Nequivack: Assessing mutation score confidence

Dominik Holling, Sebastian Banescu, Marco Probst, Ana Petrovska, Alexander Pretschner
Technische Universität München
Garching bei München, Germany
{holling, banescu, probst, petrovska, pretschn}@cs.tum.edu

***Abstract*—The mutation score is defined as the number of killed mutants divided by the number of non-equivalent mutants. However, whether a mutant is equivalent to the original program is undecidable in general. Thus, even when improving a test suite, a mutant score assessing this test suite may become worse during the development of a system, because of equivalent mutants introduced during mutant creation. This is a fundamental problem.**

Using static analysis and symbolic execution, we show how to establish non-equivalence or "don't know" among mutants. If the number of don't knows is small, this is a good indicator that a computed mutation score actually reflects its above definition. We can therefore have an increased confidence that mutation score trends correspond to actual improvements of a test suite's quality, and are not overly polluted by equivalent mutants.

Using a set of 14 representative unit size programs, we show that for some, but not all, of these programs, the above confidence can indeed be established. We also evaluate the reproducibility, efficiency and effectiveness of our Nequivack tool. Our findings are that reproducibility is completely given. A single mutant analysis can be performed within 3 seconds on average, which is efficient for practical and industrial applications.

I. Introduction

Mutation testing [1] uses fault injection for test suite assessment. By introducing faults into the tested program P, so called *mutants* $P'_1...P'_n$ are created. The test suite is then executed using the mutant thereby either *killing* it by detecting the introduced fault or keeping it *alive* by failing to detect the fault. An *equivalent mutant* is the result of applying a transformation to P creating P', where P and P' have equivalent input-output behavioral semantics (i.e. $[[P]] == [[P']]$). The *mutant score* to judge the quality of a test suite is defined as the number of killed mutants divided by the number of non-equivalent mutants. Two fundamental aspects in mutation testing are: (1) the choice of mutation operators transforming the program into mutants and (2) avoiding the creation of equivalent mutants [2]. Since equivalent mutants are not distinguishable from the original program by a test suite, they are one limitation of the practical deployment of mutation testing. For mutation testing to be applicable in practical contexts, equivalent mutants must be either detected or their creation must be limited [3]. Otherwise, increasing the fault detection ability of a test suite may cause the mutation score to decrease during the development of a program due to an increased number of equivalent mutants.

Problem: The mutation score inherently suffers from the undecidable problem of program equivalence [2]. As tools for mutation testing typically use the number of all mutants (including equivalent mutants) when computing the mutation score, the confidence in the mutation score is limited. For example, using a mutant generation tool throughout the iterations or sprints of a certain software project, the mutation score may decrease even though the fault detection ability of the test suite has increased and vice versa. The reason behind such discrepancies is the unknown number of equivalent mutants against which the test suite is assessed.

Solution: In practice, it is essential to know how close the mutation score obtained by a tool is to the actual mutation score. We propose the mutation score confidence as a measure for this closeness. It is defined as the number of proven non-equivalent mutants divided by all mutants. To compute the number of non-equivalent mutants, our tool *Nequivack* represents a Non-EQUIVAlence ChecKer using symbolic execution (see below) on a unit testing level. Nequivack is started after applying mutation to the program and before executing the test suite to compute the confidence in the to be obtained mutation score. For this purpose, Nequivack is given the original program and a mutant; it tries to find a counter-example (i.e. a test case as evidence) to their equivalence. If a test case is found, the mutant is deemed non-equivalent. In case Nequivack encounters an error times out or terminates, the equivalence of the mutant is deemed unknown.

Technically, we make use of symbolic execution [4], [5], which has been employed in software testing for the exploration of program paths leading to the automatic generation of high coverage test suites. To symbolically execute a program, symbolic instead of concrete values are used as inputs. Symbolic values represent a set of possible concrete values that could be taken on a particular program execution path. Thus, the output and condition to take the path (i.e. path condition) are expressed w.r.t. the symbolic input values. Whenever a program termination point (or error state) is reached, a test case for the taken path is created by querying an SMT-solver with the corresponding path condition. A framework for the symbolic execution of programs written in C/C++ is KLEE [6]. KLEE interprets LLVM bitcode for symbolic execution using a variety of path exploration techniques and a bit-level precision memory model. While exploring the paths, it can detect run time errors, such as *out of bound memory access* or *division by zero* and create respective test cases. In addition, it has the ability to falsify custom program assertions on each path. In particular, using symbolic execution and assertions on the outputs of two programs may generate test cases showing non-equivalence of those two programs. Note that, although the generated test cases are inherently able to kill the mutant

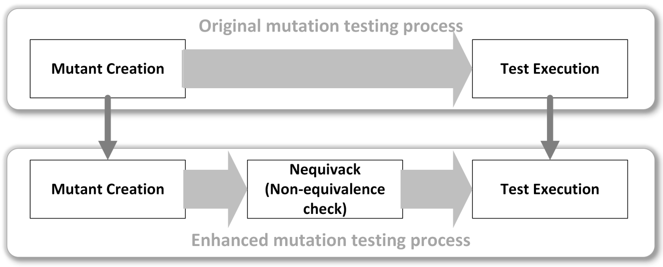

Fig. 1. Mutation testing: Original and proposed process

and would yield a high mutation score, their ability to detect realistic faults is only given iff the coupling hypothesis [1] holds.

Contribution: To the best of our knowledge, this is the first assessment of mutation score confidence. By employing our approach, we are able to give a decision-making criterion on the closeness of the mutation score obtained and the actual mutation score. Thus, we are able to give an a priori assessment whether evaluating the test suite using mutation testing is practically sensible. In case the mutation score confidence is high, the mutation score will be close to the actual mutation score as the number of equivalent mutants is acceptably low. For a low mutation score confidence, the number of equivalent mutants may be high and the sensibility of mutation testing is unknown, which may yield the recommendation to use other forms of test case assessment in practice. In addition, our approach is lightweight on a unit testing level and is offered as an open source tool that can be easily be integrated as an additional efficient step into existing mutation testing tools. Furthermore, we are able to demonstrate full reproducibility of the results of Nequivack, while at the same time having high efficiency and high effectiveness. Reproducibility is especially important in practical contexts where vastly different results analyzing the same mutant are unacceptable. Nequivack is able to efficiently analyze 10 000 mutants in approximately 6 minutes on average while effectively classifying all mutants correctly.

The structure of this paper is as follows: In Section II, we present our approach. Section III evaluates the approach w.r.t. to reproducibility, effectiveness and efficiency using well-known algorithms. Section IV puts our work in context. Finally, Section V presents a conclusion and an outlook.

II. Approach

Our approach in Figure 1 extends the common mutation testing process of mutant creation and test execution with an intermediary non-equivalence check. The source code of Nequivack and all evaluation programs can be found at https://github.com/tum-i22/nequivack. To determine the non-equivalence of two programs at unit testing level, Nequivack requires 6 steps, which we will describe in the following.

a) Step 1: Annotate all functions in the original program which are to be tested for non-equivalence. Annotations take the form of a single line comment placed right before

```
// @klee[inArraySize=5,outArraySize=4](sym,1)
int* remove_first(int a[], int n) {
   return a + n;
}
```

Listing 1. Example of function annotation

```
// @klee
int add(int a, int b) {
   return a + b;
}
```

Listing 2. Function annotation where all input parameters are symbolic

the function definition. The minimal annotation that must be provided is `@klee`, which indicates that this function will be checked for non-equivalence. If the function has an array as input parameter or an array as output parameter, then Nequivack requires specifying the desired maximum sizes of these arrays via the `inArraySize`, respectively `outArraySize` annotations placed between square brackets, illustrated in Listing 1. Nequivack also allows specifying which input parameters of the function to be tested should take concrete or symbolic values. These values can be specified via a list between parentheses at the end of the annotation as shown in Listing 1, i.e. `sym` on the first position in the list means that the first parameter of the function will take a symbolic value, while the value `1` on the second position of the list means that the second parameter of the function will be fixed to the concrete value one. If this list is not specified, then it means that all input parameters are symbolic. For example, both the input parameters of the `add` function presented in Listing 2 will be made symbolic.

Annotation of all functions in a given source code file can be automated given that all their input arguments are symbolic and the maximum size of input and output arrays are the same for all functions. After this automatic annotation a test developer can manually tune the annotations by adding concrete values for some arguments, changing the maximum sizes of arrays or removing annotations from a subset of functions.

b) Step 2: In a later step the functions to be tested for non-equivalence from the original program P and a mutant P' will be called in the same source code file. To avoid name ambiguities, this step renames the functions from P

```
// original program
// @klee
int original_add(int a, int b) {
   return a + b;
}

// mutated program
// @klee
int transformed_add(int a, int b) {
   return a - b;
}
```

Listing 3. Example of function renaming

```c
1  #include "original_Add.h"
2  #include "transformed_Add.h"
3  #include "klee/klee.h"
4
5  void test_add() {
6     int add_0, add_1;
7     klee_make_symbolic(&add_0, sizeof(add_0), "
        add_0");
8     klee_make_symbolic(&add_1, sizeof(add_1), "
        add_1");
9
10    int original_ret = original_add(add_0, add_1);
11    int transformed_ret = transformed_add(add_0,
        add_1);
12
13    klee_assert(original_ret == transformed_ret);
14 }
15
16 int main(int argc, char* argv[]) {
17    test_add();
18    return 0;
19 }
```

Listing 4. Example of KleeMain.c

and P' by prefixing them with `original_`, respectively `transformed_`. If we consider the function from Listing 2 to be part of the original program P and we consider a mutant P', where the addition operator on line 3 of Listing 2 is changed to a subtraction operator, then the result of step 2 is shown in Listing 3.

c) Step 3: Two header files are generated for each of P and P' containing the signatures of their annotated functions. If P contains any structure declarations, then these are extracted into another header file.

d) Step 4: To compare the functions in P against those in its mutant P', Nequivack generates a source code file called "KleeMain.c", which contains the main function to be executed by the symbolic execution engine KLEE [7]. This file contains calls to functions which test all annotated pairs of functions from P and P', with their corresponding set of symbolic and/or concrete parameters. Each pair of return values of all the pairs of `original_` and `transformed_` functions is placed inside a statement that asserts equality if the two return values are of integer type. In case of float integer values the comparison is made by asserting that the absolute value of their difference is lower than the machine epsilon[1]. In case the return value is a structure or an array multiple assert statements are used to check the equality of the return values. If a pair of return values are equal then their corresponding assertion succeeds, otherwise it fails, which means a counter-example that proves the pair of `original_` and `transformed_` functions to be non-equivalent has been found.

An example of step 4 corresponding to the original and mutant functions from Listing 3, is presented in Listing 4. The main function of "KleeMain.c" in lines 16-19 of Listing 4, simply calls the `test_add` function defined on lines 5-14, which tests the pair of functions from Listing 3. The first two lines in Listing 4 contain the includes of the two header files generated by step 3. Since the implementation of Nequivack depends on KLEE, line 3 includes the header file which offers KLEE's API. Part of this API is a function called `klee_make_symbolic`, which sets the memory at the address indicated by its first parameter and of size equal to its second parameter as symbolic. The third parameter of `klee_make_symbolic` is an arbitrary name used to indicate the assigned concrete value to that symbolic memory in the test cases generated by KLEE. Both input values of the original and mutated `add` functions are declared in line 6 of Listing 4). They are turned into symbolic values on lines 7 and 8 using `klee_make_symbolic`. The original and mutated functions are called using these symbolic arguments, on lines 10 and 11. Note that the order in which the `original_` and `transformed_` functions are called does not matter, because no output of one function is used as input for the other function. On line 13 another function from KLEE's API `klee_assert` is called to verify if the integer return values of the 2 functions are equal.

Note that making the variables `original_ret` and `transformed_ret` symbolic by using `klee_make_symbolic`, would not be beneficial to the goal of checking non-equivalence of the `original_` and `transformed_` functions. The reason is that once the two variables are made symbolic, they will disregard any concrete return value of the `original_` and `transformed_` functions and the symbolic variables will simply take on any value of their specific type.

e) Step 5: Compile and link all the C source code and header files using the LLVM *clang* compiler and the *llvm-link* linker. Afterwards, we run KLEE. KLEE uses the modified version of the *uClibc* library[2] to symbolically execute definitions for a subset of C library functions, which the program may call. It also uses a *POSIX-runtime*[3], which handles the majority of operating system facilities used by command line application.

f) Step 6: Nequivack analyzes the output of KLEE and classifies each mutant as either *non-equivalent* if it finds a counter-example input for which the outputs of a pair of functions is different, or *unknown* if no counter-example is found.

A. Limitations

Nequivack does not support all data types representable in the C programming language. Currently it supports primitive types, arrays, pointers and structs, as long as they only contain primitive types. The C programming language offers more possibilities like *pointers to pointers* or *pointers to arrays*. For example, Nequivack does not support arrays of character pointers[4]. Another minor limitation is that Nequivack does not also rename global variables used by the function to test,

[1]http://www.cplusplus.com/reference/cfloat/

[2]http://www.uclibc.org/

[3]Information from http://klee.github.io/tutorials/testing-coreutils/

[4]An array of character pointers is equivalent to an array of strings in other languages like Java.

TABLE I
PROGRAMS USED IN THE EVALUATION AND THEIR MUTATION SCORE CONFIDENCE (# STANDS FOR "NUMBER OF")

Category	**Name**	**Purpose**	**# Lines of code**	**# Functions**	**Total # mutants**	**# Non-equiv. mutants**	**Confidence**
Basic	Branching	All branching constructs	75	3	178	162	91%
Basic	Looping	All looping constructs	48	3	131	108	82%
Binary	MSB	Get most significant bit	17	1	72	0	0%
Binary	MSB1	Get most significant bit	18	1	105	0	0%
Math	Factorial	Get factorial	11	1	45	38	84%
Math	Fibonacci	Produce fibonacci	12	1	54	32	59%
Math	GCD	Euclid's algorithm	12	1	15	11	73%
Math	PrimePalindrome	Get next prime palindrome	65	2	159	0	0%
Math	SimpleMath	All basic arithmetic	32	5	63	54	86%
Sorting	Bubble	Sort array	16	1	109	48	44%
Sorting	Insertion	Sort array	16	1	79	37	47%
Sorting	Merge (uses recursion)	Sort array	31	2	148	5	3%
Sorting	Quick (uses recursion)	Sort array	29	1	117	5	4%
Sorting	Selection	Sort array	21	1	77	17	22%

which may cause an inconsistent state. However, this feature is straightforward to implement.

Most of the limitations of Nequivack are due to using KLEE. However, the general idea can be applied by using other symbolic execution engines instead of KLEE. For example, KLEE can only process programs written in the C language. KLEE cannot handle symbolic sizes for arrays. This means that users need to provide concrete values for array sizes via annotations. Otherwise, KLEE generates an error and silently concretizes the symbolic value. KLEE behaves unpredictably, when it analyzes a program, that calls the `exit()` statement. KLEE stops its own execution and therefore the analysis, when it reaches such a statement.

Finally, our approach has fundamental limitations due to the use of symbolic execution, i.e. scalability is limited. The runtime of symbolic execution increases exponentially w.r.t the number of branches encountered during execution. This means that if we have a loop then the number of branches includes all the iterations of that loop. Thus, we apply our approach at unit testing level as meaningful symbolic execution is possible due to low complexity of unit compared to the high complexity of complete programs.

III. EVALUATION

We evaluate Nequivack in combination with the mutation testing tool Milu [8] on an octa-core Intel Xeon E5540 at 2.5 GHz evaluation system with 40 GB of RAM. Milu allows the automated creation of mutants for programs written in the C programming language. The evaluations goal is to show (1) reproducibility, (2) effectiveness and (3) efficiency of the approach.

Firstly, symbolic execution as implemented in KLEE [6] is non-deterministic as choosing a path may occur at random. Thus, reproducibility must be assessed to evaluate if multiple executions lead to the same result. In practice, every run should lead to the same result for the mutation score to represent a continuous test suite assessment.

Secondly, efficiency of the approach needs to be evaluated. Classifying the mutants must happen with reasonable time and resource consumption, to be applicable in practice. In particular, we are interested in the execution times of our approach on the mutants with our standard hardware evaluation system.

Thirdly, the effectiveness of the approach must be evaluated concerning the accuracy of Nequivack. In our definition, a *false positive* is defined as an equivalent mutant deemed non-equivalent. A *false negative* is a non-equivalent mutant with an unknown result. Particularly, false negatives increase the mutation score confidence and could lead to unreasonable use of mutation testing in practice.

The programs used for the evaluation contain a set of representative algorithms on a unit testing level implemented in the C programming language. An overview of these programs is presented in Table I. All programs have between 1 and 5 functions and worst case complexities between $O(1)$ and $O(n^2)$. Milu applied 12 of the 14 mutation operators presented in Table II at least once to the 14 programs yielding 1352 mutants. The only mutant not applied were the OBBA and OBBN mutations as our programs did not contain the syntactic elements required for mutation.

For the experiment executed in this evaluation, Nequivack was then given each single mutant and the respective original program to find counter-examples as evidence for their non-equivalence. We limit the execution time of KLEE to 60 seconds. Thus, if a counter-example for equivalence is not found within this time, equivalence of the mutant is unknown. We deliberately do not evaluate Milu as this is done elsewhere [8] and it represents a typical mutation tool used in practice. As Nequivack is used before test suite execution, our evaluation is completely independent of the test suite to be evaluated using the mutants. Thus, we do not evaluate any test execution tool.

A. Reproducibility

To evaluate the reproducibility of our approach, we question the reproducibility of the results of Nequivack. Steps 1 to 4 and 6 of Nequivack are completely deterministic due to no user inputs and single-threading. Solely executing KLEE in step 5 has an impact on reproducibility as KLEE uses random path exploration. To examine reproducibility, we run Nequivack 10

TABLE II
APPLIED MILU OPERATORS

Acronym	Description
SBRC	Replacement of break statements by continue statements
ABS	Absolute value insertion in each arithmetic expression
CRCR	Replacement of constants by randomly chosen values or by in-/de-crementing the current value
OAAA	Replacement of arithmetic operators used in combination with assignment operators
OAAN	Replacement of arithmetic operators in the right hand side of expressions
OBBA	Replacement of bitwise operators used in combination with assignment operators
OBBN	Replacement of bitwise operators in the right hand side of expressions
OCNG	Negation of branching statement boolean conditions
OIDO	Replacement of increment operator by the decrement operator or vice-versa
OLLN	Replacement of logical operators in boolean expressions
OLNG	Logical negation of boolean expressions
ORRN	Replacement of relational operators by other relation operators
UOI	Insertion of unary operators (e.g., -,++,--,!) in arithmetic or boolean expressions

TABLE III
TOP 10 WORST CASE EXECUTION TIME: NON-EQUIVALENT MUTANTS

Program	Mut. No.	Avg. (ms)	Std. Dev. (ms)	Max. (ms)
Selection	57	24 449.5	10 338.0	38 867
Selection	60	22 516.6	7 709.2	38 433
Selection	21	18 115.6	3 209.5	21 871
Selection	46	17 475.9	994.0	18 790
Selection	50	17 362.8	724.4	18 450
Selection	76	15 091.5	3 711.2	22 000
Selection	43	14 703.9	3 347.5	18 720
Selection	45	13 939.6	1 108.1	15 750
Selection	9	13 449.5	683.6	14 600
Bubble	109	13 206.7	3 795.4	17 715

TABLE IV
TOP 10 WORST CASE EXECUTION TIME: UNKNOWN EQUIV. MUTANTS

Program	Mut. No.	Avg. (ms)	Std. Dev. (ms)	Max. (ms)
Selection	58	105 347.4	1 777.5	107 189
Selection	56	104 796.2	1 431.6	106 597
Selection	34	103 247.6	2 379.3	105 572
Selection	61	102 220.4	212.4	102 518
Selection	3	101 793.5	892.5	102 814
Selection	4	101 238.2	554.8	102 022
GCD	2	79 619.3	4 664.6	84 866
Fibonacci	66	69 409.1	4 737.4	77 668
Fibonacci	25	65 262.1	931.3	66 311
Fibonacci	11	65 020.0	896.6	66 048

times on each mutant. We deem the results reproducible, iff at least 80% of all non-equivalent mutants identified in 10 executions are classified as non-equivalent in each run.

The result of the reproducibility analysis was 100% reproducibility of all results in all runs. Mutants classified as non-equivalent and unknown equivalence were the same in all executions for all programs. Thus, we tentatively conclude that non-equivalence checking with Nequivack is highly reproducible for the presented programs.

B. Efficiency

To address the aspect of efficiency, we measure the execution time of Nequivack 10 times for each mutant. For each mutant, program and program category we compute the average execution time and its (average) standard deviation. Our baseline for adequacy in practice is the ability of Nequivack to process 1000 of our mutants within 60 minutes on average. This mutant analysis rate allows to even perform mutation testing in large scale projects like the Apache Web Server[5] core within 8 hours overnight when using 4 machines. For this purpose, we investigate the worst case execution times of all attested non-equivalent mutants and all unknown equivalence mutants. In addition, we analyze the worst case execution times per program and per category to give an indication towards mutant analysis complexity concerning the programs and categories. The indication aims to predict execution time in industry scenarios.

The worst case execution time to reach a counter-example and attest non-equivalence takes 39 seconds using the 57th mutant of selection sort as shown in Table III. On average the worst case program takes 24.5 seconds with a standard deviation of 10 seconds. We hypothesize the standard deviation to be due to the non-determinism in KLEE. The worst case execution time average also decreases significantly from 24 to 13 seconds within the mutants having the top 10 worst execution times shown in Table III. Thus, it is likely that these high values represent outliers in the analysis. This is also illustrated by the average of all non-equivalent mutants being 3.3 seconds and the median being 2.2 seconds. Thus, for a vast majority of mutants determining their non-equivalence is possible within 3 seconds, which fulfills our baseline.

The shortest execution times were produced by mutants which contain compilation errors, introduced by the mutation tool. These mutants take less than a second to classify. These times are negligible and would be the same even without the presence of Nequivack in the process.

For all unknown equivalence mutants, the worst case execution times is 107 seconds (see Table IV) where compiling took about 40 seconds worst case. However, we consider all mutant analyses above 100 seconds outliers as the average mutant analysis for unknown equivalence mutants was 21 seconds

[5]https://httpd.apache.org/

TABLE V
AVERAGE AND WORST EXECUTION TIME BY PROGRAM

Program	Average (ms)	Standard Deviation (ms)	Maximum (ms)	Worst case complexity
Branching	2 191.5	45.0	3 312	$O(1)$
Looping	3 148.6	165.5	62 451	$O(n)$
MSB	3 079.0	160.0	4 500	$O(1)$
MSB1	2 801.4	106.4	3 366	$O(1)$
Factorial	8 705.6	324.4	65 868	$O(1)$
Fibonacci	28 692.1	402.2	67 057	$O(n^2)$
GCD	22 609.1	959.9	84 866	$O(n^2)$
PrimePalindrome	1 987.4	57.9	78 591	$O(n^2)$
SimpleMath	2 497.9	76.9	11 297	$O(1)$
Bubble	27 257.0	2 841.1	67 033	$O(n^2)$
Insertion	17 078.4	2 787.0	66 345	$O(n^2)$
Merge	19 842.5	458.6	55 504	$O(n^2)$
Quick	53 999.3	76.2	64 241	$O(n^2)$
Selection	32 267.7	2 080.3	107 189	$O(n^2)$

TABLE VI
AVERAGE AND WORST EXECUTION TIME BY CATEGORY

Category	Avg. (ms)	Std. Dev. (ms)	Max. (ms)	Complex.
Basic	2 597.3	96.1	62 451	$O(n)$
Binary	2 914.3	128.2	4 500	$O(1)$
Math	8 195.3	192.7	84 866	$O(n^2)$
Sorting	30 300.8	1 446.8	107 189	$O(n^2)$

and compilation time was less than one second. The median was 3.2 seconds demonstrating the division of this set of mutants into many short mutant analyses and few very lengthy ones. The average standard deviation is 1.3 seconds hinting at stable execution time. Thus, we tentatively conclude for a vast majority of mutants determining unknown equivalence is possible within 3.5 seconds fulfilling our baseline.

Of all programs, the worst case execution time was produced by selection sort followed by GCD and PrimePalindrome as seen in Table V. Quick sort produced the worst average with 54 seconds. In particular, this was due to a large fraction of equivalent mutants and mutants not symbolically executable within feasible time by Nequivack being created. Table V particularly shows an increase of mutant analysis time with increased worst case complexity. This makes sense as the number of paths to explore increase as worst case algorithmic complexity increases.

Of all categories, sorting has by far the biggest worst case on average and in total (see Table VI). Since sorting contains the most programs with maximum worst case complexity and worst case execution times of the categories are ordered by worst case complexity, the indication is again a correlation. Thus, we tentatively conclude the evaluation to indicate a correlation between mutant analysis time and worst case algorithmic complexity. The results also indicate a non-linear correlation, which has to be further examined.

Overall the analysis of 927 mutants is performed within 10 seconds (see Figure 2). The analysis of only 425 takes longer than 10 seconds, almost half of which (178) are analyzed within 30 seconds. In addition, there are 15 mutant analyses taking longer than 70 seconds, which we consider outliers. Thus, we tentatively conclude Nequivack to be efficient for use in practice w.r.t. to the presented programs.

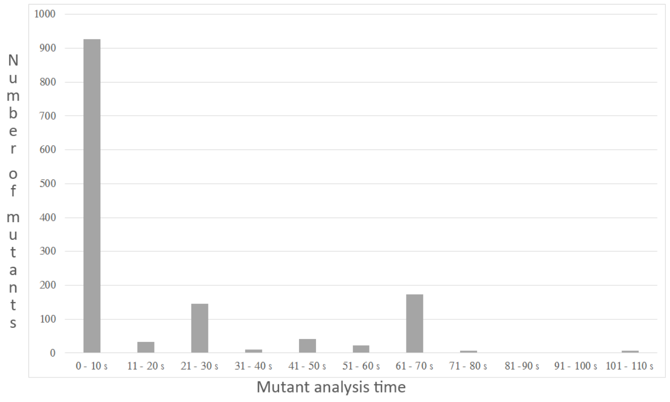

Fig. 2. Accumulated worst case mutant analysis time

C. Effectiveness and Confidence

To address the aspect of effectiveness, we analyze the true and false positives/negatives that Nequivack produces for our 14 original programs. In the mutant analysis: True positives are non-equivalent mutants labeled as non-equivalent. False positives are mutants with unknown equivalence labeled as non-equivalent. True negatives are equivalent (or faulty) mutants labeled with unknown equivalence. False negatives are non-equivalent mutants labeled with unknown equivalence. Our baseline for mutation score confidence is, somewhat arbitrarily, 80%. This is to assure a minimal number of equivalent mutants and an obtained mutation score close to the actual mutation score.

The total number of created mutants is 1352, of which: 517 mutants are labeled as non-equivalent, 432 are labeled with unknown equivalence and 403 mutants did not compile. All 517 mutants labeled as non-equivalent were associated with a test case proving the non-equivalence of the mutant w.r.t. the

```c
// Original line 4: if (a < 0)
int absolute (int a)
{
   if (a <= 0) {
      a = - a ;
   }
   return a ;
}
```

Listing 5. Exemplary equivalent mutant of SimpleMath

```c
// Original line 5: for (i = 1; i < n; i++) {
int* sort(int a[], int n) {
   int i, j, t;

   for (i = 1; abs(i) < n; i++) {
      t = a[i];

      for (j = i; j > 0 && t < a[j - 1]; j--) {
         a[j] = a[j - 1];
      }
      a[j] = t;
   }
   return a;
}
```

Listing 6. Exemplary equivalent mutant of Insertion

original program. Thus, there were no false positives. For the 432 mutants labeled with unknown equivalence, there were three cases in step 5 of Nequivack : (1) KLEE terminated exploring all paths, (2) KLEE did not terminate and was forcefully terminated and (3) KLEE terminated after encountering a run time error.

In the case where KLEE terminated after exploring all path, non-equivalence could not be proven by Nequivack. Therefore, we manually inspected all analyzed mutants (in sum 254) and found them to be equivalent to the original program.

```c
// Original line 2: while (true)
while (-1) {
   n++;
   t = n;

   while (t) {
      r *= 10;
      r += t % 10;
      t /= 10;
   }

   if (r == n) {
      d = original_sqrt32(n);

      /* Checking prime */
      for (c = 2; c <= d; c++) {
         if (n % c == 0)
         break;
      }
      if (c == d + 1)
      break;
   }
   r = 0;
}
return n;
```

Listing 7. Exemplary equivalent mutant part of PrimePalindrome

```c
// Original line 6: if (n == 1)
unsigned long fibonacci(unsigned short n) {
   if (n == 0) {
      return 0;
   }
   if (n == 302) {
      return 1;
   }
   return fibonacci(n - 1) + fibonacci(n - 2);
}
```

Listing 8. Exemplary equivalent mutant of Fibonacci

```c
// Original line 5: for (i = 1; i < n; i++) {
int* sort(int a[], int n) {
   int i, j, t;

   for (i = -1; i < n; i++) {
      t = a[i];

      for (j = i; j > 0 && t < a[j - 1]; j--) {
         a[j] = a[j - 1];
      }
      a[j] = t;
   }
   return a;
}
```

Listing 9. Exemplary run time error causing mutant of Insertion

A particular example of an equivalent mutant can be found in Listing 5. In the example, only the path taken for 0 is changed, but multiplying 0 with -1 does not change its value. Predominant was also the replacement of loop variables `i` with `abs(i)` while the loop started at 0 and increased the value of `i` as seen in Listing 6. A further example was the replacement of the constant TRUE in a while loop with -1 as seen in Listing 7. Although `while(TRUE)` and `while(-1)` may look different to a human reviewer, they have equivalent semantics.

For the 52 mutants, where KLEE was forcefully terminated by a timeout, we either found them to be equivalent or inappropriate for mutation testing. An example of the latter was a transformation of Fibonacci causing an infinite loop for a plethora of inputs. As seen in Listing 8, it transformed the constant 1 to a constant 302. Although this mutant is non-equivalent, adding it to the set of non-equivalent mutants depends on the context. In a purely functional testing context, its execution leads to no additional test suite assessment information as this mutant will always be killed by a time out. However, if non-functional aspects such as watchdogs or safe guards are to be tested, using it for test suite assessment may prove useful.

The mutants for which KLEE detected a run time error are divided into mutants causing a *divide by zero* or a *memory access out of bounds* error. Particularly loops in the mutants of: Looping, Fibonacci, GCD, SimpleMath and all sorting algorithms (in sum 14 mutants) produced a run time error after a division by the loop variable was added and the loops started at 0. Memory out of bounds run time errors were produced by all sorting algorithms (in sum 112 mutants) as these use arrays. One example is the modification `0` to `-1` to the start of the loop variable in the first loop of insertion sort. This directly leads to an out of bounds memory access in line 6 of Listing 9. We included the mutants producing run time errors into the category of unknown equivalence category as they exhibit undefined behavior according to the ANSI C standard. Thus, a test suite killing these mutants must particularly aim at finding such defects, which is typically not the case.

The remaining 403 non-compilable mutants typically suffered from a character insertion illegal to the C programming language or from an illegal removal. An typical illegal insertion was a quote in front of a function (i.e. `'void f()`). Typically illegally removed was the condition of a while loop (i.e. `while()`). Since these insertions are easy to avoid, Milu could be improved to produce fewer non-compilable mutants. In addition, Nequivack would also work using a compiler-based mutant generation technique instead of a interpreter-based technique for the transformations. The LLVM compiler would then perform the mutation transformations directly on the bit code used by KLEE and the machine code used by the test suite. This would remove non-compilable mutants and potentially increase the performance [2].

On the level of all programs, the mutant score confidence is 54% and below our baseline. However, the evaluation contains programs of different complexity and with a different mutation score confidence. Thus, we need to assess the mutation score confidence on a unit level. As seen in Tables I the mutation score confidence differs greatly among the examined programs. MSB, MSB1 and PrimePalindrome have 0% confidence as KLEE deemed all mutants unknown. We speculate the reason to be KLEE's inability to symbolically execute an operation within these program. For Branching, Looping, Factorial and SimpleMath the mutation score confidence is above 80% yielding a closeness of the to be obtained mutation score to the actual mutation score. All other programs have a mutation score confidence lower than 80% and it is unknown if the mutation score obtained after test suite execution will be sufficiently close to the actual mutation score.

In sum, we can see that some programs give rise to "good" confidence ratings and some do not. In line with our methodological considerations in the beginning of this paper, this suggests that mutation testing for programs with "high" confidence is unlikely to suffer from distortions of the mutation score as a consequence of equivalent mutants. In contrast, for programs with "low" confidence we simply do not know if non-equivalent mutants are likely to be a concern.

An interesting aspect of the evaluation is a possible correlation between the unit complexity (see Table V) and the mutation score confidence. The possibility of the existence of this correlation becomes even more evident when taking into account the fundamental scalability issues of symbolic execution. However, since we found no false positives in our evaluation, this also hints towards a correlation between the number of equivalent mutants and the program complexity.

Since we did not find any false positives and false negatives, we tentatively conclude Nequivack to be effective for the determination of non-equivalence w.r.t. to the presented unit testing level programs. Thus, the mutant score confidence is effectively determinable using Nequivack before executing any test cases.

D. Summary and Discussion

In summary, the evaluation shows the results of Nequivack to be reproducible, efficient and effective and Nequivack usable in practice for mutant score confidence determination. We are aware that the performed evaluation was on unit-size programs mostly containing a single function. We deliberately chose to perform the evaluation with these programs for two reasons.

Firstly, it is well-known that symbolic execution suffers from scalability issues [5]. [9] uses slightly larger and more complex programs to show equivalence using symbolic execution. However, Nequivack focuses on obtaining the mutation score confidence on a unit level and is only able to do so for low complexity units. On program level, this level of complexity is easily surpassed.

Secondly, when using mutation testing on the unit level, (de-)composition of the mutants and tests is possible. This means that splitting a complex function into multiple simpler functions, mutating them and executing all unit test cases yields no significant difference to mutating the complex function and executing the unit tests on the mutants of the complex function. However, it leads to difficulties when comparing our approach to other approaches in Section IV and non-equivalence on unit level may not mean non-equivalence at a higher level. As an example, take the standard function to compare strings `int strcmp(const char *s1, const char *s2);`. It compares two strings and returns a positive, zero or negative integer if s1 is less than, equal to, or greater than s2 respectively. There exist a plethora of mutants of this function on the unit level. However, if a higher level only checks for equality of strings, these mutants can be deemed equivalent. In addition, optimizations concerning which unit test cases of the test suite to execute for each mutant may be made (e.g. by using test case coverage).

There is an effort involved in annotating the source code. For functions using only primitive arguments, this effort is negligible as only a single annotation has to be added. This addition may even be performed automatically. For non-primitive or fixed arguments, the effort depends on the complexity of the function. For the C programs used in the evaluation, our effort was minimal. However, to gain generalizable results concerning the annotation effort, further (possibly case-study) research is required.

By construction it is impossible to get false positives with Nequivack (i.e. Nequivack proves non-equivalence) as the produced test cases always show non-equivalence. Thus, the mutation score confidence obtained with Nequivack is an under approximation as no false negatives cannot be guaranteed. We expect the number of false negatives to be low, given the fact that the eliminated mutants in the experiments presented in this paper were manually inspected and no false negatives were found. However, having any false negatives vastly changes the perspective on the mutation score confidence. Since some non-equivalent mutants may have unknown equivalence, the mutation score confidence given by the results of Nequivack may be lower than its actual value. Thus, we set our baseline to 80%. However, large scale research in practice is required to gain insights on practically useful baselines. One idea to detect false negatives could be to also run the test suite and check which mutants are actually killed. Nevertheless, we believe that Nequivack is still a useful tool to create an under approximation of the mutation score confidence.

IV. Related Work

This work proposes an addition to mutation testing introduced by DeMillo et. al. [1]. DeMillo extensively discuses equivalent mutants as one of the core problems of mutation testing. Although approaches have been developed, which produce fewer equivalent mutants (e.g. by Offut [3]), the equivalent mutant problem is still one of the fundamental problems increasing the ”amount of human effort” in mutation testing [2].

In previous work, the equivalent mutant problem has been addressed by using compiler-based techniques [10], program slicing [11], genetic algorithms [12], run time profiles [13] and constraints satisfaction [14], [15].

Using compiler optimizations and “deoptimizations”, Baldwin and Sayward [10] proposed six types of compiler optimization rules to detect equivalent mutants. Similarly, Papadakis et. al. [16] use trivial compiler equivalence to detect equivalent mutants and present a large-scale study on its effectiveness. These detection techniques are fast and are able to detect between 9% and 100% of all equivalent mutants in an evaluation [16]. Hierons et al. [11] created a program slicing approach to assist humans in the detection of equivalent mutants. Adamopoulos et al. [12] propose to use a genetic algorithm with a fitness function to detect equivalent mutants. Run time profiling was used by Ellims et. al. [13] to detect equivalent mutants as the run time profile is similar to the original program. Similarly, Schuler and Zeller [17] examined the coverage of mutants to determine equivalence. Since any approach showing equivalence is naturally able to show non-equivalence by negation, these approaches can also be to detect non-equivalent mutants and give a mutation score confidence. We speculate to be as efficient as compiler optimizations, run time profiling and coverage analysis and more efficient than slicing for human inspection and genetic algorithms based on the results of [9]. For a direct comparison, these approaches will have to be applied to the unit size programs that yield a sensible mutation score confidence.

The approach closest to Nequivack was proposed by Offutt and Pan [14], [15] almost two decades ago. They express mutant equivalence as a constraint solving problem concerning the path condition over all program paths. Due to advances in constraint solving improving symbolic execution [5], Nequivack is a natural evolution of Equivalencer by Offut and Pan. However, while Equivalencer aims to show equivalence, Nequivack aims to show non-equivalence. Still the same predictions about significant better efficiency compared to human equivalence checking [14], [15] apply. Since the Equivalencer tool works for programs writing in COBOL, we were not able to compare its results with the results of Nequivack on a common set of programs. Bardin et. al. [18] propose to specify test requirements as labels. These labels can also specify program equivalence as constraint problem to solve. However, the specification of the constraints is manual.

Symbolic execution has been combined with model checking to verify the equivalence of sequential and parallel numerical programs [19]. Even closer to our work, KLEE has been proposed as a tool for low-effort equivalence verification [9]. However, these works do warn about the fact that equivalence is guaranteed only on the finite set of path that are explored by the symbolic execution engine. In comparison, our approach inverts the perspective on equivalence by proving non-equivalence using a counterexample to equivalence. If no such counterexample is found, equivalence is deemed unknown and we do not consider the mutant for mutation testing.

Rice's Theorem states, that non-trivial properties of a program are undecidable [20]. Determining if two programs are equivalent is one of those non-trivial properties. This property was also declared a "grand challenge for computing research" by Tony Hoare [21]. This is because showing that two programs are equivalent requires a proof that the output of the two programs are the same for any possible input. On the other hand, our approach of showing that a mutant is not equivalent to the original program is a much easier task due to the fact that only one counter-example to equivalence is required to prove this.

Symbolic execution and mutation testing has also been combined in previous work for other purposes than our approach of non-equivalence checking of mutants. Papadakis and Malevris [22] used symbolic execution for mutants to automatically derive test suites. By using this approach mutation testing is used for test case generation instead of assessment. Thus, the created test cases can be grouped into defect-based testing [23]–[25]

The above cited works constitute technical rather than methodological contributions. In contrast, our work is both technical—by using KLEE for non-equivalence checks—and methodological, by leveraging the use of technology to the notion of mutation score confidence.

V. Conclusion

In this paper, we have proposed a new approach for assessing the mutation score by introducing the mutation score confidence. The mutation score equivalence is defined as the ratio of proven non-equivalent mutants and all mutants. The mutation score confidence is able to give an a priori assessment whether evaluating the test suite using mutation testing is practically sensible. This deviates from earlier approaches, where equivalent mutants were sorted out instead and no indication as to how many equivalent mutants remain is given. To arrive at the set of non-equivalent mutants, we provide an open-source tool called Nequivack based on the KLEE symbolic execution engine. Nequivack can be applied to programs written in the C programming language at a unit level. In mutation testing, Nequivack is placed between the creation of the mutants and the execution of the test suite as an additional step. It applies six steps to determine non-equivalence of a program starting from a simple static analysis ramping up to a symbolic execution. Using the non-equivalence results, the mutant score confidence can be calculated.

We evaluate Nequivack using a set of 14 unit size C programs containing basic algorithms on unit level. These programs are mutated using the Milu tool [8], which generated a set of 1352 mutants. Nequivack was able to filter out more than 50% of the generated mutants because they were either equivalent, not suitable for mutation testing or gave a run time or compilation error. We showed that the results of Nequivack are 100% reproducible and its efficiency by the fact that it can process a single none-equivalent mutant in 3 seconds on average and 24 seconds worst case. We were able to produce the mutant score confidence for each program resulting in a correlation to the complexity of the program. However, since no falsely classified mutants were found in the evaluation, there may also be a correlation between the complexity of the program and the resultant number of equivalent mutants.

In the future, we want to extend Nequivack to support further C data structure types. These include structures with pointers and arrays, nested structures, pointers to pointers and pointers to arrays. This would yield a significant increase in the space of programs analyzable by Nequivack. In addition, characterizing the mutation transformations and applying a directed symbolic execution as discussed by Person et. al. [26] could lead to a significant efficiency gain. Orthogonally, we plan to investigate the consequences that a high-number of false negatives of our approach would have for the mutation score confidence.

To reduce the workload of Nequivack, it is also possible to combine different techniques of mutant equivalence detection. Using trivial compiler equivalence [16] before using Nequivack could already remove many equivalent mutants. This is particularly interesting since Nequivack requires a shorter time to show non-equivalence than equivalence.

Finally, our work on non-equivalence can be applied in other areas of research such as program transformations. Program obfuscation research is one possibly interesting direction as finding non-equivalence of an obfuscated program w.r.t its original is essential in finding bugs while developing any obfuscation tool.

References

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

[2] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 649–678, Sept 2011.

[3] A. Offutt, "A practical system for mutation testing: Help for the common programmer," in *ITC*. IEEE Computer Society, 1994, pp. 824–830.

[4] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.

[5] C. Cadar and K. Sen, "Symbolic execution for software testing: Three decades later," *Commun. ACM*, vol. 56, no. 2, pp. 82–90, Feb. 2013.

[6] C. Cadar, D. Dunbar, and D. Engler, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. OSDI*, 2008, pp. 209–224.

[7] ——, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'08. Berkeley, CA, USA: USENIX Association, 2008, pp. 209–224.

[8] Y. Jia and M. Harman, "MILU: A Customizable, Runtime-Optimized Higher Order Mutation Testing Tool for the Full C Language," in *Practice and Research Techniques, 2008. TAIC PART '08. Testing: Academic Industrial Conference*, Aug 2008, pp. 94–98.

[9] D. A. Ramos and D. R. Engler, "Practical, low-effort equivalence verification of real code," in *Computer Aided Verification*. Springer, 2011, pp. 669–685.

[10] D. Baldwin and F. Sayward, *Heuristics for Determining Equivalence of Program Mutations*, ser. Department of Computer Science: Research report. Yale University, Department of Computer Science, 1979.

[11] R. Hierons, M. Harman, and S. Danicic, "Using program slicing to assist in the detection of equivalent mutants," *Software Testing, Verification and Reliability*, vol. 9, pp. 233–262, 1999.

[12] K. Adamopoulos, M. Harman, and R. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Genetic and Evolutionary Computation GECCO 2004*, ser. Lecture Notes in Computer Science, K. Deb, Ed. Springer Berlin Heidelberg, 2004, vol. 3103, pp. 1338–1349.

[13] M. Ellims, D. Ince, and M. Petre, "The csaw c mutation tool: Initial results," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION, 2007. TAICPART-MUTATION 2007*, Sept 2007, pp. 185–192.

[14] A. Offutt and J. Pan, "Detecting equivalent mutants and the feasible path problem," in *Computer Assurance, 1996. COMPASS '96*, Jun 1996.

[15] ——, "Automatically detecting equivalent mutants and infeasible paths," *Software Testing, Verification and Reliability*, vol. 7, no. 3, pp. 165–192, 1997.

[16] M. Papadakis, Y. Jia, M. Harman, and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1, May 2015, pp. 936–946.

[17] D. Schuler and A. Zeller, "Covering and uncovering equivalent mutants," *Software Testing, Verification and Reliability*, vol. 23, no. 5, pp. 353–374, 2013.

[18] S. Bardin, M. Delahaye, R. David, N. Kosmatov, M. Papadakis, Y. L. Traon, and J. Y. Marion, "Sound and quasi-complete detection of infeasible test requirements," in *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, April 2015.

[19] S. F. Siegel, A. Mironova, G. S. Avrunin, and L. A. Clarke, "Using model checking with symbolic execution to verify parallel numerical programs," in *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ser. ISSTA '06. ACM, 2006.

[20] H. G. Rice, "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, vol. 74, no. 2, pp. 358–366, 01 1953.

[21] T. Hoare, "The verifying compiler: A grand challenge for computing research," *J. ACM*, vol. 50, no. 1, pp. 63–69, Jan. 2003.

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

[23] L. Morell, "A theory of fault-based testing," *IEEE Transactions on Software Engineering*, 1990.

[24] A. Pretschner, D. Holling, R. Eschbach, and M. Gemmar, "A generic fault model for quality assurance," in *Proc. MODELS*, 2013, pp. 87–103.

[25] A. Pretschner, "Defect-Based Testing," in *Dependable Software Systems Engineering*. IOS Press, 2015, to appear.

[26] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," in *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '11. New York, NY, USA: ACM, 2011, pp. 504–515.