Informatics

# Metamorphic Testing von ZKC Infrastructure

DIPLOMARBEIT

zur Erlangung des akademischen Grades

**Diplom-Ingenieur**

im Rahmen des Studiums

**Software Engineering & Internet Computing**

eingereicht von


**Philipp Leeb, BSc**
Matrikelnummer 11808219

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof.in Dr.in sc. Maria Christakis
Mitwirkung: Valentin Wüstholz, PhD
Projektass. Dipl.-Ing. Christoph Hochrainer, BSc



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub


Wien, 15. August 2024

Philipp Leeb | Maria Christakis

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Informatics

# Metamorphic Testing of ZKC Infrastructure

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

**Diplom-Ingenieur**

in

**Software Engineering & Internet Computing**

by

**Philipp Leeb, BSc**
Registration Number 11808219

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof.in Dr.in sc. Maria Christakis
Assistance: Valentin Wüstholz, PhD
Projektass. Dipl.-Ing. Christoph Hochrainer, BSc

Vienna, August 15, 2024

Philipp Leeb    Maria Christakis



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Erklärung zur Verfassung der Arbeit

Philipp Leeb, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 15. August 2024

Philipp Leeb

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Danksagung

Ich bedanke mich bei Fredrik Dahlgren (Maintainer von *Circomspect*) für die Hilfe mit dem Parser von *Circomspect*. Weiters möchte ich mich bei Valentin Wüstholz, Christoph Hochrainer und vor allem bei Maria Christakis für die Unterstützung während der Planung, Implementierung und Evaluation der Arbeit bedanken.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

# Acknowledgements

I want to thank Fredrik Dahlgren (Maintainer of *Circomspect*) who helped me to get started with the parser of *Circomspect*. Also, thanks to Valentin Wüstholz, Christoph Hochrainer and especially Maria Christakis for supporting and guiding me during the entire planning, implementation and evaluation.

TU WIEN Bibliothek Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Kurzfassung

In den vergangenen Jahren bekamen Zero Knowledge Proofs immer mehr Aufmerksamkeit im Bereich Security. Durch die steigende Zahl an Zero Knowledge Proof Sprachen sollte auch das Testen der Compiler bedacht werden. Schon seit den frühen Tagen des Programmierens ist automatisiertes Testen von Software ein wichtiger Teil der Qualitätssicherung. Das gilt vor allem für sicherheitskritische Applikationen, wie zum Beispiel Compiler. In dieser Arbeit wird ein Test-Framework vorgestellt, welches mittels Metamorphic Testing auf der Sprache *Circom* evaluiert wird. Zero Knowledge Proof Sprachen wurden bisher noch nicht mittels Metamorphic Testing getestet, weshalb dies den Kern der wissenschaftlichen Arbeit darstellt. Das vorgestellte Framework besteht aus Metamorphic Transformern, Orakeln und zwei Fuzzern, die für die Erstellung von Testinstanzen und zur Verifizierung verwendet werden. Die durchgeführten Experimente enthüllen neue Herausforderungen in Bezug auf Metamorphic Testing in diesem Kontext. Zu den zentralen Ergebnissen dieser Arbeit zählen die 3 gefundenen Bugs, wobei die gemessene Line-Coverage im *Circom* Compiler bei 33,02% liegt. Zusätzlich wurde auch die zusätzlich benötigte Zeit für die Coverage Instrumentationen gemessen.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Abstract

Zero knowledge proofs gained a lot of attention over the last few years in the security domain. With the rising number of zero knowledge proof languages, the need for compiler testing should also be considered. Since the very early days of programming, automated software testing is an important part of quality management, especially for critical infrastructure such as compilers. In this thesis, we propose and implement a testing framework which utilizes metamorphic testing and evaluate it on the *Circom* language. Metamorphic testing on zero knowledge proof infrastructure has not been done before and is the main contribution of this work. The proposed framework consists of metamorphic transformers, oracles and two fuzzers which are utilized to generate new test instances and verify them. The experiments revealed new challenges when applying metamorphic testing in this context. The results exposed 3 bugs while covering 33,02% of the lines in the *Circom* compiler. Additionally, the time consumption of the coverage instrumentations are measured.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Contents

**Kurzfassung** **xi**

**Abstract** **xiii**

**Contents** **xv**

**1 Introduction** **1**
1.1 Background . . . . . . . . . . . . . . . . . . . . . . . . . . . . 1
1.2 Problem Definition and Contribution . . . . . . . . . . . . . . . . . . 3

**2 Related Work** **7**

**3 Methods** **9**
3.1 Testing Framework . . . . . . . . . . . . . . . . . . . . . . . . 9
3.2 Metamorphic Transformations . . . . . . . . . . . . . . . . . . . . 10
3.3 Oracles . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
3.4 Input Fuzzer . . . . . . . . . . . . . . . . . . . . . . . . . . 17
3.5 Configuration Fuzzer . . . . . . . . . . . . . . . . . . . . . . . 17
3.6 Additional Adaptations . . . . . . . . . . . . . . . . . . . . . . 18

**4 Results** **19**
4.1 Datasets . . . . . . . . . . . . . . . . . . . . . . . . . . . . 19
4.2 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . 19

**5 Discussion** **25**
5.1 Limitations . . . . . . . . . . . . . . . . . . . . . . . . . . . 25
5.2 Conclusion . . . . . . . . . . . . . . . . . . . . . . . . . . . 26

**Overview of Generative AI Tools Used** **27**

**Übersicht verwendeter Hilfsmittel** **29**

**List of Figures** **31**

**List of Tables** **33**



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

# CHAPTER 1

# Introduction

The topic of security got increasingly important in the last decades. *Hall and Wright (2018)* analyzed data breaches and their costs induced by cyber crime [17]. In the year 2014, the report of the *Center for Strategic and International Studies* assessed costs of over 400 billion dollars caused by cyber criminal activities. In the same paper, they show that in 2017 and 2018, 29% and 24% respectively of the data breaches are caused by internal errors of systems or employee error.

This motivates the need of testing software to avoid system errors where possible. Especially automated testing helps reducing errors. Although *Rafi et al. (2012)* came to the conclusion that automated testing does not replace manual testing, the test coverage is improved [28].

The next section introduces the underlying concepts and the background of the work, which is then followed by the problem definition and contribution.

## 1.1 Background

This section introduces the utilized concepts and motivates their usages in real life applications.

### 1.1.1 Zero Knowledge Proofs

The underlying concept of *Zero Knowledge Proof* (ZKP) is a minimal and secure communication between two parties without revealing private information. *Feige et al. (1987)* described that a sender $A$ does not want to reveal any information about the used language $L$ and the input $I$ while proving to a receiver $B$ that $A$ knows the state of $I$ in regard to $L$ [15]. ZKP program are functions with inputs and outputs called signals. In addition to that, intermediate signals are variables holding interim results which are not exposed externally.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

To enable this communication, mainly 3 components are utilized [31]:

- **Witness Generator** - Generates an artifact called witness, which holds the values for all intermediate and output signals.
- **Prover** - Generates an artifact called proof from the witness, which does not leak any information about the applied function nor the private inputs.
- **Verifier** - Determines if a proof in combination with the public inputs of the ZKP is valid or not.

These parts of the system enable the sharing of limited information, as shown in Figure 1.1. Alice knows a function $f$ as well as their respective public and private inputs $x$ and $y$. With this information, the witness generator computes a witness $W$ which includes the results of the function. In the next step, the prover takes the witness and produces the proof, $P(x)$ which does not expose information about the private input $y$ nor the function $f$. Bob on the other side only receives the public $x$ and $P(x)$ which can then be verified via the verifier. Only if Bob passes the correct combination of inputs to the proof, the test from the verifier is passed. There is also no way to retrieve the original private inputs or function.

Popular examples for ZKP language frameworks are *ZoKrates* [20], *Noir* [2], *Lurk* [1] and *Circom* [5]. We focus on the latter one in this thesis. These frameworks also support the usage in the *blockchain*, where applications utilize the proof system for communication. An example program of *Circom* is shown in Figure 1.2.

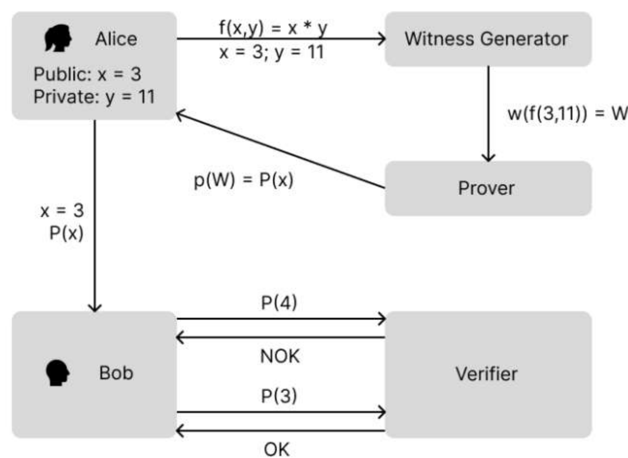


Figure 1.1: The figure shows the process of generating the witness and the proof on the sender side as well as the verification step on the receiving end.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
pragma circom 2.1.8;

template IsEqual() {
signal input in[2];
signal output out;

component isz = IsZero();

in[1] - in[0] ==> isz.in;

isz.out ==> out;
}

component main = IsEqual();
```

Figure 1.2: The code shows an example for a program written in *Circom*. This example is taken from the util library *circomlib*. The IsEqual template takes an input signal with an array size of 2 and provides one output signal in return. To calculate the result, the component IsZero is called with the subtraction of both input values and assigns its output accordingly. Below the template, a main component is defined to call the implementation.

Real life applications integrate ZKPs in the *blockchain* to hide data while providing the possibility to validate a transaction on the chain. *Zcash* is a cryptographic protocol which incorporates ZKPs for encryption, where the hiding of private keys is made possible. A current research topic is the use of ZKPs in authentication systems. The sender is able to prove the validity of a password without revealing it. This is an alternative approach to the commonly used asymmetric key cryptography [18].

### 1.1.2 Metamorphic Testing

The second concept to introduce is a testing method called *metamorphic testing*. In general, it is done by taking a seed program and transforming it with a known operation s.t. the relation from the seed program output to the corresponding target output is known [10]. After running the program, the outputs should be related to each other, which is previously determined by the transformation. For example: given an input tuple $A = (x, y)$ with a program $P(x, y) = x + y$, a generated metamorphic test case may be $P'(x, y) = x + y + 1$ with an output relation of *greater than*. This is a simple example, but the essential part is, that by mapping $P$ to $P'$, we can check the relation of both outputs where $P'(x, y) >= P(x, y)$. Figure 1.3 shows an example of a program which is transformed with a *greater than* relation. To increase the probability of finding issues within the underlying infrastructure, the programs are executed multiple times with different inputs and verify their outputs with the respective output relations.

## 1.2 Problem Definition and Contribution

As motivated in the introduction, security is an important part of software development and a lot of attacks are made possible by bugs in the target system. To find these bugs,



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

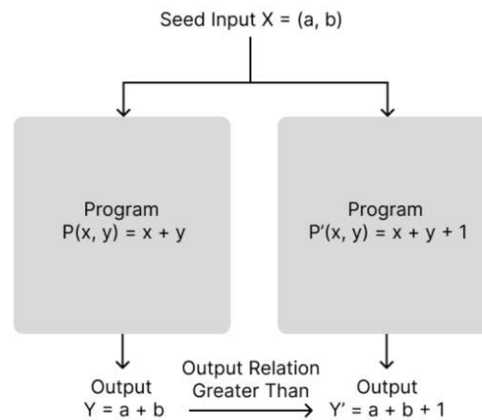


Figure 1.3: The figure shows an example of metamorphic testing with a *greater than* output relation.

sophisticated testing methods are needed. Besides the traditional unit and integration tests, randomized test are helpful to reveal a wider range of issues. Especially for compilers, metamorphic testing is a suitable method to extend testing.

In the context of this thesis, the novel part is to use metamorphic testing for a ZKP language framework, which has not been done before. For that, a new testing framework is proposed which enables the automated generation of test instances by using metamorphic transformations and facilitates more diversity due to swarm testing [16]. The automated generation of tests takes seed programs as a starting point and applies transformations on them. To maintain comparability, additional information is stored on how the transformation changed the program. These details are used during the verifying step to check whether the results are correct or not. The actual execution of the generated programs is done multiple times per instance with randomly generated inputs to cover a wider range of possible paths.

To demonstrate the steps of the framework, Figure 1.4 gives an abstract example. On the left side, a statement of a seed program is modified with different transformations and the relation to the output, namely *equals*, is forwarded to the verify step. The execution of each program is done multiple times with the inputs providing different values of $y$. The result is then compared with the original values. In this example, the operations in the expression must not change the results in comparison to the executions of the seed program. If there are differences, there must be a bug within the underlying infrastructure.

In addition to the testing framework, we also discuss the challenges and limitations when testing ZKP language frameworks with an automated testing framework. In the evaluation of the results, the performance of test runs in terms of time and coverage is presented as well as regression testing and the found bugs during the testing phase.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

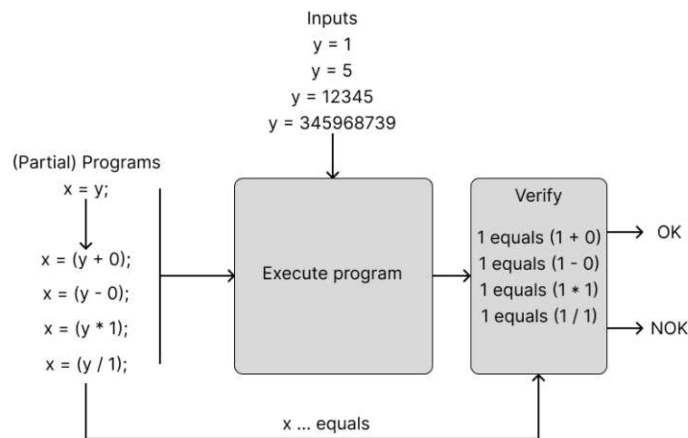


Figure 1.4: This figure provides a brief introduction to the testing process with an example. A program gets transformed, executed with different inputs and then verified.

To the best of our knowledge, there is no comparable publication so far on this topic. The building blocks of the system are described in Chapter 3.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 2

# Related Work

**Metamorphic testing for program analyzers.** Metamorphic testing is done in various forms and on different environments to ensure software quality. *Mansur et al. (2020)* introduced the STORM framework, which utilized blackbox mutational fuzzing to test SMT solvers by fuzzing input instances while keeping them satisfiable [26]. This is done by mutating existing seed instances with a fuzzer while guaranteeing, that the new SMT instance stays satisfiable. One year later, the same researchers apply metamorphic testing on Datalog programs to test the underlying engines [25]. They defined special relation types for the Datalog domain to enable the metamorphic testing approach.

**Metamorphic testing for other use cases.** *Chen et al. (2016)* introduced a different approach, where the target system is also tested with obfuscated code gathered via metamorphic transformations [11]. However, the novelty lies in comparing two obfuscated programs with each other instead of referring to the seed program. The technique of metamorphic testing is researched in various domains to benefit from the advantages [29]. *Chan et al. (2005)* utilizes this method to test Service-Oriented Architectures (SOA) and for supervised machine learning, *Xie et al. (2009)* publish an approach for testing classification algorithms building on metamorphic testing [7][32].

**Compiler testing.** The area of compiler testing is a wide field and is discussed for different languages and implementations [13][23][24][30]. Similar to metamorphic testing, *Equivalent Modulo Inputs* (EMI) is also an automated testing method. This technique is introduced by *Le et al. (2014)* and mutates programs to test compilers [22]. In particular, dead code is pruned from seed programs and a specific input to create a new test instance. Afterwards, both programs are compiled and executed with the respective input. If the results differ, a bug is found. *Kossatchev and Posypkin (2005)* looked into the different steps in compiler testing to ensure the quality thereof [21]. A summary of current testing approaches for compilers are also discussed in the paper



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

of *Chen et al. (2021)* [9]. To introduce even more diversity into the overall testing process, swarm testing is used [16]. A configuration determines the features which are applied in the test run. By automatically generating multiple configuration instances, a broader test coverage is reached. In terms of efficiency, *Chen et al. (2017)* propose a learning-to-test approach called LET, which uses the test instances of already found bugs to prioritize future test instances to find similar bugs [8]. A prominent example of a test-case generator for compiler testing is *Csmith* developed for C compilers [33]. The authors found more than 325 unknown bugs in open source compilers applying *Csmith*.

**Bug detection techniques.** Another technique for testing programs and finding bugs is graph abstraction, which is described by *Allen et al. (1976)* [3] and *Ferrante et al. (1987)* [14]. In general, this concept is well known and generally applicable for improving efficiency of opimizations or analizing programs. The tool *SSLint* proposed by *He et al. (2015)* utilizes a *Program Dependence Graph* (PDG) to find faulty usages of SSL APIs [19]. In the cryptographic domain, crypto-detectors are used to find misuses of APIs. These softwares are critical for avoiding security vulnerabilities which means that they need to be tested. *Ami et al. (2022)* introduce the *MASC* framework to introduce faulty usages of cryptographic APIs in Java programs to test detectors [4].

**Implementations of ZKPs.** For the calculation and verification of ZKPs, zk-SNARK is used [27]. It implements the witness generation, prover and verifier while keeping the proof small in terms of size (*succinctly*), distributable without the need of further communication (*non-interactively*), correct and infeasible to compute a fake proof (*argument of knowledge*) and without revealing any private information in the proof (*zero-knowledge*).

**Vulnerabilities within ZKPs Implementations.** While testing of the underlying compiler is important, *Chaliasos et al. (2024)* defined 3 categories of vulnerabilities, where only one of them is related to the computational part of the system [6]. The others are under-constrainted and over-constrainted programs in a way that fake proofs are possible or real proofs are not verified correctly. Additionally, interaction between the applied systems may also be faulty and introduce security risks. This emphasizes the fact that the testing of the ZKP infrastructure is necessary, but it does not replace the testing of other layers.

**Analysis of ZKP Implementations.** A tool for analyzing ZKP written with *Circom* is *Circomspect* [12]. It statically analyzes programs and flags all usage of the assignment operator without constraint <−− as dangerous. Instead, the constraint assignment operator <== is the safer option in most cases because it adds constraint assertions which fail on invalid states. This ensures, that no fake proofs are possible. However, by finding the occurrences, bugs are not found immediately. A review must be done to verify the correctness of the program.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 3

# Methods

In this chapter, the newly designed testing framework and each component for testing ZKP infrastructure is described. First, an overview of the structure is given, which shows the data flow within the system. Afterwards, the implementation of the metamorphic transformations and oracles are discussed, followed by the input and configuration generation. At the end, additional adaptations are mentioned to tackle various issues.

## 3.1 Testing Framework

To increase the diversity of the test cases, a number of components work together to generate new test instances, execute them against the target system and collect the results. Figure 3.1 gives an overview of the architecture with the order in which the components are called to contribute to the result. In each iteration, the next seed program is chosen from a fixed selection of programs depicted on the left top of the figure. This program $P_0$ is fed into the metamorphic transformer to generate new programs $P_{1 \cdots n}$ which deviate from the original one by the applied transformations. Additionally, for each program, input files $I_{11 \cdots 1m}$ to $I_{n1 \cdots nm}$ are generated alongside the transformations. The original program receives its own input files $I_{01 \cdots 0m}$. The settings for the transformations are configured by a generated test configuration, which is located on the top right part of the figure and fed into the metamorphic transformer and input fuzzer. In each run, this configuration is randomly created, which is described in more detail in the Configuration Fuzzer section. Afterwards, all created programs including the original program are compiled, executed and verified.

After running each test case, a number of details are collected. Most obviously, the success or failure of the test run is documented and logs are stored to enable backtracking of potential errors. Additionally, the execution time is measured for each step, namely the input generation, the generation of a new program from a seed program, the compile time of the ZKP compiler, the witness generation and the zk-SNARK execution. Before



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

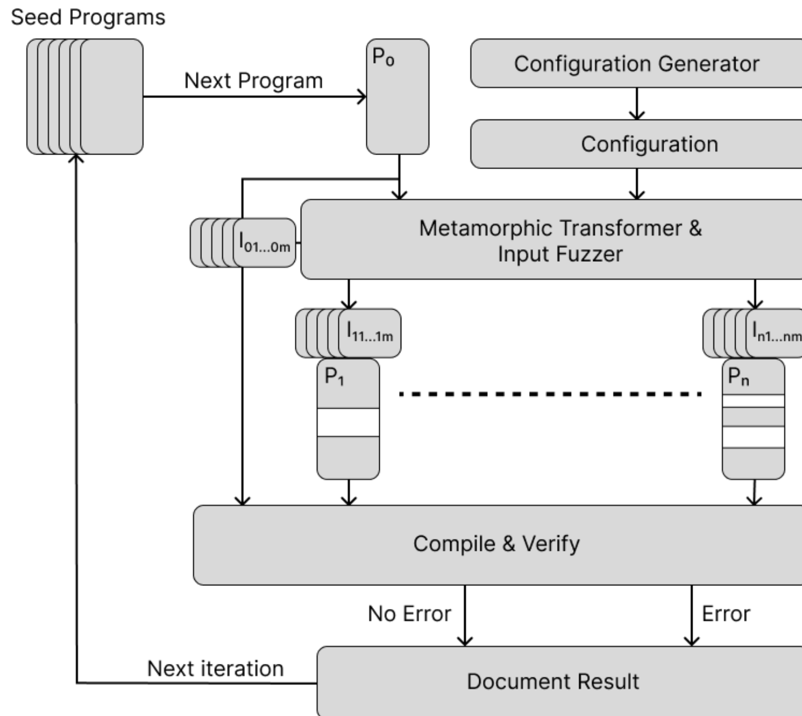


Figure 3.1: The proposed testing framework consists of the components shown in this figure. The seed programs get transformed into numerous test instances, which include the transformed program with an input file. Results are stored for evaluation.

the execution, the compiler is instrumented to enable the possibility of collecting coverage data during the test run. The data thereof is used to analyze the effectiveness of the tool and gives a hint on which regions need to be tested in more detail. After the execution, an aggregated report is generated where the number of successful programs and paths to errors are listed. Besides the success and error messages, a third category, namely warnings, are introduced which are configured to ignore certain error messages such as already found bugs or expected failure due to unsatisfied constraints or assertions within the tested programs. The collected results are further discussed in Chapter 4.

## 3.2 Metamorphic Transformations

The first step for more diversity in the test instances is the automated creation of new programs. As shown in the overview above, each iteration of the testing cycle starts by picking a new seed program from a fixed set of programs. In Figure 3.2, the left side shows an example of a seed program implementing a simple multiplication for the two input signals $a$ and $b$. The right-hand side is a transformation generated by the metamorphic



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

```
pragma circom 2.1.8;

template Multiplier2 () {
    signal input a;
    signal input b;
    signal output c;
    signal output d;

    c <== a * b;
    d <== a * b;
}

component main = Multiplier2();
```

```
pragma circom 2.1.8;

include "circomlib/comparators.circom";

template Multiplier2() {
    signal input a;
    signal input b;
    signal output c;

    c <== a * b;
    component eq1;
    eq1 = IsEqual_Copy();
    eq1.in[0] <== (a * b);
    eq1.in[1] <== <maxNumber>;
    d <== ((a * b) + (1 - eq1.out));
}

component main = Multiplier2();
```

Figure 3.2: This figure shows two programs where the left one is the seed program and the right side code is transformed with the *ConstraintAddModificationTemplate* transformation which has a *greater than or equal* output relation. $< maxNumber >$ hereby stands for the highest non-overflowing number configured.

transformer, which takes the seed program as an input. A component is used to check for a possible overflow when adding to the multiplication. Afterwards, the multiplication is extended to add one if no overflow occurs. In addition to the transformed program, an output relation file is generated, which is needed to verify the executions with the oracles described in the next section. Each output signal gets assigned one out of three possible relations, namely *equal*, *greater than or equal* and *less than or equal*. They describe the relation between the outputs of the original execution compared to the execution of the transformed program. In case of the example, the output relation *greater than or equal* is applied.

Depending on the configuration, different transformations are applied. In this paper, 19 unique operations are described and used for the evaluation in Chapter 4. Table 3.1 shows a list with a short description of each available transformation. To increase the diversity even more, chaining of transformations is possible. However, there are some exceptions to that. To ensure the output relation, in any given chain of operations, only either *greater than or equal* or *less than or equal* shall be applied. The *equal* relation is not effected by the restrictions and can be combined arbitrarily. By default, the *equal* relation does not overwrite any previous relations set in the chain. For example, if a signal $a$ already has the output relation *greater than or equal* and an *equal* operation is applied, the output relation for $a$ does not change.

To explain each transformation, a few terms need to be defined. Each *Circom* program comprises templates and functions. The difference between them is the allowed types which are embedded within the structure. Functions do not allow signals or constraints but only simple variables. Signals can be used for inputs, outputs or intermediate results which do only allow quadratic constraint assignments of the form $A * B + C$. This is the restriction of the ZKP concept. Constraints are assertions to make sure a given



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

| Metamorphic Transformations | | |
|---|---|---|
| **Name** | **Output Relation** | **Description** |
| *InputSignalOrder* | *equal* | Changes the order of the input signals in the main template (Incompatible with *AddInputSignals* and *DuplicateStatements*) |
| *Duplicate* | *equal* | Duplicates a template or function |
| *Log* | *equal* | Adds a log statement to the main template |
| *Assert* | *equal* | Adds an assert statement (tautology) to the main template |
| *DeadcodeIfFalse* | *equal* | Adds an if statement with an always false condition to the main template |
| *AddInputSignals* | *equal* | Adds new input signals to the main template (Incompatible with *InputSignalOrder*) |
| *ChangeInputSignalDimension* | *equal* | Changes input signal dimensions in the main template |
| *DuplicateStatements* | *equal* | Duplicates all statements in the main template and rename its variables (Incompatible with *InputSignalOrder*) |
| *LibCall* | *equal* | Adds a library call to the statements in the main template |
| *AddZero* | *equal* | Adds zero to an expression in the main template |
| *SubZero* | *equal* | Subtracts zero to an expression in the main template |
| *MulOne* | *equal* | Multiplies one to an expression in the main template |
| *DivOne* | *equal* | Divides by one with an expression in the main template |
| *ConstraintAddModification* | *greater than or equal* | Adds either 1 (or 0 for overflow values) to the last constraint in the main template |
| *ConstraintMulModification* | *greater than or equal* | Multiplies either by 2 (or 1 for overflow values) to the last constraint in the main template |
| *LoopConstraintAddModification* | *greater than or equal* | Adds either 1 (or 0 for overflow values) to the last constraint in the main template in a loop |
| *ConstraintSubModification* | *less than or equal* | Subtracts either 1 (or 0 for overflow values) from the last constraint in the main template |
| *ConstraintDivModification* | *less than or equal* | Divides the last constraint in the main template by a random number while removing the remainder beforehand to get an integer as a result |
| *LoopConstraintSubModification* | *less than or equal* | Subtracts either 1 (or 0 for overflow values) from last constraint in the main template in a loop |

Table 3.1: This table summarizes the metamorphic transformations with their respective output relation.

intermediate or output signal fulfills the defined quadratic expression. In contrast to signals, variables do not have these limitations, but are not used for template inputs or outputs. Templates allow a combination of signals and variables and provide their results via an output signal. Each program must have one main template, which is the defined starting point of the execution. During that, templates from other sources may be imported to call templates or functions of libraries to extend the functionality.

In the following paragraphs, we will discuss the transformations in more detail.

**InputSignalOrder.** This transformation switches the order of input signals in the main template by swapping two random signals. For signals $a$, $b$ and $c$ in this exact order, one valid transformation is the ordering $c$, $b$ and $a$. The results are not affected by the swapping and therefore, the output relation does not change for any signal.

**Duplicate.** When applying this transformation, a random template or function is copied and placed with a new unique identifier appended to its name. This new structure is never called and serves as a dead code extension for more variety in the test instance. The original template or function which is copied does not change in any way and therefore the output relations stay the same for all signals.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

**Log.** The transformation inserts a log statement into a random position of the main template. A random string is printed in the output console, which does not affect the output relations.

**Assert.** Similar to the *Log* transformation, *Assert* adds an assert statement with a tautology as argument. The execution must not stop at this statement and is not affected by it, resulting in the same output relations as before.

**DeadcodeIfFalse.** To test if-statements and their conditions, this transformation adds an if-statement with a contradiction as condition at a random position in the main template. As the body of the if-structure, a random number of statements from below the inserting position are copied. During the run of the program, the body of the if-statement must not be executed and therefore the output relations should not be affected. For the expressions within the condition, a random operator gets picked from the set [<=; >=; <; >; ==; ! =; ||; &&]. For the boolean *OR* (||) and *AND* (&&), two additional expressions are generated, which form a contradiction.

**AddInputSignals.** This transformation adds a new signal with a unique name to the main template. In addition to that, the input files for this newly generated program must be modified with a value for the new signal. While adding, nothing affects the original execution, which preserves the output relations.

**ChangeInputSignalDimension.** When applying this transformation, a random zero dimensional input signal (i.e. a non-array signal) is chosen and dimensions are added to it. To introduce more diversity while keeping the total array size reasonable for execution, 1 out of 3 possible adjustments gets picked. The first option adds a single dimension with a high size (i.e. up to 10000). In the second one, up to 3 dimensions are chosen while keeping each dimension at medium size (i.e. up to 100). The last option introduces up to 100 dimensions with each dimension only having a size of 1. Similar to the *AddInputSignals* transformation, the input files are adjusted by copying the original values of the zero dimensional signal to all array positions. Also, in all statements which use this input signal, the access needs to be adjusted as well. For that, all calls to the former zero dimensional signal gets replaced by a random access within the new array. If the former signal is called $a$ and 3 dimensions with size 10, 20 and 30 are added, a generated access may be $a[3][17][24]$. The described changes to the program should not affect the output signals in any way and the output relations stay the same.

**DuplicateStatements.** In contrast to *Duplicate*, *DuplicateStatements* does copy the statements within the main template and copies them below the original code within the same template. To preserve validity of the code, all names are renamed by appending a unique identifier. This operation also adds new input and output signals which need to be added within the input files and the output relations. All new outputs are referred



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

to the output signal they are copied from. This process is described in more detail in Section 3.3.

**LibCall.** Between the statements of the main template, 1 out of 4 templates picked from the *circomlib* libraries are called. This call includes the declaration of the template and setting the input signals thereof. The outputs are ignored, which results in no changes within the output signals from the main template and output relations.

**AddZero and SubZero.** Both transformations work similarly. They pick a random expression to add (*AddZero*) or subtract (*SubZero*) zero to it. The result of the expression should remain the same. Therefore, no changes to the output relations are applied.

**MulOne and DivOne.** When applying these transformations, a random expression is chosen and one is multiplied (*MulOne*) or the expression is divided by one (*DivOne*). Hereby, the results of the expression remain the same and no output relations change.

**ConstraintAddModification and ConstraintSubModification.** These two transformations change the output relations to *greater than or equal* or *less than or equal* respectively. Both pick the last constraint in the main template and add (*ConstraintAddModification*) or subtract (*ConstraintSubModification*) one from the expression. This is only done, if no over- or underflow would happen after the operation is applied. To detect this state, a separate mechanism is placed before the calculation. Figure 3.2 shows an example of a *ConstraintAddModification* transformation. The program is read and transformed on the output signal $d$. Before the assignment, a check for overflows is added, where the original value is checked against the maximum value. $eq.out$ returns 0 if no overflow occurs and 1 is added in the expression. Otherwise, 1 gets inserted for $eq.out$ and 0 is added to the expression. This results in an output relation of *greater than or equal* to the output signal $d$. All other untouched output signals get the default output relation *equal*.

**ConstraintMulModification and ConstraintDivModification.** Similar to the transformations in the previous paragraph, these two multiply (*ConstraintMulModification*) or divide (*ConstraintDivModification*) the last constraint in the main template. Again, an over- and underflow mechanism is placed before the constraint statement. The output signal which is assigned in the constraint either gets the output relation *greater than or equal* or *less than or equal*. On the left side of Figure 3.3, the *ConstraintMulModification* transformation is applied. The original calculation $(a * b)$ is assigned to a temporal signal $tmp1$ for calculating intermediate results. To ensure, that the multiplication by 2 at the end does not overflow, the next 4 lines check, if $tmp1$ is less than or equal to the maximum allowed number divided by 4. The divisor is picked such that the calculation within the template *LessEqThan_Copy* which is copied from the *circomlib* does not overflow on its own. This issue is discussed in more detail in Chapter



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

```
// before transformation

c <== (a * b);

// after transformation

signal tmp1;
tmp1 <== (a * b);
component lte3;
lte3 = LessEqThan_Copy(252);
lte3.in[0] <== tmp1;
lte3.in[1] <== (<maxNumber> / 4);
signal tmp2;
tmp2 <== ((2 * lte3.out) + (1 - lte3.out));
c <== (tmp1 * tmp2);
```

```
// before transformation

c <== (a * b);

// after transformation

var tmp2;
tmp2 = (a * b);
component lt1;
lt1 = LessEqThan_Copy(252);
lt1.in[0] <== tmp2;
lt1.in[1] <== 7656;
var i3;
i3 = 0;
while (i3 < 7656) {
    i3 = (i3 + 1);
    tmp2 = (tmp2 - (1 - lt1.out));
}
c <== tmp2;
```

Figure 3.3: On the left side of this figure, an example for the *ConstraintMulModification* transformation is shown. The right side is an example for the *LoopConstraintSubModification* transformation, where the number 7656 is randomly generated and placed to determine the number of iterations for the loop. Both of them only show the code which is affected. $< maxNumber >$ hereby stands for the highest non-overflowing number configured.

5. The last 3 lines sets the multiplication factor depending on the overflow check and assigns the multiplication to $c$.

**LoopConstraintAddModification and LoopConstraintSubModification.** Two transformations to iteratively add (*LoopConstraintAddModification*) or subtract (*LoopConstraintSubModification*) one from the last constraint in the main definition. Again, either the *greater than or equal* or *less than or equal* output relation is applied for the corresponding signal. A random number is generated to determine the number of iterations for the while loop. Inside the loop, a temporal variable which holds the original expression is increased or decreased by one in each iteration. After the loop, the temporal variable is assigned to the output signal. In advance, an over- or underflow check is placed. Hereby, the number of iterations for the loop is considered. If during any iteration the value would over- or underflow, nothing is added to the expression. Figure 3.3 shows an example of the *LoopConstraintSubModification* transformation on the right side. A temporal variable stores the original result and then an underflow check takes place to see, if $tmp2$ will be decreased below 0. In this example, the while loop gets executed 7656 times which is randomly generated by the metamorphic transformer. In each iteration of the loop, $tmp2$ is reduced by one if there is no underflow happening. At the end, $tmp2$ gets assigned to the output signal $c$ resulting in an output relation of *greater than or equal*.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

**Further considerations**

There are also incompatibilities between the transformation which shall not be chained together, namely *InputSignalOrder* which cannot be combined with *AddInputSignals* or *DuplicateStatement* and vice versa. This is due to the restrictions of the output file given by the actual system which is used for the evaluation. The file only stores the results of the signals in their order of declaration, without their respective names. When changing the order of the signals with a new introduced signal, the backtracking of signals is wrongly applied due to changes in the ordering.

To ease the process of program manipulation, the seed program is parsed and stored into an Abstract Syntax Tree (AST). This object is modified by the transformer and then written back to a new file.

## 3.3 Oracles

After each run of a test instance and its respective seed program, the results are compared with different oracles. These comparisons are done for each output signal of a program, where the output relations may vary for each of them. Alongside the mutated program, the transformer also produces an output relation file which maps the signals to their respective relations with the signals of the seed program.

During the verify-step, each output signal gets checked with one of the following oracles:

- **Equals Oracle** - Checks if the output signal of the transformed program equals the original one
- **Greater than or Equals Oracle** - Checks if the output signal of the transformed program is greater than the original one
- **Less than or Equals Oracle** - Checks if the output signal of the transformed program is less than the original one

For each check, a log is stored, which is included in the aggregated final report. There is also a special case where an output signal of the newly generated program has no respective signal in the seed program. This happens when a new signal get added to introduce more complexity for testing. In the respective output relation file, this special case is covered by referencing the newly introduced signal to an already existing one. For example, the *DuplicateStatements* transformation duplicates each signal and the logic of the seed program. When $a$ is the original output signal, the transformed program contains a copy of $a$ called $a'$ and a new copied output signal $b'$. In the output relation for $a'$ and $b'$ are *equal* to $a$. That way, a new relation is added, which can then be independently modified by the next step of the transformation chaining.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

## 3.4 Input Fuzzer

Another option to increase the variety of test data is by generation of different input values. The input fuzzer takes a seed program and creates an input file which fits the needs of the input signals. The allowed input signals for ZKPs are rather trivial and only allow unsigned numbers or arrays thereof, which may be arbitrarily nested. To increase the number of tests, multiple input files are generated, which all have different values for their input signals.

In addition to the features described in Section 3.2, the metamorphic transformer also copies the input files of the seed program for each generated program. If a transformation with input changes is applied, each input file is altered to fit the new program accordingly.

## 3.5 Configuration Fuzzer

The last component to describe is the configuration fuzzer which takes care of randomizing the settings of the system. The configuration is only generated at the beginning of the execution and configures the transformer and input fuzzer for all following transformations. In the following list, all configurable values are described:

- **Random program seed** - The starting seed as integer value for the metamorphic transformer when generating new programs. This seed's value is increased by one for each new generated program.
- **Random input seed** - The starting seed as integer value for the input fuzzer when generating new input files. This seed's value is increased by one for each new generated input file.
- **Number of transformations per seed program** - Defines the number of times one seed program is fed into the metamorphic transformer to generate new programs.
- **Number of inputs per seed program** - Defines the number of times one seed program is fed into the input fuzzer to generate new input files.
- **Maximum number of metamorphic transformation chaining** - Defines the maximum number of metamorphic transformations which is used during one execution of the metamorphic transformer. For each run of the transformer, the actual number is determined randomly, depending on the current random seed value.
- **Set of *equal* transformations** - A set of *equal* transformations considering the restrictions in Section 3.2.
- **Set of *greater than or equal* transformations** - A set of *greater than or equal* transformations which is empty, if any *less than or equal* transformation is chosen.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- **Set of *less than or equal* transformations** - A set of *less than or equal* transformations which is empty, if any *greater than or equal* transformation is chosen.

The option to automatically generate configurations enables the possibility of swarm testing to test with different settings [16]. Multiple instances of the testing framework run in parallel while only the configuration changes. This is the final measure to increase the diversity in the testing space.

## 3.6 Additional Adaptations

While the overall structure works fine, there are some extra adaptations to ensure a smooth execution.

In some cases, the seed programs need to be pre-processed for several reasons. Depending on the parser and write back, it may be necessary to rename variable names to a unique name. In the case of the actual implementation, for loops are transformed into while loops. The declaration of the iterating variable gets moved before the loop block, which may result in errors when writing back due to non-unique variable names. Also, some programs collected from the dataset are outdated and use invalid syntax. To test the newest version of the compiler, all seed programs need to be compatible with that exact version. Due to the simplicity of the input fuzzer, the seed program also needs to have constant input array sizes. The size of an input signal array can be calculated with an arbitrary expression or via a function. As a pre-processing step, these expressions need to be changed to a constant value to determine the size of the array which needs to be generated and placed in the input file.

Although the artifacts generated during the execution of the system may contain valuable information in case of a bug, the majority of the results are correct and there is no knowledge gained by them. All generated artifacts build up to a significant amount, resulting in unnecessarily filled disc space. As a countermeasure, a cleaning process takes place whenever a test case is successful. This way, only artifacts of faulty test instances are stored for review.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 4

# Results

The goal of the testing framework is to find bugs in the underlying infrastructure and provide a randomized testing mechanism which runs automatically on a regular basis. Besides finding bugs already present in the code base, regression testing is also possible when running with the same seed programs.

For the testing environment, the *Circom* [5] compiler with version 2.1.8 is used, which is the latest released version at the time of the execution. In our test runs we found 3 bugs where one of them was already reported by an independent individual. The remaining two were reported and one of them is already fixed and is published with the next release.

In this chapter, the dataset and evaluation of the results is presented.

## 4.1 Datasets

The dataset of seed programs is collected by querying *Circom* programs from *GitHub* repositories. The libraries imported in the seed programs are manually added, and the references are changed accordingly. For the final evaluation, a total of 109 seed programs are collected and evaluated.

During the evaluation, all seed programs are considered in each run and the results are collected from runs with different configurations.

## 4.2 Evaluation

For the evaluation, the proposed system is implemented in *Rust* to extend the *Circom* ecosystem, where the compiler is also written in this language. The test runs are performed on a server with 512GB of RAM and 2 AMD EPYC 7702 64-core processors. To get a reliable measure of time, the cores are limited per run and the number is pointed out in the experiments below.

In the implementation of the configuration fuzzer, boundaries for possible values are set to limit the runtime and overall complexity. In Table 4.1, the boundaries are shown.

### 4.2.1 Execution Time

In addition to the time it takes to run the tests, the overhead added by the coverage tooling is also measured. Therefore, the dataset is executed ten times. The first five runs utilize the code to collect the coverage data within the *Circom* compiler, while the remaining five runs do not add anything to the underlying infrastructure. All executions use 32 cores of the described hardware to make the results comparable. Table 4.2 summarizes the average results of all runs. In total, the execution without coverage took 1614,01m while with the coverage, 3145,27m are needed which results in an increase of 94,87%. The measurements are split up in 6 different steps, namely *Compile Step*, *Circuit Fuzzer*, *Input Fuzzer*, *Make Execution*, *SnarkJS Execution* and *Witness Generation*. All executions except the *Compile Step* have similar execution times with and without coverage, while the compilation takes longer with coverage to finish. On average, 138,95s are needed if the coverage is enabled while without it, only 3,91s are needed. The overhead for the instrumentation needed to collect the coverage is therefore significant and is turned off for all further measurements.

Another measure which is taken is the time until a bug occurs. For this reason, the 5 executions without the coverage data collection are analyzed. Figure 4.1 shows a diagram displaying all runs and the average time until bug in minutes. On average, the first bug

| **Configuration Fuzzer Boundaries** | |
|---|---|
| **Value Name** | **Boundaries** |
| Random program seed | Random 32-bit unsigned integer |
| Random input seed | Random 32-bit unsigned integer |
| Number of transformations per seed program | Fixed at 5 |
| Number of inputs per seed program | Fixed at 10 |
| Maximum number of metamorphic transformation chaining | Random value between (inclusive) 1 and (exclusive) 25 |
| Set of equal transformations | Contains any available *equal* transformation with a chance of 50% (minimum set size of 1) |
| Set of greater than or equal transformations | Contains any available *greater than or equal* transformation with a chance of 50% (minimum set size of 1 if *less than or equal* set has size 0) |
| Set of less than or equal transformations | Contains any available *less than or equal* transformation with a chance of 50% (minimum set size of 1 if *greater than or equal* set has size 0) |

Table 4.1: This table shows the boundaries for the values generated by the configuration fuzzer.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

| Execution Time | | |
|---|---|---|
| **Scope** | **Avg. Time with Coverage** | **Avg. Time without Coverage** |
| Compile Step | 138,95s | 3,91s |
| Circuit Fuzzer | 0,19s | 0,18s |
| Input Fuzzer | 8,98ms | 9,20ms |
| Make | 19,74s | 19,29s |
| SnarkJS | 38,81s | 37,50s |
| Witness Generation | 35,94ms | 35,02ms |
| **Total** | **3145,27m** | **1614,01m** |

Table 4.2: This table shows a summary of the execution time with different settings. Each one is an average value of 5 complete runs on the dataset, and all of them utilize 32 cpu cores.

is found after 569,87m. The second one takes 1214.27m to be found. Note, that only two bugs are found during the duration of these runs instead of the 3 bugs mentioned above. This is due to the fact, that the logging bug described further down in this chapter was discovered due to a bug in the developed tool, which happened to also reveal a bug in the compiler. Two of the five executions did only reveal one bug. This is due to the fact that the configuration for these runs didn't allow the bug to occur, which shows the importance of swarm testing in this setting.

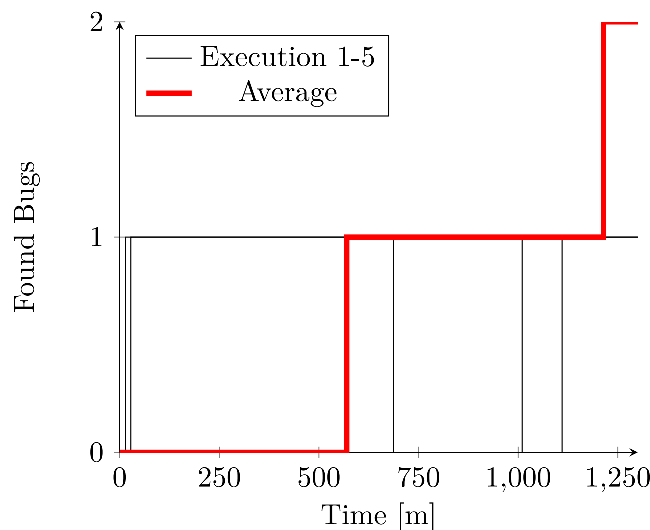


Figure 4.1: In this diagram, the time until a bug is found for each individual execution is printed.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

### 4.2.2 Coverage

In addition to the time, the coverage of the target compiler is measured, as briefly introduced in the previous paragraphs. To inspect and collect this coverage data, the *Circom* compiler is instrumented with code that writes the additional information into separate files for each execution.

We conducted a measurement on the entire dataset five times and generated a random configuration for each of them to see how much coverage the transformations add. Table 4.3 shows the coverage results where the coverage is measured in region, function and line coverage. The value for the coverage of only the seed programs, only the transformed programs and the total coverage are collected and the average value of all runs is computed and shown.

Overall, the coverage categories correlate with each other and therefore only the line coverage is discussed in detail. The coverage for seed programs is 32,95% while the coverage for the transformed programs is on average 32,29%. The fact that the coverage of the transformed programs is less comes from the implementation process of writing parsed programs back to the file. Some structures are simplified or replaced by others, which naturally reduces the coverage. The total coverage is at 33,02%. The gain by the transformations is 0,07% which is very low considering the amount of transformations in use. The reason for that is the difficulty on how to design metamorphic transformations introducing novelty into the programs while being generally applicable. This topic is also discussed in Chapter 5.

### 4.2.3 Regression Testing

The practical use case of regression testing is done by running the same 5 configurations on an older version of the *Circom* compiler. In our tests, we chose the version 2.1.0 and compared the results with the found bugs in the other runs. Upfront, there is one limitation which needed to be tackled, namely the version settings in the code. If the version in the file requires a newer release of the compiler, the compilation process fails immediately and prints out a corresponding message, which is the expected behavior. To counter this, all executions producing this message are ignored and cannot be evaluated.

As expected, the results also produce the already found bugs which are present in the newer version. However, these runs did not reveal any new errors. One partial reason

| Coverage | | | |
|---|---|---|---|
| | **Region Cov.** | **Function Cov.** | **Line Cov.** |
| Avg. seed program coverage | 41,04% | 40,29% | 32,95% |
| Avg. transformed coverage | 40,86% | 39,85% | 32,29% |
| **Avg. total coverage** | **41,18%** | **40,40%** | **33,02%** |

Table 4.3: This table shows the average results for the coverage on the entire dataset.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

for that is the trimmed pool of seed programs due to version incompatibilities. Another reason is the need for more coverage to really challenge as many paths in the compiler code as possible. Nevertheless, this can be repeated many times in all different versions to foster the regression testing process.

### 4.2.4 Found Bugs

During the entire testing phase and the evaluation of the system, a total of 3 bugs got revealed for the compiler version 2.1.8. 2 bugs are forwarded to the maintainers and the third one was already reported by an independent individual. One of them is already fixed at the time this paper is written and is released in the next version. In the following paragraphs, these bugs are discussed.

The first issue produces invalid artifacts when generating specific code with the *Circom* compiler. When introducing a line break into the code within a log statement string, this line break is copied into the CPP file, which is part of the artifacts produced in the compile step. As a result, an error message is printed when trying to work with the invalid artifact, but no error message is thrown during the compile step. Figure 4.2 shows an example of the logging bug and its message.

A second issue is reproduced when having a program with over 256 input signals. The compiler does not exit without an error and also does not shut down gracefully. The expected behavior would be to either preferably not depend on the number of input signals or to shut down gracefully.

```
pragma circom 2.1.8;

template Multiplier2 () {
   signal input a;
   signal input b;
   signal output c;

   log("abc
   abc");

   c <== a * b;
}

component main = Multiplier2();
```

```
bug.cpp:84:8: warning:
               missing terminating " character
   84 | printf("abc
      |        ^
bug.cpp:84:8: error:
               missing terminating " character
   84 | printf("abc
      |        ^~~~
bug.cpp:85:4: warning:
               missing terminating " character
   85 | abc");
      |    ^
bug.cpp:85:4: error:
               missing terminating " character
   85 | abc");
      |    ^~~
```

Figure 4.2: This figure shows an example how to reproduce the log bug and the corresponding error output when compiling the CPP artifacts produced by the *Circom* compiler.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The last found bug happens during witness generation. During the generation, an assertion within the CPP code fails. The error message printed by the compiler says:

> fr.cpp:166: void Fr_fail: Assertion ‘false’ failed.

A similar bug is already reported by an independent individual and fixed by the developers in *Circom* version 2.1.9. To validate, that this bug is the same as we found during our tests, we executed the program which revealed the bug with version 2.1.9 which is newly released at the time of evaluation of the bugs. The run succeeds, and the bug is therefore fixed.

In addition to the found bugs, in some cases, the transformation of seed programs resulted in large programs not allowed by the ceremony setup. These instances provide a way of testing this limit in a randomized way.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

# Discussion

While metamorphic testing is suitable for testing [25][29], the process for testing the *Circom* compiler has some extra challenges which need to be overcome. In the following sections, the limitations are described and a conclusion with an outlook for the future is given.

## 5.1 Limitations

While the proposed framework found bugs in the compiler, there are limitations which reduce the effectiveness and the potential of this solution. One of the biggest factors is time. While the transformation of a program is negligible, the compilation and execution process takes most of the time, as shown in Chapter 4. This limits the amount of test instances per time and therefore reduces the effectiveness.

Adding more transformations is also a way to increase the probability of finding bugs. However, it is hard to design transformations which are generally usable. Trivial ones are easier to implement, but do not contribute as much as more complex ones. While adding more seed programs helps with diversity, those programs are very likely to not be faulty because they are already tested by the owners which use them in their software.

As described before, the limitation of fuzzing the input files forces changes in the seed programs before sending them into the system. To tackle this, an expression evaluation is needed to allow arbitrary input signal dimensions. In addition to that, the input fuzzer operates in a purely random manner, meaning that there is no focus on edge cases. An additional analysis of the program would enable the generation of input values focusing on these special cases.

The parser used in the implementation of the testing framework simplifies the code before storing it in the AST. This causes the code which is written back after transformations to miss out on certain structures. Part of the lost information are comments and for-loops,



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

which are converted into while-loops. Therefore, these structures are only tested with the seed programs, while the transformed versions do not contain any comments or for-loops. This however also brings some novelty into the generated programs.

Configurations are also randomly generated and therefore special cases are unlikely to be covered. For example, applying only one transformation and chaining it to itself multiple times is possible, but statistically rare. Multiple fuzzer modes would bring the possibility of having specialized and purely random configurations for the system.

## 5.2 Conclusion

In this thesis, we proposed a new testing framework for ZKP infrastructure utilizing metamorphic testing and swarm testing. Moreover, metamorphic transformations are described and applied during execution. The results are collected by running the system multiple times with 109 seed programs and the transformations thereof. The coverage of the compiler code is measured and reached a line coverage of 33,02%. Additionally, the execution is tested with and without coverage instrumentation to show the overhead.

Overall, the testing framework enabled us to find 3 bugs, which is a success. However, the time-consuming executions and missing coverage limited the effectiveness of our tool. There is still work to be done in this domain, such as finding new complex metamorphic transformations and trying other configuration settings to force other edge cases. Also, the generation of inputs can be improved by allowing general expressions for array sizes. Besides metamorphic testing, the framework may also be extended by other similar forms of testing, such as EMI [22].

The bugs we found are forwarded to the developers of the *Circom* compiler and are partially fixed at the time of writing this thesis. The result also motivates a further look into this topic to continue the testing of this system by extensions of our tool or other similar tools. We are looking forward to seeing the automated testing techniques evolve further and increase their efficiency as well as the amount of use cases which they are applicable for.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

# Overview of Generative AI Tools Used

While writing this thesis, no generated text from AI tools is used. The text in this thesis is written by myself and only corrected by *LanguageTool* (https://languagetool.org/de) which provides feedback for grammar and spelling.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Übersicht verwendeter Hilfsmittel

Beim Schreiben dieser Arbeit wurde kein generierter Text von KI Tools verwendet. Der gesamte Text dieser Arbeit wurde von mir geschrieben und nur durch *LanguageTool* (https://languagetool.org/de) verbessert, welches Feedback zur Grammatik und zur Rechtschreibung gibt.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# List of Figures

1.1 Example for a ZKP communication. . . . . . . . . . . . . . . . . . . . . . . . 2
1.2 Example of a *Circom* program. . . . . . . . . . . . . . . . . . . . . . . . . 3
1.3 Example for metamorphic testing. . . . . . . . . . . . . . . . . . . . . . . . 4
1.4 Overview of the testing process. . . . . . . . . . . . . . . . . . . . . . . . . 5

3.1 Architecture of the proposed testing framework. . . . . . . . . . . . . . . . . 10
3.2 Example of the seed program and a transformed program. . . . . . . . . . . 11
3.3 Example of a *ConstraintMulModification* and *LoopConstraintSubModification* transformation. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 15

4.1 Diagram presenting the time until bugs are found. . . . . . . . . . . . . . . 21
4.2 Log bug and the error message when compiling the CPP artifacts. . . . . 23



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# List of Tables

3.1 List of the metamorphic transformations applied during the evaluation. . 12

4.1 List of configuration fuzzer boundaries. . . . . . . . . . . . . . . . . . . . 20
4.2 Summary of execution times. . . . . . . . . . . . . . . . . . . . . . . . . . 21
4.3 Average Coverage results. . . . . . . . . . . . . . . . . . . . . . . . . . . . 22

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

# Bibliography


[1] Lurk. https://lurk-lang.org/. accessed 04.04.2024.

[2] Noir. https://noir-lang.org/. accessed 04.04.2024.

[3] F. E. Allen and J. Cocke. A program data flow analysis procedure. In *Communications of the ACM*, volume 19(3), page 137.

[4] Amit Seal Ami, Nathan Cooper, Kaushal Kafle, Kevin Moran, Denys Poshyvanyk, and Adwait Nadkarni. Why crypto-detectors fail: A systematic evaluation of cryptographic misuse detection techniques. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 614–631. IEEE.

[5] Marta Bellés-Muñoz, Jordi Baylina, Vanesa Daza, and José L. Muñoz-Tapia. New privacy practices for blockchain software. In *IEEE Software*, volume 39(3), pages 43–49.

[6] Stefanos Chaliasos, Jens Ernstberger, David Theodore, David Wong, Mohammad Jahanara, and Benjamin Livshits. SoK: What don't we know? understanding security vulnerabilities in SNARKs. In *arXiv preprint arXiv:2402.15293*.

[7] W. K. Chan, Shing Chi Cheung, and Karl RPH Leung. Towards a metamorphic testing methodology for service-oriented software applications. In *Fifth International Conference on Quality Software (QSIC'05)*, pages 470–476. IEEE.

[8] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. Learning to prioritize test programs for compiler testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 700–711. IEEE.

[9] Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. In *ACM Computing Surveys*, volume 53(1), pages 1–36.

[10] Tsong Y. Chen, Shing C. Cheung, and Shiu Ming Yiu. Metamorphic testing: a new approach for generating next test cases. In *arXiv preprint arXiv:2002.12543*.




Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[11] Tsong Yueh Chen, Fei-Ching Kuo, Wenjuan Ma, Willy Susilo, Dave Towey, Jeffrey Voas, and Zhi Quan Zhou. Metamorphic testing for cybersecurity. In *Computer*, volume 49(6), pages 48–55.

[12] Fredrik Dahlgren. It pays to be circomspect. https://blog.trailofbits.com/2022/09/15/it-pays-to-be-circomspect/. accessed 07.04.2024.

[13] Maulik A. Dave. Compiler verification: a bibliography. In *ACM SIGSOFT Software Engineering Notes*, volume 28(6), page 2.

[14] Jeanne Ferrante, Karl J. Ottenstein, and Joe D. Warren. The program dependence graph and its use in optimization. In *ACM Transactions on Programming Languages and Systems*, volume 9(3), pages 319–349.

[15] Uriel Fiege, Amos Fiat, and Adi Shamir. Zero knowledge proofs of identity. In *Proceedings of the nineteenth annual ACM conference on Theory of computing - STOC '87*, pages 210–217. ACM Press.

[16] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 78–88. ACM.

[17] Ashley A. Hall and Carol S. Wright. Data security: A review of major security breaches between 2014 and 2018. In *Federation of Business Disciplines Journal*, volume 6, pages 50–63.

[18] Jahid Hasan. Overview and applications of zero knowledge proof (ZKP). In *International Journal of Computer Science and Network*, volume 8(5).

[19] Boyuan He, Vaibhav Rastogi, Yinzhi Cao, Yan Chen, V. N. Venkatakrishnan, Runqing Yang, and Zhenrui Zhang. Vetting SSL usage in applications with SSLINT. In *2015 IEEE Symposium on Security and Privacy*, pages 519–534. IEEE.

[20] Max Kobelt, Michael Sober, and Stefan Schulte. A benchmark for different implementations of zero-knowledge proof systems. In *2023 IEEE International Conference on Blockchain*, pages 33–40.

[21] A. S. Kossatchev and M. A. Posypkin. Survey of compiler testing methods. In *Programming and Computer Software*, volume 31(1), pages 10–19.

[22] Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 216–226. ACM.

[23] Xavier Leroy. Formal verification of a realistic compiler. In *Communications of the ACM*, volume 52(7), pages 107–115.



Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[24] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In *ACM SIGPLAN Notices*, volume 50(6), pages 65–76.

[25] Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. Metamorphic testing of datalog engines. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 639–650. ACM.

[26] Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 701–712. ACM.

[27] Maksym Petkus. Why and how zk-SNARK works. In *arXiv preprint arXiv:1906.07221.*

[28] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V. Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42. IEEE.

[29] Sergio Segura, Gordon Fraser, Ana B. Sanchez, and Antonio Ruiz-Cortés. A survey on metamorphic testing. In *IEEE Transactions on software engineering*, volume 42(9), pages 805–824.

[30] Chengnian Sun, Vu Le, and Zhendong Su. Finding and analyzing compiler warning defects. In *Proceedings of the 38th International Conference on Software Engineering*, pages 203–213. ACM.

[31] Hongbo Wen, Jon Stephens, Yanju Chen, Kostas Ferles, Shankara Pailoor, Kyle Charbonnet, Isil Dillig, and Yu Feng. Practical security analysis of zero-knowledge proof circuits. In *Cryptology ePrint Archive, Paper 2023/190.*

[32] Xiaoyuan Xie, Joshua Ho, Christian Murphy, Gail Kaiser, Baowen Xu, and Tsong Yueh Chen. Application of metamorphic testing to supervised classifiers. In *2009 Ninth International Conference on Quality Software*, pages 135–144. IEEE.

[33] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in c compilers. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 283–294. ACM.



TU Bibliothek Your knowledge hub WIEN
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.