kills

63.39% (i.e., 22.80%/35.97%) of all the mutants killed for *MLFS* and 88.26% (36.52%/41.38%) of all the *LIBU* mutants killed. Our results show that *MOTIF* can be practically useful even when the budget available for mutation testing is limited.

For *LIBU*, the number of mutants killed by *MOTIF* reaches a plateau after 1,500 seconds (25 minutes). For *MLFS*, the number of killed mutants keeps increasing over time, thus suggesting that a large test budget may help *MOTIF* identify inputs that kill mutants when they belong to a narrow subdomain of the input space.

E. RQ3 - Seeding effectiveness

1) *Design*: To discuss how *MOTIF* seeds contribute to mutation testing results, we focus on the proportion of mutants killed with seed inputs in the experiments performed to address RQ1 and RQ2.

2) *Results*: In RQ1 experiments, for *LIBU* and *ASN1lib*, one mutant (less than 1% of the mutants killed on average in 10,000 seconds) and 280 (24.15%) mutants are killed by seeds, respectively. In RQ2 experiments, seeds kill 76 *MLFS* mutants (5.43%) and 26 *LIBU* mutants (21.66%).

The percentage of mutants killed by seed inputs largely depends on the nature of the functions under test. For *MLFS* and the *LIBU* functions considered for RQ1, such percentage is low because they mainly alter mathematical operations whose mutants are killed with inputs satisfying complex constraints. For RQ2, the proportion of *LIBU* mutants killed is higher because several mutants alter conditions verifying the correctness of input strings; the seed strings generated by *MOTIF* include characters (e.g., spaces) that are targeted by such correctness controls, thus killing the mutants. Seed inputs do not introduce bias in RQ1 results since *SEMuP* kills most of the mutants killed by seed inputs (267/280 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 (RQ2-*ASN1lib*), 75.85% (i.e., 100% – 24.15%) of the killed mutants being killed by fuzzing.

F. Threats to validity

To address threats to internal validity, we manually verified that *MOTIF* and *SEMuP* correctly execute and, further, we manually inspected a large subset of the generated test cases and all the mutants killed by *MOTIF* but not *SEMuP*. 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 27,918 mutants reported as killed by the fuzzing driver, only 123 were false positives (0.4%), 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 IV-B). Further, though in Section II-A we clarified the technical reasons for not applying hybrid fuzzers, they could

be considered in future work if the applicability of their underlying technology (e.g., LLVM) improves.

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, and a data serialization component. 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.

V. 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 to process the input data generated by the fuzzer, provides appropriate chunks of 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. To compare *MOTIF* with a SOTA approach based on symbolic execution, we created an alternative pipeline that leverages symbolic execution instead of fuzzing. Our results show that the approach based on fuzzing outperforms the one based on symbolic execution, for two software subjects where symbolic execution is applicable: it kills 73.79% and 86.08% of live mutants in contrast to 26.93% and 75.56% for symbolic execution, respectively. Further, it also detects a large number of mutants (35.97% and 41.38%) for subjects where symbolic execution is infeasible. Our results therefore clearly show that fuzzing should be adopted as the preferred method to use to perform mutation testing. Further, this motivates the development of fuzzing tools dedicated to mutation testing which can, for example, prioritize inputs in the fuzzer queue based on the difference in coverage between the original and the mutated function.

ACKNOWLEDGMENT

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 Thierry Titchou Chekam to help with the development of the SEMUs pipeline.

REFERENCES

- [1] 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.
- [2] 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.
- [3] 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.
- [4] 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>
- [5] 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.
- [6] 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.
- [7] 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.
- [8] Cobham Gaisler, "RTEMS Cross Compilation System," <https://www.gaisler.com/index.php/products/operating-systems/rtems>, 2021.
- [9] 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.
- [10] 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>
- [11] P. McMinn, "Search-based software test data generation: a survey," *Software Testing, Verification and Reliability*, vol. 14, no. 2, pp. 105–156, 2004. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.294>
- [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] "FAQS project," <https://faqas.uni.lu>, 2023.
- [17] 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.
- [18] 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>
- [19] 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.
- [20] 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>
- [21] 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.
- [22] 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.
- [23] E. Vigano, O. Cornejo, F. Pastore, and L. C. Briand, "Data-driven mutation analysis for cyber-physical systems," *IEEE Transactions on Software Engineering*, vol. 49, no. 04, pp. 2182–2201, apr 2023. [Online]. Available: <https://doi.org/10.1109/TSE.2022.3213041>
- [24] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "QSYM archived repository," 2023. [Online]. Available: <https://github.com/sslab-gatech/qsym>
- [25] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '05. New York, NY, USA: Association for Computing Machinery, 2005, p. 190–200. [Online]. Available: <https://doi.org/10.1145/1065010.1065034>
- [26] Intel corporation, "Pin - a dynamic binary instrumentation tool."
- [27] 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>
- [28] 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>
- [29] CLANG, "Undefined Behavior Sanitizer," <http://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html#ubsan-checks>, 2020.
- [30] M. Zalewski, "How AFL works," 2022. [Online]. Available: https://afl-l.readthedocs.io/en/latest/about_afl.html#how-afl-works
- [31] P. Ammann and J. Offutt, *Introduction to software testing*. Cambridge University Press, 2016.
- [32] 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>
- [33] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-Based Grey-box 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.
- [34] 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.

- [35] P. Wang, X. Zhou, K. Lu, T. Yue, and Y. Liu, "The Progress, Challenges, and Perspectives of Directed Greybox Fuzzing," 2020. [Online]. Available: <http://arxiv.org/abs/2005.11907>
- [36] R. Majumdar and K. Sen, "Hybrid concolic testing," in *29th International Conference on Software Engineering (ICSE'07)*, 2007, pp. 416–426.
- [37] 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.
- [38] M. Zalewski, "American Fuzzy Lop: a security-oriented fuzzer," 2020. [Online]. Available: <http://lcamtuf.coredump.cx/afl/>
- [39] 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>
- [40] 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.
- [41] 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.
- [42] D. B. Brown, *Mutation Testing: Algorithms and Applications*. The University of Wisconsin-Madison, 2020.
- [43] 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.
- [44] 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.
- [45] T. T. Chekam, "SEMu: Symbolic Execution-based Mutant Analysis Framework," <https://github.com/thierry-tct/KLEE-SEMu>, 2023.
- [46] 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.
- [47] M. Dorigo, M. Birattari, and T. Stutzle, "Ant colony optimization," *IEEE computational intelligence magazine*, vol. 1, no. 4, pp. 28–39, 2006.
- [48] G. Fraser and A. Arcuri, "Achieving scalable mutation-based generation of whole test suites," *Empirical Software Engineering*, vol. 20, no. 3, pp. 783–812, 2015.
- [49] 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>
- [50] 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.
- [51] B. Korel, "Automated software test data generation," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 870–879, 1990.
- [52] K. Lakhota, 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.
- [53] —, "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.
- [54] K. Lakhota, "Honggfuzz," <https://github.com/kiranlak/austin-sbst>, 2022.
- [55] 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>
- [56] 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.
- [57] Z. Wang, B. Liblit, and T. Reps, "Tofu: Target-oriented fuzzer," *arXiv preprint arXiv:2004.14375*, 2020.
- [58] European Space Agency, "Space," 2021. [Online]. Available: <https://sir.csc.ncsu.edu/portal/bios/space.php>
- [59] 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>
- [60] 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.
- [61] 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>
- [62] D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *STVR*, 2013.
- [63] 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.
- [64] 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.
- [65] LLVM project, "Clang library," <https://clang.llvm.org/>, 2023.
- [66] —, "The LLVM compiler infrastructure project," <https://llvm.org/>, 2023.
- [67] LLVM, "LLVM documentation - libfuzzer – a library for coverage-guided fuzz testing," <https://llvm.org/docs/LibFuzzer.html>, 2022.
- [68] European Space Agency, "MLFS - mathematical library for space software," 2021. [Online]. Available: <https://essr.esa.int/project/mlfs-mathematical-library-for-flight-software>
- [69] ESA, "ECSS-E-ST-40C - Software general requirements," 2009. [Online]. Available: <http://ecss.nl/standard/ecss-e-st-40c-software-general-requirements/>
- [70] —, "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/>
- [71] Semantix and Neupublic, "ASN.1 certified compiler," <https://github.com/ttsiodras/asnlsc>, 2021.
- [72] 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>
- [73] 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.
- [74] 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.
- [75] 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.
- [76] 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.
- [77] 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.
- [78] Google, "Honggfuzz," <https://github.com/google/honggfuzz>, 2022.
- [79] J. Lee, E. Viganò, O. Cornejo, F. Pastore, and L. Briand, "MOTIF toolset," <https://github.com/SNTSVV/MOTIF>, 2023.
- [80] —, "Replication package," <https://doi.org/10.6084/m9.figshare.22693525>, 2023.
- [81] G. J. G. Upton, "Fisher's exact test," *Journal of the Royal Statistical Society. Series A (Statistics in Society)*, vol. 155, no. 3, pp. 395–402, 1992. [Online]. Available: <http://www.jstor.org/stable/2982890>