DYNAMUT: A MUTATION TESTING TOOL FOR INDUSTRY-LEVEL EMBEDDED SYSTEM APPLICATIONS

Darin Weffenstette and Kristen R. Walcott

Department of Computer Science, University of Colorado, Colorado Springs, USA
dweffens@uccs.edu and kwalcott@uccs.edu

ABSTRACT

Test suite evaluation is important when developing quality software. Mutation testing, in particular, can be helpful in determining the ability of a test suite to find defects in code. Because of challenges incurred developing on complex embedded systems, test suite evaluation on these systems is very difficult and costly.

We developed and implemented a tool called DynaMut to insert conditional mutations into the software under test for embedded applications. We then demonstrate how the tool can be used to automate the collection of data using an existing proprietary embedded test suite in a runtime testing environment. Conditional mutation is used to reduce the time and effort needed to perform test quality evaluation in 48% to 67% less time than it would take to perform the testing with a more traditional mutate-compile-test methodology. We also analyze if testing time can be further reduced while maintaining quality by sampling the mutations tested.

KEYWORDS

Test Development, Embedded Test Suites, Test Case Sampling, Mutation Testing

1. INTRODUCTION

When engineering a software solution, testing is essential. To ensure a test suite is effective at finding defects, it is important to evaluate the test suite with regard to quality. While code coverage metrics, such as statement or branch coverage, are useful in determining how to improve a test suite, mutation testing has been shown to be a better indicator of the ability of a test suite to find faults in code [17, 21]. Many tools have been created to automate test suite evaluation for unit tests (e.g. [1, 2, 3, 4]). Unfortunately, on embedded systems in industry, functional testing of the whole system is much more common than unit testing [11]. Thus, combined with uncommon build and runtime environments, the time overhead inherent to the embedded platform, and a lack of applicable tools, makes automated test suite evaluation challenging on embedded systems.

Mutation testing is a fault-based technique that measures the fault-finding effectiveness of test suites on the basis of induced faults [13, 15]. Mutation testing evaluates the quality of test suites by seeding faults into the program under test. Each altered version containing a seeded fault is called a mutant. Mutants of the original program are obtained by applying mutation operators. For example, a conditional statement such as if (a < b) results in multiple mutants by replacing the relational operator < with valid alternatives such as <= or !=. A test suite kills a mutant if a test within the test suite fails. After running the test suite on each mutant, a mutation score can be calculated; the mutation score is the ratio of killed mutants to generated mutants. Prior studies have used mutation adequacy to gauge the effectiveness of testing strategies [7, 8, 14, 20].

Many tools have been developed to help support mutation testing. Some of these tools (e.g. Jester [1], MuJava [26]) focus on source code mutation. Yet, modifying source code can lead to many incompilable mutants and introduces a large re-compilation cost toward the creation of all mutants. Other tools focus on bytecode mutation (e.g. Javalanche [31], Jumble [6], and PITest [4]). Bytecode mutation is favorable because changes can be made on-the-fly without recompilation. It is also simpler to mutate. However, mutating bytecode generates mutants that could have never been introduced into the source code due to the use of syntactic sugar, and generated mutants cannot be mapped back to the source code, which hampers manual inspection of mutants. More advanced tools such as MAJOR [22, 20] take a compiler-integrated approach using abstract syntax trees to introduce mutations for easy and fast fault seeding using a domain specific language to configure the mutation process for JUnit tests [20]. These tools help more at the static and runtime levels.

While tools like MAJOR and PITest have been shown to be effective, they are not practical for all applications in industry, and they do not relate to the application during runtime. Particularly in embedded systems, these tools currently have no analogue. Engineering software for embedded systems presents challenges the current tools have not yet over- come. Because most embedded systems have limited memory and processing power in comparison to traditional computers, interpreted languages such as Java are generally not used. Although mutation testing tools exist for C, like MiLu [18], they do not account for the penalties incurred by compilation for an embedded system. Tools like MAJOR and PITest mutate, build and run code all on the same machine, something that is not always possible on embedded systems. Performing all these tasks on one machine allows these tools to run quickly, but when developing on embedded systems, it may take minutes to recompile and deploy code before the test suite can be run. This increased time overhead makes the methods used by current tools inefficient and excessively time-consuming.

We utilize conditional mutation testing to reduce the costs of evaluating an embedded system and its test suite. Instead of injecting one mutation, compiling, deploying, testing, and repeating, conditional mutation injects all the mutations into the code and selectively activates one at a time as executed. With this strategy, multiple mutations can be tested without restarting the software under test (SUT), saving a significant amount of time. We also show how an existing proprietary test suite can be automated for mutation analysis. Finally, we demonstrate a method of reducing the amount of mutations needed to get representative results.

In this research, we develop a tool called DynaMut, which statically injects conditional mutations into C++ code. This tool replaces defined mutation operators with macros, and the macros contain conditional code to select mutants during runtime. DynaMut employs runtime-based conditional mutation so that the software under test only needs to be compiled once, saving overheads incurred during compilation and deployment to an embedded system. In order to allow for greater time saving in mutation testing, this work also analyzes mutation sampling techniques. Simple, evenly-spaced sampling, random sampling, and dithered sampling, a novel form of sampling inspired by electronic test and measurement equipment, are applied to the runtime mutation data gathered.

DynaMut was used to inject mutations into the embedded application, and specific tests from the larger proprietary test suite were chosen for the mutation analysis. The selected tests were automated and data was collected for the generated mutants. Our results show that conditional mutation allowed for time savings between 48% and 67% when compared with a standard mutate-compile-test methodology. Using the gathered mutation data, three sampling methods were then used to reduce the number of mutations with the goal of keeping the mutation score representative across analyses. The dithered sampling technique is shown to be more effective and efficient than either a random sampling or a simple sampling when decimating the data at ratios between one third and one sixth of the original set.

In summary, the main contributions of this paper are:

- Development of DynaMut, a static tool to insert runtime-based conditional mutations into C++ code
- A description of how to alter an embedded application and test suite to perform runtime mutation testing analysis
- An evaluation of the time overheads incurred by using conditional mutation rather than mutate-compile-deploy-based mutations
- A comparison of three mutation sampling techniques for use in a conditional mutation environment

2. Mutations and Sampling

In this section, we discuss work related to mutation analysis and sampling techniques as they relate to our work.

2.1. Mutation Analysis

Mutation analysis is a method of test suite evaluation first implemented in 1980 [10]. To perform mutation analysis, faults are seeded into the System Under Test (SUT). For each fault or mutant, the test suite is run. If the test suite fails, it is said to have killed the mutant. If the test suite succeeds, it did not detect the mutant. A test suite is given a mutation score that is the percentage of mutants killed out of the total mutants seeded. A mutant analyzer seeds faults systematically in order to ensure that the faults are introduced in an unbiased manner.

Many different types of code mutations have been proposed and tested. Unfortunately, using all variations, especially in a large SUT, can be prohibitively expensive due to the time it would take to test each mutation. Offutt et al. researched different mutation types in [25] and determined a subset of operators which are effective in mutation testing and do not lose significant data in comparison with larger sets of mutations. Based on the work by Offutt et al. [25], DynaMut focuses on implementing these same mutation operators.

Just et al. [23] perform further research to reduce the mutations needed for Operator Replacement Binary (ORB) operators. Their work notes the importance of keeping a mutant's impact on the code minimal. Trivial mutations, mutations that cause wrong output for all possible input values, should be avoided to reduce runtime of the analysis. Redundant mutations should also be avoided to reduce analysis time and also to prevent skew in the overall mutation score. The work by Just et al. [23] considers Conditional Operator Replacement (COR) and Relational Operator Replacement (ROR). For each COR operator, it was found that four mutation types are sufficient to test for non-trivial and non-redundant mutations for any one operator. Given this, each ROR operator can be replaced by only three mutants, instead of the seven that were proposed.

The case studies presented by Just et al. [23] showed that, compared to replacing all operators with all valid replacements, replacing COR and ROR operators with the sufficient set was able to reduce the total number of mutants generated by 16.9% to 32.3%, depending on the ratio of COR and ROR to all other mutant types. This resulted in improved mutation analysis runtime of between 10% and 34%. They also showed decreased overall mutations scores by 2% to 8%, leading to more accurate assessment [24, 23]. Because of these works, this paper will limit the mutations of ROR and COR operators to those in past work [24, 23]. Apart from the normal operators, the mutations include: true, false, rhs, and lhs. Rhs stands for right-hand side, meaning the right-hand side of the operator is always returned. Lhs stands for left-hand side, meaning the left-hand side of the operator is always returned.

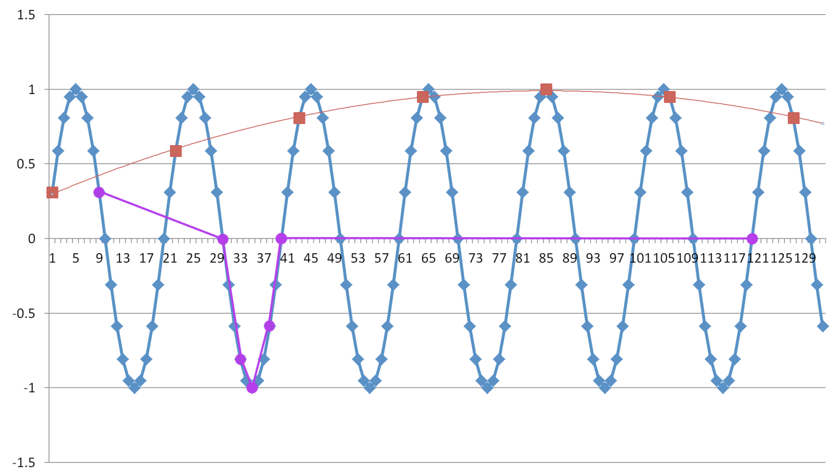

Figure 1(a): Example of a simple sampling and a random sampling technique

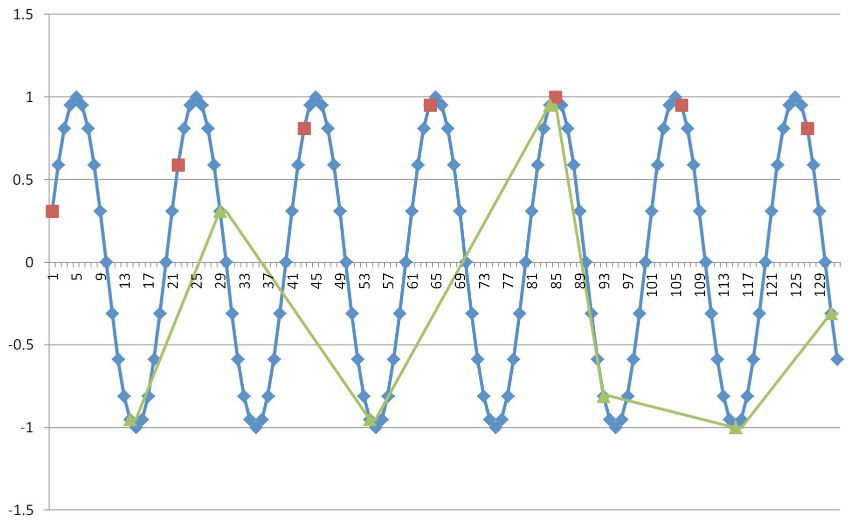

Figure 1(b): Example of a dithered sampling technique

2.2. Sampling Techniques

Given the large number of mutants that can be created using the operators discussed, one can also consider only using subsets of the created mutations. The subsets can be generated using sampling techniques.

Sampling techniques are used in many software engineering fields that gather large amounts of data including profiling (e.g. [12] and testing (e.g. [28]).

There are many sampling techniques including simple even sampling, random sampling, and dithered sampling. When sampling, we attempt to represent the full set of data, keeping a high level of quality while gathering less information. We hypothesize that these sampling techniques can be applied to other kinds of data sets, in this case, mutation testing, to reduce the cost of such testing. This work applies sampling techniques to reduce the amount of data needed to achieve representative results.

In Figure 1a, a set of data is represented by the blue diamonds, where each diamond is a data point. This data could be a typical sine wave as acquired by test and measurement equipment. If one wanted to decimate that data, one option would be to select every twenty-first point. Decimated data is represented by the red squares. As can be seen, this greatly misrepresents the actual data. If this data was presented, a user might think the signal was a sine wave at 1/21th of the frequency. A better technique would, for every 21 samples, pick one sample randomly. With this sampling technique, called dithered sampling, the signal would look like noise. However, noise can be a better representation of the data and would likely allow for some measurements to occur with greater accuracy than simple sampling.

Figure 1b shows an example of the data using a dithered sample technique. The green triangles represent this new set of data. As can be seen, it looks like noise. However, unlike the evenly sampled data, one could measure the amplitude with decent accuracy. Measurements of frequency may be incorrect, but the results may still be more accurate than that of the evenly-spaced samples. The amount of decimation in this example is extreme. Clearly, it is desirable to preserve as much of the data as possible to reconstruct the true data, but it is a good example of how sampling can affect a measurement.

Other software engineering works [32, 33] have used random sampling to reduce the number of mutations needed. Figure 1a shows an example random sampling represented by the purple circles. In this case, seven data points are sampled, and five of them are clustered. This cluster represents one part of the signal well, but as random sampling makes no attempts to spread out samples, entire sections of the data are missed. In this case, the repeated pattern of the sine wave is not represented well; much of the signal presents as a constant value.

The sequence of mutations seeded by DynaMut exhibits a repeating pattern. For each source file, different kinds of mutations are seeded from the top of the file to the bottom. This same pattern repeats across the many files. One might think of the gathered data as a kind of sine wave through the code, although it would not be as clean as the waves in Figures 1a and 1b. Unlike random sampling, dithered and simple sampling will ensure all areas of the code are represented in the mutation score. In addition, dithered sampling can ensure that the sampled data is not misrepresenting data based on recurring patterns. For these reasons, dithered sampling may provide better mutation testing data than either simple sampling or random sampling in mutation testing.

3. IMPLEMENTATION

In order to create a tool that can perform automated mutation testing on embedded de- vice applications, we created DynaMut, a conditional mutation testing tool with varying sampling rates. Firstly, DynaMut includes a static tool to insert calls to centralized functions or macros from all mutation sites in the code. DynaMut is configurable for different software projects, and it can be easily extended for other programming languages. In this section, we explain how projects can be revised and configured to work with DynaMut along with examples.

To reduce the cost of performing mutation testing on embedded software and mutation data gathering, a dynamic/conditional mutation approach is taken to assist with mutation analysis. While other tools are available to help in mutation testing and mutation test analysis in general, they are unable to work with C++ programs. For example, tools such as Nester, Major and

```
<?xml version="1.0"?>
<ProjectConfiguration>
  <WorkerThreadsCount>4</WorkerThreadsCount>
  <IncludeAbsoluteDirectory>C:\Work\Project\source</IncludeAbsoluteDirectory>
  <IncludeFileExtension>*.cpp</IncludeFileExtension>
  <ExcludeDirectory>bin</ExcludeDirectory>
  <ExcludeFile>test.cpp</ExcludeFile>
  <ExcludeFile>gui*</ExcludeFile>
</ProjectConfiguration>
```

Figure 2: Example of ProjectConfig.xml file

PiTest [3, 2, 4] cannot be easily adapted to work with C or C++ code due to their design. When working with C++ or C code, common in embedded systems, tools such as MAJOR and PITest, which both mutate Java bytecode, are unsuitable for most embedded applications. Nester does alter source code with function calls at the mutation sites, but it has not been actively developed, and it is not as configurable as this research requires. This led to our development of DynaMut- A Dynamic Mutation testing tool for embedded system applications.

DynaMut is highly configurable. In this way, it is usable on different systems with var- ied programming languages. Two configuration files are used to control it. First, Dy- naMut imports all the code files that will be mutated. Figure 2 shows the contents of a sample configuration file. With just four rules including IncludeAbsoluteDirectory, IncludeFileExtension, ExcludeDirectory, and ExcludeFile, any complicated directory structure can be navigated. One or more IncludeAbsoluteDirectory rules must be set, and DynaMut will search all children folders. One or more IncludeFileExtension rules must be set to define what types of files may be included. The remaining two rules are optional, and can be used to exclude directories and files.

```
#define MUTATION_INDEX(index) (index + 200)
// Code before mutating
for (int i = 0; i < 5; ++i)
// Code after mutating
for (int i = 0; DYNAMUT_LESSTHAN(i, 5, MUTATION_INDEX(1)); DYNAMUT_PREINC(i,
    MUTATION_INDEX(0)))
```

Figure 3: Example of code before and after DynaMut mutation

Next, DynMut configures how the code is mutated. Each mutation group can be one of three types: OperatorReplacementUnaryGroup, OperatorReplacementBinaryGroup, or LiteralValueReplacementGroup. For each mutation group, three things must be specified: RegularExpression, NumberOfMembers, and the GroupMember variations. The Regular-Expression should contain a regular expression to match the operator(s) and operand(s). The NumberOfMembers specifies how many operators the regular expression matches. Each GroupMember specifies three things: the Operator, NumberOfMutations, and the Replace-mentFunction text. The Operator should contain the operator so DynaMut can detect which operator in the group is matched. The NumberOfMutations should specify how many variations the conditional code will use to mutate a given operator. This is used by DynaMut to space out the constants placed in the function calls. The ReplacementFunction contains the function call being used.

Because of the amount of text parsing performed by DynaMut, it can be extremely resource-intensive dependent on the application. To assist in reducing the time overhead of analysis, DynaMut is implemented in a way that allows for multithreading. Each task can run in an independent thread. A task thread is created for each code file, and each is placed in a Thread

Pool. The number of threads running at one time can be controlled by the WorkerThreadCount in the ProjectConfig.xml file, as can be seen in Figure 2. Because each file is altered individually, each file's index mutation starts at zero, but these values are placed in a macro, as can be seen in Figure 3. After every file has finished being seeded, DynaMut defines the MUTATION_INDEX macro, which contains an offset to make sure that each mutation has a unique ID across the entire software project.

When adding mutations, it is important that functionality of the original code is not changed. Operators are placed in groups with operators that possess the same level prior- ity in the target language's order of operations. This ensures that order of operations does not change code functionality unintentionally. Another consideration is how the regular expression gets matched. For the groups which use left-to-right precedence, the 'lhs' capturing group ends with a question mark. This tells the regular expressions parser to match the fewest number of characters, ensuring the left-most operator gets captured first.

Rules were also added to the DynaMut code to skip regular expressions matched in certain conditions. For example, DynaMut has logic to detect if the match is in a comment or a string. If in a string declaration, nothing is changed. If the match occurred in a comment, the operator is removed to make matching faster the next iteration. DynaMut also includes rules to detect addition of strings (strings can be added but not subtracted) and subtraction of pointers (pointers can be subtracted but not added). This aids in helping the applications under test to build successfully following mutation.

4. EVALUATION

The primary goal of this paper's research is to demonstrate that mutation testing can be performed on complicated embedded systems in industry. In the evaluation of DynaMut, we will:

- Discuss the criteria used to select the tests used
- Analyze the run-time data gathered and estimate how much time was saved with run- time conditional mutation testing versus mutate-compile-deploy testing
- Explore ways of reducing cost of testing through sampling of mutations
- Discuss these results and how they can be applied to reduce the cost of mutation testing in an embedded system environment

```
#define DYNAMUT_PREINC(operand, mutationId) \
    (cDynaMut::CheckMutation(mutationId) ? --(operand) : ++(operand))

#define DYNAMUT_LESSTHAN(lhs, rhs, mutationId) \
    (cDynaMut::CheckMutation(mutationId) ? (lhs) <= (rhs) : \
     (cDynaMut::CheckMutation(mutationId + 1) ? (lhs) != (rhs) : \
      (cDynaMut::CheckMutation(mutationId + 2) ? false : \
       (lhs) < (rhs))))
```

Figure 4: Example of macros used in Keysight$_C$ to define conditional mutation behavior

Within the evaluation, we examine the time overhead of conditional mutation testing in an embedded environment versus the traditional mutation-compile-deploy approach and evaluate how sampling techniques can be applied to reduce the number of test runs without reducing effectiveness.

4.1. Case Study

DynaMut was evaluated on Keysight$_{app}$, a proprietary industry-level tool. Keysight (Keysight$_{app}$) is a tool that is an embedded application with about 1 million LOC. The development team of Keysight$_{app}$ maintains a manually-developed test suite that uses re- mote commands to perform tests; this suite (Keysight$_{suite}$ can take several hours to run, depending on the mode in which it is run.

DynaMut was run on Keysight$_{app}$ across 492 code files. This yielded approximately 121,000 mutations. In order to use Keysight$_{app}$ with DynaMut, a number of issues were identified that could cause DynaMut to make improper/uncompilable mutations.

After adding these new rules into DynaMut, Keysight$_{app}$ was compiled in release mode. However, the application would not start up fully, encountering errors. These likely are DynaMut mutations that, although not incorrect enough to cause compile errors, caused a change in behavior that proved fatal. Because of the scope of this project, it was decided to limit mutations to a single subsystem of Keysight$_{app}$, consisting of 49 code files, or 10% of the total number of files. With this limitation, the application runs normally when mutations are not in effect. While this section of code represents only a portion of Keysight$_{app}$, it is an important behavioral subsystem that is tested by the majority of the tests in Keysight$_{suite}$.

4.2. Modifications to Keysight$_{app}$

In addition to the code changes made by DynaMut, a small amount of code was added to the application to enable conditional mutation operation. Macros were created to control the mutations. Figure 4 shows two examples of the macros added to Keysight$_{app}$. As can be seen, the macros make calls to the static function cDynaMut::CheckMutation. CheckMutation returns true if mutation is enabled and the mutationId parameter matches the static variable containing the currently-active mutation. In this way, only one mutation is in effect at any point in time. In addition to controlling the active mutation, the cDynaMut class also was designed with the ability to track mutation coverage.

4.3. Automating Keysight$_{suite}$ for Data Collection

The full test suite, Keysight$_{suite}$, takes several hours to run all tests on Keysight$_{app}$. For this reason, it was decided to only use small tests from the larger suite for evaluation. MutationTestRunner performs the following tasks to automate mutation testing data col- lection:

- Imports the csv file containing the covered mutations gathering information on the tests being run only.
- Communicates with the remote device running Keysight$_{app}$ to control the mutation.
- Communicates with a remotely-controlled power strip to reboot the remote device when necessary.
- Runs Keysight$_{suite}$ command line utility and captures output.

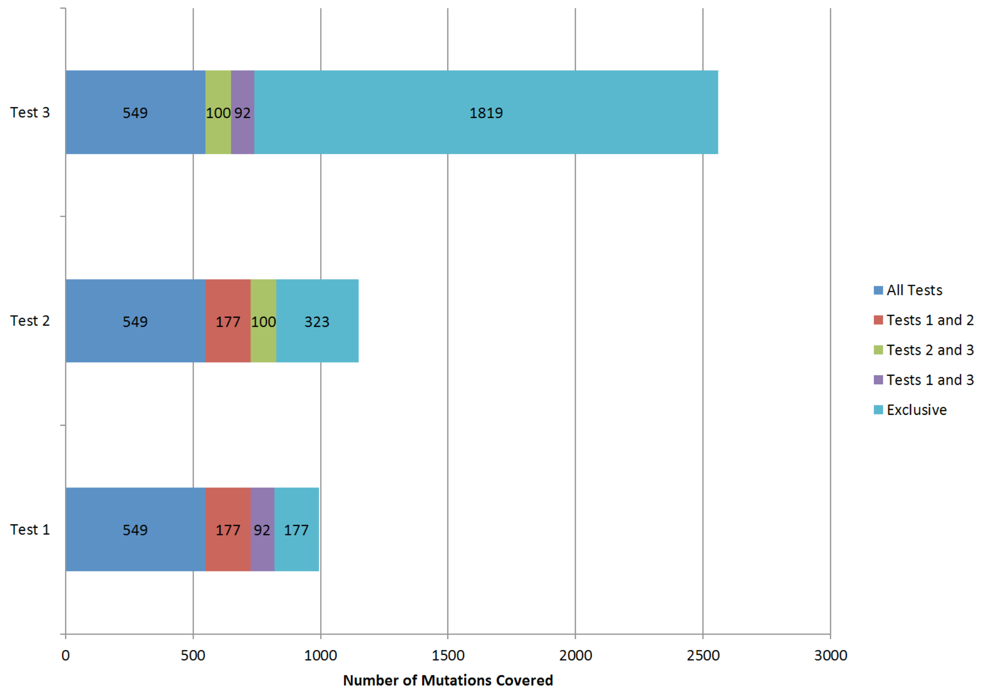

Figure 5: Mutations covered by each test and # of mutations covered by multiple tests

All changes were made to the subsystem of Keysight$_{app}$ [5], which represents the key and behavioral subsystem and represents the main functions of the application. All testing is performed on the most recent accepted build of the code. For each test subsuite, a training run was performed with the coverage tracking feature enabled. This produced a coverage file enumerating the indices of all the mutations covered by a given test subsuite. The MutationTestRunner utility was then used to automate running tests in Keysight$_{suite}$ on the mutations specified by the coverage file.

4.4. Test Selection for Test Suite and Mutation Scores

Because of the size of test suite, Keysight$_{suite}$, evaluation was performed on a limited number of tests from the entire suite. To select the three test subsuites used in this evaluation, tests were selected that 1) could be run one time in under 30 seconds, 2) that could cover at least 1,000 mutations, and 3) tests where the mutations covered by the chosen tests overlap as little as possible in order to provide differing data.

First, Keysight$_{suite}$ was run to determine the time each test would take to execute. Then for the tests that completed reliably in under 30 seconds, training runs were performed to gather the mutation coverage information for each test. With the tests that covered roughly 1,000 or more mutations, the coverage data was analyzed for overlapping coverage– that is, mutations that are covered by two or more tests. Based on this data, three test sets were chosen. They will be referred to as Test 1, Test 2, and Test 3.

With the complete sets of covered mutations, the test suites achieved the mutation scores of 20.9%, 13.3%, and 22.8% for Tests 1, 2, and 3 respectively. While these mutation scores are low, we observe that DynaMut can be useful in identifying parts of code that have not yet been tested. With approximately 121,000 mutations being added to the application, 13-22% can be identified using the existing, provided test suites.

Figure 5 shows the number of mutations covered by each test. It was observed that Test 1 covers 995 mutations and executes in about 23 seconds. Test 2 covers 1,149 mutations and executes in 18 seconds. Test 3 covers 2,560 mutations and executes in 11 seconds. For each test, these mutations are categorized by how many tests were executed and by which tests mutations are covered. There are a significant number of duplicate mutations covered by the tests due to the fact that these tests exercise the same SUT.

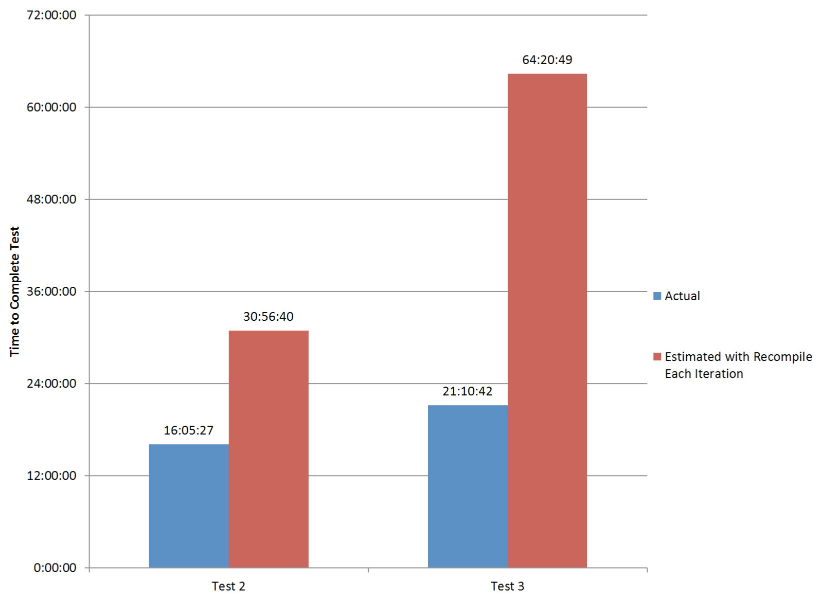

Figure 6 a: Comparison of testing time with conditional mutation and compiled mutation

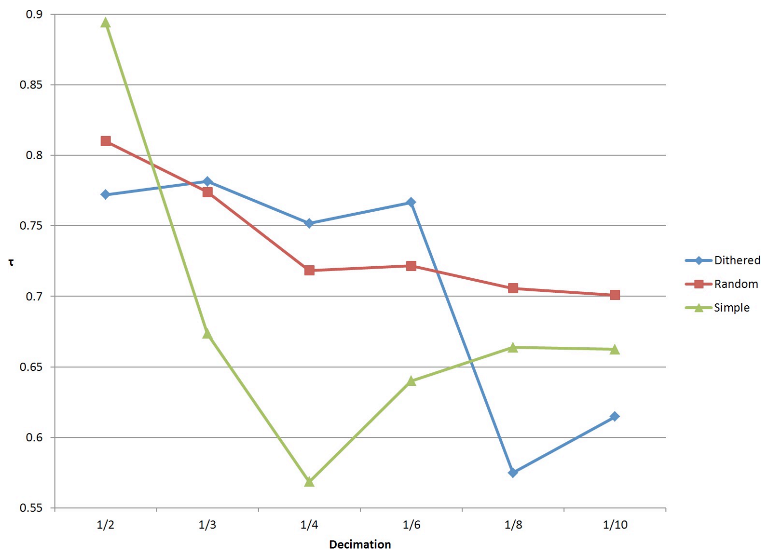

Figure 6 b: Correlation of sampled data to actual test values

4.5. Time Overhead Reduction

The time overhead was analyzed from the test runs. This analysis focuses on the data collected from Test 2 and Test 3 due to their larger mutation coverage. Because of the nature of testing on an embedded system, there is time overhead not normally associated with mutation analysis. When the SUT is running on a host, it can be killed and started quickly, depending on the startup time of the SUT. Because of this, mutation testing is often performed by running a new instance of the application for each mutation. On the embedded system for Keysight$_{app}$, the system must be rebooted to restart the SUT. The boot process, including the time it takes to fully start Keysight$_{app}$, takes about 59 seconds, which is far longer than traditional applications being tested with mutation analysis.

Due to this extra overhead, MutationTestRunner was designed to only restart the embedded system after a test failure. If the test passes with a given mutation, the conditional mutation ID is changed to that of the next mutation, and the test is run again without restarting the embedded system, thus providing significant savings in testing time. Even if the system does not need to be rebooted, about 5 seconds of overhead occurs before every test. This time is incurred when sending the remote commands to tell Keysight$_{app}$ which mutation to enable. Even though these commands are small in size, the steps necessary to ensure reliable operation cause this step to consume 5 seconds. To estimate the time it would take to perform these mutation analyses with a traditional mutate-compile-test methodology, the additional overhead of compiling Keysight$_{app}$ (with only minor changes) and deploying it to the embedded system is estimated to be 15 seconds. This method would also require the system to be rebooted after every test. Because no tools exist that can easily be used with embedded products, estimations are used based on manual testing of Keysight.

As seen in Figure 6a, MutationTestRunner completed the full mutation analysis of Test 2 in 16 hours, 5 minutes and 27 seconds (16:05:27). Of this time, 5:43:49 was spent performing the actual test, and 1:35:45 was spent in the unavoidable overhead described above. 54.47% of the time (8:45:53) was spent rebooting after failures. With a mutate- compile-test method, it is estimated that Test 2 would take 30:56:40, 92.31% more than the conditional mutation method, given the estimates described. Test 3 was performed in a total of 21:10:42. Of this time, 45.26% or 9:35:04 was spent rebooting. The compiled mutation method on Test 3 would take an estimated 64:20:49, or 203.83% more than the implemented conditional mutation method. The estimations of mutation-compile-test times are necessary as there are no tools that can easily perform these actions on embedded applications.

These time estimations assume that the same coverage data would be available for the mutate-compile-deploy method, which would require more static analysis to be performed. Even with this consideration, the conditional mutation method implemented in this work saves an estimated 48.00% of the time to evaluate Test 2 and 67.09% of the time needed to evaluate Test 3.

4.6. Mutation Sampling

To further reduce testing cost, this work evaluates methods of reducing the number of mutations tested. To perform this evaluation, the full results from each Test are analyzed. Each covered mutation either passes or fails. This data was imported into a spreadsheet, where the simple sampling, random sampling, and dithered sampling methods were applied to the data of each test across a variety of decimation factors. For the simply sampled sets, all possible sets of evenly-spaced samples were determined for each decimation proportion and test (for example, at 1/2 decimation, there are only 2 possible sets for each of the 3 tests). For the dithered and random sample sets, data was gathered for 10 samples of each decimation proportion and test. The data was then used to correlate each decimation proportion and sampling type to the score obtained from the full set of data. Correlation was calculated using [29] to get the Kendall's τ factor. The decimation proportions used are: 1/2, 1/3, 1/4, 1/6, 1/8, and 1/10 the total number of covered mutations.

Kendall's τ factor is a measure of how one set of data correlates to another. It can be from 1 to 1, where 0 means there is no correlation, 1 means there is absolute positive correlation, and -1 means there is absolute inverse correlation. For this work, closer to 1 is more desirable. Figure 6b shows the results of the correlation analysis. At 1/2 decimation simple sampling correlates better to the actual data; however, this might be misleading because the dithered and random data each have 10 data points per test compared to the simple sampling's 2 points per test. At 1/3, 1/4 and 1/6 decimation, the dithered sampling provides better correlation than both the simple sampling and the random sampling. At 1/8 and 1/10 decimation, simple sampling provides better correlation than dithered sampling, although neither provide very good correlation; random sampling manages to provide the best result at these decimation ratios.

Sampling can also be used to reduce the number of mutations tested. Dithered sampling offers better correlation to the true values; however, as the data is decimated further, the risk of gathering non-representative samples increases. This risk must be balanced between the effectiveness and efficiency of testing that is needed.

4.7. Discussion

While the sampling methods are correlated to the actual values from the full set of covered mutations, we did not determine how much time the sampling techniques would save versus other techniques due to a lack of tools that can perform similar tasks. Because of the overheads present, it cannot be assumed that testing 1/2 of the mutations would save 50% of the testing time. Assuming the decimated set of mutations exhibit a similar pass/failure rate as the whole set, the time overhead scale is predictable based on our preliminary tests.

In addition, although the correlation of dithered samples remains relatively constant between 1/2 and 1/6 decimation, that does not make them equally good options for testing. The chances of obtaining an outlier or biased result increases as the sample decreases, so 1/6 decimation would not be as accurate a method as 1/2 decimation.

Overall, we learn that DynaMut can perform analysis of an embedded application and that it can be adapted to work on other applications and languages. While every embedded program has its own specifications, DynaMut provides options to configure programs to match the tool and modifications needed to work with the tool. Sampling techniques can also be used and modified, where the user can select between multiple sampling types and rates. DynaMut in itself can provide a method to test embedded programs on a mutation level and gives sampling options given testing efficiency needs.

5. THREATS TO VALIDITY

This work tested a small portion of the tests in Keysight$_{suite}$. With a limited number of the tests from the test suite, testing with mutations in only 10% of Keysight$_{app}$, testing still took between 16 and 21 hours per test subset. There is room for improvement. However, the tests selected in this work covered the majority of the functions specified by users and main program functions.

Also, the research is only based on an industry level application. The selected application is a large, proprietary embedded application, where testing was focused on the primary functions of the application. We believe that the results can be extended to other C and C++ based embedded applications given the additional modifications that were incorporated into DynaMut. However, these need to be tested and evaluated. More applications representing embedded software are needed.

This work evaluates conditional mutation testing and sampling techniques on one embedded system and one software system. These results may not translate to other embedded systems or software packages. We tried to mitigate this possibility by using a variety of tests without regard to the system under test. While we only focus on patterns inherent in Keysight$_{app}$ and Keysight$_{suite}$, these were not the focus when designing the tool. Keysight$_{app}$ was used as a learning an evaluation tool, but general application designs were considered during DynaMut's implementation.

6. Related Work

This paper deals with efforts to reduce the costs of mutation analysis to make it practical for testing an embedded system in industry. Much of this saving is needed in the runtime of the mutation analysis due to the size of the SUT and test suite. Consideration has been put into reducing the overall number of mutations used. This work is based on the studies performed in [25, 23, 24] to reduce the number of mutations seeded in the code. In our tool, fewer mutations being seeded results in time savings during mutation testing.

Another way to save time during mutation analysis is to reduce the compilation time. Just et al. propose a method of increasing the efficiency of mutation analysis in [22]. In their work, they manage to save compilation time by introducing conditional mutation. In this method, the compiler inserts conditional code at each mutation site for all possible mutations, and a global state variable controls which mutation is in effect. The introduction of conditional code at each mutation site introduced a large amount of code overhead. On the applications tested, the instrumented code compiled to a size between 18% and 66% larger than the original program. Nester [3], a mutation testing tool for C# ported from Jester [1], takes a slightly different approach. It replaces various operators with calls to a set of central functions, instead of placing the conditional code at each location. These functions contain the conditional code to allow one mutation to occur at a time, but should incur far less code overhead. This is more important in an embedded system where memory is constrained. Our work uses conditional mutation similar to that used in [22]; however, this work uses macros to introduce the conditional code. Because the mutations were limited to a subsystem of Keysight$_{app}$, and a significant portion of the compiled binary is devoted to GUI-related non-code data, the memory overhead and time overhead introduced were not evaluated.

For embedded systems, reducing the frequency of compilation has an added bonus, which differs from systems such as Just et al. [22]. The SUT is compiled on a workstation, and then it is deployed to the embedded system. Combined with software startup time, and the time necessary to reboot the embedded system between code runs (although, in theory the embedded system could be rebooted during compilation), every deployment can add approximately one minute to the time overhead of mutation analysis. Being able to compile the SUT and deploy it only once therefore can yield much more benefit in this case than on systems where the SUT is run on the same computer on which it is compiled.

This paper also proposes a method of sampling mutations to reduce the cost of mutation testing. Dithered sampling has been used for a long time in analog and digital test and measurement equipment [30, 16]. This equipment can generate large data sets, and decimating that data can be useful for improving performance of measurements, analysis or visualization. Dithered sampling ensures that this decimation does not inadvertently misrepresent the original data. This paper shows that application of this dithered sampling can provide more representative samples than either a random sampling or an evenly-spaced simple sampling. Other works have studied sampling techniques applied to mutation testing. In [33], Zhang et al. compare random mutation sampling to techniques of reducing mutation by reducing the set of operators used. They found that random mutation sampling can be just as effective. In later work, (a different) Zhang et al. combine random sampling of mutations with reduction of operators used, and show that the combination of techniques yields precise results with far fewer mutations [32]. This second work evaluates eight different random sampling techniques. Their baseline stratagem is equivalent to the random sampling of this paper. The other strategies select a certain percentage of mutants from each set of mutants: generated from a single operator, generated inside a given program element (e.g. class or function), or a combination of the previous. In practice, the dithered sampling in this paper may behave similarly to the technique of selecting a percentage of mutations within a given program element; however, dithered sampling requires no extra code analysis to perform, making it easier to implement with a simple tool like DynaMut.

Embedded systems are often tested using model-based approaches. Tan et al. demonstrate an integrated framework for development of self-testing model-based code [27]. Bringmann and Krämer introduce a tool to perform model-based testing on automotive embedded systems in [9]. While these embedded systems are amenable to model-based testing, not all embedded systems are. Like the in-industry case study of [19], the embedded application Keysight$_{app}$ is a large piece of software that is highly configurable. The size of these applications makes model-based testing or development impractical. These applications often have evolved over a series of product iterations, during which tests have been added to a proprietary test suite. This work evolves test suite evaluation methods to work with one such test suite.

7. Conclusion and Future Work

This paper demonstrates that mutation testing can be performed on embedded systems in industry. DynaMut inserts runtime conditional mutations into a SUT, then demonstrated how to automate collection of data using an existing proprietary test suite. Conditional mutation was used to reduce the time and effort needed to perform this testing. The mutation testing was performed on three tests chosen from a larger suite of tests. It is estimated that the conditional mutation technique saves between 48% and 67% of the time it would take to perform the testing with a more traditional mutate-compile-test methodology.

The data is further analyzed to determine if testing time could be further reduced by sampling the mutations tested, rather than testing all the covered mutations. Dithered sampling proves to perform better than simple evenly-spaced sampling or random sampling in both efficiency and effectiveness.

The techniques used in this paper could be enhanced to further reduce testing costs. In future work, we would use multiple test fixtures to allow for testing of mutations in parallel. This would likely be an effective way to reduce testing time. It would also be interesting to apply the dithered sampling algorithm to larger data sets and more applications to ascertain its relative effectiveness.

8. References

[1] http://jester.sourceforge.net, September 2014.

[2] http://mutation-testing.org/, September 2014.

[3] http://nester.sourceforge.net, September 2014.

[4] http://pitest.org/, September 2014.

[5] http://www.keysight.com, September 2014.

[6] http://jumble.sourceforge.net/, January 2015.

[7] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? [software testing]. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 402–411. IEEE, 2005.

[8] J. H. Andrews, L. C. Briand, Y. Labiche, and A. S. Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on*, 32(8):608–624, 2006.

[9] E. Bringmann and A. Kramer. Model-based testing of automotive systems. In *Software Testing, Verification, and Validation, 2008 1st International Conference on*, pages 485–493. IEEE, 2008.

[10] T. A. Budd. Mutation analysis of program test data. 1980.

[11] A. Causevic, D. Sundmark, and S. Punnekkat. An industrial survey on contemporary aspects of software testing. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 393–401. IEEE, 2010.

[12] D. Chen, N. Vachharajani, R. Hundt, S.-w. Liao, V. Ramasamy, P. Yuan, W. Chen, and W. Zheng. Taming hardware event samples for fdo compilation. In *CGO '10: Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, pages 42–52, New York, NY, USA, 2010. ACM.

[13] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41, 1978.

[14] H. Do and G. Rothermel. On the use of mutation faults in empirical assessments of test case prioritization techniques. *Software Engineering, IEEE Transactions on*, 32(9):733–752, 2006.

[15] R. G. Hamlet. Testing programs with the aid of a compiler. *Software Engineering, IEEE Transactions on*, (4):279–290, 1977.

[16] M. Holcomb. Anti-aliasing dithering method and apparatus for low frequency signal sampling, May 19 1992. US Patent 5,115,189.

[17] L. Inozemtseva and R. Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. ACM.

[18] 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*, pages 94–98. IEEE, 2008.

[19] D. Jin, X. Qu, M. B. Cohen, and B. Robinson. Configurations everywhere: implications for testing and debugging in practice. In *Companion Proceedings of the 36th International Conference on Software Engineering*, pages 215–224. ACM, 2014.

[20] R. Just. The major mutation framework: Efficient and scalable mutation analysis for java. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 433–436. ACM, 2014.

[21] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, Hong Kong, November 18–20 2014.

[22] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using conditional mutation to increase the efficiency of mutation analysis. In *Proceedings of the International Work- shop on Automation of Software Test (AST)*, pages 50–56, May 23–24 2011.

[23] R. Just, G. M. Kapfhammer, and F. Schweiggert. Using non-redundant mutation operators and test suite prioritization to achieve efficient and scalable mutation analysis. In *Proceedings of the International Symposium on Software Reliability Engineering(ISSRE), pages 11–20, November 28–30 2012.*

[24] G. Kaminski, P. Ammann, and J. Offutt. Better predicate testing. In Proceedings of the 6th International Workshop on Automation of Software Test, pages 57–63. ACM, 2011.

[25] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. ACM Transactions on Software Engineering and Methodology (TOSEM), 5(2):99–118, 1996.

[26] *J. Offutt and N. Li. http://cs.gmu.edu/~offutt/mujava/, January 2015.*

[27] *L. Tan, J. Kim, O. Sokolsky, and I. Lee. Model-based testing and monitoring for hybrid embedded systems. In Information Reuse and Integration, 2004. IRI 2004. Proceedings of the 2004 IEEE International Conference on, pages 487–492. IEEE, 2004.*

[28] *K. Walcott-Justice, J. Mars, and M. L. Soffa. Theme: A system for testing by hardware monitoring events. In Proceedings of the 2012 International Symposium on Software Testing and Analysis, pages 12–22. ACM, 2012.*

[29] *Wessa. Kendall tau rank correlation (v1.0.11) in free statistics software (v1.1.23-r7). http://www.wessa.net/rwasp_kendall.wasp/, 2012.*

[30] *B. Widrow. Statistical analysis of amplitude-quantized sampled-data systems. American Institute of Electrical Engineers, Part II: Applications and Industry, Transactions of the, 79(6):555–568, 1961.*

[31] *A. Zeller. https://www.st.cs.uni-saarland.de/mutation/, January 2015.*

[32] *L. Zhang, M. Gligoric, D. Marinov, and S. Khurshid. Operator-based and random mutant selection: Better together. In Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on, pages 92–102. IEEE, 2013.*

[33] *L. Zhang, S.-S. Hou, J.-J. Hu, T. Xie, and H. Mei. Is operator-based mutant selection superior to random mutant selection? In Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1, pages 435–444. ACM, 2010.*