Killing Stubborn Mutants with Symbolic Execution

THIERRY TITCHEU CHEKAM, SnT, University of Luxembourg, Luxembourg
MIKE PAPADAKIS, SnT, University of Luxembourg, Luxembourg
MAXIME CORDY, SnT, University of Luxembourg, Luxembourg
YVES LE TRAON, SnT, University of Luxembourg, Luxembourg

We introduce *SEMu*, a Dynamic Symbolic Execution technique that generates test inputs capable of killing stubborn mutants (killable mutants that remain undetected after a reasonable amount of testing). *SEMu* aims at mutant propagation (triggering erroneous states to the program output) by incrementally searching for divergent program behaviours between the original and the mutant versions. We model the mutant killing problem as a symbolic execution search within a specific area in the programs' symbolic tree. In this framework, the search area is defined and controlled by parameters that allow scalable and cost-effective mutant killing. We integrate *SEMu* in KLEE and experimented with Coreutils (a benchmark frequently used in symbolic execution studies). Our results show that our modelling plays an important role in mutant killing. Perhaps more importantly, our results also show that, within a two-hour time limit, *SEMu* kills 37% of the stubborn mutants, where KLEE kills none and where the mutant infection strategy (strategy suggested by previous research) kills 17%.

ACM Reference Format:
Thierry Titcheu Chekam, Mike Papadakis, Maxime Cordy, and Yves Le Traon. 2020. Killing Stubborn Mutants with Symbolic Execution. 1, 1 (September 2020), 23 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Thorough testing is often required in order to assess the core logic and the 'critical' parts of the programs under analysis. Unfortunately, performing thorough testing is hard, tedious and time consuming.

To support thorough testing, mutation testing aims at guiding the design of test cases that are likely fault revealing. The key idea of mutation is to use artificially introduced defects, called mutations, to identify untested (or weakly tested) cases and to guide test generation. Thus, testers can improve their test suites by designing mutation-based test cases, i.e., test that reveal the artificially introduced defects.

The mutation testing practice has shown that it is relatively easy to detect a large number of mutants by simply covering the mutated statements [2, 25, 33]. Such trivial mutants are not useful as they fail to provide any particular guidance towards test case design [35]. However, practical experience has also shown that there are some few mutants that are relatively hard to detect (a.k.a. stubborn mutants [41]) and can provide significant advantages when used as test objectives [33, 41]. Interestingly, these mutants form special corner cases that when tested often reveal faults [39?

Authors' addresses: Thierry Titcheu Chekam, thierry.titcheu-chekam@uni.lu, SnT, University of Luxembourg, Luxembourg; Mike Papadakis, michail.papadakis@uni.lu, SnT, University of Luxembourg, Luxembourg; Maxime Cordy, maxime.cordy@uni.lu, SnT, University of Luxembourg, Luxembourg; Yves Le Traon, yves.letraon@uni.lu, SnT, University of Luxembourg, Luxembourg.

XXXX-XXXX/2020/9-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

]. The importance of using the stubborn mutants as test objectives has also been underlined by several industrial studies [6, 9] including a large study with Google developers [33].

Stubborn mutants are hard to detect mainly due to a) the difficulty of infecting the program state (causing an erroneous program state when executing the mutated/defective point) and b) due to the masking effects that prohibit the propagation of erroneous states to the program output (aka failed error propagation [5] or coincidental correctness [1]). Either being the case, the issues linked with these mutants form corner cases which are most likely to escape testing (since stubborn mutants form small semantic deviations) [39].

Killing stubborn mutants (designing test cases that reveal undetected mutants) is challenging due to the large number of potential program execution paths, constraints and data states of the program versions (original and mutant versions) that need to be differentially analysed. The key challenge here regards the handling of the failed error propagation (masking effects), which is prevalent in stubborn mutants. Effective error propagation analysis is still an open problem [27, 34] as it involves state comparisons among the mutant and the original program executions that grow exponentially with the number of the involved paths (from the mutation point to the program output).

Many techniques targeting mutation-based test generation have been proposed [4, 27, 36]. Most of these techniques focus on generating unit-level test suites from scratch, mainly by either covering the mutated point or by causing an erroneous program state at the mutation point. Interestingly, there is no work leveraging the value of existing tests to perform thorough testing by targeting stubborn mutants, which are mostly hard to propagate. Moreover, none of the available symbolic execution tools generate test inputs by targeting the strongly killing of mutants[1].

We present *SEMu*, an approach based on dynamic symbolic execution that generates test inputs capable of killing stubborn mutants. The particular focus of *SEMu* is on the effective and scalable handling of mutant propagation. Our technique executes both the original and mutant program versions with a single symbolic execution instance, where the mutant executions are "forked" when reaching the mutation points. The forked execution follows the original one and compares with it. The comparisons are performed based on the involved symbolic states and related (propagation) constraints that ensure execution divergences that are probably leading to divergent behaviours.

A key issue with both symbolic execution and mutation testing regards their scalability. To account for this problem, we develop a framework that allows defining the mutant killing problem as a search problem within a specific area around the mutation points. This area is defined by a number of parameters that control the symbolic exploration. We thus, perform a constrained symbolic exploration, starting from a pre-mutation point (a point in the symbolic tree that is before the mutation point) and ending at a post-mutation checkpoint (a point after the mutation point) where we differentially compare the symbolic states of the two executions (forked and original) and generate test inputs.

We assume the availability of program inputs that can reach the areas we are targeting. Based on these inputs, we infer preconditions (a set of consistent and simplified path conditions), which we use to constrain the symbolic exploration to only a subset of program paths that are relevant to the targeted mutants. To further restrict the exploration to a relevant area, we systematically analyse the symbolic tree up to a relatively small distance from the mutation point (performing a shallow propagation analysis).

To improve the chances for propagation we also perform a deep exploration of some subtrees. Overall, by controlling the above parameters we can define strategies with trade-offs between

[1] A list of test generation techniques can be found in the related surveys of mutation testing [27, 36]

depth and deepness. Such strategies allow the differential exploration of promising code areas, while keeping their execution time low.

We integrate *SEMu*[2] into KLEE [7] and evaluate it on 47 programs from Coreutils, real-world utility programs written in C. We also compare *SEMu* with the mutant infection strategy, denoted as infection-only, that was proposed by previous work [15, 42]. Our results show that *SEMu* achieves significantly higher killing rates (approximately +37% and +20%) of stubborn mutants, for both KLEE (alone) and infection-only strategy, on the majority of the studied subjects.

In summary, our paper makes the following contributions:

(1) We introduce and implement a symbolic execution technique for generating tests that kill stubborn mutants. Our technique leverages existing tests in order to perform a deep and targeted test of specific code areas.
(2) We model the mutant killing as a search problem within a specific area (around the mutation point). Such a modelling allows controlling the symbolic execution cost, while at the same time allows forming cost-effective heuristics.
(3) We report empirical results demonstrating that *SEMu* has a strong mutant killing ability, which is significantly superior to KLEE and other mutation-based approaches.

The paper is organized as follows. Section 2 presents the targeted problem, the working scenario of our work, the symbolic program repesentation, used through the paper and an overview of symbolic execution, as implemented in our work. Section 3 provides our modelling approach of the problem of killing mutants, where exhaustive exploration, conservative search space pruning, and heuristic search are presented. Section 4 presents the cost-control heuristics used to form our heuristic search-based mutant killing modelling approach. The empirical evaluation of our approach, including the reserach questions, experimental setup and procedure, employed tools and subjects, is presented in Section 5. The results of the empirical evaluation are presented in Section 6. The related work is discussed in Section 7. Section 8 concludes this work.

2 CONTEXT

Our work aims at the automatic test input generation for selected methods/components of the systems under test. In particular, our working scenario assumes that testers have performed some basic testing and want to dig into some specific parts of the program. This is a frequent scenario used to increase confidence in the critical code parts (encode the core program logic) or on parts that testers feel uncertain. To do so, it is reasonable to use mutation testing by adding tests that detect the surviving mutants (mutants undetected by the existing test suite) [35, 41].

We consider a mutant as detected (killed) by a test when its execution leads to a different observable output from that off the original program. According to our scenario, the targeted mutants are those (killable) that survive a reasonable amount of testing. This definition depends on the amount of the performed testing; strong test suites kill more mutants than weak ones, while 'adequate' test suites kill them all [3, 41].

To adopt a baseline for basic or 'reasonable amount of testing' we augment the developer test suites with KLEE. This means that the *stubborn mutants are those that are killable and survive the developer and coverage-guided automatically generated test suites.* The surviving mutants form the objectives for our test generation technique.

2.1 Symbolic Encoding of Programs

Independently of its language, we define a program as follows.

[2] Publicly available at https://github.com/thierry-tct/KLEE-SEMu.

Definition 2.1. A program is a Labeled Transition System (LTS) $\mathcal{P} = (C, c_0, C_{out}, V, eval_0, T)$ where:

- C is a finite set of control locations;
- $c_0 \in C$ is the unique entry point (start) of the program;
- $C_{out} \subset C$ is the set of terminal locations of the program;
- V is a finite set of variables;
- $eval_0$ is a predicate capturing the set of possible initial valuations of V;
- $T : C \times GC \rightarrow C$ is a deterministic transition function where each transition is labeled with a guarded command of the form $[g]f$ where g is a guard condition (i.e., a formula in first-order logic) over V and f is a function updating valuation of variables V. GC denotes the set of labels of the transition system, that is, the set of guarded commands over V. Thus, $(c_i, [g_i]f_i, c_{i+1})$ means that the program execution can move from location c_i to location c_{i+1} if condition g_i is satisfied. When it does, the program updates the variables' value according fo f_i.

The LTS modelling a given program defines the set of control paths from c_0 to any $c_{out} \in C_{out}$. A path is a sequence of n connected transitions $\pi_P = \langle (c_0, gc_0, c_1), \ldots, (c_{n-1}, gc_{n-1}, c_{n=out}) \rangle$ such that $(c_i, gc_i, c_{i+1}) \in T$ for all i. Any well-terminating execution of the program goes through one such path. Since we consider deterministic programs, this path is unique and determined by the initial valuation, i.e., the test input, v_0 of the variables V. More precisely, each path π_P defines a *path condition* $\phi(\pi_P)$ which symbolically encodes all executions going through π_P. This path condition consists of a Boolean formula such that the test with input v_0 executes through π_P iff $v_0 \models \phi(\pi_P)$. By solving $\phi(\pi_P)$ (e.g. with a constraint solver like Z3 [8]), one can obtain an initial valuation satisfying the path condition, thereby obtaining a test input that goes through the corresponding program path. The execution of the program, with the resulting test input, is a sequence of $n + 1$ couples of variable valuations and locations, noted $\tau_{(P,v_0)} = \langle (v_0, c_0), \ldots, (v_{n-1}, c_{n-1}), (v_{n=out}, c_{n=out}) \rangle$, such that $v_0 \models eval_0$ and for all i, $v_i \models g_i$ and $v_{i+1} = f_i(v_i)$. While v_{out} is the valuation of all variables when $\tau_{(P,v_0)}$ terminates, the observable result of $\tau_{(P,v_0)}$ (its *output*), noted $Out(\tau_{(P,v_0)})$, is the subset of v_{out} restricted only to all observable variables. Since a path π encompasses a set of executions, we can also represent the set of outputs of those executions into a symbolic formula $Out(\pi)$.

2.2 Symbolic Encoding of Mutants

A mutation alters or deletes a statement of the original program $\mathcal{P}$. Thus, a mutant results from changing the transitions of $\mathcal{P}$ that correspond to that statement, i.e., two transitions for branching statements; one for the others.

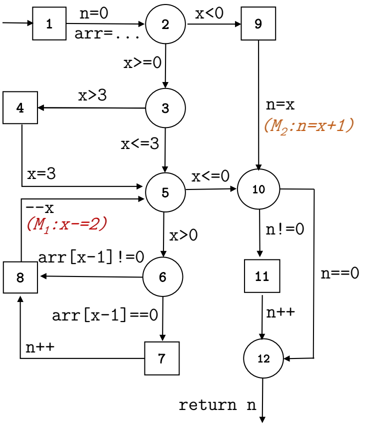
Definition 2.2. Let $\mathcal{P} = (C, c_0, V, eval_0, T)$ be an original program. A mutant of $\mathcal{P}$ is a program $\mathcal{M} = (C, c_0, V, eval_0, T')$ with $T' = (T \backslash T_m) \cup T'_m$ such that:

$$\begin{cases} T_m \subseteq T \wedge |T_m| > 0 \\ \forall (c_1, [g']f', c'_2) \in T'_m, \exists (c_1, [g]f, c_2) \in T_m : ([g']f' \neq [g]f) \vee (c'_2 \neq c_2)\} \end{cases}$$

Here, T_m is the subset of the transitions of $\mathcal{P}$ that are mutated to create $\mathcal{M}$, and T'_m is the subset of transitions of $\mathcal{M}$ that result from the mutation of T_m . Thus, $\mathcal{M}$ is created by replacing T_m, in $\mathcal{P}$, by T'_m.

It may happen that a program mutation leads to an *equivalent* mutant, i.e., semantically equivalent to the original program, that is, for any test input t, $Out(\tau_{(P,v_0)}) \equiv Out(\tau_{(M,v_0)})$. All non-equivalent mutants, however, should be discriminated, i.e., *killed*, by at least one test input. Thus, there must exist a test input t that satisfies the following three conditions (referred to as Reach, Infect,

```
int func (int x) {
1.    int n = 0, arr[] = {0,5,0};
2.    if (x >= 0) {
3.        if (x > 3)
4.            x = 3;
5.        while (x > 0) {
6.            if (arr[x-1] == 0)
7.                n++;
8.            --x; // M1 (x -= 2;)
          }
      } else {
9.        n = x; // M2 (n = x+1;)
      }
10.   if (n)
11.       n++;
12.   return n;
}
```



t1 (x = 2)		t2 (x = -1)	
Orig.	M1	Orig.	M2
1	1	1	1
2	2	2	2
3	3	9	9
5	5	10	10
6	6	11	12
8	8	12	
5	5		
6	10		
7	12		
8			
5			
10			
11			
12			
Out 2	0	0	0

Fig. 1. Example. The rounded control locations represent conditionals (at least 2 possible transition from them).

Propagate RIP [3, 10, 21]): the execution of M with t must (i) Reach a mutated transition, (ii) Infect (cause a difference in) the internal program state, i.e., change the variable valuations or the reached control locations, (iii) Propagate this difference up to the program outputs. In the remainder of the paper we state that a test reaches a mutant if the test satisfies condition (i), a test performs a mutant infection if it satisfies condition (ii) and a test causes mutant propagation if it satisfies condition (iii). One can encode those conditions as the symbolic formula: $kill(P, M) \triangleq \exists \pi_P, \pi_M : \phi(\pi_P) \wedge \phi(\pi_M) \wedge (Out(\pi_P) \not\equiv Out(\pi_M))$. Any valuation satisfying this formula forms a test input killing M. For given π_P and π_M, $kill(\pi_P, \pi_M) \triangleq \phi(\pi_P) \wedge \phi(\pi_M) \wedge (Out(\pi_P) \not\equiv Out(\pi_M))$ denotes the formula encoding the test inputs that kill M and go through π_P and π_M in P and M, respectively.

Definition 2.3. Let P be an original program and $M_1, \ldots, M_n$ be a set of mutants of P. Then the **mutant killing problem** is the problem of finding, for each mutant M_i:

(1) two paths π_P and π_{M_i} such that $kill(\pi_P, \pi_{M_i})$ is satisfiable;
(2) a test input t satisfying $kill(\pi_P, \pi_{M_i})$.

2.3 Example

Figure 1 shows a simple C program. The corresponding C code and transition system are shown in the left and middle of Figure 1, respectively. The transition system does not show the guarded commands for readability. The right side of Figure 1 shows two test inputs and their corresponding traces (as sequences of control locations of the transition system). The transition system contains 12 control locations, corresponding to the 12 numbered lines in the program. The squared nodes of the transition system represent the non-branching control locations and the circular nodes represent the branching control location. For simplicity, we assume that each line is atomic. The initial condition $eval_0$ is $x \in Int$ where Int is the set of all integers. Two mutants M_1 and M_2 are generated by mutating statements 8 and 9, respectively. M_1 results from changing the statement "$--x$" into "$x- = 2$" and M_2 results from changing the statement "$n = x$" into "$n = x + 1$". The mutants M_1 and M_2 result from the mutation of the guarded command of the transitions 8 → 5 and 9 → 10, respectively.

The test execution of t_1 reaches M_1 but not M_2, while t_2 reaches M_2 but not M_1. Test t_1 infects M_1 and t_2 infects M_2. The test execution of t_1 on the original program and mutant M_1 return 2 and 0, respectively. The mutant M_1 is killed by t_1 because $2 \neq 0$. Similarly, the test execution of t_2 on the original program and mutant M_2 return 0 and 0, respectively. Test t_2 does not kill mutant M_2.

2.4 Symbolic Execution

One can apply symbolic execution to explore the different paths, using a symbolic representation of the input domain (as opposed to concrete values) and building progressively the path conditions of the explored paths. The symbolic execution starts by setting an initial path condition to $\phi = True$. At each location, it evaluates (by calling a dedicated solver) the guarded command of any outgoing transition. If the conjunction of the guard condition and ϕ is satisfiable then there exists at least one concrete execution that can go through the current path and the considered transition. In this case, the symbolic execution reaches the target location and ϕ is updated by injecting into it the guarded command of the transition. This procedure enables the symbolic execution to discard infeasible paths (when the conjunction of the guard command and ϕ is unsatisfiable) without exploring them, thus, removing their negative impact on the symbolic exploration. When multiple transitions are available, the symbolic execution successively chooses one and pursues the exploration, e.g., in a breadth-first manner.

As the symbolic execution progresses, it explores additional paths. The explored paths can together be concisely represented as a tree [16] where each node is an execution state $\langle \phi, \sigma \rangle$ made of its path condition ϕ and symbolic program state σ (itself constituted by the current control location – program counter value – and the current symbolic valuation of variables). The path condition ϕ is dynamically constructed through the dynamic execution, i.e., by using the symbolic inputs and states on every program predicate.

Still, the tree remains too large to be explored exhaustively. Thus, one typically guides the symbolic execution to restrict the paths to explore, effectively cutting branches of the tree. Preconditioned symbolic execution attempts to reduce the path exploration space by setting the initial path condition (at the beginning of the symbolic execution) to a specific condition. This precondition restricts the symbolic execution to the subset of paths that are feasible given the precondition. The idea is to derive the preconditions from pre-existing tests (aka *seeds* in the KLEE platform) that reach the particular points of interests. This allows us to provide vital guidance towards reaching the areas that should be explored symbolically, while drastically reducing the search space. In the rest of the paper, we refer to a *preconditioned symbolic execution that explores the paths followed by some concrete executions as "seeded symbolic execution".*

Overall, one could make the following steps to generate test inputs for a program P via symbolic execution:

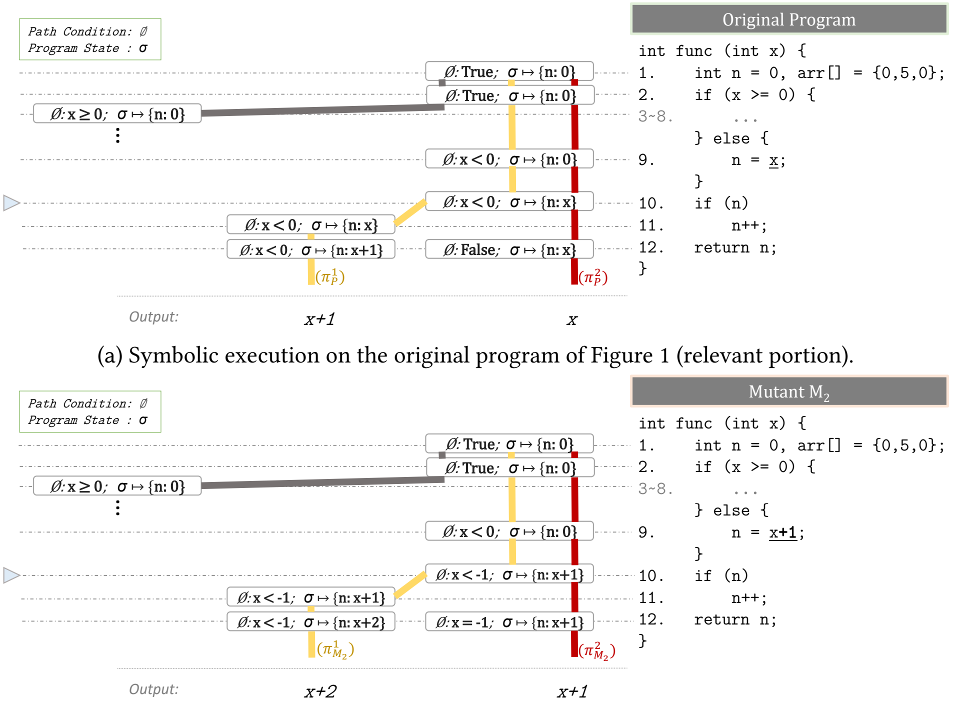
(1) **Precondition**: specify a logical formula over the program inputs (computed as the disjunction of the path conditions of the paths followed by the executions of the seeds) to prune out the paths that are irrelevant to the analysis.
(2) **Path exploration**: explore a subset of the paths of P, effectively discarding infeasible paths.
(3) **Test input generation**: for each feasible path π_P, solve $\phi(\pi_P)$ to generate a test input t that executes $\tau_{(P,t)}$ following π_P.

3 KILLING MUTANTS

3.1 Exhaustive Exploration

A direct way to generate test inputs killing some given mutants (of program P) is to apply symbolic execution on both P and the mutants, thereby obtaining their respective set of (symbolic) paths. Then, we can solve $kill(\pi_P, \pi_{M_i})$ to generate a test input that kills mutant M_i and goes through π_P in P and through π_{M_i} in M_i.

Figure 2 illustrates the use of symbolic execution to kill mutant M_2 of Figure 1. The Subfigure 2 (a) represents the symbolic execution of the original program, and the Subfigure 2 (b) represents the symbolic execution of the mutant M_2. We skip the symbolic execution subtree rooted at control



```
int func (int x) {
1.    int n = 0, arr[] = {0,5,0};
2.    if (x >= 0) {
3~8.      ...
      } else {
9.        n = x;
      }
10.   if (n)
11.       n++;
12.   return n;
}
```

(a) Symbolic execution on the original program of Figure 1 (relevant portion).

```
int func (int x) {
1.    int n = 0, arr[] = {0,5,0};
2.    if (x >= 0) {
3~8.      ...
      } else {
9.        n = x+1;
      }
10.   if (n)
11.       n++;
12.   return n;
}
```

(b) Symbolic execution on the mutant M_2 of Figure 1 (relevant portion).

Fig. 2. Example program ilustrating the symbolic execution to generate test to kill a mutant. The Subfigure (a) and (b) represent the original program and a mutant, respectively.

location 3 since the corresponding paths do not reach mutant M_2 and can easily be pruned using static analysis. Also, we do not represent the symbolic variables arr and x, which are not updated in this example. The symbolic execution on the original program leads to the paths π_P^1 and π_P^2 such that $\phi(\pi_P^1) \equiv (x < 0)$, $\phi(\pi_P^2) \equiv False$, $Out(\pi_P^1) \equiv x + 1$ and $Out(\pi_P^2) \equiv x$. The symbolic execution on the mutant M_2 leads to the paths $\pi_{M_2}^1$ and $\pi_{M_2}^2$ such that $\phi(\pi_{M_2}^1) \equiv (x < -1)$ and $\phi(\pi_{M_2}^2) \equiv (x = -1)$, and $Out(\pi_{M_2}^1) \equiv (x + 2)$ and $Out(\pi_{M_2}^2) \equiv (x + 1)$. For easier visualization, Figure 3 illustrates a side-by-side view of the symbolic executions represented in Figure 2.

The test generation that targets mutant M_2 solves the following formulae:

(1) $kill(\pi_P^1, \pi_{M_2}^1) \equiv ((x < 0) \wedge (x < -1) \wedge (x + 1 \neq x + 2))$. Satisfiable: example solution is $x = -2$.
(2) $kill(\pi_P^1, \pi_{M_2}^2) \equiv ((x < 0) \wedge (x = -1) \wedge (x + 1 \neq x + 1))$. Unsatisfiable: no possible output difference.
(3) $kill(\pi_P^2, \pi_{M_2}^1) \equiv (False) \wedge (x < -1) \wedge (x \neq x + 2))$. Unsatisfiable: infeasible path (π_P^2).
(4) $kill(\pi_P^2, \pi_{M_2}^2) \equiv (False) \wedge (x = -1) \wedge (x \neq x + 1))$. Unsatisfiable: infeasible path (π_P^2).

This method effectively generates tests to kill mutants. However, it requires a complete symbolic execution on P and on each mutant M_i. This implies that (i) all the path conditions and symbolic outputs have to be stored and analysed, and (ii) $kill(\pi_P, \pi_{M_i})$ has to be solved possibly for each pair of paths (π_P, π_{M_i}). This leads to large computational cost that makes the approach impractical.

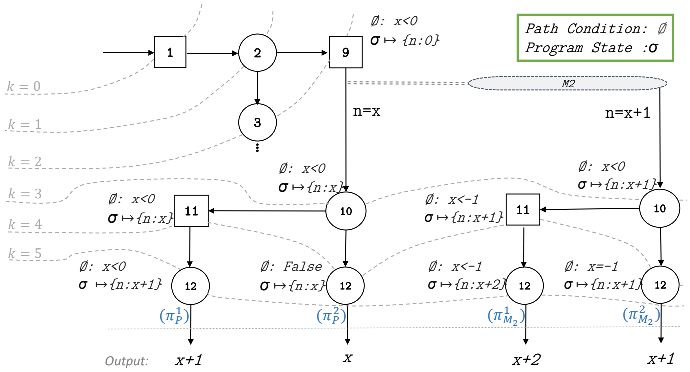


Fig. 3. Example of Symbolic execution for mutant test generation. After control location 9, the symbolic execution on the original program contains transition 9 ⟶ 10 with $n = x$ while the symbolic execution of the mutant M_2 contains transition 9 ⟶ 10 with $n = x + 1$.

3.2 Conservative Pruning of the Search Space

To reduce the computational demands induced by the exhaustive exploration, we apply two safe optimizations (preserve all opportunities to kill the mutants) that avoids exploring program paths that are not promising. We take advantage of the fact that mutants, which are results of simple syntactic alterations, share a large portion of their code with the original program. This is recommended by other studies [22] in the context of specification-based testing, creating mutants from specifications or system models. Though, here we aim at source code, which involves a lower level representation and execution.

3.2.1 Meta-mutation. Our first optimization stems from the observation that all paths and path prefixes of the original program P that do not include a mutated statement, i.e., location whose outgoing transitions have changed in the mutants, also belong to the mutants. Thus, the symbolic execution of P and that of the mutants may explore a significant number of identical path prefixes. As seen in Figure 3, the symbolic execution is identical for the original and mutant M_2 up to control location 9. Instead of making two separate symbolic executions, *SEMu* performs a shared symbolic execution based on a meta-mutant program. A meta-mutant [28, 29, 40] represents all mutants in a single code. A branching statement (named *mutant choice statement*) is inserted at each mutation point and controls, based on the value of a special global variable (the mutant ID), the execution of the original and mutant programs.

The symbolic execution on the meta-mutant program initialises the mutant ID to an unknown value and explores a path normally until it encounters a mutant choice statement. Then, the path is duplicated once for the original program and once for each mutant, with the mutant ID set to the corresponding value, and each duplicated path is further explored normally. While the effect of this optimization is limited to the prefixes common to the program and all mutants, it reduces the overall cost of exploration at insignificant computation costs and without compromising the results.

3.2.2 Discarding non-infected mutant paths. In practice, many execution paths reach a mutant (cover the mutation point) but fail to infect the program state (introducing an erroneous program state). Extending the execution along such paths is a waste of effort as the mutant will not be killed along those paths. Thus, *SEMu* terminates anticipatively the exploration of any path that reaches the mutant but fails to infect the program state. This procedure automatically stops the exploration of mutant paths for equivalent mutants that cannot infect program states. Regarding equivalent mutants that can infect the program state but never propagate the infection to the output, this procedure cannot discard them.

3.3 Heuristic Search

Even with the aforementioned optimizations, the exhaustive exploration procedure remains too costly due to two factors: the size of the tree to explore and the number of couples of paths π_P and π_{M_i} to consider. To speed up the analysis, one can further prune the search space, at the risk of generating useless test inputs (that kill no mutant) or missing opportunities to kill mutants (by ignoring relevant paths).

A first family of heuristics reduce the number of paths to explore by selecting and prioritizing them, at the risk of discarding paths that would lead to killing mutants. A second family stop exploring a path after k transitions and solve, instead of $kill(\pi_P, \pi_{M_i})$, the formula

$$partialKill(\pi_P[..k], \pi_{M_i}[..k]) \;\triangleq\; \phi(\pi_P[..k]) \wedge \phi(\pi_{M_i}[..k]) \wedge (\sigma(\pi_P[..k]) \;\not\equiv\; \sigma(\pi_{M_i}[..k]))$$

where, for any path π, $\pi[..k]$ denotes the prefix of π of length k and where $\sigma(\pi[..k])$ is the symbolic state reached after executing $\pi[..k]$. It holds that $kill(\pi_P, \pi_{M_i}) \Rightarrow \exists k : partialKill(\pi_P[..k], \pi_{M_i}[..k])$, since a mutation cannot propagate to the output of the program if it does not infect the program in the first place. The converse does not hold, though: statements after a mutation can cancel the effects of an infection, rendering the output unchanged at the end of the execution. The problem then boils down to selecting an appropriate length k where to stop the exploration, so as to maximize the chances of finding an infection that propagates up to the observable outputs.

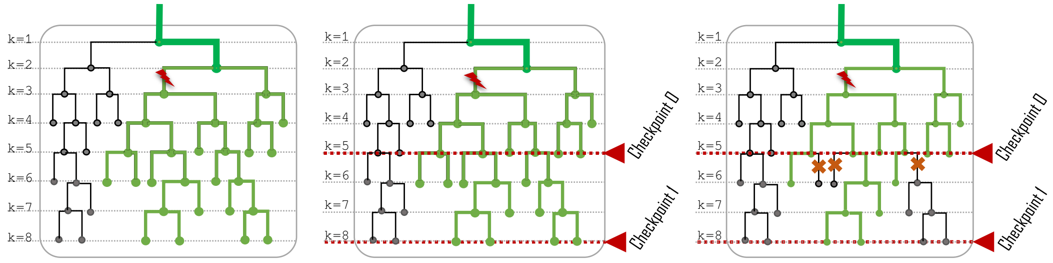
Regarding equivalent mutants, any mutant that cannot infect the program state is discarded during the conservative pruning of the search space (Section 3.2.2). Mutants that can infect the program state, but without propagating the infection to the output, are treated like killable mutants. Only note that, in this case, the appropriate length k represents the point where the effects of all potential infections are canceled.

As illustrated in Figure 3, generating a test at $k = 3$ (control location 10 and line number 10 in Figures 2*a* and 2*b*), requires to solve the constraint $partialKill(\pi_P^1[..3], \pi_{M_2}^1[..3]) \equiv (x < 0 \wedge x \neq x{+}1)$. The constraint solver may return $x = -1$ which does not propagate the infection to the output. However, generating a test at $k = 5$ (control location 12 and line 12 in Figures 1*a* and 2*b*), using the original path π_P^1 and mutant path $\pi_{M_2}^1$, requires to solve the constraint $x < 0 \wedge x < -1 \wedge (x{+}1 \neq x{+}2)$. Any value returned by the constraint solver kills the mutant.

An ideal method to kill a mutant M would explore only one path π_P and one path π_M, and up to the smallest prefix length k where the constraint solver can generate a test that kills M. However, identifying the right π_M and the optimal k is hard, as it requires precisely capturing the program semantics. To overcome this difficulty, *SEMu* defines heuristics to prune non-promising paths on the fly and to control at what point (what prefix length k) to call the constraint solver. Once path prefix candidates are identified, *SEMu* invokes the solver to solve $partialKill(\pi_P[..k], \pi_M[..k])$.

3.4 Infection-only Strategy

The infection-only strategy generates tests by aiming at mutant infection (without any exploration for propagation). To generate test inputs infecting some given mutants (of program P), the infection-only strategy applies symbolic execution on both P and the mutants, thereby obtaining their respective set of (symbolic) paths. Then, for each mutant M_i (linked with mutated statement s_{M_i}), it solves the infection condition $partialKill(\pi_P[..k_P], \pi_{M_i}[..k_{M_i}])$ to generate a test input that infects



(a) (b) (c)

Fig. 4. Illustration of *SEMu* cost-control parameters. Subfigure (a) illustrates the Precondition Length where the green subtree represents the candidate paths constrained by the precondition (the thick green path prefix is explored using seeded symbolic execution). Subfigure (b) illustrates the Checkpoint Window (here CW is 2). Subfigure (c) illustrates the Propagation Proportion (here PP is 0.5) and the Minimum Propagation Depth (here if MPD is 1 the first test is generated, for unterminated paths, from *Checkpoint 1*).

mutant M_i and goes through π_P in P and through π_{M_i} in M_i. Here k_P is the first occurrence of s_{M_i} in π_P and k_{M_i} is the first occurrence of s_{M_i} in π_{M_i}.

As illustrated in Figure 3, generating a test using the infection-only strategy, requires to solve the constraint $partialKill(\pi_P^1[..3], \pi_{M_2}^1[..3]) \equiv (x < 0 \land x \neq x + 1)$. While such an approach can be effective [28], it has limitations as the constraint solver does not consider failed propagation. For instance, in this case, the constraint solver may return $x = -1$ which infect the mutant M_2 but does not propagate this infection to the output.

4 *SEMU* COST-CONTROL HEURISTICS

SEMu consists of parametric heuristics to control the symbolic exploration of promising code regions. Any configuration of *SEMu* sets the parameters of the heuristics, which together define which paths to explore and the test generation process. *SEMu* also takes as inputs the original program, the mutants to kill and a set of pre-existing test inputs to drive the seeded symbolic execution. During the symbolic exploration, *SEMu* selects which paths to explore and when to stop the exploration to generate test inputs based on the obtained path prefix.

4.1 Pre Mutation Point: Controlling for Reachability

To improve the efficiency of the path exploration, it is important to quickly prune paths that are infeasible (cannot be executed) or irrelevant (cannot reach the mutants, i.e., the execution of the paths does not result in executing the mutated statements). To achieve this, we leverage seeded symbolic execution (as implemented in KLEE) where the seeds are pre-existing tests. We distinguish between seeds and tests in order to denote the starting points of the symbolic exploration (the paths from which we deviate) and the tests that are included in the test suites, respectively. We proceed in two steps. First, we explore the paths in seeded mode up to a given length (precondition length). Then, we stop following the seeds' executions and switch to a non-seeded symbolic execution. The location of the switching point thus determines where the exploration stops using the precondition. In particular, if it is set to the entry point of the program then the execution is equivalent to a full non-seeded symbolic execution. If it is set beyond the output then it is equivalent to a fully seeded symbolic execution. Formally, let Π denote the complete set of paths of a program P, $\{t_1, \ldots, t_n\}$ be the set of seeds, and l be the chosen precondition length. Then the sets of explored paths resulting from the seeded symbolic execution of length l and with seeds $\{t_1, \ldots, t_n\}$ is the largest set $\Pi' \subseteq \Pi$ satisfying $\pi \in \Pi' \Rightarrow \exists t_i : t_i \models \phi(\pi[..l])$.

This heuristics is illustrated in Figure 4a where the thick (green) segments represent the portion of the tree explored by seeded symbolic execution and the subtree below (light green) represents the portion explored by non-seeded symbolic execution. The precondition leads to pruning the leftmost subtree.

Accordingly, the first parameter of *SEMu* controls the *precondition length (PL)* at which to stop the seeded symbolic execution. Instead of demanding a specific length l, the parameter can take two values reflecting two strategies to define l dynamically: *GMD2MS* (Global Minimum Distance to Mutated Statement) and *SMD2MS* (Specific Minimum Distance to Mutated Statement). When set to GMD2MS, the precondition length is defined, *for all explored paths*, as the length of the smallest path prefix that reaches a mutated statement. When set to SMD2MS, the precondition length is defined, *individually for each path* π, as the length l of the smallest prefix $\pi[..l]$ of this path that reaches a mutated statement.

4.2 Post Mutation Point: Controlling for Propagation

From the mutation point, all paths of the original program are explored. When it comes to a mutant, however, it happens that path prefixes that cover the mutation point and infect the program state fail to propagate the infection to the outputs. These prefixes should be discarded to reduce the search space. Accordingly, our next set of parameters control where to check that the propagation goes on, the number of paths to continue exploring from those checkpoints, and when to stop the exploration and generate test inputs. Overall, those parameters contribute to reducing the number of paths explored by the symbolic execution as well as the length k of the path prefixes from which tests are generated.

It is worth noting that the risk to discard killing paths depends on the set value of each parameter, as well as the program under analysis and the mutant considered. For instance, lets assume that, for a given program and mutant, the killing paths are evenly distributed. By randomly discarding 50% of the paths in order to reduce the search space, roughly 50% of the killing paths will be discarded (note that only one killing path is needed to generate test to kill a mutant).

4.2.1 Checkpoint Location. The first parameter is an integer named the *Checkpoint Window* (CW) which determines the location of the checkpoints. Any checkpoint is a program location with branching statements, i.e., transitions with guarded command $[g]f$ such that $g \neq True$, that is found after the mutation point. Then, the checkpoint window defines the number of branching statements (that are not checkpoints) between the mutation point and the first checkpoint, and between any two consecutive checkpoints. The effect of this parameter is illustrated in Figure 4b. The marked horizontal lines represent the checkpoints. In this case, the checkpoint window is set to 2, meaning that there are two branching statements between two checkpoints. At each checkpoint, *SEMu* can perform two actions: (1) discard some branches (path suffixes) of the current path prefix, by ignoring some of the branches, and (2) generate tests based on the current prefix. Whether and how those two actions are performed is determined according to the following parameters.

4.2.2 Path Selection. The parameter *Propagating Proportion* (PP) specifies the percentage of the branches that are kept to pursue the exploration, whereas the parameter *Propagation Selection Strategy* (PSS) determines the strategy used to select these branches. We implemented two strategies: random (RND) and Minimum Distance to Output (MDO). The first one simply selects the branches randomly with a uniform probability. The second one assigns a higher priority to the branches that can lead to the program output more rapidly, i.e., by executing fewer statements. This distance is estimated statically based on the control flow and call graphs of the program. More specifically, for each target branch, we compute the minimum distance, on the program control flow graph, from the target branch to all output system calls statements (*printf*, *puts*, ... function calls). The branches with smaller distances are selected by the MDO strategy. Note that the minimum distances are (pre-)computed once and cached, at the start of the execution, through a reverse breadth first search that starts from all output system call statements, and sets the minimum distances of each visited branch to its depth level. This pre-computation has similar complexity with a breadth-first search traversal, thus, does not incur a significant overhead on *SEMu*. The two parameters are illustrated in Figure 4c, where the crossed subtrees represent branches pruned at Checkpoint 0.

4.2.3 Early Test Generation. Generating test inputs before the end of the symbolic execution (on the path prefixes) allows us to reduce its computation cost. Being placed after the mutation point, all checkpoints are potential places where to trigger the test generation. However, generating sooner, on the one hand, reduces the chances of seeing the infection propagate to the program output, in the case of a killable mutant. On the other hand, it also increases the chances to generate a (spurious) test, based on an infection that cannot propagate to the output, in the case of an equivalent mutant. To alleviate this risk, we introduce the parameter *Minimum Propagation Depth* (MDP), which specifies the number of checkpoints that the execution must pass through before starting to generate tests. In Figure 4c, if MDP is set to 1 then tests are generated from Checkpoint 1 (for the two remaining paths prefixes). Note that in case MDP is set to 0, tests are generated for the crossed (pruned) path prefixes at Checkpoint 0.

4.3 Controlling the Cost of Constraint Solving

Remember that $partialKill$ requires the state of the original program and the mutant to be different. The subformulae representing the symbolic program states can be large and/or complex, which may hinder the performance of the invoked constraint solver. To reduce this cost, we devise a parameter *No State Difference* (NSD) that determines whether to consider the program state differences when generating tests. When set to $True$, $partialKill(\pi_P[..k], \pi_M[..k])$ is reduced to $\phi(\pi_P[..k]) \wedge \phi(\pi_M[..k])$; however, its solution has lower chances of killing mutant M.

4.4 Controlling the Number of Attempts

It is usually sufficient to generate a single test that reaches the mutant to kill it. However, the stubborn mutants that we target may not be killed by the early attempts (applied closer to the mutation point) and require deeper analysis. Furthermore, a test generated to kill a mutant may collaterally kill another mutant. For those reasons, generating more than one test for a given mutant can be beneficial. Doing this, however, comes at higher test generation and test execution costs. To control this, we devise a parameter *Number of Tests Per Mutant* (NTPM) that specifies the number of tests generated for each mutant, i.e., the number of $partialKill$ formulas solved for each mutant.

5 EMPIRICAL EVALUATION

5.1 Research Questions

We first empirically evaluate the ability of *SEMu* to kill stubborn mutants. This is an essential question, since there is no point in evaluating *SEMu* if it cannot kill some of the targeted mutants.

RQ1 What is the ability of *SEMu* to kill stubborn mutants?

By answering this question we find a strong killing ability of *SEMu*. We, therefore, turn our attention to the question of whether the killing ability is due to the extended symbolic exploration that is anyway performed by KLEE. We thus, compare *SEMu* with KLEE by running KLEE in the seed mode (using the initial test suite as a seed for KLEE test generation) to generate additional tests. Such a comparison is also a first natural baseline to compare with. These motivate RQ2:

RQ2 How does *SEMu* compare with KLEE in terms of killed stubborn mutants?

Perhaps not surprisingly, we found that *SEMu* outperforms KLEE. This provides evidence that our dedicated mutation-based approach is indeed suitable for mutation-based test generation. At the same time though, our results raises further questions on whether the superior killing ability of *SEMu* is due to its ability for mutant infection (suggested by previous research) or due to its ability for mutant propagation (specific target of *SEMu*). In case we find that mutant infection is sufficient for killing stubborn mutants then mutant propagation should be skipped in order to save effort and resources. To investigate this, we ask:

RQ3 How does *SEMu* compare with the infection-only strategy in terms of killed stubborn mutants?

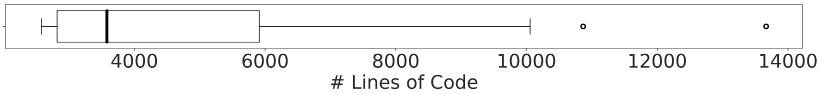
The infection-only strategy generates tests by aiming at mutant infection only (without any exploration for propagation). When the symbolic execution path exploration reaches a mutated statement having a mutation corresponding to mutant M, a test is generated by solving the mutant infection condition for M.

5.2 Test Subjects

To answer our research questions, we experimented with the C programs of GNU Coreutils[3] (version 8.22). GNU Coreutils is a collection of text, file, and shell utility programs widely used in unix systems. The whole codebase of Coreutils is made of more than 60,000 lines of C code[4].

The repository of Coreutils contains developer tests for the utilities programs which are system tests written in shell or perl scripts that involve more than 20,000 lines of code[4].

Applying mutation analysis on all Coreutils programs requires excessive amount of effort. Therefore, we randomly sampled 60 programs, based on which we performed our analysis. Unfortunately, in 13 of them mutation analysis took excessive computational time (due to costly test execution), for which we terminated the analysis. Therefore, we analysed 47 programs. These are: `base64`, `basename`, `chcon`, `chgrp`, `chmod`, `chown`, `chroot`, `cksum`, `date`, `df`, `dirname`, `echo`, `expr`, `factor`, `false`, `groups`, `join`, `link`, `logname`, `ls`, `md5sum`, `mkdir`, `mkfifo`, `mknod`, `mktemp`, `nproc`, `numfmt`, `pathchk`, `printf`, `pwd`, `realpath`, `rmdir`, `sha256sum`, `sha512sum`, `sleep`, `stdbuf`, `sum`, `sync`, `tee`, `touch`, `truncate`, `tty`, `uname`, `uptime`, `users`, `wc`, `whoami`. The following Figure presents the size of these subjects.



For each subject we selected the 3 functions that were covered by the largest number of developer tests (from the initial test suite).

5.3 Employed Tools

We implemented our approach on top of LLVM[5] using the symbolic virtual machine KLEE [7]. The version of our tool is based on the KLEE revision 74c6155, LLVM 3.4.2. Our implementation modified (or added) more than 8,000 lines of code on KLEE, and is publicly available[6]. We are

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

[4] Measured with cloc (http://cloc.sourceforge.net/)

[5] https://llvm.org/

[6] https://github.com/thierry-tct/KLEE-SEMu

planning to add support for newer versions of LLVM. To convert system tests into the format of seeds required by KLEE for the seeded symbolic execution, we use Shadow [24].

Our tool requires the targeted mutants to be represented in a meta-mutant program (presented in Section 3.2.1), which were produced using the *Mart* [38] mutant generation tool. *Mart* mutates a program by applying a set of mutation operators (code transformations) to the original LLVM bitcode program. An example of the mutation operator is to change the addition operation "+" into the substraction operation "-".

The mutants produced by the version of *Mart* used in this experiment are first-order mutants. However, our approach also support higher-order mutants. In the case of higher-order mutants, the meta-mutant program fed to *SEMu* is required to use the same mutant identifier for all sub-mutants of the higher-order mutant. When, applying *SEMu* on higher-order mutants, equivalent mutants can be detected and discarded by using existing higher-order equivalent mutant detection tools [13].

5.4 Experimental Setup

5.4.1 Selected Mutants. To perform our experiment we need to form our target mutant set. To do so, we employed *Mart* by using its default configuration and generated 172,919 mutants. This configuration generates a comprehensive set of mutants based on a large set of mutation operators, consisting of 816 code transformations. It is noted that the operator set includes the classical 5 operators [23] that are used by most of the todays' studies and mutation testing tools. The interested reader is referred to *Mart*'s paper for further details [38].

To identify the stubborn mutant set we started by eliminating trivial equivalent and duplicated mutants, and form our initial mutant set M_1. To do so, we applied Trivial Compiler Equivalence (TCE) [26], a technique that statically removes a large number of mutant equivalences. In our experiment, TCE removed a total number of 102,612 mutants as being equivalent or duplicated. This gave us 70,307 mutants to be used for our initial mutant set, i.e., M_1=70,307.

Then, we constructed our initial test suites TS (composed of the developer test suite and automatically generated tests by a simple test generation run of KLEE). To generate these tests with KLEE, we set a test generation timeout of 24 hours, while using the same configurations presented by the authors of KLEE [7] (except for larger memory limit and `max-instruction-time`, set to 9GB and 30s respectively). This run resulted in 5,161 tests (2,693 developer tests and 2,468 tests generated by the initial run of KLEE).

We then executed the initial test suites (TS) with the initial mutant set (M_1) and identified the live and killed mutants. The killed mutants were discarded, while the live ones formed our target mutant set (denoted it as M_2), i.e., M_2 is the target of *SEMu*. In our experiment we found that M_2 included 26,278 mutants, which is approximately 37% of M_1. It is noted that M_2 is a superset of the stubborn mutants as it includes both stubborn and equivalent mutants. Unfortunately, judging mutant equivalence is undecidable and thus, we cannot remove such mutants before test generation. Therefore, to preserve realistic settings we are forced to run *SEMu* on all M_2 mutants.

To evaluate *SEMu* effectiveness we need to measure the extent to which it can kill stubborn mutants. Unfortunately, M_2 contains a large proportion of equivalent mutants [35], which may result in significant underestimations of test effectiveness [18]. Additionally, M_2 may contain a large portion of subsumed mutants (mutants killed collaterally by tests designed to kill other mutants), which may inflate (overestimate) test effectiveness [25]. Although we discarded easy-to-kill mutants (mutants of M_1 that are killed by TS), it is still likely that a significant amount of 'noise' still remains.

To reduce such biases (both under and over estimations) [18, 25], there is a need to filter out the subsumed mutants by forming the subsuming mutant set [17, 27]. The subsuming mutants are mainly distinct (in the sense that killing one of them does not alter, increase or decrease, the chances of killing the others) providing objective estimations of test effectiveness. Unfortunately, identifying subsuming mutants is undecidable and thus, several testers, e.g., Ammann et al. [2], Papadakis et al. [25], Kurtz et al. [18] suggested approximating them through strong test suites. Therefore, to approximate them, we used the combined test suite that merges all tests generated by KLEE and *SEMu* across the execution of its 128 different configurations, $\bigcup_{i=0}^{n} TS_{x_i}$, where x_0 is KLEE and x_i $(0 < i \leq n)$ are the n *SEMu* configurations (refer to Section 5.4.2 for details). This process was applied on M_2 and resulted in a set of 529 mutants, forming the mutant set M_3. In the rest of the paper we call the mutants belonging to M_3 as reference mutants. We use M_3 for our effectiveness evaluation.

Overall, through our experiments we used two distinct mutant sets, M_2 and M_3. To preserve realistic settings, the former is used for test generation, while the later is used for test evaluation (to reduce bias).

5.4.2 SEMu *Configuration.* To specify relevant values for our modelling parameters (*SEMu* parameters) we performed ad-hoc exploratory analysis on some small program functions. Based on this analysis we specify 2 relevant values for each of the 7 parameters (defined in Section 4). These values provided us the basis for constructing a set of configurations (parameter combinations) to experiment with. In particular the values we used are the following: Precondition Length: *GMD2MS and SMD2MS*, Checkpoint Window: *0 and 3*, Propagating Proportion: *0 and 0.25*, Propagating Selection Strategy: *RND and MDO*, Minimum Propagation Depth: *0 and 2*, No State Difference: *True and False*, Number of Tests Per Mutant: *1 and 5*.

We then experiment with the constructed configurations in order to select the most prominent *SEMu* configuration and form our approach. It is noted that different values and combinations form different strategies. Examining them is a non-trivial task since the number of configurations is exponentially increased, i.e., $2^7 = 128$ and mutant execution takes considerable amount of time. In our study, the total test generation by the various configurations of *SEMu* and KLEE took roughly 276 CPU days (number of days for a single CPU single thread execution), while the execution of the mutants took approximately 1,400 CPU days.

To identify and select the most prominent configuration, we executed our framework on all test subjects under all constructed configurations. We restrict the symbolic execution time to 2 hours. We then randomly split the set of test subjects into 5 buckets of equal size (each one containing 20% of the test subjects). Then, we pick 4 buckets (80% of the test subjects) and select the best configuration by computing the ratio of killed reference mutants. We assess the generalization of this configuration on the left out bucket (5th bucket that includes 20% of the test subjects). To reduce the influence of random effects, we repeated this process 5 times by leaving every bucket out for evaluation. At the end we selected the median performing configuration (performance on the bucket that had been left out). It is noted that such a cross validation process is commonly used in order to select stable and potentially generalizable configurations.

Based on the above procedure we selected the *SEMu* configuration: PL = GMD2MS, CW = 0, $PP = 0.25$, PSS = RND, $MPD = 2$, $NSD = False$, $NTPM = 5$.

5.5 Experimental Settings and Procedure

To perform our experiment we set, on KLEE, the following (main) settings (which are similar to the default parameters of KLEE): a) we set a memory usage threshold of 8 GB, (a threshold never reached by any of the studied methods), b) we set the search strategy on Breadth-First Search (BFS), which is commonly used in patch testing studies [24] and c) we set a 2 hours time limit for each subject.

It is noted that our current implementation supports only BFS. We believe that such a strategy fits well with our purpose as it is important that the mutants and original program paths are explored in a lock step in order to enable state comparison at the same depth. The time budget of 2 hours was adopted because it is frequently used in test generation studies, e.g., [7]. It is noted that since *SEMu* performs a deeper analysis than the other methods, adopting a higher time limit would probably lead to an improved performance, compared to the other methods. Of course reducing this limit could lead to reduced performance.

We then evaluated the effectiveness of the generated test suites by computing the ratio of reference mutants that they kill. Unfortunately, in 11 among the 47 test subjects we considered, none of the evaluated techniques managed to kill any mutant. This means that for these 11 subjects we approximate having 0 stubborn mutants and thus, we discarded those programs. Therefore, the following results regard the 36 programs for which we could kill at least one stubborn mutant.

To answer RQ1 we compute and report the ratio of the reference mutants killed, i.e., M_3 set, by *SEMu* when it targets the 26,278 surviving mutants, i.e., M_2 set.

To answer RQs 2 and 3 we compute and contrast the ratio of the reference mutants killed by KLEE (executed in "seeding" mode), the infection-only strategy (a strategy suggested by previous research [15, 42]) and *SEMu* (for fair comparison, we used the initial test suite as seeds for the three approaches). We also report and contrast the number of mutant-killing tests that were generated. Since the generated tests may include large numbers of redundant tests, i.e., a test is redundant with respect to a set of tests when it does not kill any unique mutant compared to the mutants killed by the other tests in the set [27], we compare the sizes of non-redundant test sets, which we call mutant-killing test sets. The size of these sets represents the raw number of end objectives that were successfully met by the techniques [3, 27].

To compute the mutant-killing test sets we used a greedy heuristic. This heuristic incrementally selects the tests that kill the maximum number of mutants that were not killed by the previously selected tests.

5.6 Threats to Validity

All in all we targeted 133 functions from 47 programs from Coreutils. This level of evidence sufficiently demonstrates the potential of our approach, but should not be considered as a general assertion of its test effectiveness.

We generated tests at the system level (system tests), relying on the developers' tests suites. We believe that this is the major advantage of our approach because this way we focus on stubborn mutants that encode system level corner cases that are hard to reveal. Another benefit of doing so is that at this level we can reduce false alarms, experienced at unit level (feasible behaviors at unit but infeasible at system level), [14]. Unfortunately though, this could mean that our results do not necessarily extend to unit level.

Another issue may be due to the tools and frameworks we used. Potential defects and limitations of these tools could influence our observations. To reduce this threat we used established tools, i.e., KLEE and Mart, that have been used by many empirical studies. To reduce this threat further we also performed manual checks and made our tool publicly available.

In our evaluation we used the subsuming stubborn mutants in order to cater for any bias caused by trivial mutants [25]. While this practice follows the recommendations made by the mutation testing literature [27], the subsuming set of mutants is a subject to the combined reference test suite, which might not be representative to the input domain. Nevertheless, any issue caused by the above approximations could only reduce the mutant killed ratios and not the superiority of our method. Additional (future) experimentations will increase the generalizability of our conclusions.

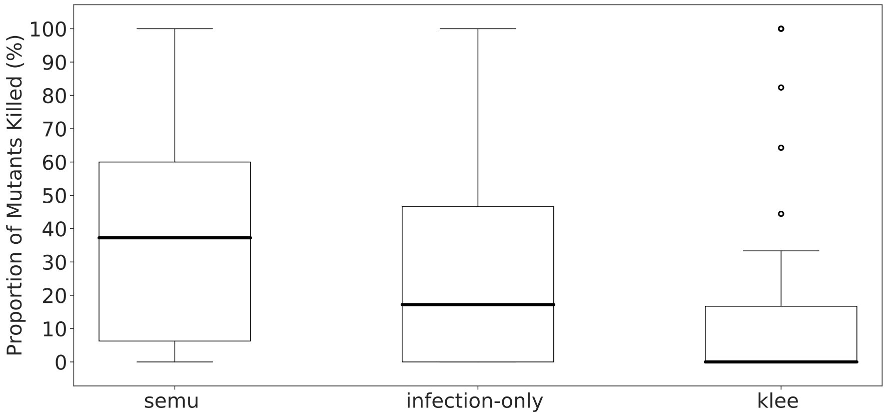


Fig. 5. Comparing the stubborn mutant killing ability of *SEMu*, KLEE and the *infection-only*.

The comparison between the studied methods (infection-only) was based on a time limit that did not include any actual mutant test execution time. This means that when reaching the time limit, we cannot know how successful (at mutant killing) the generated tests were. Additionally, we cannot perform test selection (eliminate ineffective tests) as this would require expensive mutant executions. While, it is likely that a tester would like to execute the mutants in order to perform test selection, leaving mutant execution out allows a fair comparison basis between the studied methods since mutant execution varies between the methods and heavily depends on test execution optimizations used [27]. Nevertheless, it is unlikely that including the mutant execution would change our results since *SEMu* generates less tests than the baselines (because it makes a deeper analysis than the baselines).

6 EMPIRICAL RESULTS

6.1 Killing ability of *SEMu*

To evaluate the effectiveness of *SEMu* we run it for 2 hours per subject program and collect the generated test inputs. We then execute the reference mutants with these inputs and determine the killed ones. Interestingly *SEMu* kills a large portion of the reference mutants. The median percentage of killed mutants is 37.3%, indicating a strong killing ability. To kill these mutants *SEMu* generated 153 mutant-killing test inputs (each test kills at least one mutant that is not killed by any other test).

6.2 Comparing *SEMu* with KLEE

Figure 5 records the proportion of the killed reference mutants by *SEMu*, seeded mode of KLEE and infection-only (investigated in RQ3). It is noted that the boxes include the proportions of killed mutants among the different test subjects we use. The thick horizontal line on the boxplots represents the median value of the proportion of killed mutants. It is computed from the proportion of killed mutants of all test subjects. From these results we can observe that *SEMu* has a median value of 37.3% while KLEE has a median value of 0.0%.

To further validate the difference we use the Wilcoxon statistical test (paired version) to check whether the differences are significant. The statistical test gives a p-value of 0.006 suggesting that the two samples' values are indeed significantly different. As statistical significance does not provide any information related to the volume of the difference, we also compute the Vargha

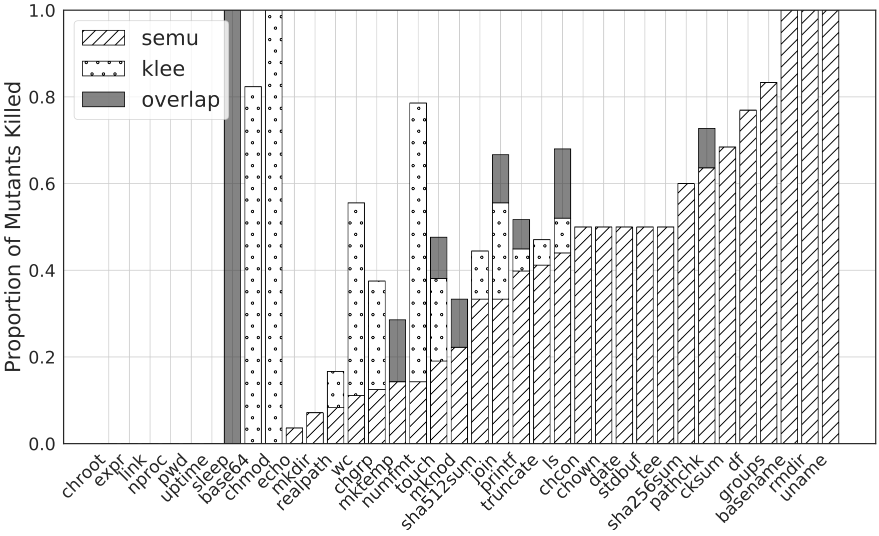


Fig. 6. Comparing the mutant killing ability of *SEMu* and KLEE in per program basis.

Delaney effect size ($\hat{A_{12}}$ value) that quantifies the frequency the observed difference. The results give a $\hat{A_{12}}$ of 0.736, which indicates that *SEMu* is superior to KLEE in 73.6% of the cases.

Figure 6 depicts the differences and overlap between the reference mutants killed by *SEMu* and KLEE, per studied subject. From this figure, we can observe that the number of programs with overlapping killed mutant is very small indicating that the two methods differ significantly. We also observe that *SEMu* performs best in the majority of the cases. Interestingly, a non negligible number of mutants are killed by KLEE only. These cases fall within a small number of test subjects. We investigated these cases and found that the differences were big either because there was only one reference mutant, which was killed by KLEE alone, or because of the large number of surviving mutants that force *SEMu* to perform a shallow search. Unfortunately, *SEMu* spends much time trying to kill every targeted mutant and thus, when a large number of them is involved, the 2 hours time limit we set is not sufficient to effectively kill them.

In fact, given a configuration of *SEMu*, the time budget needed to target all the mutants is proportional to the number of mutants. That is why, for a fixed time budget, the effectiveness of *SEMu* may decrease as the number of mutants increases, leaving some mutants untargeted when the time budget is exhausted. One way to mitigate this, when a large number of mutants is involved, perhaps is to prune the paths more aggressively or reduce the number of attempts for these cases. A future work will automatically set the appropriate parameters' values based on the program under analysis and the number of mutants, in a way that optimises the effectiveness of *SEMu*. This will enable each mutant to have a share of the time budget.

To better demonstrate the effectiveness differences of the methods we also record the number of the mutant killing test inputs (each test kills at least one mutant that is not killed by any other test). We found that *SEMu* generated 153 mutant-killing test inputs, while KLEE generated only 62.

6.3 Comparing *SEMu* with infection-only

A first comparison between *SEMu* and *infection-only* can be made based on the data from Figure 5. According to these data *SEMu* has a median value of 37.3% while *infection-only* has a median of 17.2%. Interestingly, this shows a big difference in favour of our approach. To further validate this

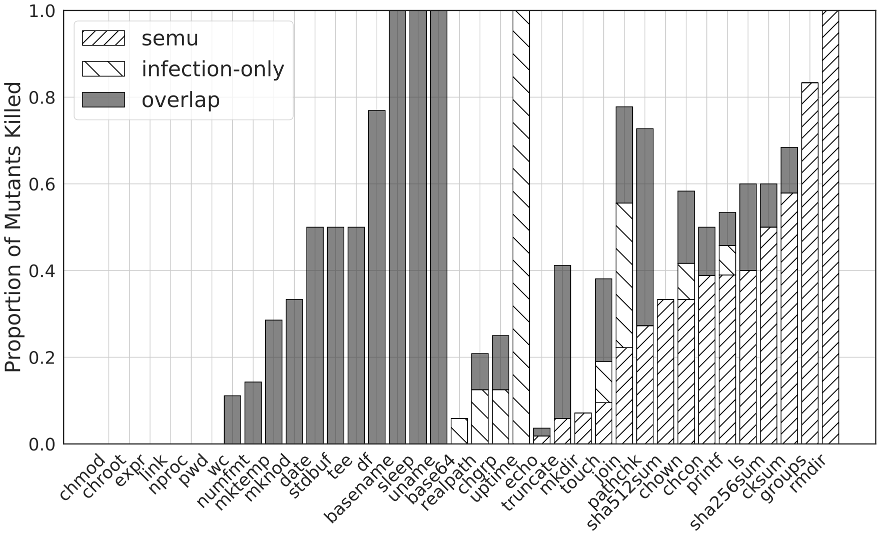


Fig. 7. Comparing the mutant killing ability of *SEMu* and *infection-only* in per program basis.

finding, we performed a Wilcoxon statistical test and got a p-value of 0.04 suggesting that the two samples' values are statistically significant (at the commonly adopted 5% confidence level). Like in RQ2 we also computed the Vargha Delaney effect size $\hat{A_{12}}$ and found that *SEMu* yields higher killing rates than *infection-only* in 61% of the cases.

To demonstrate the differences we also present our results in a per test subject basis. Figure 7 shows the differences and overlap between the killed reference mutants. From these results we observe a large overlap between the mutants killed by both approaches, with *SEMu* being able to kill more mutants for most of the cases. We also observe that in 5 of the cases *infection-only* performed better than *SEMu*, while *SEMu* performed better in 13.

Similarly, to the previous RQs we compare the strategies by counting the number of the mutant killing test inputs that were generated by the strategies. Interestingly, we found that *SEMu* generated 87% more mutant killing test inputs than the "infection-only" one (153 vs. 82 inputs) , indicating the usefulness of our framework.

7 RELATED WORK

Many techniques targeting mutation-based test generation have been proposed [4, 27]. However, most of these focus on generating test suites from scratch, by maximizing the number of mutants killed, mainly by either reaching the mutants or by targeting mutant infection. In contrast we aim at the thorough testing of specific program areas by taking advantage of existing tests and by targeting stubborn mutants that are hard to propagate.

The studies of Papadakis and Malevris [29, 30], and Zhang et al. [42] proposed embedding mutant related constraints, called infection conditions, within meta-programs. These meta-programs inject and control the mutations in order to force symbolic execution to cover them. As a result, symbolic execution modules can produce test cases that satisfy the infection conditions and have good chances to kill the mutants. Although effective, these approaches only target mutant infection, which makes them relatively weak [39].

To bypasss the abovementioned problem, the studies of Papadakis and Malevris [28] and Harman et al. [15] aimed at indirectly handling mutant propagation. The former technique searches symbolically the path space of the mutant programs (after the mutation point), while the later one searches the input program space defined by the path conditions in order to bypass constraints not handled by the used solver and to indirectly make the mutants propagate. In contrast *SEMu* aims at incrementally differentially exploring the path space by considering the symbolic states and making a relevant exploration.

Fraser and Zeller [12] and Fraser and Arcuri [11] applied Search-based testing in order to generate mutation-based tests. Their key advancement was to guide the search by measuring the differences between the test traces of the original and mutant programs. While powerful, such an attempt still fails to provide the guidance needed in order to trigger such differences.

Moreover, search techniques rely on the ability to execute test cases fast (applied at the unit level), making them less effective in cases of slow test execution (such as system level testing). Nevertheless, a comparison between search-based test generation and symbolic execution falls out of the scope of the present paper.

Much work on testing software patches has also been performed the recent years [19, 20, 37]. However, most of these methods aim at covering patches and not the program semantics (behavioural changes). Moreover, these techniques target the general patch testing problem, which in a sense assume very few patches with many changes. The case of mutation, though, involves many mutants. These are created by inducing small syntactic deviations, a fact that our method takes advantage in order to optimize the mutant killings.

Differential symbolic execution [31] aims at reasoning about semantic differences of program versions, but since it performs a whole program analysis it can experience significant scalability issues when considering large programs and multiple mutants. Directed incremental symbolic execution [32] guides the symbolic exploration through static program slicing. Unfortunately, such a method can be expensive when used with many mutants. Nevertheless, program slicing could be used to further guide *SEMu* towards the relevant mutant exploration space.

Shadow symbolic execution [24] applies a combined execution on both program versions under analysis. It relies on analysis a meta-program that is similar to the mutant's meta-program in order to take advantage of the common program parts. The major difference with our method is that we specifically target multiple mutants at the same time, limit the program exploration through data state comparisons in order to optimize performance. Since shadow targets single patches and exhaustively searches the path space (after the mutation point) it can experience scalability issues.

Overall, while many related techniques have been proposed, they have not been investigated in the context of mutation testing and particularly to target stubborn mutants. Stubborn mutants are hard to kill and their killing results in test inputs that are linked with corner cases and increase fault revelation [39].

8 CONCLUSION

This paper introduced *SEMu*, a method that generates test inputs for killing stubborn mutants. *SEMu* relies on a form of shared differential symbolic execution that incrementally searches a small but 'promising' code region around the mutation point in order to reveal divergent behaviours. This allows the fast and effective generation of test inputs that thoroughly exercise the targeted program corner cases. We have empirically evaluated *SEMu* on Coreutils and demonstrated that it can kill approximately 37% of the involved stubborn mutants within a two hour time budget. This performance is approximately 20% higher than that of the baseline (*infection-only*) strategy.

An important characteristic of *SEMu* is that it allows performing thorough testing in selected 'critical' parts of the programs under test. Therefore, it allows improving test suites by generating mutation-based test inputs that kill stubborn mutants (mutants that survive the execution with the available test suites). This is important as it allows testing corner cases that escaped testing (encoded by stubborn mutants [39]). Moreover, *SEMu* aims at handling failed error propagation (masking effects), which is challenging and prevalent in mutation testing.

Similarly to any technique that generates tests to kill mutants, the scalability of *SEMu* depends on the nature and the number of the involved mutants. However, the design of *SEMu* enables various configurations achieving different trade-offs when aiming at stubborn mutants. This makes it possible to choose, based on the number of mutants and available time budget, the configuration that evenly share the allocated time budget across all mutants.

Our future work includes the examination of additional path search strategies and a thorough evaluation of the relationship between the propagation distance and the likelihood of revealing underlying program defects. These will enable tuning further the test generation process and will improve scalability and effectiveness.

SEMu is publicly available as open-source: https://github.com/thierry-tct/KLEE-SEMu.

ACKNOWLEDGMENTS

This work was supported by the CORE Grant of National Research Fund, Luxembourg, C17/IS/11686509/CODEMATES.

REFERENCES

[1] Rawad Abou Assi, Chadi Trad, Marwan Maalouf, and Wes Masri. 2019. Coincidental correctness in the Defects4J benchmark. *Software Testing, Verification and Reliability* 29, 3 (2019), e1696. https://doi.org/10.1002/stvr.1696 e1696 STVR-18-0045.R2.

[2] Paul Ammann, Márcio Eduardo Delamaro, and Jeff Offutt. 2014. Establishing Theoretical Minimal Sets of Mutants. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*. 21–30. https://doi.org/10.1109/ICST.2014.13

[3] Paul Ammann and Jeff Offutt. 2008. *Introduction to software testing*. Cambridge University Press.

[4] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John A. Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil McMinn. 2013. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software* 86, 8 (2013), 1978–2001. https://doi.org/10.1016/j.jss.2013.02.061

[5] Kelly Androutsopoulos, David Clark, Haitao Dan, Robert M. Hierons, and Mark Harman. 2014. An analysis of the relationship between conditional entropy and failed error propagation in software testing. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 573–583. https://doi.org/10.1145/2568225.2568314

[6] Richard Baker and Ibrahim Habli. 2013. An Empirical Evaluation of Mutation Testing for Improving the Test Quality of Safety-Critical Software. *IEEE Trans. Software Eng.* 39, 6 (2013), 787–805. https://doi.org/10.1109/TSE.2012.56

[7] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. In *8th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2008, December 8-10, 2008, San Diego, California, USA, Proceedings*. 209–224. http://www.usenix.org/events/osdi08/tech/full_papers/cadar/cadar.pdf

[8] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08/ETAPS'08)*. Springer-Verlag, Berlin, Heidelberg, 337–340. http://dl.acm.org/citation.cfm?id=1792734.1792766

[9] Pedro Delgado-Pérez, Ibrahim Habli, Steve Gregory, Rob Alexander, John A. Clark, and Inmaculada Medina-Bulo. 2018. Evaluation of Mutation Testing in a Nuclear Industry Case Study. *IEEE Trans. Reliability* 67, 4 (2018), 1406–1419. https://doi.org/10.1109/TR.2018.2864678

[10] Richard A. DeMillo and A. Jefferson Offutt. 1991. Constraint-Based Automatic Test Data Generation. *IEEE Trans. Software Eng.* 17, 9 (1991), 900–910. https://doi.org/10.1109/32.92910

[11] Gordon Fraser and Andrea Arcuri. 2015. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering* 20, 3 (2015), 783–812. https://doi.org/10.1007/s10664-013-9299-z

[12] Gordon Fraser and Andreas Zeller. 2012. Mutation-Driven Generation of Unit Tests and Oracles. *IEEE Trans. Software Eng.* 38, 2 (2012), 278–292. https://doi.org/10.1109/TSE.2011.93

[13] A. S. Ghiduk, M. R. Girgis, and M. H. Shehata. 2019. Employing Dynamic Symbolic Execution for Equivalent Mutant Detection. *IEEE Access* 7 (2019), 163767–163777.

[14] Florian Gross, Gordon Fraser, and Andreas Zeller. 2012. Search-based system testing: high coverage, no false alarms. In *International Symposium on Software Testing and Analysis, ISSTA 2012, Minneapolis, MN, USA, July 15-20, 2012*. 67–77. https://doi.org/10.1145/2338965.2336762

[15] Mark Harman, Yue Jia, and William B. Langdon. 2011. Strong higher order mutation-based test data generation. In *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13th European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*. 212–222. https://doi.org/10.1145/2025113.2025144

[16] James C. King. 1976. Symbolic Execution and Program Testing. *Commun. ACM* 19, 7 (July 1976), 385–394. https://doi.org/10.1145/360248.360252

[17] Marinos Kintis, Mike Papadakis, and Nicos Malevris. 2010. Evaluating Mutation Testing Alternatives: A Collateral Experiment. In *17th Asia Pacific Software Engineering Conference, APSEC 2010, Sydney, Australia, November 30 - December 3, 2010*. 300–309. https://doi.org/10.1109/APSEC.2010.42

[18] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. 2016. Are We There Yet? How Redundant and Equivalent Mutants Affect Determination of Test Completeness. In *Ninth IEEE International Conference on Software Testing, Verification and Validation Workshops, ICST Workshops 2016, Chicago, IL, USA, April 11-15, 2016*. 142–151. https://doi.org/10.1109/ICSTW.2016.41

[19] Paul Dan Marinescu and Cristian Cadar. 2012. make test-zesti: A symbolic execution solution for improving regression testing. In *34th International Conference on Software Engineering, ICSE 2012, June 2-9, 2012, Zurich, Switzerland*. 716–726. https://doi.org/10.1109/ICSE.2012.6227146

[20] Paul Dan Marinescu and Cristian Cadar. 2013. KATCH: high-coverage testing of software patches. In *Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, ESEC/FSE'13, Saint Petersburg, Russian Federation, August 18-26, 2013*. 235–245. https://doi.org/10.1145/2491411.2491438

[21] L. J. Morell. 1990. A Theory of Fault-Based Testing. *IEEE Trans. Softw. Eng.* 16, 8 (Aug. 1990), 844–857. https://doi.org/10.1109/32.57623

[22] Omer Nguena Timo, Alexandre Petrenko, and S. Ramesh. 2017. Multiple Mutation Testing from Finite State Machines with Symbolic Inputs. In *Testing Software and Systems*, Nina Yevtushenko, Ana Rosa Cavalli, and Hüsnü Yenigün (Eds.). Springer International Publishing, Cham, 108–125.

[23] A. Jefferson Offutt, Ammei Lee, Gregg Rothermel, Roland H. Untch, and Christian Zapf. 1996. An Experimental Determination of Sufficient Mutant Operators. *ACM Trans. Softw. Eng. Methodol.* 5, 2 (1996), 99–118. https://doi.org/10.1145/227607.227610

[24] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. 2016. Shadow of a doubt: testing for divergences between software versions. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 1181–1192. https://doi.org/10.1145/2884781.2884845

[25] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*. 354–365. https://doi.org/10.1145/2931037.2931040

[26] Mike Papadakis, Yue Jia, Mark Harman, and Yves Le Traon. 2015. Trivial Compiler Equivalence: A Large Scale Empirical Study of a Simple, Fast and Effective Equivalent Mutant Detection Technique. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*. 936–946. https://doi.org/10.1109/ICSE.2015.103

[27] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Mutation Testing Advances: An Analysis and Survey. Advances in Computers, Vol. 112. Elsevier, 275 – 378. https://doi.org/10.1016/bs.adcom.2018.03.015

[28] Mike Papadakis and Nicos Malevris. 2010. Automatic Mutation Test Case Generation via Dynamic Symbolic Execution. In *IEEE 21st International Symposium on Software Reliability Engineering, ISSRE 2010, San Jose, CA, USA, 1-4 November 2010*. 121–130. https://doi.org/10.1109/ISSRE.2010.38

[29] Mike Papadakis and Nicos Malevris. 2011. Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing. *Software Quality Journal* 19, 4 (2011), 691–723. https://doi.org/10.1007/s11219-011-9142-y

[30] Mike Papadakis and Nicos Malevris. 2012. Mutation based test case generation via a path selection strategy. *Information & Software Technology* 54, 9 (2012), 915–932. https://doi.org/10.1016/j.infsof.2012.02.004

[31] Suzette Person, Matthew B. Dwyer, Sebastian G. Elbaum, and Corina S. Pasareanu. 2008. Differential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2008, Atlanta, Georgia, USA, November 9-14, 2008*. 226–237. https://doi.org/10.1145/1453101.1453131

[32] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. 2011. Directed incremental symbolic execution. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*. 504–515. https://doi.org/10.1145/1993498.1993558

[33] Goran Petrovic and Marko Ivankovic. 2018. State of mutation testing at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2018, Gothenburg, Sweden, May 27 - June 03, 2018*. 163–171. https://doi.org/10.1145/3183519.3183521

[34] Alessandro Viola Pizzoleto, Fabiano Cutigi Ferrari, Jeff Offutt, Leo Fernandes, and Marcio Ribeiro. 2019. A Systematic Literature Review of Techniques and Metrics to Reduce the Cost of Mutation Testing. *Journal of Systems and Software* (2019). https://doi.org/10.1016/j.jss.2019.07.100

[35] David Schuler and Andreas Zeller. 2013. Covering and Uncovering Equivalent Mutants. *Softw. Test., Verif. Reliab.* 23, 5 (2013), 353–374. https://doi.org/10.1002/stvr.1473

[36] Francisco Carlos M. Souza, Mike Papadakis, Vinicius H. S. Durelli, and Márcio Eduardo Delamaro. 2014. Test Data Generation Techniques for Mutation Testing: A Systematic Mapping. In *Proceedings of the XVII Iberoamerican Conference on Software Engineering, CIbSE 2014, Pucon, Chile, April 23-25, 2014*, Jaelson Castro, Claudia P. Ayala, Giovanni Giachetti, Márcia Lucena, Carlos Cares, Xavier Franch, Monalessa Perini Barcellos, Maria Lencastre, Beatriz Marín, and Ricardo Gacitua (Eds.). Curran Associates, 419–432.

[37] Kunal Taneja, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. 2011. eXpress: guided path exploration for efficient regression test generation. In *Proceedings of the 20th International Symposium on Software Testing and Analysis, ISSTA 2011, Toronto, ON, Canada, July 17-21, 2011*. 1–11. https://doi.org/10.1145/2001420.2001422

[38] Thierry Titcheu Chekam, Mike Papadakis, and Yves Le Traon. 2019. Mart: A Mutant Generation Tool for LLVM. In *Proceedings of the 27th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[39] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*. 597–608. https://doi.org/10.1109/ICSE.2017.61

[40] Roland H. Untch, A. Jefferson Offutt, and Mary Jean Harrold. 1993. Mutation Analysis Using Mutant Schemata. In *Proceedings of the 1993 International Symposium on Software Testing and Analysis, ISSTA 1993, Cambridge, MA, USA, June 28-30, 1993*. 139–148. https://doi.org/10.1145/154183.154265

[41] Xiangjuan Yao, Mark Harman, and Yue Jia. 2014. A study of equivalent and stubborn mutation operators using human analysis of equivalence. In *36th International Conference on Software Engineering, ICSE '14, Hyderabad, India - May 31 - June 07, 2014*. 919–930. https://doi.org/10.1145/2568225.2568265

[42] Lingming Zhang, Tao Xie, Lu Zhang, Nikolai Tillmann, Jonathan de Halleux, and Hong Mei. 2010. Test generation via Dynamic Symbolic Execution for mutation testing. 1 – 10. https://doi.org/10.1109/ICSM.2010.5609672