Received: 17 September 2021 | Revised: 8 April 2022 | Accepted: 19 May 2022

IET Software

DOI: 10.1049/sfw2.12063

ORIGINAL RESEARCH

Mutation-inspired symbolic execution for software testing

Kevin J. Valle-Gómez[1] | **Antonio García-Domínguez**[2] | **Pedro Delgado-Pérez**[1] | **Inmaculada Medina-Bulo**[1]

[1]Escuela Superior de Ingeniería, Universidad de Cádiz, Puerto Real, Spain

[2]Aston University, Birmingham, West Midlands, UK

Correspondence

Kevin J. Valle-Gómez, Escuela Superior de Ingeniería, Universidad de Cádiz, Avenida de la Universidad de Cádiz 10, Puerto Real, 11519, Spain.
Email: kevin.valle@uca.es

Funding information

European Commission (FEDER) and the Spanish Ministry of Science and Innovation, Grant/Award Numbers: RED2018-102472-T, RTI2018-093608-BC33

Abstract

Software testing is a complex and costly stage during the software development lifecycle. Nowadays, there is a wide variety of solutions to reduce testing costs and improve test quality. Focussing on test case generation, Dynamic Symbolic Execution (DSE) is used to generate tests with good structural coverage. Regarding test suite evaluation, Mutation Testing (MT) assesses the detection capability of the test cases by introducing minor localised changes that resemble real faults. DSE is however known to produce tests that do not have good mutation detection capabilities: in this paper, the authors set out to solve this by combining DSE and MT into a new family of approaches that the authors call Mutation-Inspired Symbolic Execution (MISE). First, this known result on a set of open source programs is confirmed: DSE by itself is not good at killing mutants, detecting only 59.9% out of all mutants. The authors show that a direct combination of DSE and MT (naive MISE) can produce better results, detecting up to 16% more mutants depending on the programme, though at a high computational cost. To reduce these costs, the authors set out a roadmap for more efficient versions of MISE, gaining its advantages while avoiding a large part of its additional costs.

1 | INTRODUCTION

Software testing is an essential part of the software development lifecycle. However, testing often requires a great deal of effort. Due to these high expenses, testing may receive less attention than required [1]. The cost of fixing a defect increases exponentially further into the development process. In other words, a defect that is not detected at an early stage will considerably increase the detection effort in successive phases. Eventually, fixing the defect during validation with the client could increase the cost considerably. As a result, testing takes on great importance: a high-quality test suite supports evolving software at a consistent velocity, as defects can be detected sooner and repaired at a lower cost. In this context, testing automation can help produce highly effective test cases in a cost-effective manner.

In the last few years, interest in software testing and its automation is increasing. This is mainly due to technical advances and a heightened awareness of the importance of software testing. The wide variety of software technologies requires addressing the software testing stage in different ways. Object-oriented systems, for example, need test cases that focus on module cohesion and separation of functionalities. Therefore, testing techniques should be adapted to the particularities of the system under test. Performing source code analysis is a common strategy to automatically generate test cases. The goal is to get enough information from the programme to generate a robust test suite. As for analysis, two popular techniques are static [2] and dynamic code analysis [3]. While the first is based directly on the source code, the second is based on its execution. Both techniques provide good results, as shown by previous studies of different authors [4, 5]. In this work, we focus on dynamic analysis of the source code.

One of the most popular techniques for the automated generation of test cases is Dynamic Symbolic Execution (DSE) [6]. This technique explores at once several paths of the source code, using symbolic values instead of concrete ones. Using a symbolic execution engine makes it possible to efficiently control which paths of the source code are traversed [7]. Among its applications, we can mention the automatic generation of unit tests, and the detection of real defects or memory overflows [8]. To assess the fault detection capability of the

generated tests, Mutation Testing (MT) [9] is acknowledged as a powerful method to evaluate and improve test quality. MT is based on the controlled injection of faults into the source code, known as mutations. The result of this injection is a new version of the programme, called mutant. A good test suite should be able to kill (detect) the mutants. In this regard, it uses mutation coverage, which is a metric that evaluates the ability of the test suite to kill mutants. It consists of the ratio of mutants killed over the whole set.

Currently, the procedures for generating test cases automatically via DSE are mainly focussed on traditional coverage criteria. It is possible to address the generation of test cases in a way that increases the mutation coverage, which will help to find more potential defects in software. This involves combining DSE with MT, so that DSE takes MT notions into account when generating test cases automatically: we call this Mutation-Inspired Symbolic Execution (MISE). In its most straightforward implementation (which we call naive MISE), this combination implies executing DSE on each mutant. These new executions can lead to the generation of different constraints to explore the paths and, therefore, to the generation of different inputs from those obtained with the original programme, resulting in new test cases.

Guided by three research questions, we first measure the mutation score of the test cases generated by DSE, using them to kill mutants derived from a set of open source utilities. Then, we see how, by combining DSE and MT, we are able to generate test cases that detect more mutants. To do so, we evaluate around 3000 mutants generated by 12 mutation operators in a set of 30 utilities. The results show that DSE is able to detect 59.9% of mutants, and naive MISE increases the number of detected mutants by up to 16%. Naive MISE, despite its benefits, entails a high cost. Thus, we propose a roadmap to develop more sophisticated variants of MISE that produce better test cases in terms of mutation coverage, while reducing the cost of combining DSE and MT. Other studies in this field also combine DSE with MT. Some improve MT by using DSE [10–12], while others use DSE to kill certain types of mutants [13–15]. Although not combining DSE with MT, DeMillo and Offut [16] lay the foundations to generate test data meeting fault-based testing criteria (such as MT). They generate manual constraints and solve them to obtain test data with better results in their purpose. Our proposal is to use DSE to perform constraint generation and resolution based on MT automatically, reducing the effort significantly.

These studies are described in further detail in Section 7. However, we go a step further, as we analyse the reasons behind these results: DSE does not take into account threshold values, does not consider slight variations in conditions, and is not concerned with variables that impact the output but not the execution flow. We propose several ways in which to extend DSE into MISE, with a view to improving the mutation coverage of the generated test cases.

The rest of this paper has the following structure: Section 2 describes the most relevant topics in this study (MT and symbolic execution). Then, in Section 3, we present the idea of combining DSE and MT to improve the automated generation of test cases. Following this, Section 4 sets out the three research questions for study. Section 5 details the experimental setup and Section 6 presents the results together with their discussion, including threats to validity. Section 7 is a compilation of related work and, finally, Section 8 shows the conclusions and future work.

2 | BACKGROUND

At a very abstract level, there are two test design approaches: black box testing and white box testing [1]. In black box testing, the programme is tested without regard to its internal structure. That is, the code is treated as an opaque box, where only the inputs and outputs are checked for correctness. On the contrary, white box strategies do consider the source code of the programme under test. Among the most popular white box testing techniques, we can find DSE and MT. While DSE is useful for generating test cases automatically, MT is used to evaluate their capability to detect faults. Both techniques, which are key in this paper, are explained in detail in the following subsections.

2.1 | Mutation testing

MT is a fault-based white box technique [9] to evaluate the quality of test cases by introducing slight changes, known as mutations. The programs that result from applying the mutations are called mutants. These changes are implemented through mutation operators. Each mutation operator is a series of instructions that describe the change made to the original programme. Mutation operators are classified by purpose (e.g. change of class modifiers or replacement of arithmetic operators). This technique is usually applied with traditional mutation operators. These kinds of mutation operators introduce simple changes in arithmetic and logical expressions or typical control structures in structured imperative languages. It aims to replicate real coding mistakes made by programmers. Listing 1 is an example of a mutant produced by an operator called ARS, where the arithmetic operator ++ has been replaced by --.

Listing 1: ARS mutation operator example

```
1  int exampleMT () {
2      int a = 10;
3      //a++;
4      /* ARS */a--;
5      return a;
6  }
```

A successful test suite should be able to detect the faults that might be present in the code. According to this logic, at least one test case in the test suite should produce a different output when run on the mutants that change the semantics of the original programme. When the test cases detect a mutant, it is said that the mutant has been *killed*; otherwise, it is known as a *surviving* or *alive* mutant. The evaluation of the test cases

is done by means of the mutation score. This metric, which ranges from 0 to 1, correlates with the ability of the test cases to kill mutants.

$$S = \frac{K}{M - E} \quad (1)$$

Formula (1) shows how this score is calculated, with S being the mutation score, K the number of killed mutants, M the total number of mutants, and E the number of equivalent mutants. A mutant is equivalent when its behaviour does not differ from the original programme. It may happen that the mutation affects a piece of code that is never reached, or that the semantics of the programme remains intact. These are changes that cannot be detected by test cases. Equivalent mutants can affect the reliability of the final result [17], so great efforts are made to identify them before calculating the mutation score. As for detecting equivalent mutants, one of the best-known techniques is Trivial Compiler Equivalence (TCE) [18]. This is a technique that allows detecting equivalent and duplicated mutants through the source code optimisations provided by the compiler. Two or more mutants are duplicated when they are equivalent to each other but not necessarily to the original programme.

A tool that allows the application of MT and is used during this work is MuCPP [19]. This MT tool for the C and C++ languages implements different types of mutation operators, some of which are shown in Section 5. MuCPP classifies mutants generated using *git* branches, and each mutation operator has an identifier assigned to it. The format in which these are named is as follows: m + ID of operator + _ + ID of location + _ + ID of attribute (selecting the specific replacement, e.g. one among several arithmetic operators) + _ + name of file (e.g. *m31_1_1_basename* would represent the ARB operator at its first location, with the first attribute selected in the basename file).

2.2 | Symbolic execution

When we put specific values into the execution of a programme, generally only one path of the programme is explored. In order to appropriately analyse the functionality of a programme, more than one execution should be performed with different inputs. Symbolic execution was presented for the first time in the 70 s [6], being used for debugging and software testing. Since then, this technique has been extensively studied in the literature [7]. This technique analyses software to check which inputs execute each part of the programme. As it uses symbolic values, it is able to simultaneously explore several programme paths. Therefore, running the programme with symbolic rather than concrete values helps check the properties of a programme. This execution is performed by a symbolic execution engine, which stores two key elements for each path:

1. A formula that describes the conditions met during the execution.
2. A dictionary that relates the variables to the symbolic values.

The formula is updated and validated throughout the symbolic execution. The validation is done through a solver, such as Z3 [20]. The solver checks that the properties are not violated and that the formula has a solution.

Listing 2: Symbolic execution example

```
1  int exampleSE (int b) {
2    int a;
3    a = b * 10;
4    if (a == 20) {
5      throw Exception( );     //Path 1
6    } else return a;          //Path 2
7  }
```

Consider the example of Listing 2 with a piece of code that throws an exception when a variable has a specific value. Assume that we execute the code using concrete execution. The variable b would be assigned a value, such as for example, 4. Then, the variable a would receive this value multiplied by 10 (e.g. $a = 40$) and would pass to the conditional statement in line 4. Since the value is different from 20, the function would return the value of a (path number 2). To explore the other path, which throws the exception, we would have to assign the value 2 to b. Although we can see at first sight that b = 2 would throw the exception, this is not so clear when we work with large programs or with complex formulae.

If we execute the code using DSE, then the variable b will take a symbolic value (from now on Ω). This value is then multiplied by 10. When line 7 is reached, the constraint solver will assign a particular constraint for each of the paths. In this example, the constraints would be 'Ω * 10 == 20' for the first path and 'Ω * 10 != 20' for the second. At this point, both paths can be independently resolved and executed. At the end of the symbolic execution, after solving all constraints accumulated during the execution, specific values are returned for Ω. In this example, they would be 2 and a value different from 2.

Despite being a powerful technique for purposes such as test case generation, two limitations undermine the effectiveness of symbolic execution [7].

1. Unsolvable Constraints: there may be cases where the solver cannot solve the path constraints. One of the simplest reasons for this is that the code depends on external functions whose code is unavailable.
2. Path Explosion: when the number of paths is so large that the time needed to explore them is too long.

To alleviate both limitations, different heuristics have been introduced to achieve high coverage without the need to explore all paths. These include randomised path exploration and the widely known DSE.

DSE, also referred to as *concolic execution*, consists of the combination of symbolic execution and concrete execution. This combination allows exploring paths more easily or maintain certain control, though it could be necessary to perform more than one execution. With respect to software testing, different tools have been developed, each one specialised in a particular environment and language. It was first introduced by DART [21], which is a tool that combines DSE with random testing and model checking techniques to run as many paths as possible and to generate unit tests for the C language. We can also find CUTE [22], which expands DART to handle multithreaded programs and thus generate, in addition to unit tests, the programme's thread queue. Other tools such as KLOVER [23] or MACKE [24] have successfully applied DSE to real-world projects in different ways.

In recent years, one of the most widely used and supported tools in the development community is KLEE [25], a DSE-based tool to generate unit tests for C and C++ projects. KLEE uses the LLVM bytecode derived from the programme as the input and requires certain guidance about what inputs should be symbolic. Using search strategies to guide the exploration of the code, KLEE can check every conditional branch. This tool is the main core of other projects that use DSE [23, 26].

3 | PROPOSAL: INTEGRATING DSE AND MT

In this work, we start from the idea that the test generation capabilities of DSE can be improved through MT. The premise is as follows: when DSE goes through the source code, it focusses its efforts on exploring as many paths as possible. This is not a sufficient criterion for MT, so we can find test suites meeting line coverage but with a low capability to detect the injected faults. This is because MT requires more specific input values (or particular combinations of them) to uncover the subtle changes introduced.

We propose the integration of both techniques to generate high-quality test cases in terms of mutation score, leading to a more sophisticated family of techniques that we call MISE. This integration may take on different forms for specific techniques within that family. In order to quickly evaluate the potential of this integration, the rest of this section will present a technique that can be implemented directly without changing DSE nor MT, called naive MISE.

This approach involves applying symbolic execution as many times as needed to try to kill all the non-equivalent mutants. As a first step, we can evaluate the mutation coverage of the tests produced by DSE from the original version of the software under test (SUT), as shown in Figure 1. DSE produces a set of test cases, and MT generates a number of mutants. The mutants are run against the DSE-generated test cases and marked as killed or surviving depending on whether a different output was detected or not.

Naive MISE (shown in Figure 2) goes a step beyond that and applies DSE iteratively to the surviving mutants as well, generating new test cases that are used to test the remaining surviving mutants. This process is repeated until all mutants have been killed, or all the surviving mutants have gone through DSE. At the end of this process, we could achieve a test suite with a potentially higher mutation coverage than the one initially achieved.

Please note that this is a direct implementation with a high cost. This is due to the need to execute DSE independently on each mutant. The aim of doing so is to check the impact that MISE can have on the generation of test cases. Further ideas for more refined MISE techniques are proposed in Section 6.3.

4 | RESEARCH QUESTIONS

The development of this study is driven by three research questions. With the first one, we want to see to what extent DSE is capable of killing mutants. With the second one, we can see how much the procedure proposed above improves the mutation score of the test suite. Finally, in the third one, we propose ways to implement a more efficient version of MISE based on the results obtained in the results of the study for RQ1 and RQ2.

RQ 1: *To what extent does DSE produce test suites that detect potential defects?*

The motivation for this question arises from the fact that DSE is an effective technique regarding classic coverage criteria [7, 25]. However, we want to check whether it is as good in terms of mutation coverage. This is a necessary first step to later evaluate whether MISE improves this coverage. In this question, we assess the ability of this technique to detect different types of defects in a set of utilities of varying complexity, following the procedure in Figure 1 of Section 3.

RQ 2: *Can DSE be combined with MT to produce higher-quality test suites in terms of mutation score?*

This question is motivated by the possibility of increasing the mutation coverage of tests generated automatically with DSE. We want to check whether applying DSE to the mutants derived from the programs under test produces new test cases that leads to an increase in the mutation score. The idea is to apply the naive MISE procedure proposed in Figure 2 of Section 3.

RQ 3: *How can current DSE solutions be improved to address the mutation coverage criterion without significantly increasing the costs?*

The motivation for this research question is to explore possible ways to improve this process by addressing complex coverage criteria such as MT. After the results obtained from the study for answering RQ1 and RQ2, it is interesting to see if DSE is prepared to generate test cases considering mutation coverage as a quality criterion. In this research question, we analyse the results in depth to see which aspects of the

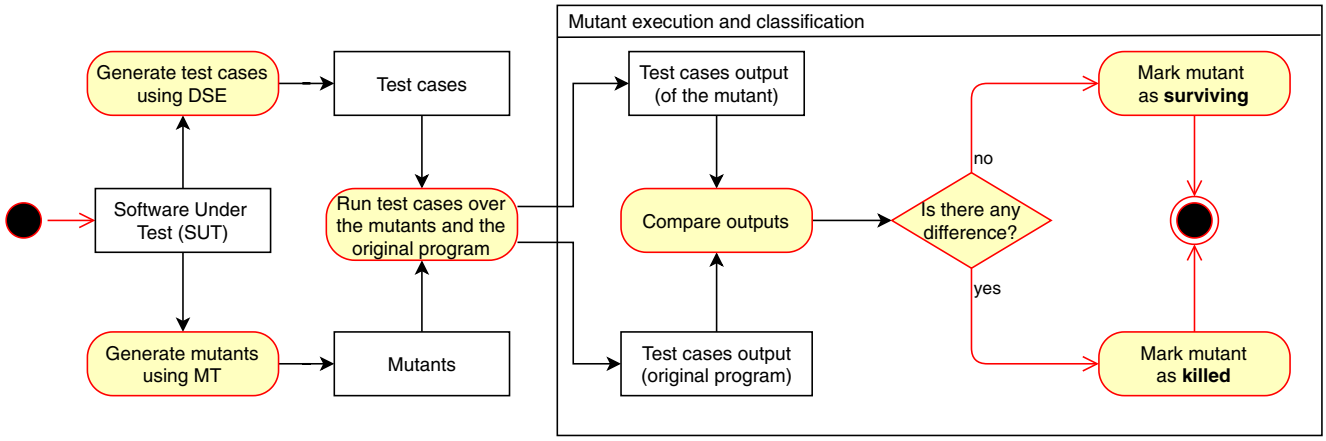

FIGURE 1 Evaluating mutation coverage from Dynamic Symbolic Execution (DSE) execution on the original SUT

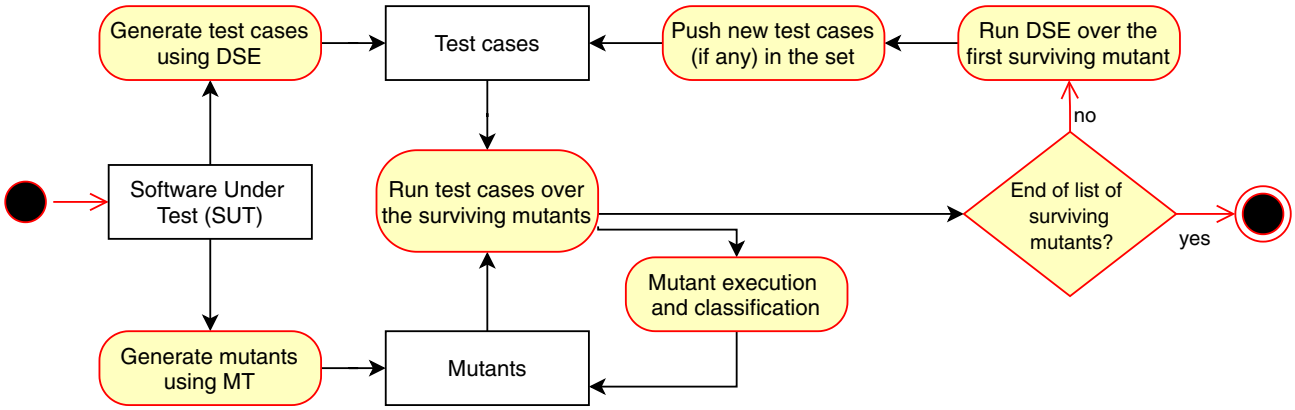

FIGURE 2 Naive MISE: combining Dynamic Symbolic Execution (DSE) and Mutation Testing (MT) for improved mutation coverage

symbolic execution process could be improved to reach a more cost-effective solution than the naive MISE procedure.

5 | EXPERIMENTAL SETUP

In order to check whether DSE is capable of finding potential defects, we use GNU Coreutils[1] as a set of utilities to analyse. These utilities, typically used in other studies involving KLEE [13, 25], are in charge of manipulating different elements of the GNU operating system, such as system properties (*logname* or *hostname*), the file system (e.g. *ls*, *chmod* or *cp*), processing files (e.g. *sort*), and so on. These utilities are included by default in most Unix systems, so they represent the main core of the tools used by millions of users. This means that they have been in active development for decades and have been extensively tested. The utilities vary in complexity. For example, we have a few small utilities with no more than 100 lines of code (e.g. *false*, *sync* or *whoami*). Most of the utilities in Coreutils are of medium size: between 200 and 500 lines (e.g. *chcon*, *chgrp*, *sum* or *touch*). Finally, we can find some more complex tools with over 500 lines of code (e.g. *expr*, *md5sum* or *wc*), exceeding 1000 lines in the case of *numfmt*. Given the variety of functions of this set and their complexity, it is often used in experimental studies, so we can obtain results that are easier to compare with other studies in the literature [13, 25]. As we execute the test cases against the mutants and the original utilities, the outputs are dumped into text files, and then they are cleaned up. Specifically, we filter elements which differ in each execution, such as processes IDs. Finally, the resulting outputs are compared to classify mutants into killed or alive.

We have narrowed down to 30 utilities from the GNU Coreutils set (see Table 2). Although this toolkit includes more utilities, it is reasonable to use a subset of these to make the study more manageable. In the selection, we have discarded some utilities for the following reasons.

- Long output: when running the test cases, some of the utilities produce considerably long outputs (more than 1 GB), which quickly exhausts the available disk space. This includes *base64*, *tee*, *users*, *printf* and *factor*.
- Variable-dependent utilities: some utilities extensively depend on variables that change over time or some external properties. This includes the current date and time or available hard disk space, such as *uptime*, *df* and *date*.

[1] https://www.gnu.org/software/coreutils/.

TABLE 1 Traditional mutation operators included in MuCPP

Operator (ID)	Description
ARB (31)	Arithmetic operator replacement (binary, unary and short-cut)
ARU (32)	
ARS (33)	
AIU (34)	Arithmetic operator insertion (unary and short-cut)
AIS (35)	
ADS (36)	Arithmetic operator deletion (short-cut)
ROR (37)	Relational operator replacement
COR (38)	Conditional operators (replacement, insertion and deletion)
COI (39)	
COD (40)	
LOR (41)	Logical operator replacement
ASR (42)	Short-cut assignment operator replacement

- Large number of mutants: although generating the mutants with MuCPP is a quick task, the time of compilation, execution and test case generation phase of the mutants depends on the tested programme. In our experiments, each mutant may take up to 1 h just to be symbolically executed. Some utilities generate a large number of mutants (between 500 and 2000) and take an unfeasible time in their execution. This includes *join* and *ls*.
- Modification of permissions or system properties: some utilities, when used with specific inputs, can modify the parameters or permissions of the underlying system, leaving the virtual machine in an unstable state. This includes *chmod*, *chgrp*, *rm*, *cp* and *mv*.

The *sleep* programme is a special case. Even though the number of mutants is acceptable, we cannot kill a mutant through its text output. In this case, the execution time is crucial. However, as it is a programme that pauses the execution during a given time, the execution time of the test cases is too high to be considered in these experiments. Some of the utilities mentioned here, such as *chgrp* and *sleep*, have been tested on RQ1 but have been discarded on RQ2 due to the limitations of running test cases on each mutant.

In order to apply DSE and generate test cases, we use KLEE [25]. We have used similar configurations as the authors of KLEE in previous work with GNU Coreutils [25]. For the main options, we have made some minor adjustments according to our context. We left a memory space of 4 GB, since the applications we analyse do not usually exceed this limit and, moreover, it fits well with the configuration of our virtual machines in the cloud. As for the symbolic execution time, a small preliminary study has been conducted to find out how much time the Coreutils applications consume individually. After running for 30 min, 1, 2 and 3 h, we found that after 1 h most applications consume the entire time budget, but the results do not vary enough to observe a significant difference. For this reason, it has been decided to keep the symbolic execution time threshold at 1 h for all applications. Finally, we used the Breadth-First Search (BFS) search strategy. This technique, used in other similar studies [13], focusses its efforts on exploring the paths affected by the inputs. As we are ultimately looking for test cases that will act as test inputs, it strongly fits with our study.

There are several options for applying MT depending on the language or the code environment [27, 28]. Given that GNU Coreutils utilities are written in the C language, we have chosen MuCPP [19], a MT tool for C and C++ projects which introduces different categories of mutation operators, such as class-level operators and traditional ones. This tool applies strong mutation at the source-code level. In strong mutation, the mutant is considered killed when there is an observable difference between its output and the output of the original version (this is different to weak mutation, where the state of the programme is checked to see if it changed right after executing its mutation). We apply a set of 12 traditional mutation operators included in MuCPP, which are described in Table 1 along with their ID of operator to help in their identification later on. Further details on their specific behaviour can be found in the paper in Ref. [19]. In this paper, we apply first-order mutation, that is, unlike higher-order mutation, we only introduce a single change in each mutant. We also apply TCE [18] to eliminate both equivalent and duplicate mutants.

Everything described in this study has been performed in virtual machines hosted on the Google Cloud platform.[2] The work has been distributed across VM instances of the *e2-medium* machine type (dual-core machines with 4 GB of RAM and Ubuntu 18.04 LTS).

Finally, we should note that we did not use other related techniques as a baseline, either because they target a different goal (e.g. equivalent mutant identification in Ref. [11] or seed generation in Ref. [12], whereas we target increasing the mutation score) or programing language (e.g. Java in Ref. [10, 14] or FORTRAN in Ref. [16], whereas we target C), and/or because they apply different tools (e.g. JPF-SE in Ref. [10] or MuClipse and a SCWR algorithm in Ref. [11], as opposed to our use of KLEE and MuCPP), a different mutant injection technique (e.g. bytecode-level mutants in Ref. [13], as opposed to our source code-level mutants), or a different mutant killing criteria (e.g. weak mutation in Ref. [13, 15], unlike our application of strong mutation), making the results hardly comparable. Nevertheless, the obtained results will be contrasted with related studies later on in Section 7 when possible.

6 | RESULTS AND DISCUSSION

In this paper, we conducted two main experiments: one to check the effectiveness of DSE in terms of mutation coverage and another one to evaluate the impact on the results when combining DSE and MT. In this section, we show

[2] https://cloud.google.com/.

TABLE 2 Killed mutants per programme in RQ1

Programme	LOC	Mutants before TCE	Mutants after TCE	Mutants killed	Mutants killed (%)
basename	132	36	29	14	48.2
chcon	446	41	18	16	88.8
chgrp	249	23	22	22	100.0
chown	258	12	12	9	75.0
chroot	197	49	35	8	22.8
cksum	225	120	110	67	60.9
dirname	98	12	11	8	72.7
echo	213	29	21	7	33.3
expr	790	257	254	135	53.2
false	2	5	5	3	60.0
link	60	12	11	8	72.7
logname	56	12	11	8	72.7
md5sum	657	280	216	25	11.6
mkdir	224	111	63	27	42.9
nproc	94	12	11	8	72.7
numfmt	1110	1071	795	254	31.9
pathchk	297	38	21	18	85.7
pwd	263	268	260	18	6.9
realpath	221	25	24	17	70.8
rmdir	171	70	43	24	55.8
sleep	105	33	23	17	73.9
stdbuf	278	103	68	39	56.5
sum	200	157	146	83	56.9
sync	45	12	11	8	72.7
touch	313	111	78	49	62.8
truncate	335	165	164	65	53.2
tty	80	12	11	8	72.7
uname	281	116	57	42	73.7
wc	624	688	546	338	61.9
whoami	63	12	11	8	72.7
Total	8087	3892	3087	1353	59.9

and discuss the results, with the purpose of answering the three research questions.

6.1 | To what extent does DSE produce test suites that detect potential defects?

In this experiment, we test the initial ability of DSE to kill mutants by using the test cases obtained from the non-mutated versions of the utilities. Table 2 shows the total number of mutants together with the number of mutants killed by DSE. We initially get a large number of mutants. However, at this point, equivalent or duplicate mutants are not considered because we apply TCE to eliminate as many of those mutants as possible. The number of mutants is considerably reduced, as we can see in Table 2. Although this does not exactly ensure that there are no equivalent or duplicate mutants left, it does allow us to eliminate a large number of them, so the results will be more reliable.

Within the set of utilities, we find varied results. There are some utilities such as *chcon*, *chgrp* or *uname* where the number of killed mutants is above 70%. There are other utilities where the mutation coverage is under 50% because DSE only killed a few mutants. There are different reasons why those test cases are unable to detect some mutants. As mentioned before, although TCE is a reliable technique, it is not able to detect 100% of the cases, so there are still some undetected equivalent mutants. In fact, a manual review of surviving mutants shows us that there are certain mutants that can be actually killed. These mutants cannot be detected in any way by a test case. We also found mutations affecting threshold values in conditional evaluations. Threshold values can be defined as those values used to determine whether a condition evaluates to true or false when they are compared to other variables; therefore, a slight difference in these variables around the threshold values can decide whether to go through one branch or another. For example, the mutant *m37_3_1_basename* affects a condition of a while loop in line 93, so the evaluation 'sp > suffix' is transformed into 'sp >= suffix'. In this case, *suffix* will act as a threshold value. When the value of *suffix* is strictly less than *sp*, the condition is evaluated to true both in the original version and the mutant. Therefore, to observe a difference between them, we need a test case where the variables *suffix* and *sp* take the same value. Only in that case, the mutant will take another path.

Answer to RQ1: DSE is able to produce test suites capable of detecting potential defects. However, it has limitations in this regard, due to its focus on traditional coverage criteria. In our setup, we managed to kill 59.9% of the mutants after applying TCE, so we can assume that the set of surviving mutants (40.1%) still contains plenty of mutants that can be killed.

6.2 | Can DSE be combined with MT to produce higher-quality test suites in terms of mutation score?

In the previous question, we were just interested in the effectiveness of these test cases in killing mutants. We have seen how there is still a percentage of surviving mutants. The goal of this experiment is to see if, by applying DSE on the mutants, this will lead to the generation of test cases to kill them.

In this section, we show the results obtained after following the procedure proposed at Section 3. Table 3 shows how the results have improved. Naive MISE managed to

improve the mutation score in 9 utilities. In several of them, such as *md5sum*, the improvement is significant, as it increases the mutation coverage by 16%.

Note that the number of utilities in Table 3 is lower than in Table 2 because we only show those utilities in which this solution increased the number of killed mutants. We should also notice that there were some utilities with little room for improvement, such as those with only 2 or 3 alive mutants (*chcon*, *chown*, *dirname*, *link*, *logname*, *nproc*, *pathchk*, *sync*, *tty* and *whoami*). As can be seen, there are still some mutants that could not be detected, either because they are equivalent or because the necessary test case was still not found. A large number of mutants makes it impractical to analyse each individual case, but non-equivalent mutants that remain undetected have been identified in most of the utilities discussed in this section. A good example is the *basename* utility, where there are almost no equivalent mutants left but only one is killed at this stage.

We also found the existence of crossfire mutants. This refers to surviving mutants that are killed by test cases designed to kill other mutants [29]. Thus, it may not be necessary to apply DSE to all surviving mutants. This has happened in three cases: *cksum*, *sum* and *md5sum*. For example, in *cksum*, the test cases associated with one of the mutants (*m34_1_2_cksum*) have killed six other mutants (*m33_1_2_cksum*, *m33_2_1_cksum*, *m33_2_2_cksum*, *m34_10_1_cksum*, *m34_16_1_cksum* and *m34_17_1_cksum*). The same happens with *sum*, where the test cases generated for one mutant (*m35_11_3_sum*) kill three other mutants (*m34_11_1_sum*, *m34_24_1_sum* and *m34_30_1_sum*).

At this point, it is appropriate to study the reason why some mutants that survived the experiments of RQ1 are killed at this stage. A manual analysis of the newly killed mutants has led us to identify several reasons for this, which are described below. These reasons are shown in Table 4 along with the name of some exemplary mutants found and their respective mutation operators. As there are 241 mutants, we only show the most significant ones, where the described situations can be seen.

R1: The point of mutation is not analysed in the first symbolic execution: by analysing the line coverage as an auxiliary measure, we see which lines are covered in each symbolic execution. We found some mutation locations that were not reached in the execution of DSE on the original programme but are later covered in the executions of DSE on the mutants. In our case of study, this usually happens when the execution finishes without evaluating certain conditions. It is also possible that the generated constraint is too complicated to be resolved in the first place. Other factors, such as those described in Section 6.4, may limit the generation of test cases for certain cases and somehow be addressed by the mutant. It is convenient to carry out an extensive empirical study specific to this case. This causes the generation of new test cases which can kill the mutant.

R2: The initial test case values are far from the conditional thresholds: some mutation operators, such as ROR, affect the comparisons. Often, the symbolic execution engine assigns values that are far from the threshold values in those comparisons. When applying DSE on the mutant, this value is used to create a new test case.

R3: The mutation is not directly applied to the conditions, but it does indirectly impact their evaluation. For instance, the mutation may impact the handling of a variable that one of the condition variables depends upon (e.g. the condition variable

TABLE 4 Reasons identified for the presence of surviving mutants, accompanied by illustrative mutants generated by different operators in diverse programs

Reason	Programme	Mutant	Operator
R1	Numfmt	m37_10_1_numfmt	ROR
	Cksum	m34_18_1_cksum	AIU
R2	Basename	m37_3_1_basename	ROR
	Touch	m35_4_3_touch	AIS
		m35_6_1_touch	AIS
R3	Sum	m31_3_1_sum	ARB

TABLE 3 Mutants killed by percentage

Programme	Mutants killed (initial tests) (%)	New mutants killed	Mutants killed by % (RQ2)
basename	48.2	+1	51.7
cksum	60.9	+18	77.3
expr	53.2	+6	55.5
md5sum	11.6	+35	27.7
numfmt	31.9	+3	32.3
realpath	70.8	+1	75.0
stdbuf	56.5	+4	63.2
sum	56.9	+14	66.4
Touch	62.8	+10	75.6

may be calculated from it). This case is slightly more complex, as it directly affects the way restriction queries are built up in DSE.

These cases could be reduced if the test cases had been generated with mutation coverage in mind. When the point of mutation is not analysed in the first symbolic execution (R1) it seems certain variables are not considered during the analysis, since others provide enough influence to go one path or another. However, when the mutation operator is applied, the relative importance of the variables changes, which is why in the experiments of Section 6.2 a new test case is generated that kills the mutant. R2 and R3 are closely related. However, it is easier to solve R2, since it is simple to know the threshold values in the conditions, being useful for mutation coverage. R3 is a case that deserves further analysis since it would be necessary to take into account the changes in the values before reaching the conditions.

One point to note is that most of the new killed mutants belong to the operators `AIU`, `AIS` and `ROR`. We have found that they are the mutation operators that generate most of the mutants. Still, it makes sense if we compare the changes made by these operators with the reasons given in this section. The operators `AIU` and `AIS` introduce arithmetic operators (`-`, `++` and `--`). This causes small changes that can be difficult to detect, entering the above case for R2 and R3. On the other hand, `ROR` replaces arithmetic operators. This situation causes changes in the conditions, fitting in the case exposed for R1 and R2.

As for the mutants that remain alive, these three operators reappear, due to the number of mutants they generate. However, the appearance of the `COI` operator is remarkable, where this method rarely kills any mutant. This mutation operator negates a condition. When DSE goes through the paths in the code, the negation of the condition will not affect it, since it will still access both paths with the same data. This means that the same test cases will be generated, only in a different order. Therefore, when generating test cases for this type of mutants, a new one is rarely obtained using DSE.

Answer to RQ2: The combination of DSE with MT makes it possible to generate new test cases capable of killing more mutants. Even on some occasions, the influence of crossfire mutants allows new test cases to kill other mutants. The proposed implementation of MISE, although it incurs high expenses, gives evidence of the potential of this family of techniques.

6.3 | How can current DSE solutions be improved to address the mutation coverage criterion without significantly increasing the costs?

The experiments carried out to answer RQ2 show that the combination of DSE with MT is promising. However, both techniques present a high cost in terms of time and resources, and the expenses of executing DSE are multiplied by the number of mutants when applying naive MISE. To address this issue, an alternative approach is to directly improve the internal mechanics of DSE with the knowledge extracted from mutants, thereby saving the high cost of their execution. At this point, we have identified three opportunities to enhance the test cases generated by DSE to improve the mutation coverage without significantly increasing the costs: considering threshold values, altering constraints and considering variables that impact programme output when generating test cases. These three opportunities for improvement are described in further detail below. Please note that, for the sake of clarity, from now on we resort to simple examples or simplified fragments of code to better illustrate the proposed improvements instead of using the real source code of the utilities where we detected similar situations.

6.3.1 | Considering threshold values

When analysing the test cases manually, we have noticed that the values generated by DSE are often very far from the threshold values, such as 16,843,009 or −2,147,483,648. While it is true that these values suffice to cover paths or meet traditional coverage criteria, they are not sufficient in terms of mutation coverage. As DeMillo and Offut comment [16], mutant-killing requires test data with the ability to detect faults. This includes branch coverage, domain analysis and extreme/threshold values. The latter is essential for certain types of mutants and although the values proposed by KLEE are certainly extreme, they are not for the conditions that may be encountered during source code evaluation. By reapplying DSE on living mutants, there are many cases where that threshold is considered as a new value in the test cases, and therefore the mutant is killed.

This was the case of the mutant generated by ROR in the utility *numfmt* in line 477, where the operator < is changed to <=. To understand what this involves, let us go back to Listing 2 in Section 2.2. As we saw earlier, two are the constraints that should be handled by the solver at line 7: 'Ω * 10 == 20' and 'Ω * 10 != 20'. Therefore, the test cases 'Ω = 2' and 'Ω = 3' would be able to cover both paths by meeting the first and second constraint, respectively. Now suppose we apply a mutation operator that changes the '==' operator to '<='. With these two test cases, we would not be able to appreciate the change in the code, so this mutant would not be killed, even though it is not equivalent. The value 'Ω = 1', however, would allow exploring the second path in the original programme but the first in the mutant, thus detecting the difference modelled by this mutant. These values are listed in Table 5, where we can see which path would be taken depending on the value.

The incorporation of MT-specific elements when generating test data input can solve this issue. The process should explore the values surrounding the threshold value. For this reason, it is convenient to generate test cases that contemplate

TABLE 5 Paths explored in Listing 2 with different values for the test cases

	Test cases		
Version	**'Ω = 2'**	**'Ω = 3'**	**'Ω = 1'**
Original	Path 1	Path 2	Path 2
Mutant (== by <=)	Path 1	Path 2	Path 1

three elements: the threshold value, a lower one and a higher one. Not only that but also consider other variations that mutation operators may introduce, such as small variations in text strings. It would be necessary to generate more data for the same test cases that DSE generates in the current state. More options would then be considered, which could help to kill more mutants without incurring the costs of applying DSE to the mutants as in naive MISE.

6.3.2 | Altering constraints

A way to improve the test cases is by taking advantage of the generation of test data after the symbolic execution. For this step, we propose the modification of the restrictions of the queries through the *KQuery* language, a textual representation of the constraints in KLEE.[3] The solver obtains the inputs for the programme through these expressions, which serve as the input values for the test cases.

Let us see an illustrative example. Listing 3 shows a small piece of code from one of the examples provided by KLEE. There are two checks in this method: '$x == 0$' and '$x < 0$'. As a result, three test cases are generated, each with a different value for x: 0, less than 0 and greater than 0.

Listing 3: Get Sign example

```
1  int get_sign (int x) {
2    if (x == 0) return 0;     //Path 1
3
4    if (x < 0) return -1;     //Path 2
5    else return 1;            //Path 3
6  }
```

Listing 4 shows the query which is solved for one of the test cases. This query generates a value focussed on reaching the path 3. In this example, we can assume that N0 represents the variable x, which is given a symbolic value in line 4. First, to avoid exploring the path 1, the query imposes that the value of N0 has to be different from 0 (see lines 2 and 3). Then, to reach path 3, the query requires the value of N0 to be greater than 0 (*Slt* stands for *Signed less than*). The result of solving this query will be 16,843,009, which is greater than 0.

[3] https://klee.github.io/docs/kquery/.

Given the KQuery shown in Listing 4, we propose a strategy to explore other paths of the same piece of code apart from path 3. In this example, this strategy would allow exploring path 2 by reusing the query built to explore path 3—without having to run DSE again. In line 5 of Listing 4, as we have pointed out before, the query requires that the value of N0 is greater than 0 (N0 could not be 0, since it would then have taken path 1). By removing Eq false (see line 5 marked in red of Listing 5), the query changes the condition, now requiring the value of N0 to be less than 0. The result of solving the mutated query is −2,147,483,648, a value which effectively is less than 0 and therefore useful to explore path 2.

Listing 4: KQuery of a test case

```
1  array a[4]: w32 -> w8 = symbolic
2  (query [(Eq false
3               (Eq 0
4                   N0:(ReadLSB w32 0 a)))
5           (Eq false (Slt N0 0))]
6           false [] [a])
```

Listing 5: Modified KQuery

```
1  array a[4]: w32 -> w8 = symbolic
2  (query [(Eq false
3               (Eq 0
4                   N0:(ReadLSB w32 0 a)))
5           (Slt N0 0)]
6           false [] [a])
```

According to this example, it is possible to introduce small modifications to the queries to obtain new test data input (following the same logic of mutation operators). The cost of solving these queries is usually much lower than that of generating them. Therefore, after a single DSE run, we can take advantage of existing queries to generate new test data inputs that may increase the mutation coverage, avoiding the high cost of running DSE multiple times.

6.3.3 | Considering variables that impact programme output

Dynamic Symbolic Execution is designed to generate test cases that maximise structural coverage and does not consider mutation coverage by itself. This can mean that it will ignore variables that do not impact the execution flow, even if they impact the output.

Consider Listing 6 from the *sum* programme, which shows the mutated part of *m31_3_1_sum*. In this mutant, the ARB mutation operator replaced the + in the original version of line 2 with a −. This mutant survived the test suite generated by KLEE from the original program.

Listing 6: Sum example of variable

```
1  …
2  checksum = (r & 0xffff) /* ARB */ - (r >> 16);
3
4  printf ("%d %s", checksum,
5      human_readable (total_bytes, hbuf,
6                      human_ceiling, 1, 512));
7  …
```

In this fragment, the value of the variable `r` eventually propagates to the programme output through `checksum`, but it does not impact the execution flow of the programme. The same lines will always be executed regardless of the value of `r`, so KLEE will not consider it for its restrictions unless there are other conditions outside this fragment. For instance, $r = 0$ would work just as well if we only want to cover those two lines. However, $r = 0$ would not help detect the mutation from + to – done by ARB, as we would be adding or subtracting 0 in either case. We would need $r \ll 16$ to produce a non-zero result in order to be able to tell the difference between the original and the mutant.

As it is, KLEE does not have the ability to detect variables that are relevant to the output and consider various conditions that would impact their values. In a way, KLEE would have to be extended to be mutation-aware: this involves several challenges. One challenge is to detect the variables themselves that should be made symbolic, and the other is to generate values that will produce different outputs from the mutants.

Regarding the first challenge, beyond manual annotation by the developer, one option could be to have KLEE aware of a number of functions that 'produce output' and have it follow data dependencies backwards from their arguments. As a first approximation, the appropriate C standard library functions could be annotated in this way: `printf` would be an obvious choice for a function that produces output. Looking at the arguments, `checksum` would be found and following its data dependencies backwards would reveal `r`. This new capability would need to be limited to certain thresholds to be set by the user, however, in order to keep computational costs under control.

In relation to how to produce values relevant to the mutants, one first approximation would be to generate a random set of values for `r` rather than a single value. With multiple values, there is a higher chance the mutant will be killed, but it still would not be guaranteed. A better way would be to look at the subexpressions that involve `r` (such as `r & 0xffff` or `r >> 16`) and set out conditions to produce different results from them (e.g. zero/non-zero). For instance, if we had one test case where `r >> 16` produced a non-zero value, we would be able to kill the highlighted mutant. Other authors have also considered various ways to make KLEE smarter at generating specific values, such as strings (Yoshida et al. [23]), or arrays and bit vectors (Dustmann et al. [30]): our proposal would be a new source of constraints for KLEE.

Answer to RQ3: There are multiple ways to implement MISE in order to automatically generate test cases. In addition to the naive implementation shown in this paper, current tools can be adapted to alter constraints, to consider also variables that impact the output and not just the execution flow, and to select values that can kill more mutants through threshold values and subexpressions. In this way, DSE tools could leverage MT to potentially achieve a similar mutation coverage than that of naive MISE but maintaining a single run and reducing the costs significantly. Designing and implementing these extensions (e.g. by implementing dedicated solvers or performing additional analyses) is part of our future work.

6.4 | Threats to validity

As far as construct validity is concerned, we find two main threats: equivalent mutants and search strategy. The existence of equivalent mutants is a typical limitation of any work with MT. Sometimes, when the number of mutants that remain alive is not too high and there is an in-depth knowledge of the subjects under test, they can be reviewed to detect them manually. In our study, however, we had to resort to an automated method such as TCE. As mentioned before, this technique significantly reduces the number of duplicate and equivalent mutants. However, its effectiveness is limited, so there is no guarantee that all of them have been detected. The search strategy used in DSE is also important in this respect. We have set the BFS search strategy for all the executions with KLEE. This is a popular strategy in this kind of studies [13]. However, KLEE is compatible with other strategies, such as Depth-First Search, Random Path Selection or Non-uniform Random Search. The choice of other search strategies may lead to different results and should be considered according to the nature of the experiment being run. In this work, we constantly compare the outputs of two runs (i.e. the one of the original programme and the one of a mutant), so we considered BFS appropriate to avoid as many random elements as possible.

We also take into account external validation, finding three main threats: the high cost of DSE, the configuration of the experimental setup and the use of GNU Coreutils. As we have seen before, one of the limitations of DSE is the path explosion. The number of possible paths increases with the size of the programme, which usually ends up consuming most of the available resources. To avoid this, we allowed a maximum time span of 1 h for each execution, as in other DSE studies. We believe this is sufficient with the utilities of GNU Coreutils, as many of these utilities do not consume the entire time budget. Regarding the experimental setup, the configuration of MuCPP and KLEE may affect the final outcome of the experiments. We have applied the set of traditional mutation operators included in MuCPP, as the utilities we tested are developed in C. On the other hand, KLEE presents a large number of configurable options, including time budget, different search strategies or the number of symbolic arguments. In this study, we have applied the same configuration proposed by its

authors in their initial study; however, we believe that it is possible for certain mutants to be killed by configuring KLEE for each individual case. Although it is feasible in theory, in practice it may be too resource-intensive compared to the possible benefits. This possibility will be further investigated in future experiments. Finally, despite the benefits of using GNU Coreutils in this research, it also poses some threats to external validity. The size of the utilities is not very large (see Table 2) and, as such, it is unclear whether the results from this study can generalise to other large-scale programs. Nevertheless, KLEE is typically applied to unit tests dedicated to specific system modules, rather than the entire system; therefore, the approach only needs to scale to handle the largest module in the system. Further studies and adaptations that include different tools (apart from KLEE and MuCPP) and programing languages (apart from C) are required in the future, both as test subjects and test generation tools.

7 | RELATED WORK

7.1 | Software testing automation

Automatic test generation is a diverse and active topic in the research community. In this field, some tools have shown a remarkable ability to automatise the generation of test cases following different approaches. For example, two well-known tools are EvoSuite and Randoop. EvoSuite [31] generates test cases specifically designed for object-oriented programing. During the test construction process, this tool uses a Search-Based Software Engineering approach by applying a genetic algorithm. Randoop [32], on the other hand, is a tool for the generation of unit tests through feedback-oriented random testing. This technique consists of running tests with random or semi-random data. These are generated taking into consideration some knowledge of the programme under test. As an example of their effectiveness, Randoop has been used to find bugs in Microsoft.NET code [33], while Evosuite has also been used to find real bugs in financial software [34]. The latter has also obtained the best score in recent tool competitions [35].

Another popular technique for automatic test case generation is symbolic execution. This technique has been extensively studied in the literature [7]. As we have seen before, this technique presents two open challenges: path explosion and code coverage. Different authors work to overcome them and maximise the benefits of the results [23, 36, 37]. Yoshida et al. propose KLOVER [23], a framework for automatic test generation on industrial software systems. It is based on three tools, including a modified version of KLEE. This approach addresses the scalability challenge by only exploring the paths that can lead to new test cases as well as by minimising the resulting test suite. Another study introduces Munch [36], a framework that combines symbolic execution with fuzzing [38], a technique that covers numerous test cases using invalid data as input. In doing so, the authors manage to alleviate the problem of path explosion, thus achieving better function coverage than both techniques separately. To maximise code coverage, Mossberg et al. propose Manticore [37], a Symbolic Execution framework tailored to analyse binaries and smart contracts. In addition, Manticore features a user-friendly interface that allows customising the analysis. Furthermore, there are a wide variety of tools dedicated to software testing via symbolic execution, such as DeepState [39] for C and C++ programs and mCute [40] for UML state machines.

7.2 | Combining DSE with MT

The combination of DSE with MT is an area of active research in recent years. Although not using the DSE technique to generate or solve constraints, DeMillo and Offut [16] are one of the first authors to explore the idea of generating test data by combining MT with algebraic constraint solving, with the aim of finding certain types of mutants. They thus lay the foundations for the work that has been published since then in this line of research. Similar to our research, the authors recognise that fault-based testing techniques, such as MT, are more effective than traditional coverage criteria for finding real defects in source code. However, the cost of generating test cases in these situations entails a high cost. In their proposal, the authors introduce mutations into a set of Fortran and C programs and then generate an algebraic constraint that must be satisfied in order to kill the mutant. By solving the constraint, they obtain test data which produces different results in the mutant when used from the test cases. The distinctions with our work are also notable, as they do not use DSE, so they rely on an external programme (Godzilla) to satisfy the constraints they generate in their proposal. In addition, they apply a different set of mutation operators than ours, although including a few traditional operators (`AOR`, `LCR` and `ROR`). The main advantage of our work is that, by using DSE, the whole process of constraint generation and resolution is automated. Our proposal is also applicable to C and C++ programs, which together with the most current MuCPP mutation operators, we obtain results that are more comparable with the rest of the current work on this subject.

Some studies manage to improve MT with the help of DSE [10]. In that paper, the authors use different techniques, including DSE, in order to introduce certain mutations in the programs. During their experiments, they check the mutation score of the test cases generated by three different test tools. It is shown how the tools are effective in terms of branch coverage but not so effective in killing mutants, which is in line with the initial experiments in our study. By introducing mutants into the test case generation process, mutation coverage is significantly increased. Although they combine different techniques, we can see that the inclusion of MT improves the results similarly to what is proposed in our study. In another related work, Ahmed et al. propose a strategy known as *Detecting Equivalent Mutants using Dynamic Symbolic Execution* [11]. With DSE they are able to classify mutants and detect the equivalent ones. Their set of mutation operators is very similar to that of our study and they obtain good results in

detecting equivalent mutants. However, in our work, we discarded a subset of the equivalent mutants using TCE, so the results are not comparable. Although both studies combine DSE with MT, their approach is focussed on improving MT through DSE. Contrarily, our goal in this work is to improve DSE through MT.

There are some approaches that aim to improve the automatic generation of test cases by combining DSE with MT. Papadakis and Malevris [14] propose an extension to DSE to automatically generate test data based on MT. Incorporating their approach in an automated framework for producing mutation-based test cases, they are able to kill most of the non-equivalent mutants. The process they developed uses mutant schemata and control flow graphs and uses DSE to produce conditions to reach the mutation, infect the programme state and propagate the state to the output. In their case study, they manage to achieve a mutation coverage of more than 85% over the mutants identified as killable. In our work, with naive MISE we killed 75% of all mutants in the utilities *cksum*, *realpath* and *touch*. However, these values are not comparable. While the authors of that study select five tools in the small-medium size range (varying from 40 to 500 lines of code), we analyse the GNU Coreutils set of utilities, which is composed of dozens of independent utilities (altogether, they reach thousands of lines of code). Our purpose is to verify the mutation score of naive MISE in an environment similar to what we find in a real development, so it seems appropriate to apply it to this widely used toolkit. In addition, we have not manually identified the equivalent mutants in our study, but we estimate that our actual mutation coverage would be higher after removing the remaining equivalent mutants. They also highlight the high cost associated with applying both techniques.

Zhang et al. [15] propose PexMutator, a test generation tool for MT using DSE. They generate a meta-programme from the SUT and embed the mutations in specific constraints. By doing so, DSE can generate test cases which target the constraints and kill the mutants in most cases. This tool, in its case study, is capable of killing up to 80% of non-equivalent mutants, which is close to the 75% of all mutants that naive MISE is capable of killing in our best-case scenario. Again, the percentage of killed mutants in both studies is not comparable. On the one hand, the authors use a set of five mutation operators (`ABS`, `AOR`, `LCR`, `ROR` and `UOI`), while in our study we apply a total of twelve mutation operators, as detailed in Table 1. Please note that the acronyms of the mutation operators may vary between the mutation tools used. On the other hand, the authors of PexMutator use a limited set of 5 small-sized utilities taken from a real library (containing between 1 and 12 methods). Unlike these studies, we do not contemplate the use of meta-mutations: our long-term aim is to extend DSE with the idea underlying MT (as discussed in Section 6.3) to have it run on the original programme and still produce better test suites in terms of mutation coverage, without incurring the significant costs that generating and executing mutants can introduce.

A recent tool that stands out for its good results combining DSE with MT for software testing is SEMu [13]. The authors also seek to benefit from mutants that remain alive (i.e. stubborn mutants) to improve DSE but at a more reduced cost than that of exhaustive exploration. To that end, they implemented SEMu as an extension of KLEE, which combines different strategies to make the process more efficient, mainly the use of meta-mutation and a family of heuristics to reduce the number of paths to explore. Thanks to the meta-mutation, the paths shared by the original programme and the mutant are not explored again in each mutant. The optimisation of the heuristic search allows for a better use of the available search budget to explore the paths affected by the mutation. Since they do not perform an exhaustive exploration, this increases the chances of finding an infected state that also propagates to the outputs. There are some differences between their work and ours. For example, SEMu mutates code at the bytecode level, unlike our MuCPP tool which works at the source code level, and the authors use test cases manually developed by the authors of GNU Coreutils as a seed in the execution of the tool. The authors reported better results than KLEE when killing stubborn mutants in a selection of Coreutils programs. Similarly to us, they reported high costs and had to discard a number of Coreutils programs due to those costs. While their approach improves the performance of the technique, the general time is still bounded by the number of existing stubborn mutants. Also, the more exhaustive the configuration for the heuristic search, the more the paths explored and the higher the required computational cost. As such, there is still room for improvement if we could take advantage of the information provided by mutants without executing DSE for each mutant (the naive MISE approach discussed in Section 3).

Another testing tool that combines DSE with MT is SAFL [12]. In this approach, DSE is first used to generate some initial seeds, which will later serve to feed a mutation-based fuzzing process. SAFL proves to be an efficient tool that is able to explore deep paths easier and earlier thanks to both its fuzzing algorithm and the classification of seeds according to the path coverage. However, the data is fuzzed without being aware of the exact mutations injected in the programme, so the approach is in principle limited when it comes to killing some more sophisticated mutants, especially in complex software systems (e.g. those that require some specific values for different variables before reaching the mutation). As with previous work, it is difficult to compare the set of tools used. We should note, however, that they use DSE to obtain an initial set of qualified seeds instead of random ones for the later fuzzing process, while we aim to employ DSE as the main technique to generate test cases able to detect faults.

All these studies are successful in demonstrating how DSE and MT are techniques that, when combined, are of great benefit and serve as a motivation and complement to further develop this family of techniques. This paper is the consolidation of an idea initially outlined as a short 4-page conference paper, written in Spanish [41]. In that outline paper, the hypothesis that combining DSE and MT could potentially lead to a mutation coverage increase was introduced, and a pilot study was carried out considering a small set of utilities. These preliminary results motivated further research in this line, leading

to the comprehensive study presented in this paper. More specifically, this paper expands the idea further by introducing three research questions (see Section 4) and conducting an experimental study based on the diagrams shown in Figures 1 and 2. Additionally, in this paper, we propose possible improvements of the mechanics of DSE based on the obtained results (shown in Tables 2 and 3).

8 | CONCLUSIONS

The contribution of this paper is a study of the effect of combining DSE and MT, introducing the idea of MISE. Its purpose is to incorporate MT into the automatic generation of test cases with symbolic execution in order to increase the mutation coverage. First, there is a study performed on the GNU Coreutils utilities set where we check the initial mutation coverage of the test cases generated with DSE. Second, we exploit the information provided by surviving mutants by applying naive MISE, running DSE on each of these mutants to generate new test cases not contemplated in the initial execution on the original programme. Finally, we analyse the situations in which MISE is able to kill mutants that survived the initial execution in order to find out the reasons behind this outcome.

In our setup, where we use KLEE to detect the mutants produced by 12 mutation operators included in MuCPP, DSE is able to kill 59.9% of them on average in the tested programs. As we have seen in the results, the distribution of surviving-killed mutants varies in each programme, being very low in some cases and close to 100% in others. About one third of mutants survive, so a good number of plausible defects would be left uncovered. It is desirable to improve the way DSE generates test cases to increase the initial percentage of killed mutants.

We have found that naive MISE increases the number of mutants killed in some of the utilities by up to 16%. The test cases generated with a surviving mutant sometimes kill other mutants, such as in the case of *sum*, *cksum* and *md5sum*. This allows for automatically generating a set of test cases with a higher mutation coverage while reducing the required effort since it is not always necessary to apply DSE to every surviving mutant. In conclusion, MISE can produce higher-quality test suites regarding mutation score. However, the direct combination of DSE and MT in naive MISE implies a high cost, and there are still some mutants that remain undetected. To reduce costs, we propose three approaches in Section 6.3: considering threshold values, modifying path constraints and considering variables that impact programme output. One solution is to develop a new solver that generates more test cases by taking into account the current limitations related to mutation coverage. The conclusion is that there is room for improvement and that it is necessary to find an approach adapted to the needs of DSE and the different software projects. This perspective presents new research opportunities to be considered in the development of this technique.

The procedure can be extrapolated to all sorts of software systems, so we intend to start applying it in real environments, such as the one presented in a previous work [42]. In this sense, although it can help the industry to find defects by reducing testing effort, there are different risks than those found in traditional projects. For example, it could be the case that we find unreachable dependencies, or that the symbolic execution needs much more time than usual to generate a set of test cases. In any case, it is convenient to see how it behaves in this type of systems and to measure the benefits of its application.

ACKNOWLEDGEMENTS

The work was partially funded by the European Commission (FEDER) and the Spanish Ministry of Science and Innovation (projects RTI2018-093608-BC33 and RED2018-102472-T).

CONFLICT OF INTEREST

The author declares no conflict of interest.

DATA AVAILABILITY STATEMENT

Data available on request from the authors: the data that support the findings of this study are available from the corresponding author upon reasonable request.

ORCID

Kevin J. Valle-Gómez https://orcid.org/0000-0001-6066-9441
Antonio García-Domínguez https://orcid.org/0000-0002-4744-9150
Pedro Delgado-Pérez https://orcid.org/0000-0003-1568-9288
Inmaculada Medina-Bulo https://orcid.org/0000-0002-7543-2671

REFERENCES

1. Myers, G.J., Sandler, C., Badgett, T.: The Art of Software Testing. John Wiley & Sons, New York (2011)
2. Louridas, P.: Static code analysis. IEEE Softw. 23(4), 58–61 (2006). https://doi.org/10.1109/ms.2006.114
3. Korel, B., Laski, J.: Dynamic program slicing. Inf. Process. Lett. 29(3), 155–163 (1988). https://doi.org/10.1016/0020-0190(88)90054-3
4. Kirchmayr, W., et al.: Integration of static and dynamic code analysis for understanding legacy source code. In: IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 543–552 (2016)
5. Tzermias, Z., et al.: Combining static and dynamic analysis for the detection of malicious documents. In: Proceedings of the Fourth European Workshop on System Security, pp. 1–6 (2011)
6. King, J.C.: Symbolic execution and program testing. Commun. ACM 19(7), 385–394 (1976). https://doi.org/10.1145/360248.360252
7. Baldoni, R., et al.: A survey of symbolic execution techniques. ACM Comput. Surv. 51(3), 1–39 (2018). https://doi.org/10.1145/3182657
8. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM 56(2), 82–90 (2013). https://doi.org/10.1145/2408776.2408795
9. Papadakis, M., et al.: Mutation testing advances: an analysis and survey. Adv. Comput. 112, 275–378 (2019). https://doi.org/10.1016/bs.adcom.2018.03.015
10. Papadakis, M., Malevris, N.: Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based

testing. Softw. Qual. J. 19(4), 691–723 (2011). https://doi.org/10.1007/s11219-011-9142-y
11. Ghiduk, A.S., Girgis, M.R., Shehata, M.H.: Employing dynamic symbolic execution for equivalent mutant detection. IEEE Access 7, 163767–163777 (2019). https://doi.org/10.1109/access.2019.2952246
12. Wang, M., et al.: SAFL: increasing and accelerating testing coverage with symbolic execution and guided fuzzing. In: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, pp. 61–64 (2018)
13. Chekam, T.T., et al.: Killing stubborn mutants with symbolic execution. ACM Trans. Softw. Eng. Methodol. 30(2), 1–23 (2021). https://doi.org/10.1145/3425497
14. Papadakis, M., Malevris, N.: Automatic mutation test case generation via dynamic symbolic execution. In: IEEE 21st International Symposium on Software Reliability Engineering, pp. 121–130 (2010)
15. Zhang, L., et al.: Test generation via dynamic symbolic execution for mutation testing. In: IEEE International Conference on Software Maintenance, pp. 1–10 (2010)
16. DeMillo, R.A., Offutt, A.J.: Others: constraint-cased automatic test data generation. IEEE Trans. Softw. Eng. 17(9), 900–910 (1991)
17. Grün, B.J., Schuler, D., Zeller, A.: The impact of equivalent mutants. In: International Conference on Software Testing, Verification, and Validation Workshops, pp. 192–199 (2009)
18. Papadakis, M., et al.: Trivial compiler equivalence: a large scale empirical study of a simple, fast and effective equivalent mutant detection technique. In: IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 1, pp. 936–946 (2015)
19. Delgado.Pérez, P., et al.: Assessment of class mutation operators for C++ with the MuCPP mutation system. Inf. Softw. Technol. 81, 169–184 (2017)
20. Moura, D., et al.: An efficient SMT solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems, pp. 337–340 (2008)
21. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 213–223
22. Sen, K., Agha, G.: CUTE and jCUTE: concolic unit testing and explicit path model-checking tools. In: International Conference on Computer Aided Verification, pp. 419–423 (2006)
23. Yoshida, H., et al.: KLOVER: automatic test generation for C and C++ programs, using symbolic execution. IEEE Softw. 34(5), 30–37 (2017). https://doi.org/10.1109/ms.2017.3571576
24. Ognawala, S., et al.: Compositional analysis of low-level vulnerabilities with symbolic execution. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, pp. 780–785 (2016)
25. Cadar, C., Dunbar, D., Engler, D.: 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, pp. 209–224 (2008)
26. Li, G., et al.: GKLEE: concolic verification and test generation for GPUs. In: Proceedings of the 17th ACM Symposium on Principles and Practice of Parallel Programming, pp. 215–224 (2012)
27. Chekam, T.T., Papadakis, M., Le.Traon, Y.: Mart: a mutant generation tool for LLVM. In: Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), pp. 1080–1084 (2019)
28. Coles, H., et al.: Pit: a practical mutation testing tool for Java. In: Proceedings of the 25th International Symposium on Software Testing and Analysis, pp. 449–452 (2016)
29. Smith, B.H., Williams, L.: An empirical evaluation of the MuJava mutation operators. In: Testing: Academic and Industrial Conference Practice and Research Techniques-Mutation (TAICPART-MUTATION), pp. 193–202 (2007)
30. Dustmann, O.S., Wehrle, K., Cadar, C.: PARTI: a multi-interval theory solver for symbolic execution. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, pp. 430–440 (2018)
31. Fraser, G., Arcuri, A.: EvoSuite: automatic test suite generation for object-oriented software. In: SIGSOFT/FSE 2011 - Proceedings of the 19th ACM SIGSOFT Symposium on Foundations of Software Engineering, pp. 416–419 (2011)
32. Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for Java, pp. 815–816 (2007)
33. Pacheco, C., Lahiri, S.K., Ball, T.: Finding errors in .NET with feedback-directed random testing, pp. 87–95
34. Almasi, M.M., et al.: An industrial evaluation of unit test generation: finding real faults in a financial application. In: Proceedings - IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track, pp. 263–272. ICSE-SEIP (2017). https://doi.org/10.1109/icse-seip.2017.27
35. Campos, J., Panichella, A., Fraser, G.: EvoSuite at the SBST 2019 tool competition. In: Proceedings of the IEEE/ACM 12th International Workshop on Search-Based Software Testing, pp. 29–32. SBST (2019). https://doi.org/10.1109/sbst.2019.00017
36. Ognawala, S., et al.: Improving function coverage with Munch: a hybrid fuzzing and directed symbolic execution approach. In: Proceedings of the 33rd Annual ACM Symposium on Applied Computing, pp. 1475–1482 (2018)
37. Mossberg, M., et al.: Manticore: a user-friendly symbolic execution framework for binaries and smart contracts, pp. 1186–1189 (2019)
38. Liang, H., et al.: Fuzzing: state of the art. IEEE Trans. Reliab. 67(3), 1199–1218 (2018). https://doi.org/10.1109/tr.2018.2834476
39. Goodman, P., Groce, A.: DeepState: symbolic unit testing for C and C++. In: NDSS Workshop on Binary Analysis Research (2018)
40. Ahmadi, R., Jahed, K., Dingel, J.: MCUTE: a model-level concolic unit testing engine for UML state machines. In: 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1182–1185 (2019)
41. Valle-Gómez, K.J., et al.: Ejecución Simbólica y Prueba de Mutaciones: mejora de la generación automática de casos de prueba. In: Abrahão. Gonzales, S. (ed.) JISBD2021 (2021). http://hdl.handle.net/11705/JISBD/2021/040
42. Valle-Gómez, K.J., et al.: Software testing: cost reduction in industry 4.0. In: 2019 IEEE/ACM 14th International Workshop on Automation of Software Test (AST), pp. 69–70 (2019)

How to cite this article: Valle-Gómez, K.J., et al.: Mutation-inspired symbolic execution for software testing. IET Soft. 16(5), 478–492 (2022). https://doi.org/10.1049/sfw2.12063