P8: Software Test Suite Optimisation for Complex Software Systems

Milestone Report – GS-IMTR

Maik Betka

July 14, 2022

Advisors: Prof. Dr. Stefan Wagner, Prof. Dr.-Ing. Steffen Becker, and Dr.-Ing. André van Hoorn

Main examiner: Prof. Dr. Stefan Wagner

Co-examiner: Prof. Dr. sc. Michael Pradel

Mentors: Gerd Bleher and Martin Heinrich

1 Introduction

Our modern electronic devices and digital infrastructure strongly relies on the correctness and robustness of integrated circuits and semiconductor chips. Ensuring correctness and robustness of these is a highly challenging task that requires specialised and equally reliable hard- and software. Advantest[1] is a leading manufacturer of Automated Test Equipment (ATE) which is used exactly for that purpose. However, due to the growing chip complexity and because transistor size shrank so much that physical limits are approached, upcoming chip generations require novel and improved methods for test and reliability. Therefore, Advantest and the University of Stuttgart formally established in 2019 the Graduate School Intelligent Methods for Test and Reliability (GS-IMTR).[2] Ten projects aim to improve semiconductor testing from different angles and form collaborations for interdisciplinary challenges to profit from expertise of different institutes [1].

This milestone report summarises the research work done so far and the program ahead of project P8: Software Test Suite Optimisation for Complex Software Systems. In the context of the GS-IMTR, project P8 aims to improve the operating system and development environment of the ATE, i.e., the software that is used to write and execute chip test programs. The newest generation of that software is called Smartest8, a highly complex software system that is used to compile and load chip test programs into the tester hardware, assist developers to write and debug them, enable the use of various test head cards to perform measurements of, e.g., signals and voltages, and many other features. The software contains more than 16 million lines of code, is written in multiple programming languages, is thoroughly tested, and includes a diverse set of modules ranging from low-level driver code to high-level code for graphical user interfaces. Correct operation of that software is vital for the entire chip manufacturing and testing process. In particular in times of chip shortages, any outages or interruptions due to malfunctioning software is unacceptable and has serious economical impacts.

This PhD project targets to develop and evaluate a novel approaches to optimize software test suites for correctness, robustness, and performance. In particular, we want to improve testing of such a complex software system but also similar ones in general and work on the following three research topics:

Test Suite Analysis: The code base of complex software systems naturally grows over time. The respective test suites also evolve and are written by many developers who have different experience levels and deviating coding styles. Development practices usually change over time as well, which additionally influence how code and tests are written. Thus, quality and effectiveness of existing tests vary, but code coverage alone can be misleading to evaluate the effectiveness of a test suite. Mutation testing [2] is a technique to address this issue. We investigate how usable mutation testing for large test suites is and which pieces are missing for wide industry adoption. Besides functional test suites, a large set of performance regression tests also exist that require advanced analysis techniques to quickly find the root cause for performance losses in new software releases. Therefore, we also want to investigate methods and techniques of how to improve analysis of such benchmarks.

Test Case Generation: In the context of Smartest8, most of the tests are written by hand. These type of tests are particularly strong because they incorporate the knowledge of the developers. However, the final chip test programs are written by customers of Advantest and are executed by Smartest8. These chip test programs are very versatile and are not available to the developers for intellectual property reasons. Because developers only have a limited time budget for testing, this poses the threat to miss to test some important system boundaries and combinations of features which lead to eventually overlooking software faults that are then found by the users of the software. To prevent late software fault detection, we investigate methods of how to automatically generate test cases. In particular, we aim to use and advance fuzzing, which has proven to be a very successful technique in finding real world software faults [3].

Combination of Manual and Generated Test Cases: To find the optimal process to create a cost efficient and fault-revealing test suite, we need to understand what the strengths, limitations, and overlaps of manual and automated test generation approaches are. We want to clarify these points in this project but also combine both approaches and see whether they work well together. For that, we want to create generators for fuzzing from existing hand-crafted unit tests and metamorphic tests [4] by incorporating the expertise of developers and automated tests from fuzzing approaches.

[1] `https://www.advantest.com/`
[2] `https://www.gs-imtr.uni-stuttgart.de/`

2 Mutation Testing

Software test engineers and developers mainly rely on code coverage to conclude to which extent a software system is tested. However, many previous studies [5] have shown that code coverage can be a misleading measure to reason about how effective the existing test suite actually tested the executed lines of a software. One technique to address this issue is mutation testing.

Mutation testing consists mainly of two steps: First, the software under test is slightly modified in a controlled manner, i.e., "mutated," such that it behaves differently from the original software. This version of the software is metaphorically called "mutant." In the second step, the existing test suite is re-run to check whether any test fails and, thus, detects the different behaviour. If no test case fails, it is said that the mutant "survived". Otherwise, it is said the mutant is "killed." This way, the procedure detects code that may have coverage but is not really verified to work the expected way. In particular, test suites for regression testing that kill the majority of mutants are particularly strong in detecting software faults. Related work [6, 2] describes mutation testing as an optimisation problem where the test suite should be enhanced towards a mutation score of 1. The mutation score is the number of killed mutants divided by the total number of mutants.

One major challenge of the technique is the vast amount of mutants that are generated by tools that implement it. Famous examples for Java tools are PIT [7], muJava [8], or Javalanche [9]. Traditional mutation testing performs mutations on an instruction level. For example, comparison operators like `==` may be changed to `!=` or `>` to `>=`. The number of mutants that are produced that way is usually overwhelming for developers, which often leads to rejection in using the technique [6].

To reduce the number of generated mutants, Niedermayr, Juergens and Wagner introduced extreme mutation testing in 2016 [10]. In contrast to traditional mutation testing, extreme mutation testing mutates whole methods instead of instructions, which is also the reason why it is called "extreme." The idea of the approach is to empty method bodies and replace return values with default values. If still no test case detects these drastic code changes, then the mutated method is considered to be "pseudo-tested" because it can be entirely removed without changing the test outcome. Categorizing methods this way is also easier to comprehend than to analyse why an instruction-based mutant survived. Furthermore, since the number of methods in a software project is normally much smaller than the number of instructions, the number of mutants is much smaller as well. However, this performance gain comes at the expense of loosing precision.

A previous study of Niedermayr, Juergens, and Wagner [10, 11] showed that pseudo-tested methods are very common, even in well-tested software projects. In total, 19 open-source projects were studied and each contained pseudo-tested methods. The median proportion of pseudo-tested methods was about 10%. Vera-Pérez, Monperrus and Baudry [12] confirmed that observation in their study where more than 28,000 methods were analysed. However, in the same study, lead developers of three software projects were interviewed, and it was found that only 30 out of 101 pseudo-tested methods were considered to be relevant enough to create a new test case for them. These findings challenged the usefulness of the technique in practice.

For this project, we followed that stream of research and put extreme but also traditional mutation testing as techniques to analyse the strength of a test suite into focus. For that, we conducted two case studies on Smartest8. The goal of the first case study [13] was to find out how the existing tool PIT performs in practice when used with both supported engines: *Gregor* for traditional and *Descartes* for extreme mutation testing. We also wanted to find out what prevents wide adoption in industry. At that time, Smartest8 was already tested by more than 11,000 unit tests and the number of tests continues to grow, thus, making it a good target to evaluate mutation testing. Besides running mutation testing, we also did a qualitative analysis of 25 pseudo-tested methods and discussed these methods with two experienced developers to triangulate our case study results and conclusions.

Our main findings of this study [13] include:

- Pseudo-tested methods are relevant for developers, but not every pseudo-tested method justifies the additional effort to enhance or write further tests for it.
- The execution time difference between traditional and extreme mutation testing hardly matters in practice, but their required times to comprehend and analyse detected issues does matter.
- It does make a difference at which point in time, in the development process, mutation testing is applied.

The first finding confirms the observation made by Vera-Pérez, Monperrus and Baudry [12] in their study. That is, pseudo-tested methods reveal issues that developers want to know about so that they can act upon. However, some of these methods are meant to stay, as they are either not important enough or out of scope. For example, testing whether a logging-method prints the correct log-message is not considered worth testing. This is similar for some cases where a method is executed as a side effect of the unit test that actually targets to test a different method. Finding that an assertion in a unit test is missing, however, is very well noteworthy to act upon.

The second finding regarding the execution time of both approaches was very surprising, since we expected a large difference in execution time between both techniques. Extreme mutation testing finished in 13 minutes and was about three times faster than its traditional counterpart, which finished in 37 minutes. Albeit this difference seems large, it barely matters in practice if executed asynchronously in a development pipeline that builds and tests the software project. Both times are in the order of minutes and therefore negligible when analysing their findings in a separate session at a later point in time. Considering the large amount of unit tests for Smartest8 this difference becomes even less significant for smaller software projects. In the tool demonstration of the extreme mutation testing engine Descartes by Vera-Pérez, Monperrus and Baudry [14] the reported difference was often in the order of hours. We were only able to reproduce this difference in execution time when enabling all traditional mutators of the default engine Gregor. This is, however, not recommended and not the default setting of PIT.

In contrast, the time to analyse detected issues by both techniques noticeably differs. Out of 7,989 traditional mutants, we found that 2,176 mutants survived. Out of 2,041 methods, we found 291 to be pseudo-tested. The time to analyse all issues with each technique could only be estimated. However, it is evident that developers have more than a seven times higher workload by using the traditional approach compared to the extreme approach when looking at absolute numbers. In fact, pseudo-tested methods additionally have the advantage of being pre-analysed. This means, that it is easier to comprehend why a method is pseudo-tested than to deduce why a traditional mutant survived. Therefore, we concluded that, in terms of time investment, extreme mutation testing is superior.

The last finding originated from the interview with the developers. When we asked in which situation, during development, pseudo-tested methods would be most likely addressed, the answer was: while writing unit tests. This differed from other industrial studies like the ones at Google [15, 16] where mutation testing is only applied during code review. This means that many pseudo-tested methods would be addressed during unit testing, as the effort is considered low enough to fix them at that point in time, but too high after a code review.

After that first case study, we have seen that both techniques are fast enough to be executed and applied to software projects in practice. Extreme mutation testing is faster to analyse, but some methods will never be fixed because they are not important enough. We also observed that development processes influence how the technique is used. These findings, however, revealed many research gaps: Since there are pseudo-tested methods present that will never be addressed, we did not know how many traditional mutants are generated on these methods that can be ignored as well. That would mean that using both techniques in conjunction would seem reasonable to filter these traditional mutants. However, it was not clear which issues the traditional approach can find, which the extreme one cannot. It was also not clear under which circumstances developers were actually willing to enhance the test suite and, thus, to kill mutants.

To fill these research gaps, we conducted a larger consecutive case study [17]. For that, we manually inspected more than 1,000 mutants. We selected mutants in such a way to have samples for each of the following categories:

- Mutator types (e.g., conditional boundary, return values, negating conditionals, ...)
- Return types (e.g., integer, strings, objects, ...)
- Testing verdict (i.e., pseudo-tested or not pseudo-tested)
- Mutant status (i.e., survived or killed)

We did not use runtime information by using, e.g., a debugger, but instead relied on the mutation testing report, the code coverage information, the testing verdict, and the code with its documentation within the scope of a method or class of the respective mutant. This approach enabled us to process many mutants, categorize them, and detect patterns across a variety of different properties. Similar to our first case study, we triangulated our results and conclusions by conducting a focus group with five developers. We presented 9 pseudo-tested methods and 15 traditional mutants with their code coverage information and asked if, how, and why an issue would be fixed.

Our main findings of this study [17] include:

- A noticeable amount of traditional mutants are placed on pseudo-tested methods that can be ignored.
- The mutation score is neither a good metric for test adequacy, since it is strongly influenced by redundant or equivalent mutants, nor has it any meaning for developers in practice.
- Developers only want to mutate important methods and not the whole code of the project.
- Developers would only perform mutation testing when a method has high code coverage.
- If a method raises an exception during a unit test then mutation of that method should be performed with a different strategy.
- Similar to extreme mutation testing, traditional mutation testing can also be used to derive testing verdicts.
- Tooling and reporting of mutation testing can be significantly improved when combining code coverage and strategic application of traditional and extreme mutation testing.

We found that about 9% of all traditional mutants, resided on pseudo-tested methods. Thus, running the extreme approach first, indeed, filters out a lot of traditional mutants that can be ignored. This is advisable to do, because the overhead of running extreme mutation testing first seems negligible if traditional mutation testing does not mutate these methods again.

Similar to other research work [18, 19], we found that there are a lot of redundant and equivalent mutants generated that inflate the mutation score. Redundant mutants are different mutants that point to the same issue. Equivalent mutants are mutants that have the same behaviour as the original program. Due to redundant mutants, the score does not necessarily change linearly when killing a single mutant, because additional mutants are also killed. Furthermore, the score cannot reach the desired value of 1 without detecting and removing equivalent mutants from the calculation, which is not possible without manually analysing them. The focus group additionally revealed that not all mutants are relevant for the developers, and some mutants are not meant to be killed. All these findings strongly challenge the widely established assumption that mutation testing should be seen as an optimisation problem where the mutation score must be maximized towards 1. We found compelling evidence to actually ignore it.

Interestingly, we have not seen any particular type of mutant that is more important than others for developers. Instead, intentions to kill mutants and, thus, improving the test suite, depend on the importance of the implemented method. For example, methods that process sensitive data or expose API functionality for customers were consistently marked as important. Less important methods would only be addressed when mutation testing results would be conveniently available during development.

We also spotted a noteworthy relationship between code coverage and mutation testing. Code coverage information is always required by the developers to guide writing new unit tests. However, mutation testing would only be applied to methods which have a high code coverage. Methods with low coverage are usually known to be not tested very well and generating mutants on these methods does not add new information. This could be a measure for a filter that could ignore these methods for mutation testing.

Yet, none of the well-known mutation testing tools [7, 8, 9] for Java implements such a filter. Furthermore, we observed a lot of mutants that are not worth to be created when code coverage information is available. For example, mutators that change a decision on an if-statement are perceived less meaningful if a code coverage report shows that a particular branch of that if-statement is never executed. Mutators that change that decision would lead to executing a previously not executed code block. Developers can already read from the code coverage report that this branch was not thoroughly tested since not all branches are covered. They would prioritise the growth of code coverage over killing mutants in these cases.

From manual inspection of mutants, we found that mutation testing tools should consider test cases where an exception is thrown by a method differently than the ones where none are raised. Some mutants will always be killed if a method is expected to throw an exception. For example, if any test case exists that raises a necessary exception, emptying the body of a method will always result in killing that extreme mutant because the exception will never be raised. Technically, it is not a pseudo-tested method. However, if the other tests would not detect that the code was removed, test oracles that verify the application logic might be missing, which would make the method somewhat pseudo-tested. Thus, differentiating these cases is reasonable.

Extreme mutation testing can derive pseudo-tested methods because the mutators are strategically applied. For example, because the whole method body is emptied it can be detected that a method is truly pseudo-tested. Additionally, multiple mutators are required to also verify that different return values can be returned without noticing it. However, if a method is not pseudo-tested then it does not mean that the return value is actually meaningful. We found many methods where the return value was never verified but the method body could not be removed. For these methods, a similar testing verdict like "return values are not tested" could be concluded just by considering all traditional mutants that target return statements of a particular method. In fact, we see a similar opportunity to do strategic application of mutators for class fields. For example, a mutator could change a class field to a different value and keep it constant to observe whether it influences test outcome. If all mutants survive then the usage of that class field was never verified which implicates to write a test that relies on that field. Currently, each instruction that uses that class field is just mutated and presented at different spots in a report with code coverage. Bundling that information should save time and enable developers to quicker detect and judge about issues.

We conclude that tooling and reporting of mutation testing should change to establish mutation testing as a widely accepted concept of enhancing test suites. At the current state of writing, mutation testing results are presented by PIT as code coverage reports that highlight whether mutants survived or were killed. However, the concept of having a "mutant" is not an optimal form to describe a weakness in the test suite. Having a report which tells that a method is "pseudo-tested" or that its "return value is never verified" is much more comprehensible. Tooling should enable to focus on meaningful areas of code, avoid unnecessary mutations, and produce testing verdicts that describe issues in the test suite instead of only showing whether mutants survived or not.

3 Performance Regression Testing

Performance regression tests are expensive to establish but are vital to detect performance degradations that are introduced with new software releases. First, a representative workload must be created. Second, execution of the software must be split into measurable time sequences and instrumented. Third, the former and the new software version must be executed and compared to each other to conclude whether a performance regression occurred. Lastly, the root cause of the performance degradation must be determined and remediated.

Any performance issues in Smartest8 may have a strong economical impact as it directly affects chip manufacturing. Thus, Advantest already introduced such a setup to remediate performance regressions before a new release version is shipped. Executing representative workload and comparing it to former releases is automated. To instrument sequences inside the software, an instrumentation framework is in place that is used by the developers by manually adding method- or function-calls. The instrumentation framework uses a data format very similar OpenTelemetry.[3] This data format mainly consists of *spans* and *traces*. Spans represent logical units of work that have a start- and a stop-timestamp and may point

[3] `https://opentelemetry.io`

to a parent span. The reference to the parent span is used to indicate in which context a span is started. A trace is a set of spans which logically belong together.

For this project, we focus on how to improve root cause analysis of a performance regression. All steps before the root cause analysis are automated, but analysing the root cause itself is mostly manual work. Some scripts already exist that calculate statistical metrics of traces which are then used to identify issues. Furthermore, a visualization where spans are ordered on a time axis with their parent span relations also exists. However, the current setup can still be improved by introducing other means to analyse performance issues.

We are currently investigating additional visualizations to aid performance analysis like differential flame graphs [20] or indented pixel tree plots [21] in cooperation with project P2 of the graduate school which works on visual analytics for post-silicon validation [1]. The major challenge we face is to detect the root cause of performance regressions by having to compare millions of spans that very often run in parallel. This poses limits on how interactive analysis might be and we are not yet sure whether other heuristic methods or models are more suitable. Thus, it remains an open research challenge that we want to address in future work.

4 Fuzzing

Developers mainly use manually written tests to verify that a program works as expected. This approach is very reliable in finding software faults and benefits from the expertise of the developer who is familiar with the software under test. However, a developer is limited by the available time to write the test and cannot test every possible combination of inputs to test a program. Therefore, many critical software faults may remain undetected if solely relying on hand-crafted tests. Hence, there exists a need for test automation techniques to close this gap. One technique that has proven to be very successful in that area, by finding many real software faults, is fuzzing.

Fuzzing refers to automated (semi-)random input generation for software testing. The most prominent variation of the technique is mutation-based greybox fuzzing. The approach starts with a small set of diverse inputs for a program under test which is called "seed corpus." Controlled changes, i.e., "mutations," are made to these inputs to create slightly different inputs which are then executed by the target application. The target application then returns feedback in form of code coverage. If the mutated input results in returning new code coverage then it is kept and added to the seed corpus. If the target application crashes instead then a new software fault is found and, thus, saved in a different location. Otherwise, the new input is discarded. This way, new inputs are gradually generated in an evolutionary cycle that explore different code areas by the guidance of code coverage. It is a greybox technique because it is categorized between whitebox and blackbox fuzzing. It does not analyse the code and derives new input from it like in whitebox fuzzing. However, it does use feedback from the target application by instrumenting it. Thus, it is different to blackbox fuzzing where no feedback from the target application is obtained.

One famous implementation of that approach is the "american fuzzy lop" (AFL) which was written by Michał Zalewski.[4] Research work made many improvements to the formerly described fuzzing loop and used for that the AFL implementation as a baseline. This work either targeted to improve the feedback that is returned by the fuzzer to find different classes of software faults [22, 23] or addressed input generation and mutation to improve it for various file formats [24, 25]. One of AFL's strongest limitations is the mutation of highly structured inputs that require grammars. AFL only uses bit-level mutations and a list of keywords to mutate inputs. This makes it hard for the fuzzer to surpass the parser code of the target application and reach the actual application logic. Although other grammar-based coverage-guided fuzzers emerged, they are either not build on top of AFL [26], which makes re-implementing existing improvements a necessity, or they use parsers [27] to parse inputs first and secondly mutate the derivation tree, which is not optimal in terms of performance due to the algorithmic complexity of the parsing algorithm. Thus, generating structured inputs and incorporating it into existing fuzzers remains an open challenge.

For this project, we focus on how to use this powerful test generation technique to find software faults in Smartest8 and take advantage of past improvements to AFL. For that, we split our work in two different

[4] `https://lcamtuf.coredump.cx/afl/`

studies: The first study focused on creating a file format that can be mutated by AFL's bit-level mutations but would still result in providing syntactically valid input to the target application. This study is under peer review at the time of this writing. In the next study, we will focus on fuzzing domain-specific files of Smartest8.

For the first study, we aimed to surpass the parser stage and reach the main application logic of programs by creating syntactically valid inputs. The idea was to keep AFL's evolutionary design, that is proven to work very well to find software faults, and make input generation very fast to compete with other fuzzers later. We decided to only introduce a new file format with a respective decoder and keep the rest of the fuzzer as it is. AFL would apply fast binary-level mutations to the new file format which changes the input. However, the application would only see the structured (plaintext) input because the decoder interprets the new file format first. To accomplish that, the file format must be affected by binary-level mutations in a way to introduce changes after decoding while being able to keep most of the remaining information that a parse tree encodes to preserve the evolutionary cycle of AFL. This is difficult because defined structure in a file format may be mutated as well.

We eventually designed a file format on top of a context-free grammar, which is used to describe the syntax of the input. To describe the context-free grammar we used ANTLR [28]. An ANTLR grammar consists of multiple rules that describe which children a parse tree node may have. Internally, ANTLR uses augmented transition networks (ATNs) [29] to describe these grammar rules. ATNs are recursive state machines where each path from a start- to a stop-state represents a parse tree node. We encoded these paths in a bit-sequence and introduced a chunk-based design to handle random mutations to the file. We have implemented our decoder in Go.

To evaluate the effectiveness of our new approach we conducted an exploratory study by fuzzing SQLite. We have chosen SQLite over Smartest8 for this study for various reasons. First, since SQLite is openly available, this makes reproduction of our experiments easier for other researchers and fosters open science. Second, ANTLR grammars for SQLite already exist and can be reused. Third, the software is already fuzzed by the SQLite developers, which means that strong fuzzing seeds already exist that can be reused for a realistic comparison. Lastly, it is similar to Smartest8 in a way that it is already very well tested and accepts structured inputs.

We ran in total six 48h fuzzing campaigns with 80 CPU cores on the upstream branch of SQLite. Three fuzzing campaigns used AFL with the plaintext encoding (default approach) and three fuzzing campaigns used AFL with our newly implemented decoder and file format (encoded approach). The default approach used the 512 already existing strong fuzzing seeds of the SQLite development team whereas our encoded approach started by a single randomly generated input from the grammar. Our main conclusions of the newly introduced file format and decoder are:

- The encoded approach finds complex bug triggers that require a combination of grammar elements, which the default approach is highly unlikely to produce.
- The encoded approach inherits the evolutionary design of AFL and finds more paths than the default approach.
- The encoded approach generates more valid inputs than the default approach which are also more compact.
- The encoded approach is resilient to small grammar errors.

We found 3 distinct crashes in SQLite. One crash was caused by a falsified assertion inside the code which was thought to be unreachable. I.e., the developers assumed that there does not exist an input to reach that statement, which we have proven wrong. Another crash indicated a possible use-after-free bug that was also found by one of their fuzzers. The last crash was a bug in the application logic with the possibility of a wrong query result. This crash in particular had a very complex setup. A valid table must be created. That table must have an index which involves the use of the COLLATE operator. Then, a subsequent DELETE statement must reference that index in an expression where the index is part of the right operand of an "==" or "IS" operator.

Finding these crashes in SQLite was very astounding because SQLite has 100% branch coverage, millions of existing test cases, many specialised tests, and the developers additionally run AFL and another

structure-aware fuzzer at all times.[5] Their own structure-aware fuzzer is particularly designed to generate SQL queries and alter databases. Yet, our approach has found two new crashes that have not been found by all of these tests and one of it even revealed a logic bug. In addition to that, our decoder is not even aware of semantics but only of syntax and still produced rather long semantically valid sequences. This is only possible, because using the new file format with the decoder inherits the evolutionary design of AFL. Ensuring syntax validity apparently leads to additionally finding more paths, which is a measure used by AFL to distinguish code coverage.

Eventually, we analysed how valid our input files were. By "valid" we mean "semantically correct," i.e., input files that surpass the parsing stage of the code. For that, we executed all inputs that were found by both approaches and analysed their logs for syntax and other errors. We found less, but still some, syntax errors in the seed corpus of the encoded approach. These syntax errors were caused by minor errors in the grammar. However, in total, we found that the encoded approach produced more files that were overall valid and have also observed that the number of executed statements was only a quarter of the default approach. This shows that our newly developed approach is resilient to minor grammar errors while still producing more valid and compact inputs.

Our results are very promising and a good basis for future work. In our next study, we will focus on fuzzing domain-specific files of Smartest8 by using our newly designed file format and decoder. In particular, we plan to target specification files of Smartest8 that are used to specify initial settings for an instrument and, thus, test head cards that are used by the ATE. These properties include, depending on the instrument, e.g., setting voltage levels or defining timings of a signal. Many of these settings are optional or have a well defined range. We want to test whether the compiler of Smartest8 can actually cope with all these different settings and guarantee correct compilation. We also see a chance to apply metamorphic testing [4] to the found set of fuzzing inputs to have a better test oracle at hand than only considering crashes. Besides that, we also plan to increase external validity of our work by comparing it to other grammar-based coverage-guided fuzzers [27, 26] and investigate other feedback metrics than code coverage alone from existing implementations that extend AFL [22, 23].

It is noteworthy to mention that we have an ongoing collaboration with project P5 of the graduate school in that research area [1]. In fact, the project already uses our file format to generate code snippets in order to create system-level tests for semiconductor systems.

5 Conclusions and Next Steps

We investigated how to evaluate the strength of a test suite by applying and comparing traditional and extreme mutation testing. We concluded that both techniques are usable in practice but also found that creating mutants is not the best way to describe a weaknesses in the test suite. If mutation testing is strategically applied, similar testing verdicts as in extreme mutation testing can be created. Doing so would improve acceptance of the technique since these testing verdicts are easier to comprehend than to analyse mutants. To eventually conclude the research area of test suite analysis, we want to improve root cause analysis of performance regressions. For that, we plan to work together with project P2 of the graduate school to find out which visualization can cope with the large amount of (parallel) spans that current performance regression tests create.

We created a new file format with a decoder to generate highly structured inputs for our fuzzing campaigns. Our first exploratory study on SQLite showed that the approach is very effective in findings complex bug triggers that originate from combinations of grammar elements. This is a good foundation when we start to fuzz domain-specific files of Smartest8 and perform test case generation in a more complex software system. We will continue to collaborate with project P5 of the graduate school that already uses our newly designed input generation approach for system-level tests of semiconductor systems.

Lastly, we plan to combine automatically generated fuzzing results and use them for metamorphic testing to see whether we can combine automated and manual test generation that is guided by the expertise of developers. We hope to find synergistic effects between these techniques that will lead us to create cost efficient and fault-revealing test suites for complex software systems.

[5] `https://sqlite.org/testing.html`

References

[1] Hussam Amrouch, Jens Anders, Steffen Becker, Maik Betka, Gerd Bleher, Peter Domanski, Nourhan Elhamawy, Thomas Ertl, Athanasios Gatzastras, Paul R. Genssler, Sebastian Hasler, Martin Heinrich, André van Hoorn, Hanieh Jafarzadeh, Ingmar Kallfass, Florian Klemme, Steffen Koch, Ralf Küsters, Andrés Lalama, Raphael Latty, Yiwen Liao, Natalia Lylina, Zahra Paria Najafi-Haghi, Dirk Pflüger, Ilia Polian, Jochen Rivoir, Matthias Sauer, Denis Schwachhofer, Steffen Templin, Christian Volmer, Stefan Wagner, Daniel Weiskopf, Hans-Joachim Wunderlich, Bin Yang, and Martin Zimmermann. Intelligent methods for test and reliability. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2022*, page 1–6, 03 2022.

[2] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. Mutation testing advances: an analysis and survey. *Advances in Computers*, 112:275–378, 2019.

[3] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. *The Fuzzing Book*. CISPA Helmholtz Center for Information Security, 2021.

[4] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. A survey on metamorphic testing. *IEEE Transactions on Software Engineering*, 42(9):805–824, 2016.

[5] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, page 435–445, New York, NY, USA, 2014. Association for Computing Machinery.

[6] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2010.

[7] Henry Coles, Thomas Laurent, Christopher Henard, Mike Papadakis, and Anthony Ventresque. Pit: A practical mutation testing tool for java (demo). ISSTA 2016, page 449–452, New York, NY, USA, 2016. ACM, Association for Computing Machinery.

[8] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: an automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

[9] David Schuler and Andreas Zeller. Javalanche: Efficient mutation testing for java. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, page 297–298, New York, NY, USA, 2009. ACM, Association for Computing Machinery.

[10] Rainer Niedermayr, Elmar Juergens, and Stefan Wagner. Will my tests tell me if I break this code? 2016 IEEE/ACM International Workshop on Continuous Software Evolution and Delivery (CSED), pages 23–29. IEEE, 2016.

[11] Rainer Niedermayr. *Evaluation and improvement of automated software test suites*. PhD thesis, University of Stuttgart, Fakultät 5 - Informatik, Elektrotechnik und Informationstechnik; Pfaffenwaldring 47, 70569 Stuttgart, 2019.

[12] Oscar Luis Vera-Pérez, Benjamin Danglot, Martin Monperrus, and Benoit Baudry. A comprehensive study of pseudo-tested methods. *Empirical Software Engineering*, 24(3):1195–1225, 2019.

[13] Maik Betka and Stefan Wagner. Extreme mutation testing in practice: An industrial case study. In *2021 IEEE/ACM International Conference on Automation of Software Test (AST)*, pages 113–116, 2021.

[14] Oscar Luis Vera-Pérez, Martin Monperrus, and Benoit Baudry. Descartes: a pitest engine to detect pseudo-tested methods: tool demonstration. 2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 908–911. IEEE, 2018.

[15] Goran Petrovic, Marko Ivankovic, Bob Kurtz, Paul Ammann, and René Just. An industrial application of mutation testing: Lessons, challenges, and research directions. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW), pages 47–53. IEEE, 2018.

[16] Goran Petrović and Marko Ivanković. State of mutation testing at google. In *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '18, page 163–171, New York, NY, USA, 2018. ACM, Association for Computing Machinery.

[17] Maik Betka and Stefan Wagner. Towards practical application of mutation testing in industry – traditional versus extreme mutation testing. *Journal of Software: Evolution and Process*, 2022.

[18] Mike Papadakis, Christopher Henard, Mark Harman, Yue Jia, and Yves Le Traon. Threats to the validity of mutation-based test assessment. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 354–365, New York, NY, USA, 2016. Association for Computing Machinery.

[19] Bob Kurtz, Paul Ammann, Jeff Offutt, and Mariet Kurtz. Are we there yet? how redundant and equivalent mutants affect determination of test completeness. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 142–151, 2016.

[20] Cor-Paul Bezemer, Johan Pouwelse, and Brendan Gregg. Understanding software performance regressions using differential flame graphs. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 535–539, 2015.

[21] Michael Burch, Michael Raschke, and Daniel Weiskopf. Indented pixel tree plots. In *International Symposium on Visual Computing*, pages 338–349. Springer, 2010.

[22] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. Perffuzz: Automatically generating pathological inputs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2018, page 254–265, New York, NY, USA, 2018. Association for Computing Machinery.

[23] Rohan Padhye, Caroline Lemieux, Koushik Sen, Laurent Simon, and Hayawardh Vijayakumar. Fuzzfactory: Domain-specific fuzzing with waypoints. 3(OOPSLA), oct 2019.

[24] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru Răzvan Căciulescu, and Abhik Roychoudhury. Smart greybox fuzzing. *IEEE Transactions on Software Engineering*, 47(9):1980–1997, 2021.

[25] Caroline Lemieux and Koushik Sen. *FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage*, page 475–485. Association for Computing Machinery, New York, NY, USA, 2018.

[26] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. *Proceedings 2019 Network and Distributed System Security Symposium*, 2019.

[27] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. Superion: Grammar-aware greybox fuzzing. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 724–735, 2019.

[28] Terence Parr. *The definitive ANTLR 4 reference*. Pragmatic Bookshelf, 2013.

[29] Terence Parr, Sam Harwell, and Kathleen Fisher. Adaptive ll(*) parsing: The power of dynamic analysis. *SIGPLAN Not.*, 49(10):579–598, oct 2014.