SOFTWARE EQUIVALENCE CHECKING BASED ON UNIT TESTING AND SYMBOLIC EXECUTION

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF SCHOOL OF INFORMATICS
OF
MIDDLE EAST TECHNICAL UNIVERSITY

BY
BURAK ÜNALTAY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
INFORMATION SYSTEMS

NOVEMBER 2018

Approval of the thesis:

SOFTWARE EQUIVALENCE CHECKING BASED ON UNIT TESTING AND SYMBOLIC EXECUTION

submitted by **BURAK ÜNALTAY** in partial fulfillment of the requirements for the degree of **Master of Science in Information Systems Department, Middle East Technical University** by,

Prof. Dr. Deniz Zeyrek Bozşahin
Dean, Graduate School of **Informatics** __________________

Prof. Dr. Yasemin Yardımcı Çetin
Head of Department, **Information Systems** __________________

Assoc. Prof. Dr. Altan Koçyiğit
Supervisor, **Information Systems Dept., METU** __________________

Examining Committee Members:

Assoc. Prof. Dr. Aysu Betin Can
Information Systems Dept., METU __________________

Assoc. Prof. Dr. Altan Koçyiğit
Information Systems Dept., METU __________________

Assoc. Prof. Dr. Pekin Erhan Eren
Information Systems Dept., METU __________________

Assoc. Prof. Dr. Tuğba Taşkaya Temizel
Information Systems Dept., METU __________________

Assist. Prof. Dr. Tülin Erçelebi Ayyıldız __________________
Computer Engineering Dept., Başkent University

Date: **14.11.2018**

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

Name, Last name: Burak Ünaltay

Signature :

ABSTRACT

SOFTWARE EQUIVALENCE CHECKING BASED ON UNIT TESTING AND SYMBOLIC EXECUTION

Ünaltay, Burak
M.S., Department of Information Systems
Supervisor: Assoc. Prof. Altan Koçyiğit

November 2018, 45 pages

Hardware is one of the best representatives of our ever-changing world. For most of the end users this is nothing but a nuisance as they have to renew their electronics each year. For the embedded system designers who has to deal with this change in frontier however it is a definite threat. Embedded system designers have a close relationship to the hardware as the software runs on it highly tuned for the platform it runs on. It is when hardware is completely obsolete this impact reaches its peak because customized software will need heavy refactoring and often times a complete rewrite. One has to be sure to have a functionally equivalent product after this refactoring effort. This requires a costly and lengthy validation process. Sum of all validation efforts for this purpose could be identified as equivalence checking. Here in this study, we lay out a method that automatically deals with problem of equivalence checking. Our method is tested against two evaluation scenarios. In first scenario, our method is tested against small function bodies. In second scenario, it is tested against a larger code example that is closer to a real production code. In both ways of evaluation, our method is able deduce equivalency with a score of 5 out of 6.

KEYWORDS: Software Equivalence, Software Equivalence Checking, Automatically Generated Unit Tests, Symbolic Execution.

ÖZ

SEMBOLİK YÜRÜTMEDEN ELDE EDİLEN BİRİM TESTLERİN YAZILIM DENKLİĞİNİ KANITLAMADA KULLANILMASI

Ünaltay, Burak
Yüksek Lisans, Bilişim Sistemleri
Tez Yöneticisi: Assoc. Prof. Altan Koçyiğit

Kasım 2018, 45 sayfa

Günümüzde her şey nasıl değişime uğruyorsa, donanım komponentleri de bu değişimin parçası olmak zorunda. Bu durum son kullanıcı için küçük rahatsızlıklar yaratmakta. Fakat gömülü sistem tasarımcıları için bu durum çok daha kritik, çünkü gömülü yazılımlar üzerinde çalışacakları donanıma yüksek bir oranda bağlı olduğundan herhangi bir donanım değişikliği yazılım geliştirme süreçlerine büyük külfetler getirmekte. En büyük zorluk ise donanımın artık üretilmemesi durumunda ortaya çıkacaktır. Böyle bir durumda eski donanım için tasarlanmış yazılım ağır bir yenileme sürecine girecek ve hatta tamamen baştan yazılmak zorunda kalacaktır. Bu yenileme sürecinden sonra eski yazılım ve donanım ile yeni yazılım ve donanım ikilisi ile fonksiyonel anlamda denk olmalıdır. Bu denkliği test etmek oldukça uzun ve etraflı bir süreçtir ve denklik testi olarak adlandırılır. Bu çalışmada denklik testini otomatik olarak gerçekleştirebilen bir method ortaya koyuyoruz. Methodumuz iki değerlendirme yöntemiyle test edilmiştir. İlk yöntemde küçük boyutlu fonksiyonlarla ikinci yöntemde ise gerçek hayat senaryolarına daha yakın bir kod örneği kullanılmıştır. İki yöntemde de methodumuz 6 üzerinden 5 başarı oranı sağlamıştır.Bu tezden elden edilen veriler, mevcut kanadın statik test sonuçlarına katkı sağlayacaktır.

ANAHTAR KELİMELER: Yazılım Denkliği, Yazılım Denkliği Testi, Otomatik Oluşturulmuş Birim Testler, Sembolik Yürütme.

This dissertation is dedicated to my beloved family. Without their support, I could not have done it.

ACKNOWLEDGEMENTS

I would like to thank my advisor Assoc. Prof. Dr. Altan Koçyiğit. Thanks to his support and guidance, I was able to see through this study. I would also like to thank my girlfriend Meltem Özyıldız for her support during the whole process.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

NIST	National Institute of Standards and Technology
CBMC	C Bounded Model Checker
CT	Combination Testing
AES	Advanced Encryption Standard
VLSI	Very Large Scale Integration
DSP	Digital Signal Processor
MBT	Model Based Testing
SUT	System Under Test
SDL	Specification and Description Language
DART	Directed Automated Random Testing
KLEE	KLEE LLVM Execution Engine
CIVL	Concurrency Intermediate Verification Language
LLVM	Low Level Virtual Machine
CPU	Central Processing Unit
GHz	Gigahertz
GB	Gigabytes
RAM	Random Access Memory

CHAPTER 1

INTRODUCTION

In today’s fast-moving world, everything is subject to change and this definitely includes hardware components. This situation causes a minor inconvenience for the end users since they constantly have to change their electronics to keep up with the technological advancements. However, it poses a much more serious challenge for embedded system designers as hundreds of components go end of life each year. As embedded software is often highly tuned for the hardware it runs on, changing hardware components has a huge impact on the software development process. The real challenge is when the hardware is completely obsolete and has to be replaced by a newly designed one because the software will need heavy refactoring and sometimes even a complete rewrite to work with the new hardware. After this refactoring process, one has to be sure that the new hardware and software stack has the identical behavior with the previous one. This can be a quite tedious and lengthy validation process. Summation of this validation efforts could be identified as equivalence checking.

1.1. Why is Equivalence Checking Important ?

Equivalence checking in low level software is quite important for several reasons. One of the reasons is that majority of the embedded software is expected to run for several years at a time without any intervention. This stems from the fact that cheap low level electronics are so common that even simplest of devices have several of them. The devices we mention could be anything and everything around us such as:

- A subsystem in a mobile phone
- A thermostat in a living room
- A sensor measuring vitality signals of a patient
- Controller of an irrigation system
- A meteorological measurement system situated at the top of a mountain

These examples could be expanded with ease, as we are surrounded billions of such devices. According to a research, there will be 30 billion connected devices by the year 2020 [1]. The main connecting point of our examples is updating, patching, fixing them in terms of software means is highly unlikely and sometimes downright impossible especially the embedded software driving them. Therefore, it is of absolute importance that these run, for lack of a better term, like a clock since the world as we know revolves around them. This brings us to our main point, which is how do we make sure that our software acts exactly like it is designed even when hardware is constantly changing. It is an undeniable fact that hardware grows old and as technological advancements are pushing limits old technology is destined to disappear. In this constant stream of change, it is not surprising to come across to a point where our trusted microchip is not produced anymore. Hence the new iteration of whatever product we might be working on, has to be based on some other architecture, processor, instruction set, etc. As software engineers working on such embedded devices, it is imperative that they must treat hardware and software as a whole system. Hence, the work of porting the software to a new hardware platform should not cause any change of features. Occurrence of unexpected changes and hiccups will interfere with the whole system. If the aim and scope of the product you are working on aligns with the principles mentioned above, then the equivalence of the software running both on the new and old platforms is of utmost importance.

This thesis explores this issue and lays out an automated equivalence checking framework. Our scope will be limited to embedded systems only, therefore ‘C’ will be our choice of programming language for the software components under equivalence investigation. The ‘equivalence checking’ process will be conducted using unit tests. Our aim is to automatically generate unit test cases for both original and modified components and later on use them to reason about how the modified software component compared to original one.

In the next chapter, there will be a literature review covering different methods that could be used in such a framework. Main methods that will be investigated are already existing equivalence checking methods and testing methods for software components. The third chapter will explain our proposed method from an architectural point of view and presents a basic implementation that can be used to check equivalence of software components. The fourth chapter presents evaluation of the proposed framework by means of two cases. Final chapter is where we conclude our achievements and discuss about future work that could be done to improve and build on top of our current efforts.

CHAPTER 2

LITERATURE REVIEW

This chapter contains an investigation of methods that could be used to build a software equivalence checking platform. The “software” we mention here assumed to be developed in C programming language as embedded systems are heavily coded in C. We propose two different implementations of a software component with a common interface can be checked for equivalence by having a rigid and through testing strategy. Well thought out test scenarios could expose faulty refactoring of the software component early on, hence avoiding costly and untimely bug fixes. However, it is a well-known fact that testing is costly. But it should also be noted that software bugs are even costlier. According to a report made by Tricentis, a software test company, a total of 548 software errors affected 4.4 billion people and cost 1.1 trillion in assets [1]. To give a little perspective, another report compiled for NIST (National Institute of Standards and Technology) shows that the cost of inadequate infrastructure for software testing is estimated to range from $22.2 to $59.5 billion [3]. Although a direct comparison is not exactly meaningful, there is no denying that the cost of software errors grows rapidly. Against this rapid growth, it is only natural that an abundance of research on software testing is present. Within all this research, our aim is to find the ones that automate the process of testing, so that we could test both original and refactored components for equivalence without excessive costs.

It is important to remind the reader that our focus is the embedded domain. This results in several difficulties. Software in particular is not guaranteed to have proper documentation, a model nor requirements at all. In the embedded domain, this holds true as well, maybe even more than the other domains of software. As we focus on proving functional equivalence between two software components implemented in C programming language with embedded domain restrictions, the method to prove functional equivalence between software components should not be picky. In other words, the equivalence testing method should be able to take in the source code itself with implementation and interface parts and it should suffice for equivalence analysis. This restriction is not in place to undermine any possible equivalence checking method but simply a requirement brought out by necessity. This necessity stems from affairs in embedded software industry, as it is perfectly natural to have missing or no documents at all, nor requirements which is clarifying what this particular software product is responsible with. Often times when faced with a refactoring process of a legacy software component, the software developer have only source code itself as the guide for whole process. Therefore, expecting a complete documentation, requirements for software or a model encompassing the entire software capabilities which can be used to generate code, tests and other utilities is out of the question. Now these expectations could be perfectly valid for other domains, but for our case we assume that the code does have such amenities and for good reason too.

2.1. Software Equivalency Checking Methods

In this subheading, there will be an investigation of how current state of the art deals with checking software equivalence. A change of target hardware platform is not the only case where you might need to refactor your components so that they will stay functionally equivalent to the previous ones. A change in the toolchain that is being used could also require you to refactor your software to make it stay functionally equivalent. In another situation, you might want to check whether the version e.g., 1.5 is functionally equivalent to version 1.7 or you might be interested to upgrade the version of a library that is being used in your code base and you want to be sure that it will not break any important bits and pieces. Aforementioned situations along with many of them will put you in a situation where you will have to confirm software equivalence to ensure a safe transition period.

Jiang et al. [4] introduce an algorithm to automatically mine functionally equivalent code fragments of arbitrary size. They treat the concept of function equivalence based on the set of inputs and their respective set of outputs. They set the core of the algorithm as automated random testing and by getting inspiration from Schwartz's randomized polynomial identity testing [5] . The algorithm first extracts the candidate code fragments from the source code, later on, random inputs are generated to separate code fragments depending on their output values for respective inputs. They investigated the algorithm on the source code of Linux Kernel 2.6.24. They claim that they have found many code fragments that are although syntactically different, they are functionally equivalent. The study mentions that this method can scale million-line programs and is able to analyze the Linux kernel within several days with parallel processing. Although this method is mainly designed to detect duplicate code segments, it is possible to extend it to analyze equivalence of the components as well.

Post et al. [6] analyzed the functional equivalence between two AES(Advanced Encryption Standard) implementations. This checking process is done using automatic bounded model checking [7], which is a successful technique in hardware domain used for equivalence checking. Cryptographic algorithms have long been using bit-level operations. This situation in particular makes them suitable for bounded model checking. It is mentioned in the study that equivalence proved for the first three rounds of AES encryption routines semi-automatically. In addition to this, they were able to achieve the full proof of equivalence by manual intervention.

Godlin et al. [8] focus on verification of programs by regression. It is mentioned that real programs cannot be specified with high level invariants or temporal properties. To add to this fact, it is even harder to describe what a specific part of the code should do.

Therefore to avoid these challenges, the industry takes on a different approach with regression testing. Regression testing has no need for a formal specification. A deep understanding of the code is not required either. Having a practical approach, Godlin et al. include regression testing in their toolset while proving functional equivalence between two programs. Though this technique has long been used by the industry, it comes with some shortcomings. The tooling used in the study and the choice of the programming language have been decided by lack of maturity of the tools in other programming languages. Achieving scalability is another concern of theirs.

Matsumoto et al. [9] present an equivalence checking method for two C programs. Their main method of finding out the equivalence is symbolic simulation. However, in their claim, verifying equivalency of all the variables takes a considerable amount of time. To tackle this problem, they make use of textual differences between descriptions. Using these differences, they aim to minimize the number of checks hence shortening the required time. Through several experiments, their method provides shorter execution times for the symbolic simulation. It is important to note that the "C programs" that are being investigated are not hand written code but generated as an output of VLSI(Very Large Scale Integration) design process.

Feng et al. [2] introduce an idea for formal verification of equivalence of structurally similar software. The idea revolves around the concept of cutpoints which is a method for formal equivalence verification of combinational circuits. They took this initial idea and implemented it in the software domain. They claim that they have better execution time performance compared to that of the previous approaches, although false inequivalences are still a problem. Their future work suggestions include improvements on false inequivalences and scalability.

Ciobaca et al. [3] introduce a language-independent proof system for full equivalence. Given two programs as input, a proof tree deducts whether they are equivalent or not. They showcase their method on two programs which calculates Collatz sequence. The programs are implemented in different languages. It is also noted that this example is particularly interesting because it is now known beforehand whether the sequence will terminate or not. In the study, it is shown that programs are equivalent although neither termination nor divergence could not be established.

Wood et al. [4] argue that when programmers change their code, they intend to keep some parts of the program's behavior. Following this logic, they propose a formal criterion to characterize the preserved part of the program behavior. The criterion is "two program versions are equivalent up to a set of affected objects A, if executions of these versions correspond at each execution step when the objects in A is not considered". Then they move on to propose a sufficient condition for this criterion. It is sufficient to establish that traces of calls to the methods and returns between A objects and the rest of the objects are equivalent. They claim that examination of stack and heap at each execution step is not necessary. The proof they provide to this condition is verified using Dafny program verifier [5].

Papadakis et al. [6] propose Trivial Compiler Equivalence, which is a method that makes use of the already available compiler facilities to address identification of mutants in programs. It is argued that their method is applicable to real-world programs and can also aid already existing tools to detect equivalent and duplicated mutants. Their method is able to discard more than \%7 and \%21 of all the equivalent and duplicated mutants.

Now that we have enough background information on how equivalence checking is dealt with it is time to gauge each of these methods for our case. Jiang et al. [7] propose a method to identify functionally equivalent code in their study. Their algorithm finds the possible equivalent parts of code in the entire code base and then moves on to the part where it checks whether they are actually equivalent or not by using random testing approaches. The part of the algorithm where it extracts parts of possible functionally equivalent code is of no value for our problem as it is already known which parts of code are functionally equivalent in our case. The part where they confirm the equivalence using random testing can prove useful for our case although whether random testing can provide a meaningful test suite to catch errors that might have gone unnoticed otherwise, is still questionable. Post et al. [8] propose a solution using bounded model checking. In their solution, however they had to use several manual steps. These manual steps include modifications to the source code so that the code would conform the tool they are using to perform bounded model checking (CBMC) [9] This is undesirable for us as we want to automate this process in its entirety if possible. Godlin et al. [10] also use CBMC as the underlying engine in their verification approach. They are also involving user in certain parts of the algorithm. Matsumoto et al. [11] are making use of symbolic execution at the core of their algorithm, however they are lightening the load on symbolic execution by making use of textual differences on the code. This algorithm designed to work on generated C code, hence may not be as performant as it is shown in the study when code is hand coded and not generated. Code generators often work with similar patterns because their main responsibility is mapping certain models to certain sentences of programming language. As such, finding textual differences in generated code is not the same task as finding textual differences on hand coded programs because hand coded programs will most definitely be different with naming, indenting, choice of words, placement of statements, argument type and so on. Although studies of Post et al. [8] Godlin et al. [10] and Matsumoto et al. [11] do not exactly align with our goals and assumptions for this research, yet they are providing the idea of using a model or path checker as the base of our approach. As the model and path checkers are mathematically proven, even though we are not aiming for formal equivalence checking, they will provide us fidelity, nevertheless. Feng et al. [2] adopt notion of cutpoints from equivalence checking of electrical circuitry. Their idea is cutting up the code in bite size pieces so that symbolic execution times will be manageable. These cutpoints should be points where in between the state should not change. In the study, the programming language is an assembly dialect for a certain DSP(Digital Signal Processor) series. This is a fairly simple language compared to C programming language. Finding such cutpoints means finding candidates for certain code structures that can act as barriers around a certain state. In assembly language, these structures are often reads and writes, memory barriers, instruction barriers, branches, system calls, interrupts and so on. In a higher order programming language such as C, these structures could practically be anywhere hidden behind library calls, system calls or any function for that matter. Therefore, finding such cutpoints in higher order software is much more complicated as it requires a complicated parsing process, maybe even a symbolic engine on its own. Therefore, we refrain from this effort, because higher order languages deal with statements that can change the state every other line, completely defeating the purpose of the study. Ciobaca et al. [3] provide a mathematically sound method for equivalence checking which is also language agnostic. However it is unable to handle symbolic statements, rendering it useless for our case. Wood et al. [4] lay out an interesting approach. It is however focused on two different versions of the same code one is predecessor of the other one. In our case, it is imperative for us to deal with a complete rewrite as long as the interface stays the same, however. Finally, the method of Papadakis et al. [6] is not suitable for our purposes either. Because it is designed to find equivalent code pieces in large code bases, and can give false alarms.

2.2. Testing Methods

A literature review reveals 5 main testing related research areas which are namely:

- Symbolic Execution Based Testing
- Model Based Testing
- Combinatorial Testing
- Random Testing
- Search Based Testing

2.2.1. Symbolic Execution Based Methods

Symbolic execution is executing a program for a set of classes of inputs, rather than a set of sample inputs. In that way, each symbolic execution increment might be equivalent to a large number of normal test cases. Obtained results can be checked against a ground truth for correctness [12].

Symbolic execution engine maintains a state where it keeps information about the next statement which will be evaluated, a symbolic store where symbolic or concrete values are kept and path constraints which is the list of assumptions on symbols to reach that particular state [13]. By exploring the program in this manner while doing the bookkeeping of program state, one can identify all unsafe inputs to a program that will possibly cause an unwanted situation. However actually achieving this on a real-world program comes with challenges. Baldoni et al. [13] present these challenges under five headings as such:

- Memory: The way symbolic execution engine handles memory is important. It should be able to handle both simple and complex data structures. Also, special data types such as pointers should also be handled both symbolically and concretely.
- Environment and third-party components: How does the engine handle interactions between different software products? Calls to third party libraries or system calls can change the system state which should be dealt with by the symbolic execution engine.
- State space explosion: Loops and other programming structures can increase the number of possible execution states. Symbolic execution engine should be able to handle this situation in a reasonable amount of time. Santelices et al. state that traditional approach to symbolic execution where each execution path analyzed one by one does not scale since a typical program has many paths and the number of paths grows with the size of the program. This problem is also called path explosion [12].
- Constraint Solving: When symbolic execution was first introduced, constraint solvers were a serious limitation. However significant advances in constraint solving has now made symbolic execution viable compared to 70s when constraint solvers pose a serious liability since symbolic execution cannot generate an input if the symbolic path constraint along a feasible execution path contains formulas that cannot be (efficiently) solved by a constraint solver. An example to this might be nonlinear constraints [14].
- Binary code: Whether the symbolic engine can analyze the program without needing the source code is also an important question where the source code is not available.

The challenges listed above did not hinder the advancements in this area of research. Modern symbolic execution engines also make use of concrete execution together with symbolic execution. One such example is called “Concolic Testing”.

Concolic Testing performs symbolic execution dynamically, while the program is executed on some concrete input values. Concolic testing maintains separate states for both concrete and symbolic values. Concrete state represents the mapping between variables to their concrete values. Symbolic state however only keeps the mapping for variables that only have non-concrete values. Concolic execution also needs initial concrete values for inputs. It executes a program with some random input, extracts symbolic constraints on branch points and then uses a constraint solver to choose next execution path. This is repeated systematically or heuristically until all possible execution paths are explored, or until a user defined criteria is met [15].

Another approach to modified symbolic execution engines is Execution-Generated Testing [15]. The EGT approach works by separating concrete and symbolic state of a program. Later on, for each execution of and operation it checks whether all the values are concrete. If this is indeed the case, the operation is executed as it is. On the other hand if there is at least one symbolic variable then the operation is executed symbolically, by updating the path condition of the current path.

2.2.2. Model Based Testing

Modeling is about capturing the know-how about a system and reusing it as the system grows. This practice is quite beneficial for the design team, however for the testing team the software model is especially valuable because the model contains information regarding what the system should be doing. By having such a model, test engineers now have a way to define how the system reacts to specified inputs. Test scenarios can now be described as a sequence of actions to the system. Since all the actions revolve around the model, it provides great reuse as the model will grow with the system. However, it brings in a caveat that the model should be the heart of the development and should be continuously maintained so that the test scenarios stay relevant with the changing system. [16]

Anand et al. [17] define Model-based testing (MBT) as a light-weight formal method which uses models of software systems for the derivation of test suites. Contrary to traditional formal methods, which aim at verifying programs against formal models, MBT aims at gathering insights in the correctness of a program using often incomplete test approaches. In model-based testing, the system under test (SUT) is treated as a black box system which simply produces outputs for a range of inputs. The internal state of SUT is changing with every increment of the execution. Since model at hand describes the input/output relationship, a test selection algorithm can derive test cases by choosing a finite subset of the input/output relationship represented by the model. Depending on the tooling, test suites might be generated in the desired language.

There are three main ways to approach to the problem. They are namely:

- Axiomatic approaches: These methods are based on logic calculus.
- Finite State Machine approaches: In the FSM approach, the model is formalized by a Mealy machine, where inputs and outputs are paired on each transition. Test selection derives sequences from that machine using some coverage criteria. [17]
- Labeled Transition System approaches: Labeled transition systems (LTS) are a common formalism for describing the operational semantics of process algebra. They have also been used for the foundations of MBT.

According to Orso et al., MBT has several advantages over the other test generation techniques. Coverage based techniques specifically are not reliable because all of the code is treated equally, and an ideal way does not exist to generate test cases covering more than unit tests. MBT on the other hand can result in better test generation because of the domain knowledge, expertise and abstraction the model provides. [18]

As there are a number of ways to model a system, model-based testing approaches also vary heavily. Kerbrat et al. proposes a requirements-based approach where a system model is created using requirements that are being expressed in SDL(Specification and Description Language). Then the system model is used to create test cases related to the requirements [19]. Nebut et al. have a similar approach where they formalize requirements based on use cases extended with contracts. Later on, a transition system is automatically built which is used synthesize test cases [20].

The main disadvantage of MBT techniques is that not every software has a formal model. Even if the stakeholders are willing to put an effort to have such a model there are limitations that come with it as well. The generated test cases are either limited by the modeling language or the tooling that comes with it. In addition to the technical challenges, there are also organizational challenges. The adoption rate of the modeling language and the tools by both technical and managerial staff is quite important [21]. All in all, MBT requires a high level of commitment from all stakeholders participating in software development because it will only be beneficial where the model is center of all phases of software life cycle.

2.2.3. Combinatorial Testing

Suppose that Software Under Test (SUT) is a game software that will run over the network. Inner workings of such a game depend on numerous parameters.

These parameters could be the operating system the software is running on, or the number of players, type of graphics processing unit and many more. Also note that these parameters can have values in different variety. Interactions between these parameters could cause unexpected erroneous states. For example, think of a certain piece of hardware not being compatible with a certain operating system. Parameters and their possible values make up a large combination space. Trying out every combination in such a space is often out of the questions because of limitations such as time and budget. Provided that you are not bound by such limitations, testing all of the combinations will still be considered a waste because many of the combinations in said combination space does not cause any problems at all. Combinatorial testing provides practical ways to detect failures caused by specific combinations of parameters while still adhering to real world limitations. [22]

Originally, combination testing(CT) used for test case generation where parameters and their values are system inputs and each row of the covering array can be considered as a test case. It was also applied for test protocol conformance. Recent applications of combinatorial testing samples configurations to be tested. Software product lines are a popular area of research under CT. Since software product lines have a well-defined parameter set, a CT model can be extracted from these parameters. There is also a modified version of CT called sequence-based CT where each and every parameter becomes a location within a sequence and values of those parameters are repeated at every location. This approach has been used to test GUIs. [17]

2.2.4. Random Testing

Hamlet states that the technical meaning of random testing refers to an unsystematic choice of test data, such that there is no correlation among test cases [23]. This contrast between “random” and “systematic” stems from physical measurements being unpredictable. Hamlet mentions two major points:

- Selection of random points is algorithmically easy, and this can be used to generate a multitude of test cases
- Statistical independence among test points allows statistical prediction of significance in the observed results.

Because of a lack of systematic approach, random testing is considered one of the weakest method of testing. However, combined with different approaches, it can be quite formidable. Yue et al. introduce an enhanced form of random testing called Adaptive Random Testing [24]. This technique seeks to distribute test cases more evenly within the input space. It stands on the idea that for nonpoint types of failure patterns, an even spread of test cases is more likely to detect failures using fewer test cases than ordinary random testing. They claim that adaptive random testing does outperform ordinary random testing significantly (by up to as much as 50\%) for the set of programs under study.

Another different approach is called DART [25] which is short of Directed Automated Random Testing, it is the first tool to use concolic testing. In this approach, a combination of techniques are being used namely:

- automated extraction of the interface of a program with its external environment using static source-code parsing
- automatic generation of a test driver for this interface that performs random testing to simulate the most general environment the program can operate in
- dynamic analysis of how the program behaves under random testing and automatic generation of new test inputs to direct systematically the execution along alternative program paths.

Selling point of DART is testing can be performed completely automatically on any program that compiles.

2.2.5. Search Based Testing

Search based software testing is making use of meta-heuristic optimizing search technique, such as Genetic Algorithm to automate or partially automate a test process. One example of this could be generation of test data. The most important part of the optimization process is the fitness function. Fitness function is responsible for guiding the search to good solutions within an almost infinite search space, while still being committed to a practical time limit. [26]

Search based testing aims to treat software engineering problems as search problems. These search problems however should not be confused by textual or hyper textual searching. Search problems in search-based testing domain is a problem in which optimal or near optimal solutions are searched in space of possible solutions. [27]

McMinn [26] mentions that for a testing problem to be applicable for a search-based optimization it needs to have two attributes :

- Representation: Candidate solutions should be encodable so that they can be processed by a search algorithm.
- Fitness Function: Each fitness function is problem specific and needs to be defined for every problem.

As flexible as they are, search-based testing approaches are still vulnerable in certain areas. For the best solution, optimization approaches use a fitness function that encodes domain knowledge, however this process may pose a challenge for specific domains. For example, to test a password cracking problem in Unix/Linux one needs knowledge about Digital Encryption Standard. Furthermore, since there is no guidance in the system they will fall into random search because there is no intermediate state between decrypted password and failure of decryption. Search space of such a problem is flat with one success point which is the case where candidate string actually matching the user password [22].

Search-based software testing is a relatively new area of research that is also quite promising as it is applicable to many problems. To get the most of it however, domain knowledge should be adhered to create a fitness function that will make the solution accessible in a practical amount of time and space.

CHAPTER 3

PROPOSED EQUIVALENCE TESTING METHOD

In this chapter we propose an approach to the problem of equivalence checking. We will explain flow of our algorithm by separating it into three stages. Later on our rationale about the selection of tools is followed together with a simple example of its usage. Finally, a prototype implementation will be explained step by step.

3.1. Equivalence Testing and Our Approach

Equivalence testing for software domain can be defined as the summation of all efforts to validate that a refactored software component works functionally equivalent to the original component. Our approach of equivalence testing is based on unit tests. This approach relies on testing both components and comparing results against each other for equivalence. In this approach it should be noted that implementation details are not of importance and a black box approach is taken, therefore it is assumed that the external interface of the software component will not change. Generation of the unit tests will be done using a symbolic engine, the reason of this choice is explained in the following section. In figure 3.1, a visual representation of the proposed method is given.

In our approach, we start with extracting each function to be tested. This stage is called parser. Our choice of target programming language is C, therefore our parser is responsible for extraction of functions from software component to pass it onto next stage adhering to the C syntax rules. In order to construct a testing program (such as the one given in listing \ref{lst:chap3_lst2}) parser needs to provide function name, parameter types for the function and return type.

The next stage is responsible for taking in functions to be tested and then generating tests for each function. The tests need to be generated for original and modified components. In our approach, original component taken as a baseline and modified component is expected to work exactly as the original one. Therefore, it is important for us to not only extract test cases from the original component but from the modified one as well since any new test cases modified component could generate should produce the same output as the original component. After being fed with each function, symbolic engine state is also responsible for formatting each of them accordingly and setting up required test benches for them as well. This stage should also contain the necessary means to communicate with the test generator itself. in our case this is KLEE (KLEE LLVM Execution Engine) symbolic engine [15].

The final stage is called test executor and responsible for comparing the components and evaluating their equivalence. Next section explains how the selection of symbolic engine has been made and what are the important factors on this selection as well as the implementation of our approach.

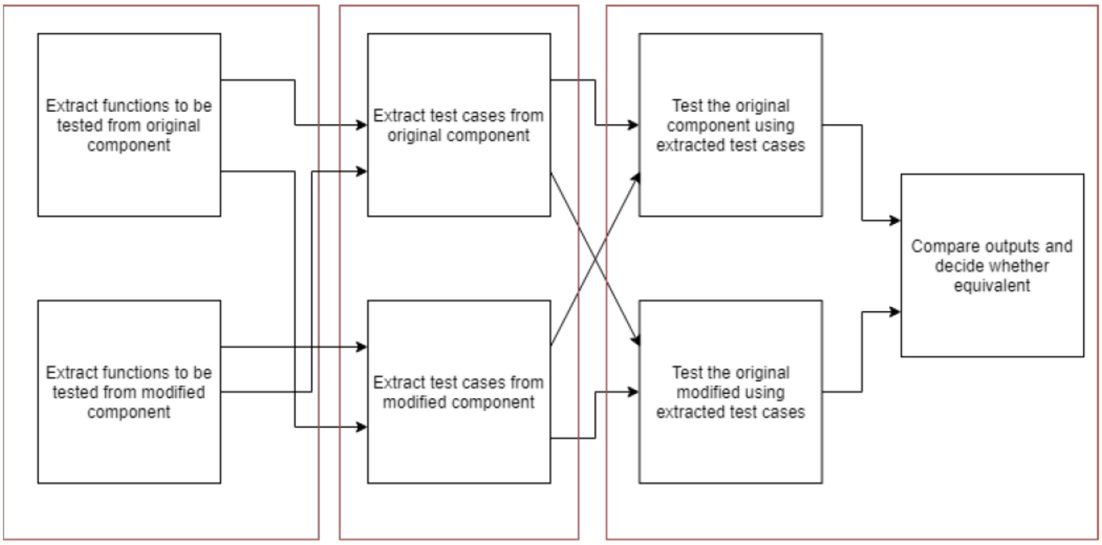

Extract functions to be tested from original component
Extract functions to be tested from modified component
Extract test cases from original component
Extract test cases from modified component
Test the original component using extracted test cases
Test the original modified using extracted test cases
Compare outputs and decide whether equivalent

Figure 3.1 : Our approach for software equivalence testing

3.2. Testing Approach Employed

Having investigated all five of these testing methods explained in Chapter 2, and focusing on the possible methods that will enable us to automatically generate test cases, one can see that symbolic execution outshines them all in our case. Instead of expanding on its advantages, it is beneficial to understand why other methods would not be as useful as symbolic execution. Therefore, in this subheading, shortcomings of these testing methods will be discussed.

- Model Based Testing : Model based testing fails to be our testing method of choice for this study for several reasons. First and most important reason is that the fact that not every software has a designated model. This is even more true when topic of consideration is embedded software. It is particularly important to note that any effort to make use of model based testing will require an already existing and verified model which is the heart of model based software development. Model based software development requires the model to be the center of all activities so that it can be assured the model is up to date and represents the entire business logic. This way automatic code generation, test generation, formal verification is possible. However this investment is not possible for every software development endeavor and sometimes it is simply impossible. The fact that model based techniques being tied to a modeling language or tool makes it even harder.

- Combinatorial Testing : Combinatorial based techniques does not have required tool support or means to generate unit tests. This reason alone is enough to make it undesirable for our case.
- Random Testing : In its solitude, random testing does not offer critical test cases that will give us the confidence while we compare two different implementations. Advanced versions of random testing make use of combined techniques to have more accurate scenarios. These techniques however heavily used by symbolic execution engines as well consequently making random testing less desirable against symbolic execution based methods.
- Search Based Testing : This method is particularly useful if there exists an abundance of test cases. Making use of search based testing, one can reduce the number already existing test cases while having the same coverage and confidence.

Contrary to the methods mentioned above, symbolic execution provides a way to generate test cases directly from source code itself with tools such as KLEE [15] . These test cases are more desirable because symbolic execution engines analyze branching points of the program. Branch points make up valid test cases because they are important points where the decision of next instruction to be executed is made. Therefore, one can say that by looking at the branch points, the behavior of the program can be deduced. This behavior is to be expected to be the same for original and refactored versions of our programs. By testing these branching points symbolic engine provides, we can reason about equivalency of the different implementations of the same component. For all the reasons listed above, main method to extract test cases for equivalency check is decided to be the symbolic execution-based testing.

3.3. Tools Selection

We build our abstractions based on the requirements of a conceptual tool that will be used to question two software components' equivalency using unit tests which are generated by using test cases extracted from a symbolic execution engine. The components of such system could be listed below as:

- A parser to extract necessary information from the software component. This parser is responsible for providing information about functions of the component, their names, return values, arguments and their types.
- The symbolic engine itself. Engine will provide us ways to symbolically execute our code excerpt and generate pivotal points for that execution sessions for us to use in our unit tests.

- A test scenario generator which will use the information parser and symbolic engine provides to generate our test scenario.
- A unit test engine to run our scenario.
- An OS communicator module which will let us operate the tools to generate our test environment.

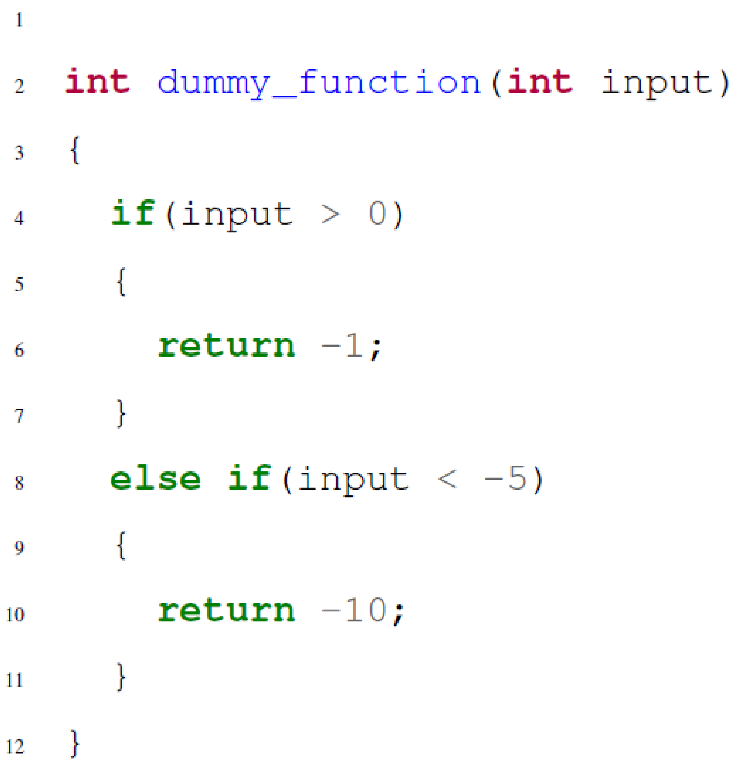
The most important component is naturally the symbolic execution engine. As our aim is to build a software equivalency checker, there will not be an attempt to build our own engine. Fortunately, we have several options to choose from engines that are able to symbolically execute programs in C programming language. Those are namely:

- Crest
- Otter
- CIVL
- KLEE

Choosing our symbolic execution engine among these options is considerably easy for several reasons. First and foremost, advancements in overall computing power and constraint solvers made symbolic execution viable. Exploring the vast solution space, especially when there are many paths to explore, could potentially take days. To make use of these novel methods, it is reasonable to pick the engine with most vibrant community and best available support. KLEE stands out from the rest based on these prerequisites because it has the most recent code base compared to other engines. It is still being updated and improved in 2018 while other engines seem to be deprecated. Another reason that makes KLEE the best option out of these four is the tooling around it. Thanks to this tool, extraction of information about the symbolic execution session is a breeze, extensive parsing of dump files is not involved. Lastly it is easy to build with the help of its website explaining the process, and also available in container forms for people who does not want to bother with building the engine from scratch. However, it is important to note that it is possible to use any other symbolic engine in our approach, KLEE is only chosen because of the convenient points mentioned above.

3.4. KLEE Symbolic Execution Engine

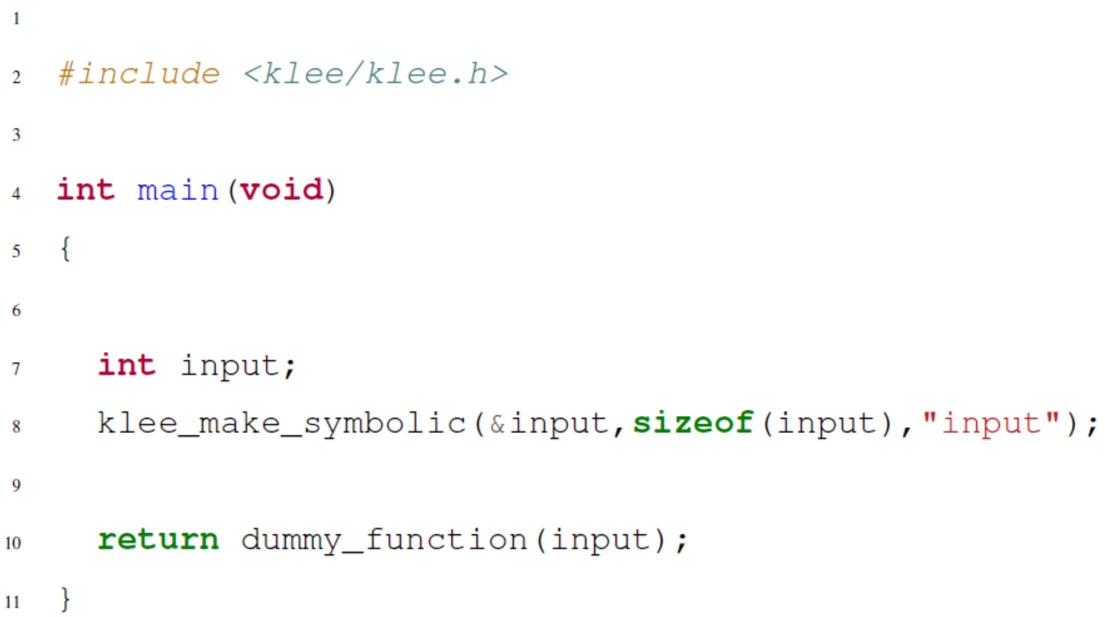
To explain the usage of KLEE, the simple example given in Figure 3.2 and Figure 3.3 are provided. Suppose we have a function named “dummy_function” as defined in Figure 3.2 :



```

int dummy_function(int input)
{
  if(input > 0)
  {
    return -1;
  }
  else if(input < -5)
  {
    return -10;
  }
}
```

Figure 3.2 : Sample function

To make this function execute symbolically we make use of KLEE’s provided functions as such as the one given in Figure 3.3



```c

#include <klee/klee.h>

int main(void)
{

   int input;
   klee_make_symbolic(&input,sizeof(input),"input");

   return dummy_function(input);
}
```

Figure 3.3 : Example program

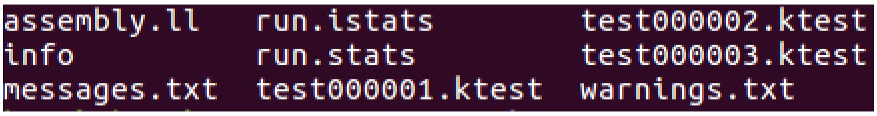
This small program now will be compiled using C compiler Clang using the command:

$ clang -I ../../include -emit-llvm -c -g main.c

This will provide us the intermediary llvm bytecode(.bc) file which will be used as :

$ klee main.bc

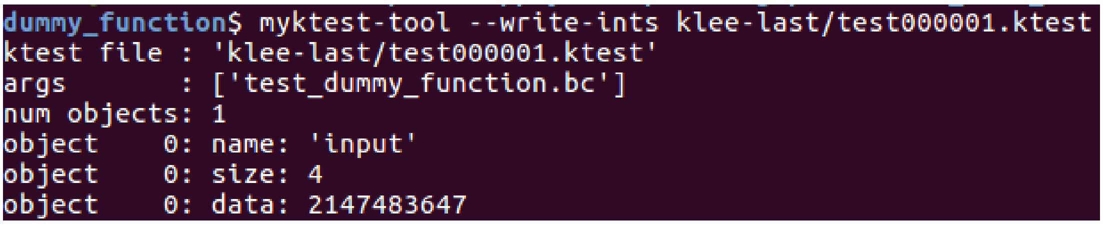
Now KLEE will generate metadata about the symbolic run and put them in a folder called /klee-last with the content given in Figure 3.4.:



```
assembly.ll    run.istats        test000002.ktest
info           run.stats         test000003.ktest
messages.txt   test000001.ktest  warnings.txt
```

Figure 3.4 : Folder that contains metadata of the sample KLEE run

In our provided example, we have three test cases generated for us. Making use of the test tool KLEE provides along with the engine, we can conveniently see what these cases as shown in Figure 3.5:

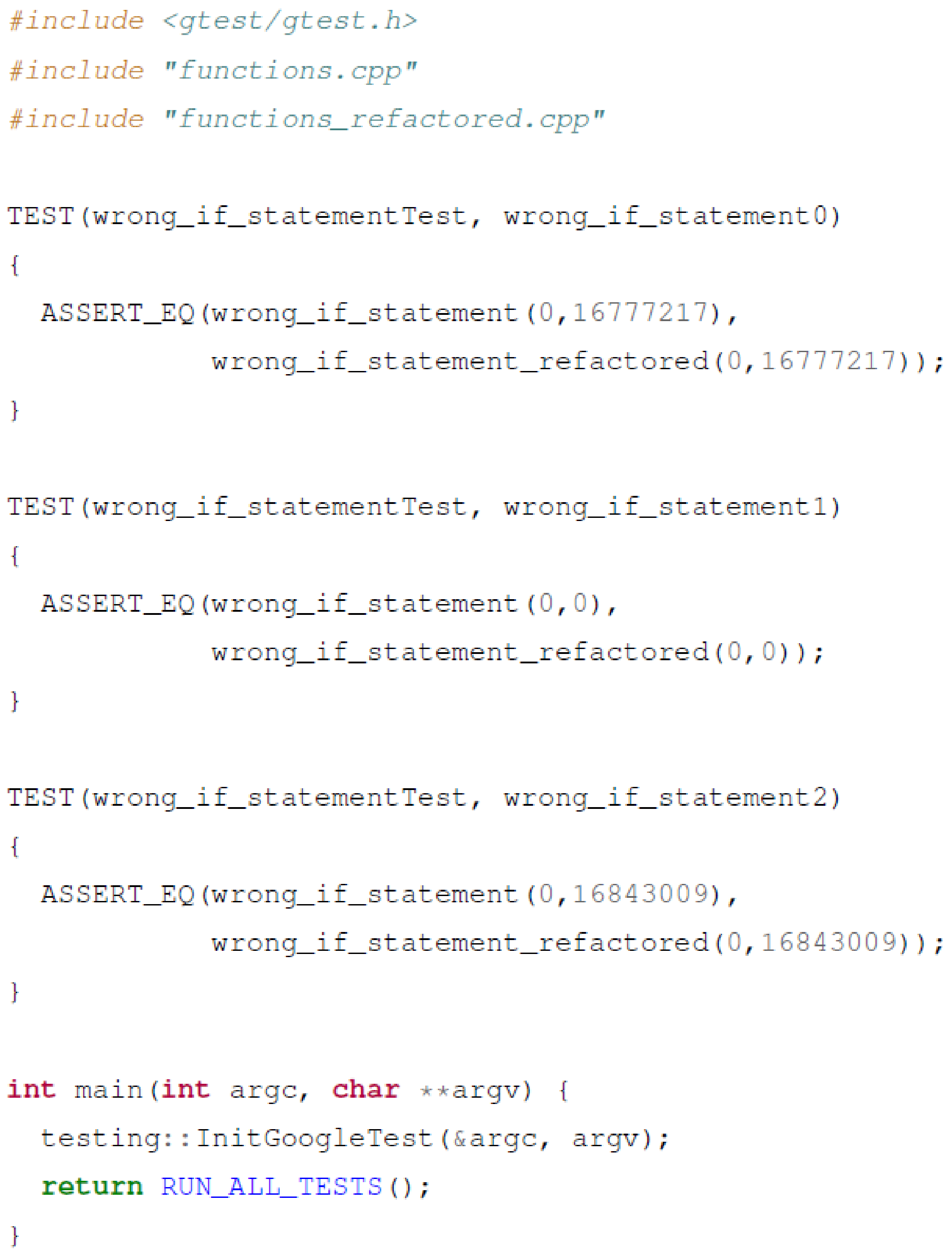


```
dummy_function$ myktest-tool --write-ints klee-last/test000001.ktest
ktest file : 'klee-last/test000001.ktest'
args       : ['test_dummy_function.bc']
num objects: 1
object    0: name: 'input'
object    0: size: 4
object    0: data: 2147483647
```

Figure 3.5 : Example KLEE test tool run

3.5. Testing Approach Employed

A prototype implementation of the proposed approach illustrated by the flowchart given in Figure 3.7. For ease of testing and convenience to the user, all steps in the flowchart conducted by a python script given in Appendix 3 and fully automatic. Our script starts with copying both the original and modified components to a test folder. After this, using header file of the original component, the function prototypes are extracted by invoking Ctags software. These function prototypes are then used to extract information such as the name of the function, return type of the function and list of the function arguments. This extraction is done by a custom parser implemented in our script. For functions that are using other functions of the same component, the cases are generated so that the file contains every function implementation used by this function to properly compile the test case, parser is also responsible for this. Then, this information is used to create a file in the form of Figure 3.3. This file will be fed to the symbolic engine. After this file creation, a Makefile is created which contains the recipe to generate LLVM bytecode and necessary steps to generate test cases from this bytecode by making use of the symbolic execution engine. A template of this Makefile can be found in Appendix 3 'make_test_cases'. Both of these files are then put in a folder and Make is executed. After the execution of Make, KLEE will generate test cases under this folder. Test cases are generated from both modified and original components, therefore enabling us to catch more errors. Following this, test cases for each function are gathered and a file as in Figure 3.6 generated along with a Makefile which contains required recipe to compile and link this file into an executable. While naming the generated test cases, first "Test" is appended to the end of the function name for denoting the unit being tested. After this, each case is named by adding a number at the end of the function name starting by zero. For example, for function "wrong_if_statement" one can see that the unit being tested named as "wrong_if_statementTest" and test cases starts from "wrong_if_statement0". The values such as "0" and "16777217" are automatically generated by KLEE. For environment mocking purposes one can use the compiler flag -DMOCK to provide a platform independent replacement for platform dependent code regions. The keyword "ASSERT_EQ" is macro that checks whether two values are equal. With this way platform dependent code can also be tested with our approach. In the final step, this executable is run, and tests results will be printed out on the console.



```
#include <gtest/gtest.h>
#include "functions.cpp"
#include "functions_refactored.cpp"

TEST(wrong_if_statementTest, wrong_if_statement0)
{
  ASSERT_EQ(wrong_if_statement(0,16777217),
            wrong_if_statement_refactored(0,16777217));
}

TEST(wrong_if_statementTest, wrong_if_statement1)
{
  ASSERT_EQ(wrong_if_statement(0,0),
            wrong_if_statement_refactored(0,0));
}

TEST(wrong_if_statementTest, wrong_if_statement2)
{
  ASSERT_EQ(wrong_if_statement(0,16843009),
            wrong_if_statement_refactored(0,16843009));
}

int main(int argc, char **argv) {
  testing::InitGoogleTest(&argc, argv);
  return RUN_ALL_TESTS();
}
```

Figure 3.6 : File tests.c

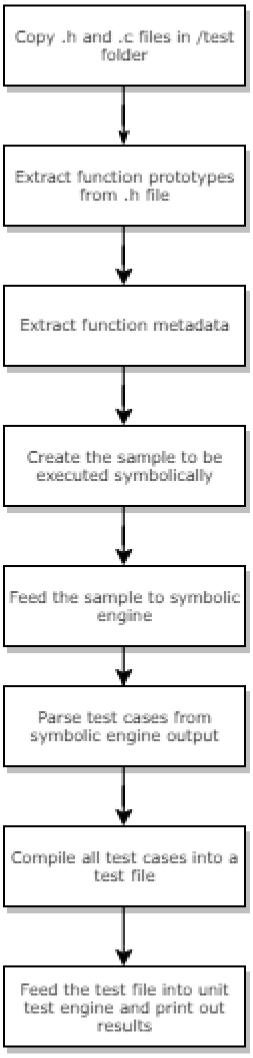

Copy .h and .c files in /test folder
Extract function prototypes from .h file
Extract function metadata
Create the sample to be executed symbolically
Feed the sample to symbolic engine
Parse test cases from symbolic engine output
Compile all test cases into a test file
Feed the test file into unit test engine and print out results

Figure 3.7 : Flowchart of our approach

CHAPTER 4

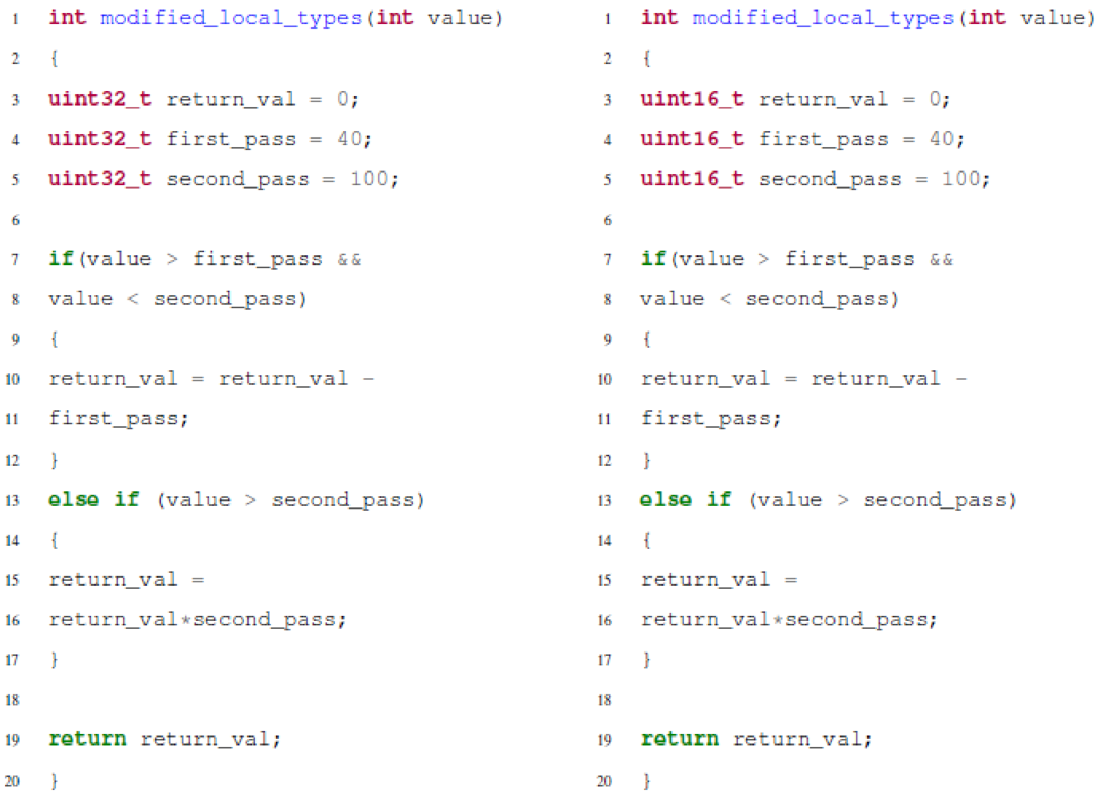
EVALUATION OF THE PROPOSED METHOD

We evaluated the method via two cases. First evaluation scenario is for checking the success of our approach on small functions. Second evaluation scenario focuses on finding errors in different versions of a more complex algorithm that is implemented by several functions. In the first case, our tool will be tested against two synthetic software components. Every function in the modified component contains an error. In the second case, we used two different implementations of a mock algorithm having a common interface. The second implementation was also introduced with several software bugs. As well as introducing bugs, the second implementation consists of refactored implementations for every function involved in the first implementation of the algorithm. Each of these cases is employed to check whether our tool is able to pinpoint common errors that can possibly be seen in every software project. All of the test cases were run on a Ubuntu (build 4.15.0-34-generic) virtual machine in VirtualBox [28] on top of a machine with the following specifications : Intel i7 CPU 2.8 @ GHz, 16 GBs of RAM, 1 TB of HDD, 64-bit Windows 10 operating system.

4.1. Error Insertion

In this section, we will explain how software errors are inserted in the functions to be tested in two cases. We are only providing three examples for brevity. Rest of the code snippets that we have tested our cases against could be found in Appendix.

The first error case stems from modified local types (Figure 4.1). Changing local data types without planning ahead could lead to overflows and many other undesired cases. Our tool was able to find 5 unique test cases for this example below. Out of these 5 cases, input values 41 and 44 was able to showcase failure cases.

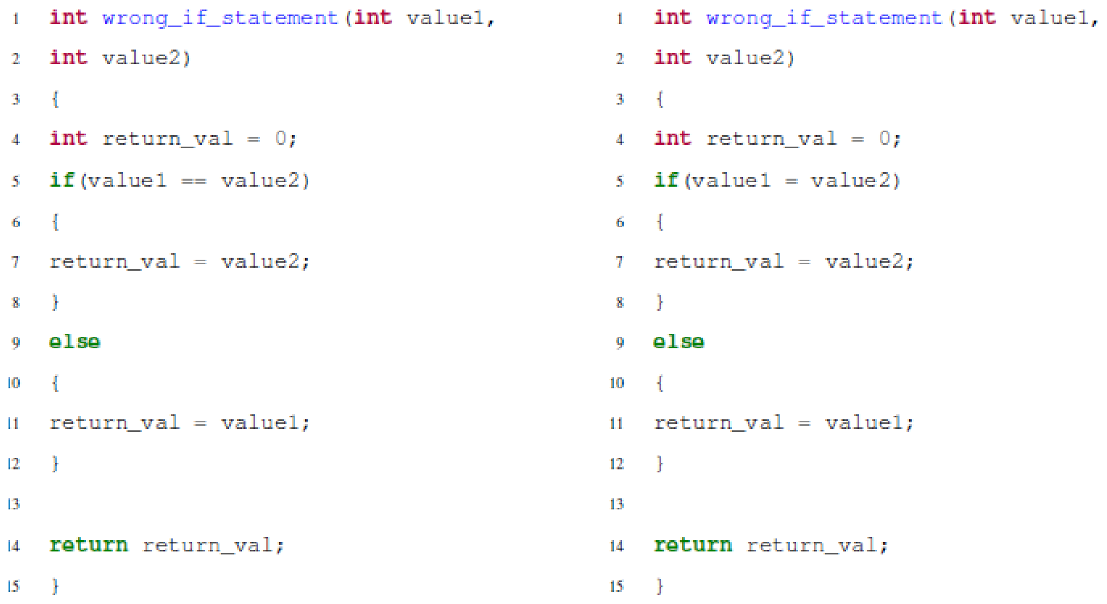


```
int modified_local_types(int value)
{
uint32_t return_val = 0;
uint32_t first_pass = 40;
uint32_t second_pass = 100;

if(value > first_pass &&
value < second_pass)
{
return_val = return_val -
first_pass;
}
else if (value > second_pass)
{
return_val =
return_val*second_pass;
}

return return_val;
}
```

```
int modified_local_types(int value)
{
uint16_t return_val = 0;
uint16_t first_pass = 40;
uint16_t second_pass = 100;

if(value > first_pass &&
value < second_pass)
{
return_val = return_val -
first_pass;
}
else if (value > second_pass)
{
return_val =
return_val*second_pass;
}

return return_val;
}
```

Figure 4.1 : Modified local types

The second example contains a wrongly used if statement (Figure 4.2). Notice that on the fifth line of the original code sample there exists a check between **value1** and **value2**. However, on the modified version of the function one can see that one '=' is missing. Although it is clearly not what the developer intended to do it is a valid C statement. Nowadays modern toolchains can easily catch errors like this. However, working with the bleeding edge toolchains are not always possible when embedded systems are in consideration. Improper use of compiler flags can also cause this error to go unnoticed.

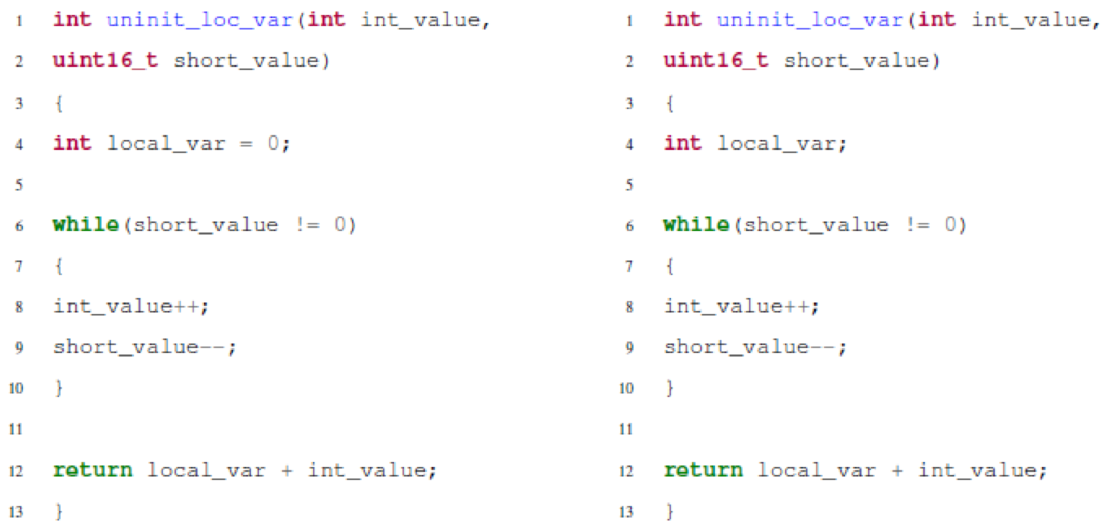


```c
int wrong_if_statement(int value1,
int value2)
{
int return_val = 0;
if(value1 == value2)
{
return_val = value2;
}
else
{
return_val = value1;
}

return return_val;
}
```

```c
int wrong_if_statement(int value1,
int value2)
{
int return_val = 0;
if(value1 = value2)
{
return_val = value2;
}
else
{
return_val = value1;
}

return return_val;
}
```

Figure 4.2 : Wrong if statement

The third example features an uninitialized local variable **local_var** (Figure 4.3). It is easy to fix but can have devastating consequences if not realized in development time. This example in particular will cause you to have different output values depending on your compiler and linker options.



```c
int uninit_loc_var(int int_value,
uint16_t short_value)
{
int local_var = 0;

while(short_value != 0)
{
int_value++;
short_value--;
}

return local_var + int_value;
}
```

```c
int uninit_loc_var(int int_value,
uint16_t short_value)
{
int local_var;

while(short_value != 0)
{
int_value++;
short_value--;
}

return local_var + int_value;
}
```

Figure 4.3 : Uninitialized local variable

4.2. Evaluation with Common Error Scenarios

This approach will investigate the performance of our approach with several software bugs that are found in some of the prevalent open source software projects. Our tool will do the equivalence checking of a synthetic component containing each of these popular software bugs in several methods. The modified component contains the ‘fixed’ versions of these methods. These common software bugs [29] contains cases such as :

- Modified local variable types
- Uninitialized local variable
- Forgotten break statement
- Operator precedence
- Incorrect dereference
- Incorrect loop control statements

In the first scenario, we seeded common software bugs into simple functions as explained in Section 4.1. Both versions of the functions tested are given in Appendix 4.1. Then, we applied our testing tool. Our tool is able to extract 36 test cases out of both modified and original components as can be seen from Table 4.1. The script took 24.54 seconds to execute in the machine configuration mentioned in above. Out of these 36 test cases, 19 of them passed and 17 of them were failed. It was able to detect each error case mentioned above except incorrect dereference example where it failed to generate proper test cases in time and simply aborted. These error cases are extracted from widely used open source projects [29] can very well end up in your code base too.

Table 4.1 : Overview of Results

	Total Cases	Cases Passed	Cases Failed	Successful
Modified local variable types	6	4	2	Y
Uninitialized local variable	4	0	4	Y
Forgotten break statement	8	6	2	Y
Operator precedence	10	5	5	Y
Incorrect dereference	0	0	0	N
Incorrect loop control statements	8	4	4	Y

4.3. Evaluation with a Larger Code Example

This approach will use two different implementations of the same algorithm. Our approach extracts test cases from each implementation and test both implementations using these cases. Implementation of the algorithm can be seen in appendix. The flowchart of the algorithm is given in Figure 4.4. To check whether our approach is able to find errors in a larger code example, we inject each of the software bugs from section 4.1

Below we will explain how our approach performed for each case. It was able to generate 28 test cases in total and 11 of them failed as can be seen from Table 4.2. The script took 127.36 seconds to execute in the machine configuration given above, note that this number is quite different than the first example in section 4.2 because symbolic engine had to symbolize arrays of integers instead of single integers this time. Please note that you can only find source files and the python script to execute our approach but not the test cases themselves. The reason for that is, the test cases generated are quite verbose and including them in this document itself will simply blow the size of it out of proportions. Instead, for readers' convenience, we made all the source files, scripts and a user manual explaining how to setup your own environment available in a Github repository [30]. This way, readers who are interested can also reproduce the case for themselves and even modify it to use for their cases.

For modified local variable types we introduced a bug to the refactored version of quicksort algorithm. In the modified version of the quicksort function, one of the sub functions called "split" has a local variable called "part_element". This variable's type changed into int_16t. Our approach was able to catch the erroneous condition by generation 6 test cases out of 3 failed.

"find_min" method contains an uninitialized variable in its modified implementation. The nature of the algorithm calls for a variable with a big value so that it can do the correct comparison for the whole span of integer values. However, in the modified case this is simply forgotten. Our approach was able to generate 4 test cases for this function and out of this 4, one of them failed.

Function named "scale" and "exit_filter" contains the forgotten break statement scenario. Our approach generated five test cases for this method "scale". However, none of the test cases was able to pinpoint the problem. When inspected, each test case generated has different values for input parameter "scale_factor". However, for the first input parameter for array "*in" one can see the all five test cases contains an array full of zeros. Since scaling these "zero arrays" will only produce another array filled with zeros. For function "exit_filter" our approach generated 8 test cases where 2 of them failed. In short, we were able to catch the forgotten break statement but only one flavor of it.

Function named "custom_filter" is the example case for operator precedence. Notice that in second implementation has braces around the and operation with flag. Our approach was able to uncover this error by generating 1 failing case out of 4 test cases generated.

In function "reverse_array", one can see that in the first version a dynamic array is created and passed onto the out pointer, however in second case a local variable created and passed on instead. In second case, since the local variable will get destructed after we get out of the function context it is for sure a bug. Our approach however fails to detect this case. It is able to generate 2 test cases and none of them was able to pinpoint the error case, failing to catch incorrect dereference scenario.

In function "offset_array", the modified version does the offsetting 4 elements at a time. However, when looked carefully, one can see that the increment is done until the loop variable reaches "arrlen/4". This causes the indexing to be faulty in main body of the for loop. Our approach was able to generate 3 test cases for this specific scenario and, out of those cases, one of them was able to detect the error case.

Table 4.2 : Overview of Results

	Total Cases	Cases Passed	Cases Failed	Successful
Modified local variable types	6	4	2	Y
Uninitialized local variable	4	0	4	Y
Forgotten break statement	8	6	2	Y
Operator precedence	10	5	5	Y
Incorrect dereference	0	0	0	N
Incorrect loop control statements	8	4	4	Y

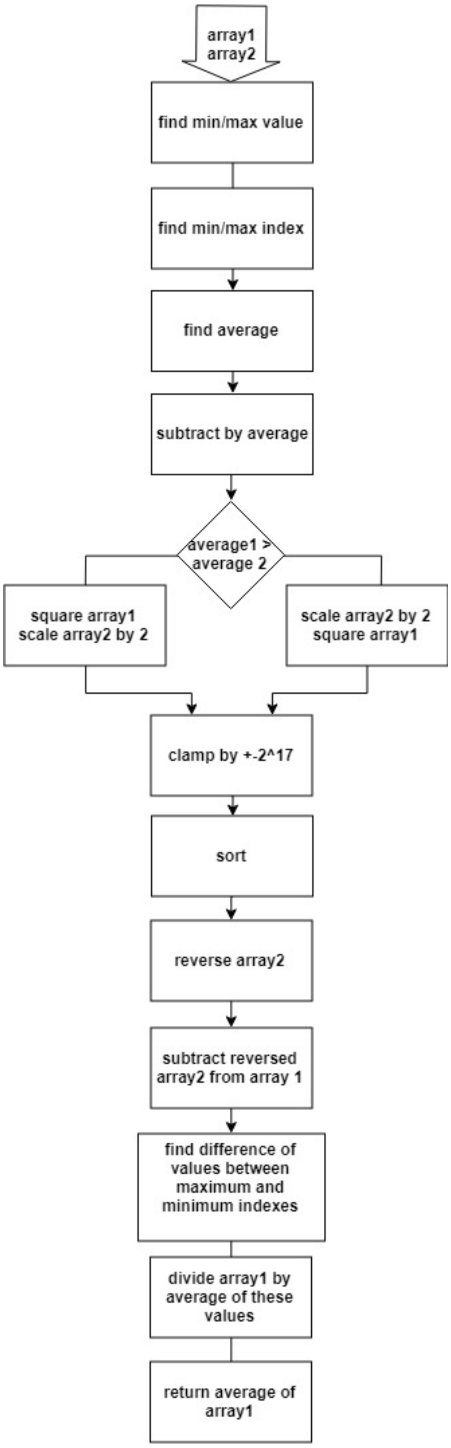


Figure 4.4 : Flowchart of the algorithm in discussion

4.4. Suggestions for Incorrect Dereference Case

One can see that in both evaluation cases our approach fails to catch incorrect dereference case. The failed cases we have observed stems from the fact that generated case for an input array is simply filled with zeroes. This causes program to not follow the branch that will have error on its path and error stays undetected. For situations like these the user may need to perform some modifications on the source code to help the symbolic engine generate more meaningful test cases. These modifications can be:

- Offsetting the input array with a nonzero value
- Changing the branching order of the program
- Changing the branching conditions of the program

CHAPTER 5

CONCLUSION

In this final chapter, we criticize of our approach, mention our contributions, shortcomings and possible future improvements. The problem at hand was to see whether we will be able to detect whether two software components are functionally equivalent or not. For this endeavor, a literature research focusing on equivalence testing is conducted. Out of many candidates, we dwindled down the number of studies into eight. Within these studies the ideas are quite varying. Some of them are focusing on formal methods, while others are borrowing ideas from electrical engineering and quite interestingly some of them are even making use of textual differences to find equivalence between software components. The approach of Jiang et al. [7] and Papadakis et al. [6] for example is much better suited for finding equivalent code pieces in large code bases compared to our approach. Post et al. [8]cannot handle complex C structures as they are bounded by the tool they are using, therefore need manual intervention. Godlin et al. [10] also have the same problem because of the same reason. Although our approach does not have the same shortcomings with Post et al. [8] and Godlin et al. [10] with respect to manual intervention there is still issues with scalability that we are also experiencing. Our approach is quite similar to Matsumoto et al. [11] although it can be said that they are more better suited for automatically generated programs compared to our approach. Approach of cutpoints by Feng et al. [2] can provide us with better execution time if it can be extended to use C programming language. Ciobaca et al. [3] provide better coverage with their language agnostic approach and it is also mathematically proven, once their approach is able to tackle on parametrized functions it will be better suited to use in domains such as aerospace and medical industry which requires absolute precision and correctness. If one is looking for a solution where each commit is compared against the previous one, approach of Wood et al. [4] is much more suitable then our approach.

In this study, we based equivalence checking on unit tests. Moreover, the process is aimed to be automatic. Therefore we have shown that symbolically executing the software and extracting important branch points, and later on using them as our test cases is a viable method to check equivalence. To prove what we have deduced, we provide a prototype implementation of our approach. In order to achieve this, we've looked into how industry deals with automatically generated test cases and symbolic execution engines. This literature research yielded the symbolic engine to be used and the outline of the architecture of our approach. Finally, we evaluated the success of our approach by means of two sample cases.

The first one was focusing in synthetic software functions introduced with common software errors. Our approach was able to generate 36 test cases for these functions in 25.54 seconds. Out of these cases 17 of them failed. Therefore, except the incorrect dereference error, all software bugs inserted are detected by our tool. In the incorrect dereference example, our tool failed to generate proper test cases in the predefined time interval and simply aborted.

The second case was to insert these bugs on a larger code example and to see whether our solution is able to detect it. Our solution was quite successful in first case of evaluation where it detected all software bugs with ease except "incorrect dereference" scenario. However, second method of evaluation proved more challenging for us, as our approach was not able to detect "incorrect dereference" scenario. Our approach generated 28 test cases in 127.36 seconds where 11 of them failed. For this method of evaluation success rate of the cases is 5 out of 6.

Under light of these evaluations, our achievements can be said to include automatic generation of unit tests for a specific software component and its modified counterpart by simply executing a script. It was important for us to make this process fully automated since it could be a part of nightly build process in a professional software development environment. The script also leaves the user with a test bench that can easily be extentable for further modifications to the cases.

Future work is needed to extend it to be used with C++. Further investigation is also needed as our approach struggle to different flavors of the same error type. This problem stems from the fact that branch points generated from symbolic engine are sometimes simply useless values as a test case. To tackle this problem future work could be invested on building better test programs to feed into symbolic engine. Right now the test program is quite straightforward where it simply makes input parameters symbolic and calls the function with those values.

REFERENCES

[1] *Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions),* 2018 (accessed March 3, 2018).

[2] X. Feng, A. J. Hu, D. Computer Science and U. British Columbia, "Cutpoints for Formal Equivalence Verification of Embedded Software," 2005.

[3] S. Ciobaca, D. Lucanu, V. Rusu and G. Ro?u, "A language-independent proof system for full program equivalence," *Formal Aspects of Computing,* vol. 28, pp. 469-497, 2016.

[4] T. Wood and S. Drossopoulou, "Program Equivalence through Trace Equivalence," 2013.

[5] Microsoft, "Dafny: A Language and Program Verifier for Functional Correctness," 2008. [Online]. Available: https://www.microsoft.com/en-us/research/project/dafny-a-language-and-program-verifier-for-functional-correctness/.

[6] M. Papadakis, Y. Jia, M. Harman and Y. Le Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," *Proceedings - International Conference on Software Engineering,* vol. 1, pp. 936-946, 2015.

[7] L. Jiang and Z. Su, "Automatic mining of functionally equivalent code fragments via random testing," *Proceedings of the eighteenth international symposium on Software testing and analysis - ISSTA '09,* p. 81, 2009.

[8] H. Post and C. Sinz, "Proving functional equivalence of two AES implementations using bounded model checking," *Proceedings - 2nd International Conference on Software Testing, Verification, and Validation, ICST 2009*, pp. 31-40, 2009.

[9] A. Biere, A. Cimatti and E. M. Clarke, "Bounded Model Checking," vol. 58, 2003.

[10] O. Strichman and B. Godlin, "Regression verification - A practical way to verify programs," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics),* vol. 4171 LNCS, pp. 496-501, 2008.

[11] T. Matsumoto, H. Saito and M. Fujita, "An equivalence checking method for C descriptions based on symbolic simulation with textual differences," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences,* Vols. E88-A, pp. 3315-3322, 2005.

[12] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology (NIST),* p. 309, 2002.

[13] R. Baldoni, E. Coppa, D. C. D'Elia, C. Demetrescu and I. Finocchi, "A Survey of Symbolic Execution Techniques," pp. 1-53, 2016.

[14] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Communications of the ACM,* vol. 56, pp. 82-90, 2013.

[15] C. Cadar, D. Dunbar and D. R. Engler, "KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs," *Proceedings of the 8th USENIX conference on Operating systems design and implementation,* pp. 209-224, 2008.

[16] L. Apfelbaum and J. Doyle, "Model Based Testing," *Procedings of the 10th Internation Software Quality Week,* pp. 1-14, 1997.

[17] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *Journal of Systems and Software,* vol. 86, pp. 1978-2001, 2013.

[18] A. Orso and G. Rothermel, "Software testing: a research travelogue (2000-2014)," *Proceedings of the on Future of Software Engineering - FOSE 2014,* pp. 117-132, 2014.

[19] A. Kerbrat, T. Jéron and R. Groz, "Automated test generation from SDL specifications," in *SDL Forum*, 1999.

[20] C. Nebut, F. Fleurey, Y. L. Traon and J. M. Jezequel, "Automatic Test Generation: A Use Case Driven Approach," 2006.

[21] A. Hartman, "Model Based Testing : What ? Why ? How ? and Who cares ?," *Agenda*, 2006.

[22] G. Antoniol, "Search based software testing for software security: Breaking code to make it safer," *IEEE International Conference on Software Testing, Verification, and Validation Workshops, ICSTW 2009,* pp. 87-100, 2009.

[23] R. Hamlet, "Random testing," *Encyclopedia of Software Engineering,* pp. 970-978, 1994.

[24] T. Y. Chen, H. Leung and I. K. Mak, "Adaptive Random Testing," *Advances in Computer Science - ASIAN 2004,* pp. 3156-3157, 2005.

[25] P. Godefroid, N. Klarlund and K. Sen, "DART: directed automated random testing," *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation,* pp. 213-223, 2005.

[26] P. McMinn, "Search-Based Software Testing: Past, Present and Future," *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops,* pp. 153-163, 2011.

[27] M. Harman, Empirical Software Engineering and Verification, vol. 7007, 2012.

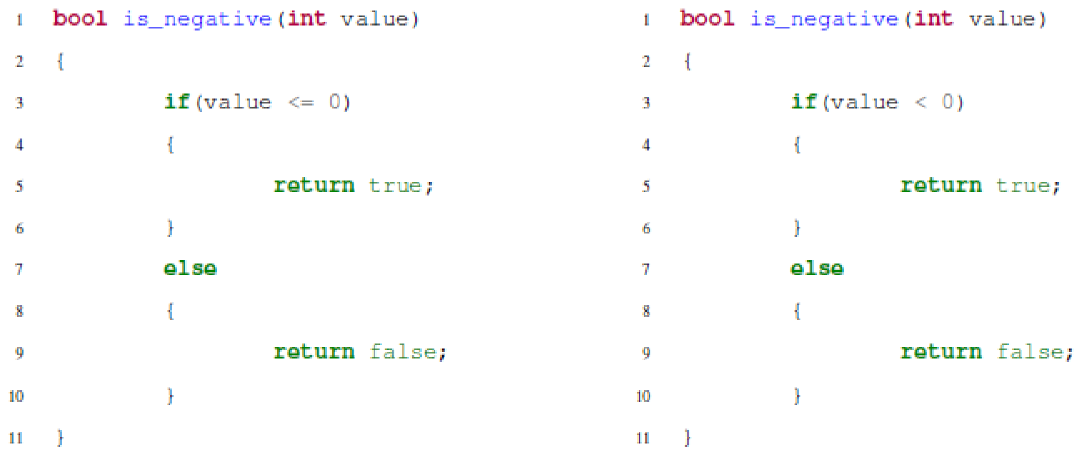
[28] "VirtualBox," 2018 (accessed September 29, 2018). [Online]. Available: https://www.virtualbox.org/.

[29] E. R. Andrey Karpov, *100 bugs in Open Source C/C++ projects,* 2012 (accessed March 3, 2018).

[30] B. Ünaltay, "thesis," 2018 (accessed September 29, 2018). [Online]. Available: https://github.com/burakunaltay/thesis.

APPENDIX

ERROR CASES



```
bool is_negative(int value)
{
        if(value <= 0)
        {
                return true;
        }
        else
        {
                return false;
        }
}
```

```
bool is_negative(int value)
{
        if(value < 0)
        {
                return true;
        }
        else
        {
                return false;
        }
}
```

Figure A.1 : Forgotten equal sign

```c
int modified_local_types(int value)
{
uint32_t return_val = 0;
uint32_t first_pass = 40;
uint32_t second_pass = 100;

if(value > first_pass &&
value < second_pass)
{
        return_val =
         return_val - first_pass;
}
else if (value > second_pass)
{
        return_val =
         return_val * second_pass;
}

return return_val;
}
```

```c
int modified_local_types(int value)
{
uint16_t return_val = 0;
uint16_t first_pass = 40;
uint16_t second_pass = 100;

if(value > first_pass &&
value < second_pass)
{
        return_val =
         return_val - first_pass;
}
else if (value > second_pass)
{
        return_val =
         return_val * second_pass;
}

return return_val;
}
```

Figure A.2 : Modified local types

```c
int wrong_if_statement(int value1,
int value2)
{
int return_val = 0;
if(value1 == value2)
{
return_val = value2;
}
else
{
return_val = value1;
}

return return_val;
}
```

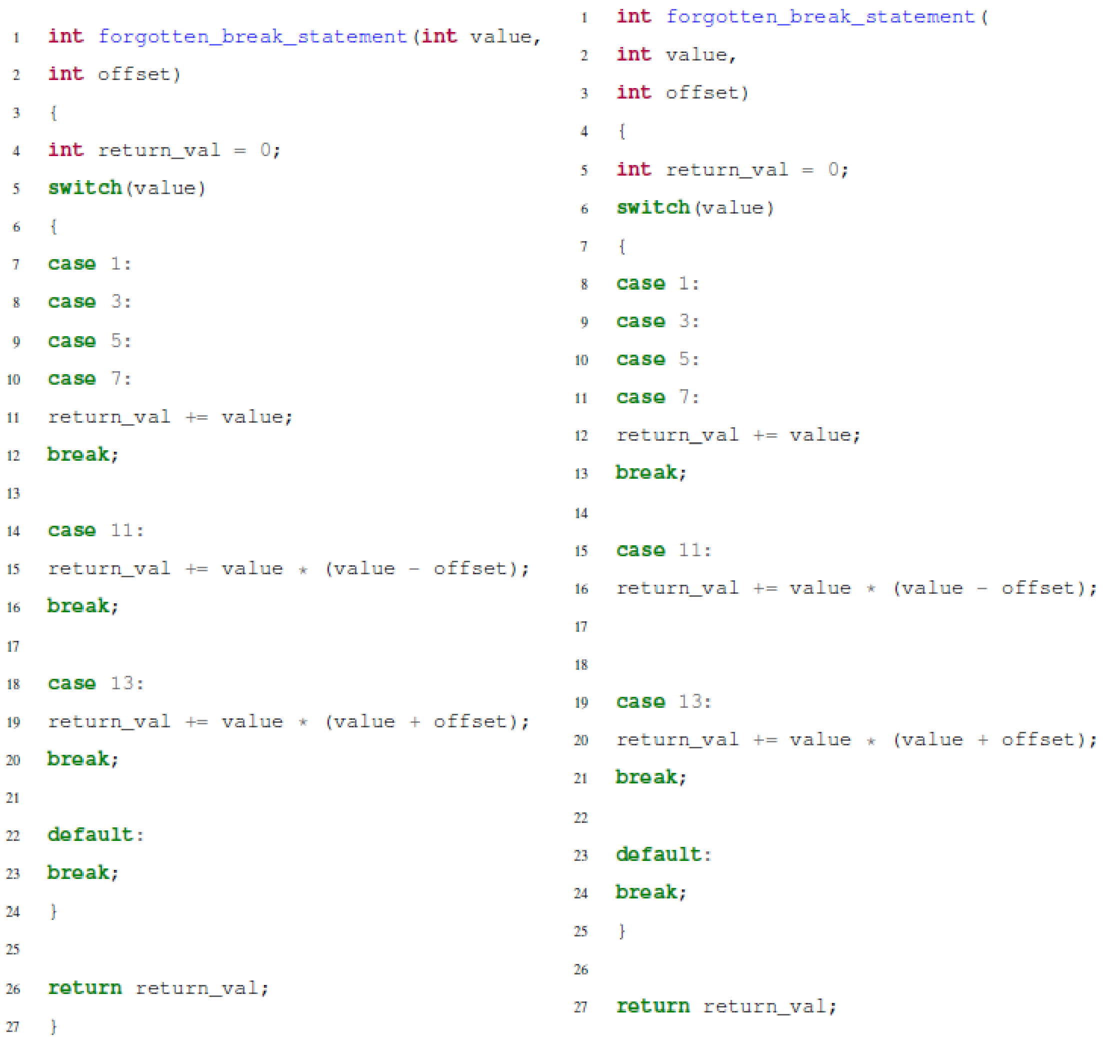
```c
int wrong_if_statement(int value1,
int value2)
{
int return_val = 0;
if(value1 = value2)
{
return_val = value2;
}
else
{
return_val = value1;
}

return return_val;
}
```

Figure A.3 : Wrong if statement

```c
int uninitialized_local_var(
int int_value,
uint16_t short_value)
{
int local_var = 0;

while(short_value != 0)
{
int_value++;
short_value--;
}

return local_var + int_value;
}
```

```c
int uninitialized_local_var(
int int_value,
uint16_t short_value)
{
int local_var;

while(short_value != 0)
{
int_value++;
short_value--;
}

return local_var + int_value;
}
```

Figure A.4 : Uninitialized local variable



```c
int forgotten_break_statement(int value,
int offset)
{
int return_val = 0;
switch(value)
{
case 1:
case 3:
case 5:
case 7:
return_val += value;
break;

case 11:
return_val += value * (value - offset);
break;

case 13:
return_val += value * (value + offset);
break;

default:
break;
}

return return_val;
}
```

```c
int forgotten_break_statement(
int value,
int offset)
{
int return_val = 0;
switch(value)
{
case 1:
case 3:
case 5:
case 7:
return_val += value;
break;

case 11:
return_val += value * (value - offset);


case 13:
return_val += value * (value + offset);
break;

default:
break;
}

return return_val;
```

Figure A.5 : Forgotten break statement

```
int operator_precedence(int value,
bool cond1,
bool cond2)
{
if (!(cond1 & cond2)) {

value = value * 2;

}

return value;
}
```

```
int operator_precedence(int value,
bool cond1,
bool cond2)
{
if ((!cond1) & cond2) {

value = value * 2;

}

return value;
}
```

Figure A.6 : Operator precedence I

```
int operator_precedence_2(int value)
{
int* ptr = &value;

*ptr++;

return value;
}
```

```
int operator_precedence_2(int value)
{
int* ptr = &value;

(*ptr)++;

return value;
}
```

Figure A.7 : Operator precedence II

```
bool operator_precedence_3(int value1,
int value2)
{

if(value1 & FLAG == value2)
{
return true;
}

return false;
}
```

```
bool operator_precedence_3(int value1,
int value2)
{
if((value1 & FLAG) == value2)
{
return true;
}

return false;

}
```

Figure A.8 : Operator precedence III