Hybrid Differential Software Testing

Dissertation

zur Erlangung des akademischen Grades

doctor rerum naturalium (Dr. rer. nat.)
im Fach Informatik

eingereicht an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Humboldt-Universität zu Berlin

von
Yannic Noller, M.Sc.

Gutachter/innen:	1.	Prof. Dr. Lars Grunske
	2.	Prof. Dr. Gordon Fraser
	3.	Prof. Dr. Corina S. Păsăreanu

Tag der mündlichen Prüfung: 18. September 2020

Yannic Noller:
Hybrid Differential Software Testing
June 2020

ABSTRACT

Differential software testing is important for software quality assurance as it aims to automatically generate test inputs that reveal behavioral differences in software. The concrete analysis procedure depends on the targeted result: differential testing can reveal divergences between two execution paths (1) of different program versions or (2) within the same program. The first analysis type would execute different program versions with the same input, while the second type would execute the same program with different inputs. Therefore, detecting regression bugs in software evolution, analyzing side-channels in programs, maximizing the execution cost of a program over multiple executions, and evaluating the robustness of neural networks are instances of differential software analysis with the goal to generate diverging executions of program paths.

The key challenge of differential software testing is to simultaneously reason about multiple program paths, often across program variants, in an efficient way. Existing work in differential testing is often not (specifically) directed to reveal a different behavior or is limited to a subset of the search space.

This PhD thesis proposes the concept of Hybrid Differential Software Testing (HyDiff) as a hybrid analysis technique to generate difference revealing inputs. HyDiff consists of two components that operate in a parallel setup: (1) a search-based technique that inexpensively generates inputs and (2) a systematic exploration technique to also exercise deeper program behaviors. HyDiff's search-based component uses differential fuzzing directed by differential heuristics. HyDiff's systematic exploration component is based on differential dynamic symbolic execution that allows to incorporate concrete inputs in its analysis.

HyDiff is evaluated experimentally with applications specific for differential testing. The results show that HyDiff is effective in all considered categories and outperforms its components in isolation.

ZUSAMMENFASSUNG

Differentielles Testen ist ein wichtiger Bestandteil der Qualitätssicherung von Software, mit dem Ziel Testeingaben zu generieren, die Unterschiede im Verhalten der Software deutlich machen. Solche Unterschiede können zwischen zwei Ausführungspfaden (1) in unterschiedlichen Programmversionen, aber auch (2) im selben Programm auftreten. In dem ersten Fall werden unterschiedliche Programmversionen mit der gleichen Eingabe untersucht, während bei dem zweiten Fall das gleiche Programm mit unterschiedlichen Eingaben analysiert wird. Die Regressionsanalyse, die Side-Channel Analyse, das Maximieren der Ausführungskosten eines Programms und die Robustheitsanalyse von Neuralen Netzwerken sind typische Beispiele für differentielle Softwareanalysen.

Eine besondere Herausforderung liegt in der effizienten Analyse von mehreren Programmpfaden (auch über mehrere Programmvarianten hinweg). Die existierenden Ansätze sind dabei meist nicht (spezifisch) dafür konstruiert, unterschiedliches Verhalten präzise hervorzurufen oder sind auf einen Teil des Suchraums limitiert.

Diese Arbeit führt das Konzept des hybriden differentiellen Software Testens (HyDiff) ein: eine hybride Analysetechnik für die Generierung von Eingaben zur Erkennung von semantischen Unterschieden in Software. HyDiff besteht aus zwei parallel laufenden Komponenten: (1) einem such-basierten Ansatz, der effizient Eingaben generiert und (2) einer systematischen Analyse, die auch komplexes Programmverhalten erreichen kann. Die such-basierte Komponente verwendet Fuzzing geleitet durch differentielle Heuristiken. Die systematische Analyse basiert auf Dynamic Symbolic Execution, das konkrete Eingaben bei der Analyse integrieren kann.

HyDiff wird anhand mehrerer Experimente evaluiert, die in spezifischen Anwendungen im Bereich des differentiellen Testens ausgeführt werden. Die Resultate zeigen eine effektive Generierung von Testeingaben durch HyDiff, wobei es sich signifikant besser als die einzelnen Komponenten verhält.

ACKNOWLEDGMENTS

First of all, I need to say thank you to my amazing wife **Vanessa Noller**, who always supported and motivated me in my work. Other big thanks to my parents **Birgit Noller** and **Roland Noller** because without them this thesis would not have been possible. Thank you for your help and your love!

Thank you to my adviser **Lars Grunske**, who supported me, gave me so much flexibility, and constantly sharpened my scientific skills. A special thanks to **Corina Păsăreanu**, who facilitated my research visits in California, which significantly influenced my PhD work and the rest of my scientific life.

Thank you as well to **Simon Heiden** for sharing the office with me and making this long time so much better, joyful, and entertaining! Other special thanks to **Birgit Heene**, who made all organizations around my PhD work so much easier, who always had a helpful advice, and a good cup of coffee.

Finally thank you to all my family, friends, colleagues, and collaborators, who constantly supported me (wittingly and unwittingly) in persuading my doctoral degree. Sometimes it was some motivating words, some probing asking about the progress, some fruitful discussions, or sometimes it was just a smile ☺ which helped me to continue the path.

Thank you to all of you (in alphabetical order): Samira Akili, Maike Basmer, Marcel Böhme, Stefan Boldt, Alexander Boll, Marc Carwehl, Jesus Christ, Zhen Dong, Stephan Druskat, Stephan Fahrenkrog-Petersen, Antonio Filieri, Aymeric Fromherz, Martin Gasch, Sinem Getir, Divya Gopinath, Timo Gruber, Herta Guhr, Isabell Guhr, Jörg Guhr, Gerda Hanson, Jim Hanson, Gregor Häußermann, Silvan Häußermann, Tobias Häußermann, René Heideklang, Lanae Isaacson, Pascal Jochmann, Timo Kehrer, Rody Kersten, Kurt Kielwein, Waltraud Kielwein, Edmund Klöpfer, Xuan-Bach D. Le, Ritchie Lee, Kasper Luckow, Kaushik Madala, Patrick Mauritz, Sergey Mechtaev, Paul Metzger, Fabian Müller, Sebastian Müller, Hoang Lam Nguyen, Manh-Dung Nguyen, Philipp Niethammer, Shirin Nilizadeh, Filippa Noller, Lilly Noller, Nadine Noller, Ada Păsăreanu, Esteban Pavese, Quoc-Sang Phan, Arut Prakash Kaleeswaran, Robert Prüfer, Wolfgang Reisig, Marius Riegraf, Abhik Roychoudhury, Sascha Russer, Sonja Russer, Stefan Russer, Jonas Scheurich, Birgit Schiefner, Lukas Schmidt, Arno Schneider, Bernd Schuler, Carolin Schuler, Heike Schuler, Sophie Schuler, Alexander Schultheiß, Vaibhav Sharma, Jan Arne Sparka, Jörg Stanuschewski, Youcheng Sun, Minxing Tang, Sebastian Tewes, Matthias Tichy, Saeid Tizpaz-Niari, Marvin Triebel, Han van der Aa, André van Hoorn, Willem Visser, Thomas Vogel, Kim Völlinger, Duc Anh Vu, Sebastian Weichbrodt, Matthias Weidlich, Katharina Weihbrecht, Junye Wen, Sven Wockel, Nick Würdemann, Guowei Yang, Bo Zhao, and Sebastian Zillessen.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LISTINGS

1 INTRODUCTION

Software engineering is the ”systematic application of scientific and technological knowledge, methods, and experience to the design, implementation, testing, and documentation of software”, as defined by the IEEE [212]. A key insight in the community is that "programming-in-the-large" [210] is much more complex than "programming-in-the-small", and hence, requires a proper *engineering* approach [215]. Early works in software engineering research urge for appropriate programming languages [210] as well as tools and methods [209]. The objective of software engineering *research* has to be the study of how to develop software (in the large), in order to provide the scientific knowledge and methods to do so. It searches for techniques to improve, simplify, and support software development. A crucial part of software development is *software quality assurance* facilitated by *software testing* [212], as it searches for *errors* in software. Therefore, software testing also plays an important role in software engineering research [11]. A special area in this research field is *differential software testing*, which aims to identify *behavioral differences* in software.

TERMINOLOGY – BEHAVIORAL DIFFERENCE

Behavioral difference means a difference in the execution behavior of a program. Such a difference can be represented by several forms, e.g., a difference in the direct output of a program, a difference in the execution time, or also a difference in the covered code fragments. ***Differential program analysis*** (here also called *differential software testing*) means the analysis of one or multiple programs in order to reveal behavioral differences.

Such a testing technique is usually applied in *software maintenance* to perform regression testing [85, 92, 93, 96, 97, 101], where the goal is to reveal differences between two successive software versions. Software maintenance is an important application area of software testing. Software does evolve with the reality, in which it is deployed [213, 215]. Therefore, new and modified functionalities need to be verified with regard to various quality attributes. Other flavors of differential program analysis are important for areas like *software security* (e.g., worst-case complexity analysis [131, 145, 146] or side-channel analysis [104, 110, 121]) and *software reliability* (e.g., robustness analysis of neural networks [157, 181, 184, 187]). This thesis focuses on differential program analysis of software and how such a testing approach can be performed efficiently. This chapter continues with the *motivation* for differential software testing, the *research problem*, an overview of the *state of the art*, and current *limitations*. Finally, this chapter provides an overview of the *research methodology* and a summary of the *contributions* made by this thesis.

1.1 MOTIVATION

Software testing is an essential part of software development, which is widely applied by practitioners and is also in the focus of numerous research projects [11]. The goal is to provide confidence in the correctness of software by searching for errors in its behavior. The manual creation of test cases can be very expensive and time consuming [99]. Therefore, recent research has focused on *automated test input generation* [54, 55, 7, 9, 62, 11]. Identified errors can be investigated and fixed, and so, software testing contributes to the quality of the software. Differential software testing specifically investigates behavioral differences and comes in various flavors.

For instance, automated *regression test generation* aims to generate inputs that trigger divergences between two successive software versions [85, 87, 92, 93, 96, 97, 101, 103]. Such differences can be observed, e.g., along the control-flow or in the actual output of the execution. Changes in software can interfere with the existing functionality, and hence, can have *unintended* side-effects [99, 103], also known as *regression errors*. This includes changes that intend to fix a prior error, but introduced another or have been incomplete, also known as *faulty bug fixes* [86, 94, 98]. Having efficient techniques for regression testing is crucial for testing evolving software [87].

In the area of software security, differential analysis can be used for automated *vulnerability detection*, e.g., with regard to *worst-case execution paths* or *side-channels*. *Algorithmic complexity vulnerabilities* can be exploited to cause a denial of service attack [131, 140, 145, 146]. The research field for algorithmic complexity analysis is very active and many serious attacks have been demonstrated, e.g., attacks based on vulnerable hash table implementations [134] as well as on the Linux kernel [132, 152]. Other examples are attacks on network security systems [148, 149] and attacks on string searching algorithms [151]. Besides the vulnerability detection, the characterization of the algorithmic complexity of a program helps to reason about performance bottlenecks [133] and to identify and design compiler optimizations [141]. Another type of security attack involves the exploitation of side-channels. Side-channel vulnerabilities have the potential to reveal sensitive information during program execution: A potential attacker could observe the non-functional characteristics of the program behavior, such as the execution time, memory consumption, response size, network traffic, or power consumption, to infer secret information. The related work on side-channel analysis shows evidence that side-channel attacks are practical and can have serious security consequences [108, 109, 113, 119]. For example, some early work on timing attacks identifies vulnerable implementation of cryptography algorithms like Diffie-Hellman, RSA, and DSS [116]. More recently, exploitable timing side-channels have been discovered for Google's Keyczar Library [126], the Xbox 360 [127], and implementations of RSA encryption [108, 109]. Other popular examples are the *Meltdown* [118, 125] and *Spectre* [115] side-channel attacks, which show how to exploit critical vulnerabilities in modern processors to uncover secret information. The above mentioned security vulnerabilities show that there is an increased need for techniques to efficiently discover such vulnerabilities, so that they can be mitigated or resolved.

A novel application of differential analysis is the robustness analysis of neural networks for input classification. This kind of analysis aims to identify two inputs that differ only very slightly, i.e., for a human almost imperceptible, but for which the neural network produces a different output (i.e., it results in different classifications) [161, 185]. The differences between such inputs are also called *adversarial perturbations* and represent major safety and security issues. For example neural networks are applied in autonomous cars for the identification of street signs or other traffic participants, where perturbations are easily possible, e.g., by changes in the lighting conditions or fog [186]. Adversarial perturbations in such an environment mitigate the software's *reliability*, and hence, can have serious consequences for the *safety* of the involved traffic participants [181, 184], as it could be observed in recent accidents [190, 194]. Neural networks are also used in the area of software *security*, where adversary inputs can be exploited to bypass security measures [180]. Therefore, it is crucial to have efficient techniques to systematically test neural networks.

1.2 RESEARCH PROBLEM

This thesis addresses the problem of *differential* software testing: the automated generation of test inputs that reveal behavioral differences in software. As discussed in the previous section, differential software testing is an active research area with numerous applications. The search for behavioral differences can be separated into two categories: It can reveal divergences between two execution paths (1) of different program versions or (2) within the same program. Figure 1 illustrates these two types of differential analysis. Category

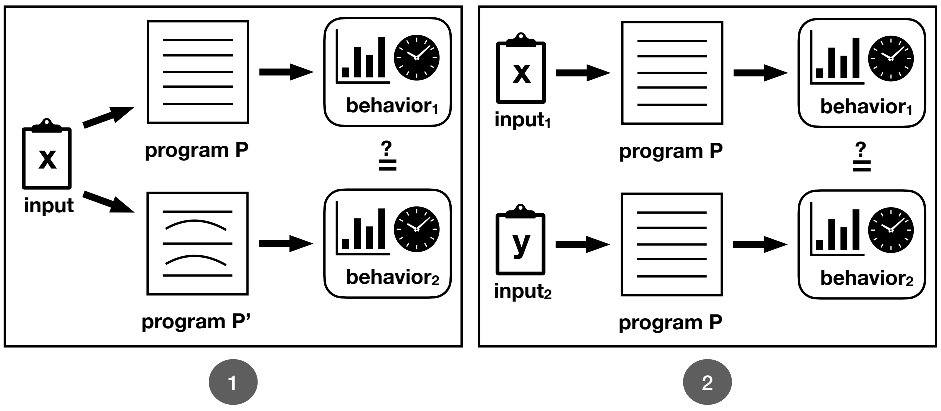


Figure 1: Two categories of differential software testing.

(1) (on the left side of Figure 1) searches for input x that leads to a different execution behavior between program P and its successive variant P'. Note that generally P and P' can be completely different programs. However, this thesis is focusing here on regression testing, which assumes that there are only *some* changes between P and P'. Category (2) (on the right side of Figure 1) searches for two different inputs x and y that lead to a different execution behavior for (the same) program P. Depending on the application it can be interesting how similar the two inputs are.

Both analysis categories require multiple program executions, which makes differential software testing a *challenging* problem. This research problem is investigated in the context of the mentioned applications:

A1 **Regression analysis** – The search for behavioral differences with the same input in successive program versions.

A2 **Worst-case complexity analysis** – The search for worst-case triggering inputs that perform significantly *different* than the average case.

A3 **Side-channel analysis** – The search for side-channel vulnerabilities in security-critical applications, which involves analyzing correlations between resource usage over multiple program paths.

A4 **Robustness analysis of neural networks** – The search for adversarial behaviors in neural networks, which requires reasoning about multiple network executions.

1.3 EXISTING APPROACHES AND LIMITATIONS IN A NUTSHELL

Regression analysis. Many techniques for regression analysis are based on symbolic execution [85, 96, 97, 74, 101, 102] in order to perform a systematic exploration and are directed to execute updated areas in the new program version. These techniques can generate test inputs that execute updated program areas, but they are not designed to also exercise the changed program behavior because they do not include the old program version in their analysis. Therefore, the results can include false negatives. Shadow symbolic execution [93] attempts to solve this problem by executing both program versions simultaneously. However, it might miss divergences due to its strong focus on the concrete inputs of an existing test suite. Search-based techniques like BERT [92], DiffGen [100], and EvoSuiteR [99] are either not specifically directed to expose regression errors or are limited because they cannot solve specific constraints in the program.

Worst-case complexity analysis. The related work on worst-case complexity analysis has a strong focus on static analysis [131, 145], which requires an exhaustive symbolic execution and might not be feasible in practice. Fuzzing based techniques [143, 146, 153] apply random mutations, and hence, might not be able to reach deep program behavior [81].

Side-channel analysis. Work on side-channel analysis covers techniques like static analysis [104, 110], abstract interpretation [112, 117, 120], and symbolic exploration [121], which have the general problem to miss vulnerabilities due to unrealistic models and assumptions, and have scalability issues. Other approaches rely on Monte Carlo sampling [111, 114] to quantify the information leakage, but might lead to imprecise results in practice.

Robustness analysis of neural networks. Many existing testing techniques for neural networks are guided by quantitative coverage metrics [170, 174, 178, 184, 196], and hence, are not suited for a differential exploration or the generation of adversarial inputs. Other techniques focus on differential testing of different neural networks [181, 197], which can lead to the identification of adversarial inputs. However this is not the intended purpose. Adversarial input detection at runtime [189] or defensive mechanisms [180] can help to mitigate the consequences of adversarial attacks, but do not provide an actual differential analysis of neural networks. There are promising approaches that try to generate inputs that trigger miss-classifications by applying hardly perceptible perturbations to the input [165], but they rely on expert knowledge of the erroneous root cause (e.g., blurry images, fog, or rain) [186], the solving of hard optimization problems [158, 161, 171, 176, 185], or expensive matrix manipulations [179]. So far there are only a few works that apply classic program analysis techniques to neural networks [163, 164].

CONCLUSION – RELATED WORK

Existing techniques in the various areas of differential testing come with their own disadvantages: Many of them are not directed to differential behavior, are not able to solve necessary constraints to reach deep program behavior, or rely on exhaustive exploration. They are limited in the effectiveness of their analysis. An efficient and effective testing approach asks for a hybrid execution setup [75, 146]. Therefore, there is a need for a hybrid differential testing technique.

1.4 OVERVIEW OF THE RESEARCH METHOD

This thesis proposes the concept of **Hy***brid* **Diff***erential Software Testing* (HyDiff), which combines search-based testing with a systematic exploration technique. More specifically, this concept aims to combine the *speed* of search-based fuzzing [81] and the systematic exploration of symbolic execution in a *parallel* setup. Both components perform their own *differential* analysis, while they exchange (interesting) inputs to support each other. This provides a generally applicable, differential software testing approach, which in particular can be applied to the mentioned application scenarios.

The evaluation of HyDiff investigates whether it can reveal behavioral differences in software and how the hybrid combination performs in contrast to its components in isolation. HyDiff is evaluated based on a quantitative analysis with benchmarks taken from the four application scenarios: (A1) regression analysis, (A2) worst-case complexity analysis, (A3) side-channel analysis, and (A4) robustness analysis of neural networks.

1.5 CONTRIBUTIONS

In summary, the core contributions made by this thesis are:

C1 the concept of differential fuzzing (e.g., to reveal side-channels); note that the testing for output inconsistencies between two different implementations of the same thing (e.g., two implementations of a PDF viewer) has been also denoted as differential testing / fuzzing [34], but represents a different type of analysis

C2 the concept of differential dynamic symbolic execution, as a technique to perform a symbolic exploration driven by differential heuristics

C3 the concept of hybrid analysis in differential program analysis

C4 the concept of a hybrid setup for applying fuzzing and symbolic execution in parallel (as an alternative to the execution in sequence [77, 81, 82])

1.6 THESIS OUTLINE

This thesis first introduces the *foundations* of this work (Chapter 2) followed by the *research method* (Chapter 3). The following three chapters represent the core of this thesis and describe hybrid differential software testing and its components in detail: *differential fuzzing* (Chapter 4), *differential dynamic symbolic execution* (Chapter 5), and their *hybrid combination* (Chapter 6). Afterwards the presented approach is *evaluated* in four application areas of differential software testing (Chapter 7). The thesis ends with a *conclusion* about the conducted research and its results (Chapter 8). The following paragraph describes each chapter in more detail:

***Chapter* 2 - Foundations** describes existing work on which HyDiff is based on. This chapter introduces fuzzing, symbolic execution, and hybrid testing approaches first and then explains the existing approaches for a hybrid program analysis. Afterwards this chapter discusses the related work on differential program analysis in the areas of regression analysis, worst-case complexity analysis, side-channel analysis, and the robustness analysis of neural networks.

***Chapter* 3 - Research Method & Contribution** presents hybrid differential software testing as the combination of differential fuzzing and differential dynamic symbolic execution.

This chapter also presents the investigated research goal and its research questions. Afterwards the chapter discusses the chosen evaluation strategy and how it aligns with the evaluations of the related work.

Chapter 4 - **Differential Fuzzing** introduces the concept and technical details of differential fuzzing. The chapter shows how fuzzing can be guided to perform a differential analysis and illustrates the concept with an example taken from the area of regression testing. This chapter concludes with a preliminary evaluation of the method by comparing it with coverage-guided fuzzing.

Chapter 5 - **Differential Dynamic Symbolic Execution** introduces the concept and technical details of differential dynamic symbolic execution. It explains its differential symbolic exploration strategy and discusses the central data structure in detail. This chapter illustrates the concept with the same example as for differential fuzzing and concludes with a preliminary evaluation by comparing it with coverage-based symbolic execution.

Chapter 6 - **Hybrid Differential Analysis** combines the previously described techniques to a hybrid differential analysis, in which both components are executed in parallel. This chapter provides the details of the hybrid setup and illustrates the concept with an example, which shows how the strength of both components can be combined to overcome their individual challenges.

Chapter 7 - **Validation** presents an extensive evaluation of the presented approach in four application areas of differential software testing: regression analysis, worst-case complexity analysis, side-channel analysis, and the robustness analysis of neural networks. For each type of differential analysis this chapter provides examples and details of how the proposed hybrid concept can be applied. It discusses the obtained results and answers the research questions formulated in Chapter 3. Furthermore, the chapter discusses the threats to validity of the conducted research.

Chapter 8 - **Conclusion** summarizes the contributions and discusses the impact of the conducted research as well as future work in the area of hybrid differential software testing and the applied test input generation techniques.

2 FOUNDATIONS

This chapter discusses the background of the existing work in the area of *fuzzing*, *symbolic execution*, and *hybrid analysis*. Furthermore, it discusses the related work on the applications of differential analysis: regression analysis, worst-case complexity analysis, side-channel analysis, and robustness analysis of neural networks. This chapter summarizes the related work and highlights their limitations. The content is partially based on the background and related work sections in the published works of Badger [3], DifFuzz [1], HyDiff [6], Shadow$_{\text{JPF}}$ [4], and Shadow$_{\text{JPF+}}$ [5].

2.1 FUZZING

The term fuzzing was coined by Miller et al. [30] in their work on reliability testing of UNIX utilities. Their goal was to crash the utilities by using random generated input strings. They have been able to crash 24% of the tested subjects, which has been unexpected for the authors because they had the intuition that since the kernel and utility programs are used frequently, they should be well tested. They presented their approach as a complement for existing testing and verification techniques, to generate inputs in an inexpensive way that reveal failures (i.e., crashes or hangs). Furthermore, they already proposed the application of fuzzing to identify security holes.

2.1.1 *Blackbox, Greybox, and Whitebox Fuzzing*

Nowadays numerous fuzzing techniques have evolved, which can be classified based on how much application knowledge is used during the testing process. The fuzzing community distinguishes *blackbox* fuzzing, *greybox* fuzzing, and *whitebox* fuzzing [17].

Blackbox Fuzzing. *Blackbox* fuzzing techniques cannot use the source code for their analysis, and hence, apply random testing on the program. Popular representatives are the Peach fuzzer [46] and Randoop [31]. Peach is a commercial fuzzing framework for, e.g., the automated testing of device drivers. Randoop randomly generates sequences of method calls for the testing of Java programs. It uses the observation of the program output to avoid redundant and illegal inputs.

Whitebox Fuzzing. *Whitebox* fuzzing techniques can use the complete source code for their analysis, i.e., they can apply a wide range of program analysis techniques to guide the input generation process. A popular representative is Sage (Scalable Automated Guided Execution) [21] as whitebox fuzzing technique for security testing. Sage uses dynamic symbolic execution to collect the constraints for a concrete input. These constraints are systematically negated and solved with a constraint solver in order to generate new inputs. Their directed approach to generate new constraints by negating all conditions in a path constraint is called *generational search* and maximizes the number of newly generated inputs with one symbolic execution run. Sage gets applied at Microsoft [21, 44] with a remarkable impact. Other popular whitebox (symbolic execution based) fuzzing approaches are KLEE [54] and S2E [18], which systematically traverse the state space (also see the following

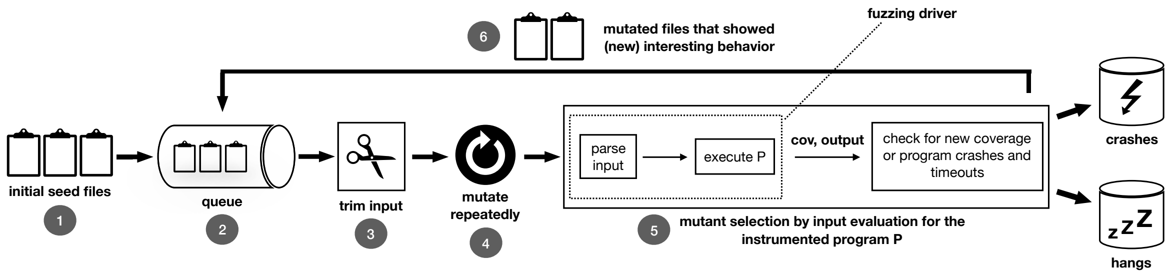

6
mutated files that showed (new) interesting behavior
fuzzing driver
initial seed files
1
queue
2
trim input
3
mutate repeatedly
4
parse input
execute P
cov, output
check for new coverage or program crashes and timeouts
5
mutant selection by input evaluation for the instrumented program P
crashes
hangs

Figure 2: Conceptual overview of coverage-based, mutational fuzzing as performed by AFL [38].

Section 2.2). Additionally, there also taint-based approaches like BuzzFuzz [19], which uses a dynamic taint analysis to identify regions in the original seed input file that are relevant to reach deeper program behaviors. It uses the information about these regions to direct the fuzzing process.

Greybox Fuzzing. The goal of *greybox* fuzzing is to keep the computation overhead low but still be able to guide the input generation. Typically, it uses some lightweight instrumentation of the application to gain some knowledge, e.g., about the program coverage, to guide the fuzzing process. Popular representatives are LibFuzzer [37] and American Fuzzy Lop (AFL) [38]. AFL builds the basis for the implementation of the differential fuzzing approach proposed in this thesis, and hence, it is described in more detail later in this section.

Fuzzing approaches can not only be distinguished based on their context knowledge, but also on how they generate new inputs: *search-based* fuzzing and *grammar-based* fuzzing [48].

2.1.2 *Search-based Fuzzing*

The goal of search-based software engineering (SBSE) is to solve software engineering problems with machine-based search algorithms [22]. The leveraged search algorithms vary from very simple, random generation of inputs (*random search*), over *hill climbing* that optimizes a fitness function to climb to a local optimum, *simulated annealing* as a similar approach as hill climbing but which avoids local optima, up to *genetic algorithms* to perform a global search by mutating initial inputs to a broader, more interesting population [22]. *Search-based fuzzing* covers the problem of finding test inputs by using a search algorithm [48]. The choice of the underlying search strategy depends on the actual problem or application under test. Search-based fuzzers typically implement a genetic algorithm for the input generation because they generate new inputs by mutating existing ones, and hence, reuse existing semantic context that has been present in the previous population [23]. These fuzzing techniques are also called *mutative* approaches [23]. A popular representative is the already mentioned AFL [38, 39], whose workflow is shown in Figure 2.

American Fuzzy Lop (AFL). AFL operates on input files, i.e., all mutations are performed on the bit or byte level of the file. Step 1 in Figure 2 represents the input for AFL's fuzzing process, which are some initial seed input files. AFL technically requires at least one *non-failing* input file. Non-failing means that AFL can execute the program under test with the given file without any crash or timeout.

The central data structure of AFL is the fuzzing queue, which holds the mutation corpus (cf. step 2 in Figure 2). The input size has a large impact on the fuzzing performance [39], and hence, the typical approach is to first trim the test inputs before mutation (cf.

step 3 in Figure 2). Afterwards, AFL applies its mutation operators repeatedly on these inputs to generate new (mutated) inputs (cf. step 4 in Figure 2). Therefore, AFL uses two types of fuzzing strategies: *deterministic* and *random* [39, 41]. The deterministic mutation operations include sequential bit and byte flips, simple arithmetics like sequential addition and subtraction of small integers, and sequential insertions of known interesting integers (e.g., 0, 1, Integer.MAX_VALUE). The random mutation operations include, e.g., random single-bit flips, random block deletions and insertions, and input splicing (i.e., the crossover operator on two test inputs at a random location). In addition to the mutation operators, AFL can be setup with a dictionary that can be filled with input specific syntax tokens. This mechanism is used to simulate an input grammar (see paragraph below). Each generated mutant needs to get evaluated for its usefulness in the following fuzzing process.

The original AFL implementation is purely coverage-based: it will keep mutated inputs in its fuzzing corpus, if they increase the overall branch coverage. In order to measure the coverage AFL uses a lightweight instrumentation of the program. AFL executes the mutated input with the instrumented program and tries to fit the input in three categories: coverage-increasing, crashing, and hanging. Coverage-increasing inputs are kept in the fuzzing queue for further mutations (cf. step 6 in Figure 2). Crashing inputs are copied in the *crashes* bucket, so that the user can use this input to debug the program. Similarly, AFL stores inputs, whose execution runs into a timeout, in the *hangs* bucket for the user. All other inputs get discarded. AFL can be directly applied on the program under test, but if the analysis requires some pre-processing of the input, then the common way is to implement a *fuzzing driver* (cf. step 5 in Figure 2). The fuzzing driver is a small program on its own, which parses the input file, extracts the necessary information, and executes the program under test with the required setup.

While AFL is applicable for C programs, there exist extensions for other program languages. For example Kersten et al. [26] present KELINCI, a tool that applies AFL on instrumented JAVA programs. AFL has been applied on many applications and has found many vulnerabilities for example in Adobe Reader, Mozilla Firefox, Internet Explorer, Apple Safari, and the iOS kernel [38]. Additionally, AFL is part of OSS-FUZZ [45], which provides open source projects the ability to apply fuzzing for their projects. As of the January 2020, OSS-FUZZ reported that it has found over 16,000 bugs in 250 open source projects [45]. Furthermore, AFL has been the starting point for many research activities around fuzzing.

Advanced fuzzers based on AFL. Böhme et al. [17] formulate coverage-based greybox fuzzing as a systematic exploration of the state space of a Markov chain. They identified that coverage-based greybox fuzzers like AFL generate many inputs that actually trigger the same behavior, and hence, loose efficiency. They present their tool AFLFAST, which focuses its search on low-frequency paths by leveraging power schedules to control the number of mutants generated from an existing input in the fuzzing corpus. In their evaluation, Böhme et al. show that AFLFAST generates significantly more crashes than AFL by increasing the covered program locations.

Later on, Böhme et al. [16] extended the coverage-guided fuzzing idea of AFL to *directed greybox fuzzing* with their tool AFLGO. It enables directed fuzzing for, e.g., patch testing, crash reproduction, static analysis report verification, or information flow detection.

The work by Lemieux and Sen [28] targets the same problem as AFLFAST [17], namely the low program coverage of AFL. In contrast to AFLFAST, their approach does not determine which input to mutate, but how to mutate the input. Additionally, they focus on hitting rarely covered branches, instead of low-frequency paths as in [17]. With their tool FAIRFUZZ, which is also based on AFL, they propose a targeted mutation strategy that tries to focus on mutation operators that likely will generate mutants to hit so far rarely covered branches.

Li et al. [29] propose with their technique STEELIX a combination of coverage-based fuzzing with a light-weight static analysis. They propose to use this light-weight static analysis together with some binary instrumentation to not only provide coverage information but also state information that pinpoints interesting bytes in the input. Interesting bytes are meant to influence constraints in the code that hinder fuzzing to get deeper into the program's logic. This information helps the fuzzer to concentrate its mutations on these parts, and hence, overcome the constraints. More specifically, the static analysis filters out comparisons in the program that appears to be relevant for deeper program's behavior. The binary instrumentation uses this information to specifically instrument the program's binary at these locations. Therefore, not the complete binary is instrumented so that the execution overhead is kept low. Furthermore, the fuzzer uses the feedback from the instrumentation to pinpoint relevant bytes in the input to guide its adaptive mutations to increase the coverage.

Pham et al. [35] show that random byte mutations can be ineffective for complex input structures, and hence they propose new mutations operators that act on an abstract view of the input structure. Their work on smart greybox fuzzing (aka AFLSMART) implements these mutation operators combined with a novel power schedule that aims to spend more time on fuzzing inputs that will likely pass the input parsing stage of applications. Their evaluation showed that AFLSMART can improve the coverage and identify more vulnerabilities than AFLFAST and AFL.

Guarantees of fuzzing. As discussed in this section, fuzzing can be very effective in identifying program bugs, especially security vulnerabilities. But since it does not perform a systematic exploration of the search space, it is in general not possible to provide formal correctness guarantees. Böhme [15] proposes a framework that models software testing and analysis as species discovery (STADS). STADS provides statistical correctness guarantees for fuzzing approaches like AFL. It helps to answer questions like "When should fuzzing be stopped?" or "What is the probability of discovering a crash with the next generation?".

2.1.3 *Grammar-based Fuzzing*

Grammar-based fuzzing belongs to the *generative* approaches that use a grammar that defines valid input structures. The grammar enables the fuzzer to generate syntactically valid inputs for a program [20, 23, 36]. It addresses the problem of traditional fuzzers like AFL, which generates new inputs with random mutation operators. This approach might fail to generate valid inputs for programs, which expect highly structured inputs [36].

Godefroid et al. [20] propose grammar-based whitebox fuzzing as a dynamic symbolic execution technique, which generates constraints that can be checked with a custom grammar-based constraint solver. This constraint solver checks whether the path constraint is satisfiable with regard to a given grammar-based specification of the valid input space. In their experiments they compare their approach with the code generation module of Internet Explorer 7. They show that their approach significantly increases code coverage while using three times fewer tests.

Wang et al. [36] present SUPERION, a grammar-aware greybox fuzzing technique, which contains (1) a grammar-aware trimming strategy, and (2) two grammar-aware mutation strategies. Input trimming in fuzzers is used to minimize the test input in order to allow a fast processing, but can in general lead to violations of the input's structure as they are performed grammar-blind [36]. Superion's trimming strategy incorporates an input grammar to ensure syntax-validity of trimmed test inputs. The standard mutation operators in fuzzers perform random mutations like bit flips or value insertions, which are again grammar-blind and hence can violate the input's structure. Superion's includes grammar-aware mutation strategies, e.g., insert values only at syntactically valid positions, as they operate on the level of the abstract syntax tree (AST). Their evaluation, which compares Superion to AFL and jsfunfuzz [43], shows that grammar-aware fuzzing significantly improves the bug finding capabilities by also increasing the code coverage.

Pavese et al. [33] use an input grammar to generate structured inputs that are very different from the common usage scenarios. They take as input a context-free grammar together with a set of sample inputs. By parsing the sample inputs Pavese et al. determine the probability of the grammar productions. Assuming that the sample inputs represent common inputs, they can retrieve uncommon inputs by inverting the probabilities.

Holler et al. [23] present LangFuzz, a blackbox fuzzing tool that combines generative and mutative fuzzing in order to generate syntactically and semantically correct code fragments to test interpreters. LangFuzz takes three input sources: a language grammar, some sample code and a test suite. It works in three phases: (1) It learns code fragments from the sample code and the test suite. Afterwards, (2) it generates test cases based on the provided language grammar and based on mutations of the code in the test suite. The knowledge about the learned language fragments is used during the mutation process to know which parts can be mutated. Finally, (3) LangFuzz uses the obtained test cases to execute the interpreter under test, while it checks for crashes and assertion failures. Holler et al. applied their grammar-based fuzzer on the Mozilla JavaScript interpreter and discovered several severe vulnerabilities.

The work by Padhye et al. [32] also combines generative and mutative fuzzing. They leverage a parametric generator, which maps a sequence of bytes to a syntactically valid input for the application. These generators are meant to generate a structured input (e.g., an XML file) based on the given parameter represented as sequence of bytes. Padhye et al. propose the idea that *bit-level mutations* in this parameter correspond with *structural* mutations in resulting generated input. Therefore, their implementation Zest performs standard mutations (e.g., random insertions and deletions of bytes) on this parameter to generate various valid inputs to explore the semantic stage of the application.

Even though the grammar specification of the application's input might be publicly available in some cases [36], the need for such grammars is a general limitation of grammar-based fuzzers. The same is valid for parametric generators as in the case of [32]. In order to mitigate this limitation Höschele and Zeller [24, 25] propose their technique AUTOGRAM to automatically mine input grammars.

SUMMARY – FUZZING

This thesis focuses on coverage-guided, mutational fuzzers because the above discussed research around AFL has shown that such a search-based fuzzing approach can be highly effective and does not make strong assumptions like the existence of a grammar specification. In order to compensate the limitations of fuzzing, a complementary component in a hybrid approach can be a systematic exploration of the search space as performed by symbolic execution. Therefore, the next section discusses the foundations of symbolic execution.

2.2 SYMBOLIC EXECUTION

Symbolic execution means the execution of a program not with *concrete* inputs but with *symbolic* inputs. Symbolic inputs represent a set of concrete inputs, usually only limited to a specific datatype. Since the symbolic inputs are variable in their nature, symbolic execution will end up with exploring all reachable paths of a program, and hence, represents a *systematic* exploration technique. The original idea of symbolic execution was independently developed by James C. King [61, 62], Lori A. Clarke [55, 56], and Boyer et al. [52]. King [61, 62] defined symbolic execution for a simple programming language that does only allow integer variables. His motivation was to provide a practical alternative between program *testing* and program *proving*. Clarke [55] proposed an automatic test data generation tool based on symbolic execution that analyzes programs written in ANSI FORTRAN. In a later work Clarke and Richardson [56] propose different models for symbolic execution, which determine how to choose the next path for exploration. They already define the notion of dynamic symbolic execution and also discuss applications like verification, test data selection, and debugging. Boyer et al. [52] proposed a system similar to King [61, 62] but for the analysis of programs written in a subset of LISP.

2.2.1 *Example*

In order to illustrate symbolic execution please consider the code snippet in Listing 1 and the corresponding symbolic execution tree in Figure 3.

Listing 1: Sample program symbolic execution.

```
1  int x, y;
2  if (x > y) {
3     x = x + y:
4     y = x - y;
5     x = x - y;
6     if (x > y)
7        assert false;
8  }
```

The sample code represents a conditional swap of values: it swaps the content of two integer variables x and y so that the larger value will be finally contained in variable y. Such a swap operation can be used for example in sorting algorithms. The symbolic execution tree represents the state space explored during symbolic execution. Each state during symbolic execution is characterized by the *path condition*, the *mapping* between variables and *symbolic expressions*, and a *pointer* to the *next instruction*. The path condition (or path constraint) represents the accumulated constraints on the control flow of the program, which need to be satisfied to reach the state.

Symbolic execution uses symbolic values for the program execution and the following explanation assumes that both variables are treated symbolically. Therefore, both get initialized with symbolic values (cf. node 1 in Figure 3). The capital X and Y hereby denote corresponding symbolic values. The constraint `[TRUE]` in the front of node 1 shows that the analysis does not assume any preconditions to reach the first node. The next step in symbolic execution is the branching instruction in line 2 (cf. Listing 1). Node 2 in the symbolic execution tree shows the condition using the symbolic values from the mapping. The path constraint remains unchanged.

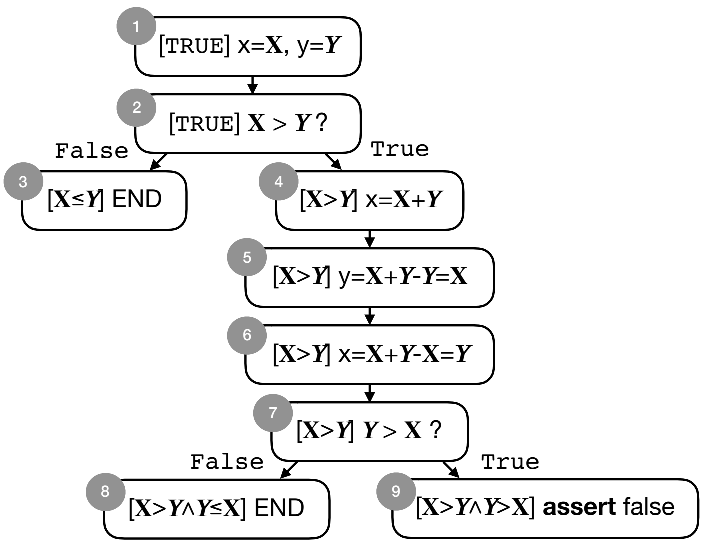


Figure 3: Symbolic execution tree for the sample code in Listing 1.

In the next step symbolic execution needs to fork the execution because it can either follow the `False` branch or the `True` branch. At this point both branches are satisfiable with regard to the current path constraint. The exact exploration strategy of symbolic execution depends on its implementation. This examples follows a deterministic depth-first search beginning with the `False` branch. Therefore, the next execution step is the node 3. The path constraint is updated based on the made choice: the condition in line 2 is assumed to be `False`, which means that the path constraint is updated to the negated condition: $X \leqslant Y$.

Afterwards, symbolic execution uses a constraint solver to solve the updated path constraint. If a path constraint is no longer feasible, i.e., the constraint solver cannot find a satisfiable model, symbolic execution will abort this execution path and backtrack to the previous branching point. Node 3 represents the code part where no swapping is necessary and ergo the execution is finished afterwards, denoted by the `END`. Also in this case symbolic execution backtracks to the previous branching point in order to investigate unexplored branches. Therefore, the next step is to enter the `if` condition in line 3 that leads to the creation of node 4. The path constraints is updated to $X > Y$ and the instruction in line 3 is executed. The arithmetic instruction `x = x + y` needs to be applied in the symbolic domain, and hence, the mapping between variable `x` is updated to the symbolic expression $X + Y$. Similarly, the following assignments in lines 4 and 5 (represented in node 5 and 6) update the mapping so that eventually `x` holds the symbolic expression Y and `y` holds the symbolic expression X. In other words: `x` and `y` swapped their values. The path constraint is not updated during this process because there is no branching involved.

Afterwards, symbolic execution reaches the branching instruction in line 6 (represented in node 7) and forks the execution accordingly. It begins again with `False` branch (cf. node 8) and updates the path constraint to $X > Y \wedge Y \leqslant X$. In this branch the swapping was successful and the program finishes, which is denoted with the `END`. Then symbolic execution backtracks and investigates the `True` branch (cf. node 9). This branch checks whether there has been an error during swapping so that the value of `x` is still greater than the value of `y`. The path constraint is updated to $X > Y \wedge Y > X$. At this point symbolic execution would not continue exploring this branch because this path constraint is unsatisfiable: Y cannot be smaller and greater than X at the same time. Therefore, the assertion failure in line 7 is not reachable. Symbolic execution explored all feasible paths, and hence, is finished. The obtained feasible path constraints can be used to generate test inputs to test these paths or to generate a test suite for a specific coverage criterion.

2.2.2 *Tools*

Early limitations in all of those techniques have been the limited reasoning ability since they usually applied only on simple programming languages and the lack of efficient constraint solving techniques. However, symbolic execution experienced renewed interest in the recent years because of the increased availability of computational power and decision procedures [11]. Popular symbolic execution engine are for example KLEE [54] and Symbolic PathFinder (SPF) [60, 67].

KLEE applies on LLVM bytecode and includes an extensive environment modeling. A large issue in symbolic execution is the handling of native operations and interactions with the environment like I/O operations on the file system. Typically there are two solutions: (1) switch to a concrete mode to perform a dynamic symbolic execution for the interaction (which of course limits its state space exploration as a concrete execution only represents one possible path), or (2) leverage models that contain the semantics of the desired action. KLEE contains such models, e.g., for the interaction with the file system, which makes KLEE highly applicable in a practical context.

SPF is based on Java PathFinder (JPF) [59, 72], a model checker for Java bytecode. JPF builds it own virtual machine so that it can interpret each bytecode instruction. SPF adapts this implementation to handle the bytecode execution symbolically.

2.2.3 *Advanced Symbolic Execution Techniques*

Symbolic execution comes with a couple of limitations: *path/state explosion*, *complex path constraints*, and the handling of *native calls*.

Tackling path explosion. The path explosion problem of symbolic execution is inherent due to its nature of a systematic exploration: Since it tries to explore all possible branches in a program, there is an exponential growth in the number of execution paths. Therefore, pure symbolic execution quickly reaches its scalability limitations in a practical environment. Loops and recursion are for special concern since they may lead to an infinite number of execution paths. This general issue gets usually addressed by a *bounded symbolic execution* (BSE). In BSE the exploration of a path will be aborted after a pre-defined bound on the search depth or the number of extracted path constraints. Alternatively, one can stop the analysis when a desired coverage is achieved.

The introduction of abstractions in the analysis can help to reduce the number of paths and broaden the coverage of symbolic execution. Loop abstractions or summaries [58, 68, 71] help to incorporate the semantics of loops in symbolic execution to reduce the exploration effort and increase the overall coverage. Abstract state matching in symbolic execution [73] means to check whether a symbolic state subsumes another symbolic state. This includes to check the subsumption of the symbolic data objects and the valid implications of the corresponding path constraints.

Compositional symbolic execution. Another research direction is *compositional* symbolic execution by Anand et al. [49]. A compositional analysis summarizes the results for lower-level function in the program so that these results can be reused in a larger analysis. For example when a function gets called multiple times in the program, symbolic execution can first analyze this function in isolation and reuse the resulting path conditions in a larger exploration. Additionally, Anand et al. proposes *demand-driven* compositional symbolic execution. It means that there is a certain target to reach with less effort as possible.

Similarly, Ma et al. [63] propose *directed symbolic execution* to guide the exploration to designated targets in the program. Both approaches leverage an *interprocedural control flow graph* (ICFG) to drive their analysis. The interprocedural control flow graph connects various (intraprocedural) control flow graphs (CFGs) that describe the control flow within a function or a method. The ICFG describes how the control flows of the functions are connected based on their calling hierarchy inside the program. The key component of both approaches [49, 63] is to efficiently identify a feasible path in the ICFG to the designated target by traversing feasible paths in the local CFGs. Distance calculations on the ICFG like the shortest distance between two nodes in graph or the reachability calculation of nodes can be used to guide such an exploration.

Parallel symbolic execution. Furthermore, symbolic execution can be accelerated by *parallelization* [53, 70] because in general there is no sharing between the sub-trees in the symbolic execution tree. Therefore, the symbolic execution tree can be partitioned in two or more parts than can be processed separately. The challenge in partitioning is to identify in good balancing between the partitions: a well balanced partition can lead to a linear speedup, whereas a poor balancing results in no speed up at all. The literature describes simple static partitioning approaches [70], as well as dynamic partitioning [53].

Decision Procedures. In terms of solving complex path constraints the performance of symbolic execution strongly depends on the decision procedures. Decision procedures are used to determine the satisfiability of a path constraint. Symbolic execution techniques typically use *Satisfiability Modulo Theory* (SMT) solvers, which can compute the satsifiability of formulas in first-order logic with associated background theories like integer arithmetic, arrays, bit-vectors, and Strings. State-of-the-art solvers are Z3 [64] and CVC4 [51]. Additionally, there exist specialized solvers for handling, e.g., formulas including String constraints, like ABC [50].

Dynamic symbolic execution. Pure symbolic execution is a static analysis technique, which might be faced with the problem that some program components cannot be evaluated symbolically. For example native calls or function calls to third-library code. This limitation can be tackled by the usage of dynamic variants of symbolic execution to simplify the constraints. *Dynamic symbolic execution* in general means that the analysis follows the execution of a concrete input but collects the symbolic constraints and expressions along this path.

DART / Concolic testing. Godefroid et al. [57] propose DART as a automated unit testing technique based on dynamic symbolic execution. They collect the constraints during concrete execution and systematically negate conjuncts of the constraints, e.g., the last conjunct, to generate new inputs that explore new execution paths. If a modified constraint is not feasible, e.g., because the constraint is too complex for the decision procedures, then DART would leverage random concrete values to solve the constraint. On a high level, their approach works in three steps: (1) the automated extraction of the interface of the program under test by using static source code parsing, (2) the automatic generation of a test driver for random testing of the program, and (3) the dynamic analysis of the program and automatic generation of new test inputs. Sen et al. [69] follow a similar approach with their technique CUTE, but specifically tackle the problem of constraints over dynamic data structures that include pointer operations. They introduce the term *concolic testing* as a cooperative combination of concrete and symbolic execution. The already discussed technique SAGE by Godefroid et al. [21] (cf. Section 2.1) present their whitebox fuzzing approach as an extension of concolic testing. They use *generational search* to maximize the modified constraints resulting from one run.

Păsăreanu et al. [66] propose mixed concrete and symbolic reasoning, where the path constraint gets splitted in a simple and in a complex part (e.g., native calls or non-linear constraints). The simple part can be solved with an existing decision procedure. The resulting solution is used to simplify the complex part.

Fitness guided symbolic execution. Furthermore, there are dynamic symbolic execution techniques to address the path explosion problem like Fitnex by Xie et al. [74]. They propose to use a fitness guided path exploration which assigns *flipping* priorities to branching nodes based on a calculated fitness value. The constraints at the node with the highest flipping priority gets *flipped*, i.e., negated to follow unexplored branches. The fitness values are calculated to measure how close the execution is to cover a given target predicate.

Online vs. offline symbolic execution. The literature distinguishes in general between two types of symbolic execution: *online* and *offline* [75, 80].

Offline symbolic execution runs the program and extracts a path constraints that is solved by a decision procedure. Afterwards it re-runs the program with the newly obtained input to extract another path constraint to be solved. This offline techniques is applied in concolic testing with CUTE [69], DART [57], and SAGE [21].

Online symbolic execution represents the traditional symbolic execution as implemented in KLEE and SPF. The symbolic execution forks at a branching point by duplicating states and path constraints. Therefore, online symbolic execution does not require re-runs but can simply backtrack to the last branching point. However, this type of symbolic execution usually leads to a high memory usage because it needs to store the state information, which might become quickly infeasible for complex programs.

SUMMARY – SYMBOLIC EXECUTION

This thesis focuses on an offline dynamic symbolic execution that can be guided by several *differential metrics*. Since it is driven by concrete inputs, it can incorporate the results from a fuzzing engine (cf. previous Section 2.1). For an efficient test input generation technique, both components (fuzzing and symbolic execution) are combined in a hybrid analysis approach. Such hybrid techniques are discussed in the next section.

2.3 HYBRID ANALYSIS

This section starts with summarizing the strengths and limitations of fuzzing and symbolic execution, and further describes existing hybrid solutions for combining these two techniques.

2.3.1 *Strengths and Limitations of Fuzzing*

Mutational fuzzing is considered as an effective testing technique, which can quickly generate a lot of inputs. Due to its random nature it can quickly generate malformed and unexpected test inputs to check the program for crashes and unexpected behaviors [30, 80]. Therefore, it is known to find shallow bugs [81], e.g., in the input parsing component (also called syntactic stage [32]). While it is considered to be *inexpensive*, it is also known to be *incomplete* [76, 30, 81] because it might miss bugs that occur deeper in the program execution (also called semantic stage [32]). Such deeper program behaviors are guarded by specific constraints that are unlikely to be hit with random mutations. Therefore, it is usually recommended as a complementing testing technique [30].

2.3.2 *Strengths and Limitations of Symbolic Execution*

Symbolic/concolic execution represents a systematic exploration of the state space of the program, and hence, can solve constraints that guard deeper program behaviors [81]. While it is known for its *input reasoning abilities* [81], it suffers from the path explosion issue, which makes it hardly applicable in a practical environment because it can, similarly to fuzzing, get *stuck* in shallow parts of the program [78, 81].

2.3.3 *Existing Hybrid Techniques*

Existing research on hybrid software testing combines fuzzing and symbolic execution in a couple of setups, e.g., fuzzing and symbolic execution in an alternating way or beginning with symbolic execution to generate seed inputs for a subsequent fuzzing run. Note that the main goal in the existing techniques is to increase the coverage with the generated test inputs and not to perform a differential analysis.

Symbolic execution followed by fuzzing. Brian S. Pak [80] proposes hybrid fuzzing, which starts with symbolic execution of the program under test to discover *frontier nodes* within a given resource budget. Frontier nodes are either nodes at the end of an execution path or are intermediate nodes, at which symbolic execution aborted the execution because it exceeded the resource budget. Pak's technique [80] generates concrete inputs that reach the identified frontier nodes and uses them as initial seed inputs for a fuzzing run. Therefore, symbolic execution is used to generate valid seed inputs for fuzzing. Pak's evaluation showed that their hybrid concolic testing can cover almost 4x the branches that are covered by random testing and almost 2x that of concolic testing. However, the concept misses a feedback loop back from fuzzing to symbolic execution.

Fuzzing followed by symbolic execution. Ognawala et al. [79] propose a compositional fuzzing approach that is aided by targeted symbolic execution. Their technique WILDFIRE builds on the compositional testing framework MACKE [65] and starts with parallel fuzzing of isolated functions based on AFL. It collects the reported crashes and afterwards determines the feasibility of reported vulnerabilities by leveraging targeted symbolic execution with an extended version of KLEE. Furthermore, WILDFIRE is a fully automated approach which, e.g., also generates the fuzzing drivers.

Alternating offline and online symbolic execution. Cha et al. [75] propose the technique MAYHEM by introducing *hybrid symbolic execution* as a combination of online and offline symbolic execution. In that sense MAYHEM is not explicitly combining fuzzing with symbolic execution but explores hybrid combinations of symbolic execution techniques. In order to identify exploitable bugs in binaries, MAYHEM's analysis alternates between online and offline mode. It starts in the online mode and as soon as the analysis reaches a memory limit it stops and generates *checkpoints*. These checkpoints contain the state information about the current non-finished execution paths. Afterwards MAYHEM picks one of the checkpoints and uses offline symbolic execution to efficiently restore this execution. Then MAYHEM restarts its online execution from there. Cha et al. [75] show in their evaluation that hybrid symbolic execution needs less memory than online symbolic execution and is faster than offline symbolic execution.

Alternating fuzzing and symbolic execution. One of the earliest work on hybrid analysis is the paper on *hybrid concolic testing* by Majumdar and Sen [77]. They propose the combination of random testing and concolic testing as an interleaving of both techniques. It starts with the concrete execution based on randomized input (aka random testing) and as soon as it cannot make any coverage improvements, it switches to concolic testing of the last seen input. Therefore, it concollicaly executes the input and generates new path constraints by negating parts of the constraint. As soon as concolic testing identifies an input that improves the program coverage it switches back to concrete execution with an updated input based on the results on concolic testing. The testing technique stops as soon as there is a bug/crash found. Otherwise it alternates between random testing and concolic testing within a given resource budget.

EVOSUITE by Fraser and Arcuri [7–9] is a test case generation techniques based on a genetic algorithm with the goal to generate a test suite, which fulfills a given coverage criterion. The genetic algorithm in EVOSUITE suffers from the general problems of covering specific program parts only with a low probability. Their intuition is that dynamic symbolic execution can help to find these values, although symbolic execution comes with its own problems (see above). Therefore, Galeotti et al. [76] propose a hybrid setup as an adaptive approach, which combines EVOSUITE with dynamic symbolic execution. The search starts with the genetic algorithm and based on its feedback it is determined whether the current problem is suitable for dynamic symbolic execution. Therefore, they do not apply expensive symbolic execution when it is actually not necessary.

The technique DRILLER by Stephens et al. [81] proposes a sequential setup, in which fuzzing and concolic execution are executed in sequence. Starting the exploration with fuzzing, concolic execution gets started as soon as fuzzing gets *stuck*, i.e., it does not make any progress in terms of coverage. The concolic execution component tries to generate inputs that would push the fuzzer deeper in the program execution. Afterwards fuzzing get started again and the process repeats. The technique MUNCH by Ognawala et al. [78] proposes a similar approach, in which fuzzing and symbolic execution are executed in sequence, although they differ on how they guide the symbolic execution. Both techniques, DRILLER and MUNCH, execute only one technique at once, and hence, might miss potential analysis power.

Yun et al. [82] report that hybrid fuzzers like DRILLER [81] still do not scale to real-world applications and name the concolic execution component as the main performance bottleneck. They therefore propose QSYM as a concolic executor that is tailored for the support of hybrid fuzzing by applying a couple of optimizations. For example they symbolically execute only a small subset of the instructions that are required to generate the necessary constraints. The instruction selection is based on a fine-grained instruction-level taint tracking, which is feasible because QSYM can quickly switch between execution models. QSYM does not use any models for external environments but simply considers them as black boxes which get executed concretely. This can result in unsound test cases that do not produce new coverage, but is cheaper than symbolic execution of the actual environment or the models. Furthermore, they apply *optimistic* solving which tries to solves only parts of a path constraint when the complete constraint is not solvable.

SUMMARY – HYBRID ANALYSIS

All in all, the software testing community has already produced hybrid testing strategies that combine fuzzing and symbolic execution. However, the existing approaches mostly aim at increased program coverage and do not focus on differential analysis.

2.4 DIFFERENTIAL PROGRAM ANALYSIS

The previous sections described the fundamentals in software testing, in particular with the search-based technique fuzzing and with the systematic exploration technique symbolic execution. This section presents the fundamentals of *differential program analysis*, its possible application areas, and the state-of-the-art techniques for the specific applications. The goal of differential analysis is to identify *behavioral differences* as introduced in Chapter 1.

While most related work on differential program analysis mean the comparison between multiple program variants (aka regression analysis) [85, 87, 92, 93, 96, 97, 101, 103], differential analysis can also mean to find different execution paths within the same program instance. For example such an analysis can try to find very expensive program paths. Expensive paths show a different behavior in terms of a given cost function compared to the other paths in the program. This analysis is called worst-case complexity analysis [131, 140, 145, 146]. Other occurences of differential analysis are side-channel analysis [109, 113, 119] and the robustness analysis of neural networks [181, 184]. Independent of the specific kind of analysis, differential program analysis is a hard problem since it needs to reason about multiple program executions. The following sections describe the foundations and related work on the mentioned differential analysis types.

2.4.1 *Regression Analysis*

Regression testing/analysis is concerned with validating the change(s) between two successive software versions [101]. Figure 4 shows the recap of the two categories in differential

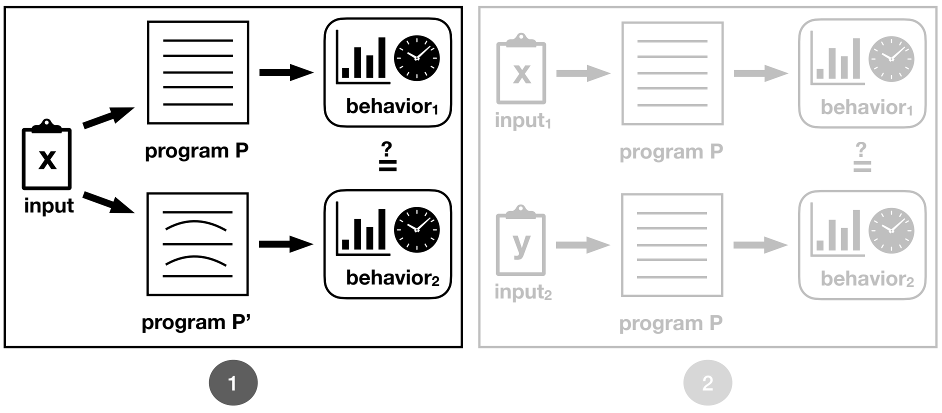


Figure 4: First category of differential software testing: regression analysis.

analysis as introduced in Section 1.2. Regression testing represents category (1) (the left side of Figure 4). It searches behavioral differences between two program variants executed with the same input. The typical approach uses an existing test suite and reruns it on the new software version [92]. However, running a complete test suite has two problems: (1) It might be not feasible because there are too many tests. Therefore, it is necessary to perform test case selection by prioritization or minimization of the test suite [87, 103]. (2) The existing test suite might not test the changed behavior because it was not relevant before, and hence, it needs some test suite augmentation [87, 92, 101, 103]

The testing techniques presented in this thesis do not focus on generating test suites, but focus on generating test inputs (for an existing test driver) that specifically trigger a changed behavior. The goal is to identify *regression errors*. The following paragraphs summarize the relevant research in the related areas, which can be separated in techniques based on *symbolic execution*, *search-based* techniques, and *verification* approaches.

Regression Analysis based on Symbolic Execution. Person et al. [96] argue that identifying differences based on differing source code fragments is to imprecise and often leads to false positives, e.g., by simple formatting changes. Therefore, they propose *Differential Symbolic Execution* (DSE) for the analysis of two program versions based on method summary generation and equivalence checking. DSE starts with symbolic execution of the two program versions and constructs over-approximating method summaries. Afterwards DSE checks whether the summaries are equivalent, for which they use two different notions for equivalence: *functional* equivalence and *partition-effects* equivalence. Functional equivalence checks for the same black-box behavior: the same input must lead to the same result. Partition-effects equivalence checks for functional equivalence and whether the programs equivalently partition the input space: the same input must lead to the same result by taking the same path. If the summaries are not equivalent, then DSE will provide deltas for the precise characterization of the inputs that show a different behavior.

In a subsequent work Person et al. [97] propose *Directed Incremental Symbolic Execution* (DiSE), which combines static analysis and symbolic execution to characterize the effect of program changes. It starts with an inter-procedural analysis to detect the parts of the control flow graph (CFG) that are affected by the changes. This information are used to guide the symbolic execution of the new (modified) version of the program by only allowing the exploration of paths that can reach the affected locations.

Yang et al. [102] introduce *memoized symbolic execution* (Memoise) as a technique to reuse previous symbolic execution runs. Memoise uses a trie-based data structure to store the key elements of symbolic execution, i.e., the branching decisions that included symbolic values. An already executed path can be efficiently replayed by following the stored choices in the trie. Therefore, memoized symbolic execution can significantly reduce the cost for multiple symbolic executions of a program during software testing. Additionally, Yang et al. propose the usage of Memoise for the regression analysis by re-executing symbolic execution only on the change-affected trie parts.

Shadow Symbolic Execution. Palikareva et al. [93] propose *shadow symbolic execution* (SSE) to generate inputs for testing software patches. Similar to the previous discussed techniques [97, 102], SSE does not aim to generate a high-coverage test suite, but to generate test inputs that specifically exercise divergences between two program versions. In contrast to the other techniques SSE does not apply only on the new program version, but executes both the old (buggy) and new (patched) versions in the same symbolic execution instance. This requires that both programs are *merged* into a *change-annotated*, unified version by applying simple annotation rules. Each change-annotation represents a function call `change(oldExpression, newExpression)`, where the arguments show the old and the new expressions respectively. Table 1 shows the change-annotations as introduced by Palikareva et al. [93].

Table 1: Change-annotations by shadow symbolic execution [93].

Change Type	**Example**
Update assignment	`x = x + change(E1, E2);`
Update condition	`if(change(E1, E2)) ...`
Add extra assignment	`x = change(x, E);`
Remove assignment	`x = change(E, x);`
Add conditional	`if(change(false, C)) ...`
Remove conditional	`if(change(C, false)) ...`
Remove code	`if(change(true, false)) ...`
Add code	`if(change(false, true)) ...`

The last two change types in Table 1 show a conservative handling of code changes in case no other more fain-grained change-annotation can be found. These change types simply represents the removing or adding of straightline code blocks.

Every change-annotation does in principle introduce a so-called `Shadow` expression that holds information about both program versions. For example in the case of updating an existing assignment: `x = x + change(E1, E2);` the variable `x` holds two expressions, `x + E2` for the new version and `x + E1` for the old version.

SSE performs dynamic symbolic execution on such a unified program version, which is implemented in two phases: (1) the concolic phase, and (2) the bounded symbolic execution (BSE) phase. In the first phase, SSE simply follows the concrete execution of test inputs from an existing test suite, while it checks for divergences along the control-flow of the two versions. This exploration is driven by the idea of *four-way* forking. In traditional symbolic execution every branching condition introduces two forks to explore the `true` and `false` branches. Shadow symbolic execution instead introduces four forks to investigate all four combinations of `true` and `false` branches for both program versions. As long as there is no concrete divergence, SSE follows the so-called $same_{\texttt{True}}$ and $same_{\texttt{False}}$ branch, which denotes that both concrete executions take the same branches. Additionally, SSE checks the satisfiability of the path constraints for the other two branching options, where both versions take different branches. These branches are called $diff_{\texttt{True}}$ and $diff_{\texttt{False}}$ paths. For every feasible *diff* path, SSE generates a concrete input and stores the divergence point for later exploration by the second phase. As long there is no concrete divergence, SSE continues until the end of the program.

When SSE hits the mentioned addition or a removal of straightline code blocks, it immediately stores a divergence point. This conservative handling leads to an over-approximation of the *diff* paths because the added/deleted code may not necessarily lead to an actual divergence.

The second phase performs bounded symbolic execution (BSE), *only* on the *new* version, from the stored divergence points to further investigate the divergences.

At the end, Palikareva et al. [93] perform some post-processing of the generated inputs to determine whether they expose some observable differences, e.g., by comparing the outputs and the exit codes. Palikareva et al. [93] implemented their approach on top of the KLEE symbolic execution engine [54].

Limitations of Shadow Symbolic Execution. Shadow symbolic execution as introduced by Palikareva et al. [93] is driven by concrete inputs from an existing test suite. While this exploration strategy tries to focus on constraining the search space, it might miss important divergences as it strongly depends on the quality of these initial test inputs. In particular

SSE might miss deeper divergences in the BSE phase because of limiting prefixes in the path constraints. Since BSE is started from the identified divergence points, it inherits the path constraint prefix from the concrete input that has been followed to find this divergence. In general, when there are several paths from the beginning of the program to this divergence, the concrete input might lead to a prefix that is not necessary and could limit the further exploration of deeper divergences.

Furthermore, SSE might miss divergences straight away due to a similar reason: Inputs that touch changes, i.e., they trigger the change-annotations, still could miss divergences when the concrete execution does not touch an actual divergence point. Hitting at least one change-annotation is trivially necessary since otherwise no change is exercised. The change-annotations introduce the `Shadow` expressions that are necessary to detect a divergence in the control-flow. However, this is necessary but not sufficient to trigger a divergence. Therefore, the existing inputs have to cover changes *and* the divergence points so that SSE can find them.

Search-Based Regression Testing. Orso et al. [89, 92] introduce *behavioral regression testing* (BERT) that essentially uses a blackbox regression technique to generate random test cases that expose behavioral differences between two software versions. BERT works in three phases: (1) In the first phase it generates large number of test cases for a specific software module. (2) In the second phase it runs the generated test case on the two software versions to check for differences. (3) In the last phase it collects the differences and filters duplicates, so that the reports can be presented to the developers. Their test case generation is based on Randoop [31] and Agitar's JUnit Factory [12].

Taneja and Xie [100] follow a similar approach with their technique DiffGen as they also use random testing to generate test cases that expose observable differences. They add additional branches to the code with the goal to expose a regression error. For example, given two versions of a Java class, DiffGen first detects modified methods. Afterwards it synthesizes a driver that sequentially calls both versions of such a method. Then DiffGen adds a comparison at the end of the driver that compares the outcomes and raises an assertion error when a difference is detected. Afterwards, they use random test case generation to test this synthesized and instrumented driver. Therefore, as soon as a generated test case covers the new inserted branches, then DiffGen has identified a difference.

Shamshiri et al. [99] propose an extension of EvoSuite [7–9] for regression testing, called EvoSuiteR. Shamshiri et al. use a genetic algorithm to generate test cases that specifically propagate regression errors to an output difference between the two software versions. EvoSuiteR starts with an initial population of a random generated test suite and applies random mutations. The mutant selection is based on a fitness function that takes into account three differential metrics: (1) structural coverage, (2) object distance, and (3) control-flow distance. Structural coverage is used to generally increase the coverage by the test cases. The object distance measure the difference between the execution traces and the resulting objects. The control-flow distance measures for every branch how close the executions are to a divergence. After the analysis, EvoSuiteR collects the difference revealing test cases and adds assertions into the test so that the developer can easily see the behavioral differences. Although, Shamshiri et al. aim at generating a whole test suite to identify regression errors and not only specific test inputs, their approach is relevant for this thesis because they use similar differential metrics.

Although there exists no regression fuzzing approach so far, existing techniques on directed fuzzing like AFLGo [16] (cf. Section 2.1.2) could be applied on regression testing problems. However, this directed analysis can only be guided to suspicious locations, but cannot explicitly be targeted to a differential behavior. Nevertheless, guiding the fuzzing process into areas of changed code is a key ability in finding regression bugs.

Regression Verification. Böhme et al. [84, 85] introduce *Partition-Based Regression Verification* (PRV). PRV provides an incremental way to verify differential input partitions in context of regression verification. The differential input partitions are calculated via symbolic execution and are specified by a symbolic condition, which defines the subset of valid inputs for the partition. The inputs will get grouped into the same partition if they reach the same syntactic changes, and if they propagate the same differential state to the output. Afterwards, the input partitions get classified to be either equivalence-revealing or difference-revealing. The incremental approach allows to still give guarantees even when the analysis is interrupted or only a partial analysis is possible. Therefore, PRV represents an alternative to regression test generation.

Another differential verification approach is for example SymDiff by Lahiri et al. [90]. SymDiff aims at identifying output differences and uses the modular verifier Boogie [83] to create verification conditions that can be solved with the SMT solver Z3 [64].

The techniques proposed in this thesis do not provide any formal guarantees, although it would be interesting to explore the employment of statistical guarantees [15]. However, it remains a tradeoff between providing results with formal guarantees and the scalability necessary for the analysis of real-world applications.

SUMMARY – REGRESSION ANALYSIS

Shadow symbolic execution by Palikareva et al. [93] shows some great potential in guiding the exploration to divergences, but its strong dependency on concrete inputs limits its exploration capabilities. Search-based approaches like Shamshiri et al. [99] have shown to be effective, although still might miss divergences due to their random nature. Xu et al. [101] studied several test suite augmentation techniques and concluded that genetic and concolic approaches are effective and complement each other in terms of code coverage. Therefore, this thesis focuses on a hybrid technique.

2.4.2 *Worst-Case Complexity Analysis*

Figure 5 shows the recap of the two categories in differential analysis as introduced in Section 1.2. Worst-case complexity analysis and the other applications described in the following Sections 2.4.3 and 2.4.4 represent category (2) (the right side of Figure 5). In this category the differential analysis searches behavioral differences within the same program that is executed twice with different inputs. The goal of worst-case complexity analysis (WCA) is to identify an input that performs very *different* than most of the other inputs: an input that maximizes the specified cost metric. In this context the analysis can be called *differential*. However, this analysis is simpler than the other presented applications because it does not reason about multiple paths at the same time. Inputs that trigger worst-case executions represent serious algorithmic complexity (AC) vulnerabilities and could be used to perform complexity attacks [132, 134, 148, 149, 151, 152].

WCA based on fuzzing. Recent techniques in WCA have explored fuzzing for its purpose. Petsios et al. [146] propose SlowFuzz as a resource-usage-guided evolutionary search. Their fitness function counts the number of instruction executed for a particular mutated

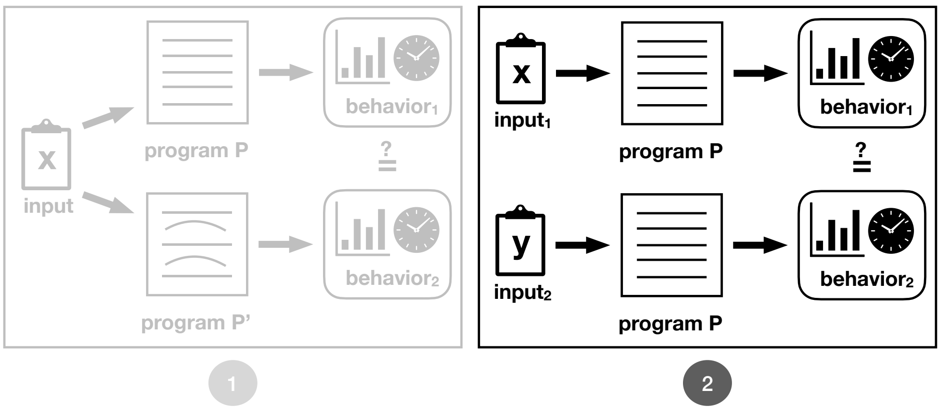


Figure 5: Second category of differential software testing: worst-case complexity analysis, side-channel analysis, and robustness analysis of neural networks.

input and keeps inputs that exceed the so-far observed execution cost. SlowFuzz builds on top of LibFuzzer [37] and is a domain-independent as it simply generates inputs for a given application.

Similarly, PerfFuzz by Lemieux et al. [143] generate inputs that trigger algorithmic complexity vulnerabilities. In contrast to SlowFuzz, they maximize not the number of executed instructions, but the execution counts for all program locations. Therefore, they collect for each execution the hit counts of each edge in the control flow graph and keep inputs that executes more edges than any other previous input.

Wei et al. [153] propose Singularity, a technique that performs *pattern* fuzzing for WCA. Instead of searching for specific inputs, they look for input patterns that maximize the resource usage of the application.

Le et al. [142] use grammar-based fuzzing to identify inputs that are not covered by the initial grammar but trigger worst-case behaviors in the target program. Le et al. [142] start with the intuition that the given grammar is only *approximate*, and hence, need to be first extended to correctly represent the input space of the target program. Therefore, their technique Saffron works in two phases: (1) First they generate valid inputs files with the given grammar, which are used as seed inputs for traditional coverage-based fuzzing. The fuzzing will generate inputs that are not covered by the grammar but might be still accepted by the target program. Afterwards they repair the initial grammar with the identified inputs. (2) In the second phase, Saffron uses the updated grammar to generate inputs in a cost-guided manner by favoring rules that lead to an increased cost.

WCA based on symbolic execution. Burnim et al. [131] propose the usage of symbolic execution for the automated test input generation to perform WCA. Their technique is called WISE (Worst-case Inputs from Symbolic Execution) and uses an exhaustive state exploration for small input configurations. Based on the resulting path constraints they learn branching policies that lead to a worst-case execution, ideally also for larger input configurations. They apply these path policies for larger input sizes and show that their technique can discover AC vulnerabilities. Luckow et al. [145] later introduce SPF-WCA for a similar analysis. Their path policies also take into account the *history* of executions and the calling context of the analyzed procedures.

Chen et al. [133] use probabilistic symbolic execution to generate performance distributions of an application. They take as input the target program and a usage profile, which defines the likelihood of inputs. Their approach works on two phases: First they symbolically execute paths with a high probability, which will represent the paths that get executed by the majority of the inputs. In the second phase they concentrate the exploration on low-probability paths to identify special conditions for the program's behavior. They generate test inputs for the extracted path constraints and execute them on the target program. The resulting cost measurements can be used to generate a *performance distribution* that plots the input probability over the program execution times. Their tool PerfPlotter can be used to reveal performance problems with their associated usage probabilities. Furthermore, they can compare the performance distributions of multiple program versions to detect performance regressions.

Zhang et al. [155] use symbolic execution to automatically generate a test suite for load testing. They perform an exhaustive symbolic exploration up to a specific exploration depth. While incrementally increasing this depth, they focus on paths that show a high resource consumption. The resulting path constraints can be solved to generate a test suite.

Other techniques for WCA. Static analysis techniques [129, 136, 137, 150] leverage techniques like invariant generation and expression abstraction to compute upper bounds on the time complexity of programs.

Holland et al. [140] propose a two-step approach for the detection of AC vulnerabilities. First of all, they use a technique based on static analysis to identify loops in the application as potential reasons for AC vulnerabilities. In the second step, they apply a dynamic analysis on the identified loops to trigger an excessive resource consumption. Their static analysis investigates the control-flow of the application and displays the resulting loop structures to a human analyst. The analyst can then select loops, which should be further analyzed. The dynamic analysis automatically instruments the loops with workload probes to measure the timing behavior.

Many techniques focus on the worst-case execution *time* (WCET) analysis with a strong focus on real-time systems [138, 139, 144, 154]. In order to manage the analysis the typical approach is to limit the analysis of loops to finite bounds while estimating the worst-case execution time for the system.

Profilers [130, 135, 147] are used to identify potential performance bottlenecks in applications. Their results depend on the test inputs used for profiling. Profiling techniques can be used to pinpoint suspicious components of the application, which can later be used as targets for a more detailed performance analysis.

SUMMARY – WORST-CASE COMPLEXITY ANALYSIS

The existing WCA techniques mostly focus on fuzzing, static analysis, or symbolic execution. Fuzzing techniques like SlowFuzz [146] are promising although they lack the general problems on mutational fuzzing. The evaluation for WCA in this thesis is partially based on [146]. The fuzzing technique proposed later in this thesis follows a similar approach but adds more metrics to have a broader fitness function for the differential analysis. Furthermore, it adds a dynamic symbolic execution to complement the fuzzing process. The existing WCA techniques based on symbolic execution perform exhaustive explorations, which may not be feasible in practice as they do not scale on large programs.

2.4.3 *Side-Channel Analysis*

Side-channel analysis tries to identify leakages of secret information by observing the non-functional system behavior, like execution time, memory consumption, or response message sizes. Imagine an authentication scenario at a webpage: a potential attacker wants to guess the password for an existing user. The main-channel in this scenario would be the actual response, which includes the information of the password check: *correct* password or *incorrect* password. A side-channel could be for example the execution time or the memory consumption; basically every metric, which can be gathered during a program execution. The side-channel analysis searches for an observation that depends on the secret information, in this case the server-side password. If there is a dependent observation, then it would mean that the program execution leaks information about the secret.

Example. A typical example for a side-channel vulnerability is a password comparison algorithm, which leaks information about the stored (secret) password via the runtime of the algorithm, also referred as *timing side-channel*. Listing 2 shows such an *unsafe* password comparison algorithm. The algorithms takes two arrays as parameters, one array for the (public) user input `pub` and the stored (secret) password `sec`. It starts with comparing the length of both arrays and will return `false` if both lengths do not match. As long as both arrays have the same length, the algorithm continues with comparing the passwords byte by byte. As soon as there is a mismatch, the algorithm will return `false`, and only if all byte values match, the algorithm will finally return `true`. Therefore, this *unsafe* algorithm has two early-returns, in lines 3 and 7, which are the reason for the vulnerability.

Listing 2: Unsafe password checking sample

```
1  boolean pwcheck_unsafe(byte[] pub, byte[] sec) {
2     if (pub.length != sec.length) {
3        return false;
4     }
5     for (int i = 0; i < pub.length; i++) {
6        if (pub[i] != sec[i]) {
7           return false;
8        }
9     }
10    return true;
11 }
```

In order to avoid the timing side-channel, it would be necessary to iterate over the complete public input without having such early returns.

Non-interference. If there is no possibility that a potential attacker can infer any secret information by an observation of the system, then the program would be considered *secure*. This intuitive property is called *non-interference*.

One way to check this property is *self-composition* [106]. The high level idea of *self-composition* is to reduce the problem of information flow analysis to the analysis of two copies of the same program, where the secret inputs are different, but the public values stay the same. The program analysis then checks whether these two copies create the same observation. More formally, let P be a program, and $P[\![pub, sec]\!]$ denote the execution of the program P with inputs pub and sec. As it is common in the security literature, the input is broken down to a tuple of public (low) values and secret (high) values. Furthermore, let

$c(.)$ denote the evaluation of a program execution with respect to a particular cost function representing the resource usage (e.g., execution time or response size) of the program. The non-interference requirement can then be formalized as follows:

$$\forall pub, sec_1, sec_2 : c(P[\![pub, sec_1]\!]) = c(P[\![pub, sec_2]\!])$$

The property states that any two secrets are *indistinguishable* through the side-channel observations and therefore can not be revealed by an attacker.

Coming back to the example of the password checking algorithm (cf. Listing 2). Such an analysis based on self-composition would consider one public input (the password guess by the attacker) and varies the secret input (the server-side password). The only difference for these two execution setups is the difference in the secret input, and hence, every difference in the observations is caused by the difference in the secret input. Therefore, if there is a difference in the observation for these two execution setups, then it would mean that the observation depends on the secret input and that the potential attacker can observe a different behavior for different passwords. Such an observation could eventually help her/him to retrieve information about the server-side password [122, 123].

ϵ-bounded non-interference. In practice the non-interference requirement might be too strict for the side-channel analysis of real-world programs. Particularly for timing channels, small differences in computations may be usual but by an attacker imperceptible. Recent research in side-channel detection [104, 110] have observed this problem and proposed the checking of an ϵ-bounded non-interference in [110]: not only programs with zero interference can be accepted as secure, but also programs where the difference between observations is too small (below a threshold ϵ) to be exploitable in practice. The corresponding formalization looks like:

$$\forall pub, sec_1, sec_2 : |c(P[\![pub, sec_1]\!]) - c(P[\![pub, sec_2]\!])| < \epsilon$$

The above property can be checked by enumerating all possible combinations of public and secret values, and measuring all possible resource usage metrics along corresponding the program executions. However, this becomes quickly infeasible for real-world programs.

Please note that the mentioned dependency between observation and secret input is only correct under the assumption that there is no other influence on the program behavior besides the given inputs, which might be not true when for example the program uses some seed to randomize its non-functional behavior to avoid side-channel attacks. In such a case the presented technique would be less effective in detecting the vulnerabilities, also because they are less exploitable (also check the paragraph below that explains the underlying attacker model). Please further note that such analysis only provides the identification of the vulnerability. It is not clear whether such a vulnerability can be exploited, i.e., whether a potential attacker can create an attack strategy to actual gain information about the secret. Nevertheless, the vulnerability can be identified and should be considered dangerous as long as it is unclear whether it can be exploited.

Attacker model. The attacker model defines which assumptions are made about the abilities of a potential attacker. Therefore, these are the assumptions made for this research work, which are similar to the assumptions by state-of-the-art previous work on side-channel detection [104, 110]:

- The program is deterministic and the side-channel measurements are precise.

- The attacker can not observe anything else besides the side-channel information. This also means that the attacker does not use the main-channel to infer information.
- Any variations measuring resource usage are caused by the application itself. Therefore, the analysis is not focused on side-channels related to the hardware architecture or the physical environment.

These assumptions are realistic as stated in the related work [110]. For better illustration please consider the following attacker scenario, which has been similarly described by [110]. In a server-client scenario in a distributed environment, the attacker is physically separated from the victim application. Therefore, the attacker has no chance to observe any physical side-channel. For a network with encrypted communication, the attacker has no access the the content of the network packages like the actual content of sent messages. The attacker can only infer information based on the observation during the communication with the server, e.g., the response times and response sizes. Furthermore, due to the physical distribution between attacker and the victim application, the attacker cannot instrument the application to observe any hardware-level side-channel.

Side-channel detection based on static analysis. Two state-of-the-art side-channel detection techniques are Blazer by Antonopoulos et al. [104] and Themis by Chen et al. [110]. Both are based on static analysis.

Blazer [104] uses decomposition instead of self-composition to prove the absence of timing side-channels. They construct a partition of the execution traces by splitting the traces in secret-independent branches. They collect traces that have the same cost when having the same public input. Afterwards they show that each component in the partition fulfills the property that the running time of the traces depends on the public and not on the secret input. Therefore, they prove that the program does not include any timing side-channel. On the other hand, when their techniques cannot find such a partition, they conclude that the program is not secure.

Themis [110] introduces the notion of the already mentioned ϵ-bounded non-interference for the side-channel detection in realistic environment and proposes the usage of Quantitative Cartesian Hoare Logic (QCHL). Hoare Logic [10] proposes a formal system to reason about the correctness of programs. The key element in Hoare Logic is the Hoare Triple $P\ \{Q\}\ R$, which states the connection between precondition P, the program Q and the postcondition R. The idea is to annotate the program with assertions, i.e., expressions that represent the current program state, which can be later used to infer the postcondition from the precondition. The formal system of Hoare Logic defines several axioms and rules that can be used to derive new assertions. Cartesian Hoare Triples are formulas that relate to k-different program runs, and hence, allow to reason about multiple programs runs. Themis's QCHLs further allow to quantify the expression in the formulas, which enables the detection of resource-related side-channel vulnerabilities.

Side-channel detection based on symbolic execution. Păsăreanu et al. [121] propose the usage of symbolic execution and Max-SMT solving to detect and quantify side-channel vulnerabilities. Max-SMT attempts to find a satisfiable assignment for a weighted first-order formula while maximizing the sum of the satisfied clauses. They first use symbolic execution to extract the path constraints that depend on the public and the secret information. They partition the path constraints based on their resulting cost. Then they formulate a Max-SMT problem by constructing clauses for each cost observation and give each clause the weight 1. The Max-SMT solver returns then a value for the public input that maximizes the number of clauses, i.e., the number of different observation. The number of possible observations correlates with the information gain about the secret. Therefore, this methodology is used to generate a public input that leads to the maximum information leakage. Furthermore, Păsăreanu et al. [121] propose a multi-run analysis, where they generate a sequence of public inputs that lead to the maximum leakage, which corresponds to a multi-run attack. Additionally, by leveraging model counting Păsăreanu et al. can assess the uncertainty about the secret after certain number of side-channel measurements.

Based on the work of Păsăreanu et al. [121], Bang et al. [105] propose the computation of leakage specifically for programs that perform string manipulations. They focus on side-channels where the attacker can explore each segment of a secret, called side-channels with segmented oracles.

Other techniques like [122, 123] focus specifically on the synthesis of side-channel attacks with similar techniques to [105, 121], namely symbolic execution, Max-SMT solving and model counting.

Brennan et al. [107] propose a symbolic path cost analysis for the detection of side-channel vulnerabilities. Their approach called CoCo-Channel tries to detect paths in the control flow that differ with regard to their execution cost. First of all they detect branch conditions that depend on the secret input information by leveraging taint tracking. Then they decompose the control flow of the program to a set of nested loops and branch components. They assign symbolic cost expressions to each component based on their nesting relationships. By leveraging a constraint solver CoCo-Channel checks whether a difference in the cost is satisfiable.

Other side-channel detection techniques. Xiao et al. [128] propose Stacco that uses differential, non-directed fuzzing for the detection of side-channel vulnerabilities related to SSL/TLS protocol implementation.

Chothia et al. [111] and Kawamoto et al. [114] propose the usage of Monte Carlo sampling to perform statistical estimation of the information leakage in Java programs. Although they show promising results by providing quantitative measures, their techniques might be imprecise in practice. Other successful approaches use abstraction interpretation to identify side-channel vulnerabilities related to the application's cache [112, 117, 120].

Furthermore, Hawblitzel et al. [88] propose a complete, full-system verification approach to prove end-to-end security. Their technique includes the proving of non-interference of the information flow. However, it lies in the nature of a full-system verification that it does not scale to real-world programs.

SUMMARY – SIDE-CHANNEL ANALYSIS

The existing approaches for the detection of side-channels are strongly based on static and symbolic analysis. Therefore, they have scalability issues and depend on models to enable an effective analysis. This thesis proposes the usage of dynamic techniques like directed fuzzing for the detection of side-channels in combination with a novel variant of shadow symbolic execution.

2.4.4 *Robustness Analysis of Neural Networks*

Artificial neural networks (ANNs) belong to a class of statistical learning algorithms in machine learning [168]. They are inspired by the biological neural networks in brains and

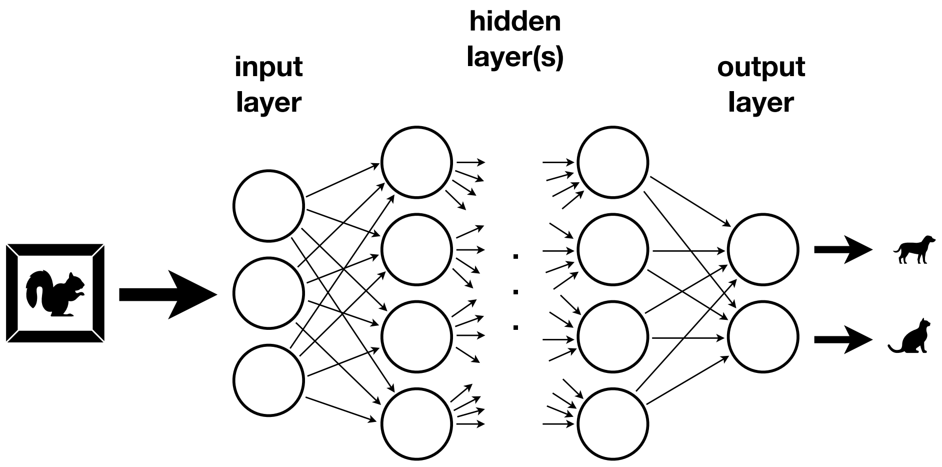


Figure 6: General architecture of a neural network.

represent organized networks of various neurons [167, 172]. In the remaining thesis ANNs are denoted simply as *neural networks* (NNs). Neural networks are used for the computation of information based on data, where the neurons represent the processing units. Similar to signal processing, the input data are processed in multiple stages up to the final output of the network. NNs are used for classification problems like pattern recognition in signals, images, and texts in areas like natural language processing [159], financial trading systems [188], and medical diagnoses [173]. Neural networks are also used for filtering and clustering of data [156, 166], and to synthesize game strategies [183]. This work focuses on neural networks for classification problems like image classification.

Figure 6 shows the general architecture of a neural network: The neurons are typically organized in multiple layers. There is an input layer that takes in the input data, and an output layer that represents the final outcome. In between there can be multiple hidden layers. A deep neural network (DNN) is a neural network typically with many hidden layers and is associated with deep learning strategies [195]. The neurons are connected to the neurons in the other layers, while each connection has an associated weight that determines the impact of the value at the source neuron on the value of the target neuron. Furthermore, the propagation of the value at a neuron for the next neuron can be influenced by so-called propagation/activation functions. The weights are learned during the training phase, for which there exist different training strategies depending on the actual application and available training sets.

In *supervised learning* the weights are learned with existing training data that consist of concrete inputs with their expected outputs. The learning process works in two phases: (1) forward- and (2) backward propagation. In the first phase the network simply executes the given test input with the current weight values (in the beginning they are usually initialized with random values). The generated output is compared with the expected output. The second phase modifies the weight values so that the error, i.e., the difference between expected and obtained output, is minimized.

Unsupervised learning follows the same principle, but instead of having test data with expected outputs, the testing data come with a cost function. The backward propagation would then modify the weights so that the cost function is minimized.

A general issue of machine learning (ML) techniques is their explainability [182]. For example in the case of neural networks the sequence of neuron activations and their weights hardly support the understanding of *why* the output is generated. Due to their limited explainability it is important to have techniques to thoroughly test neural networks. One type of testing is the robustness analysis of neural networks, which aims to determine how the network behaves when inputs slightly differ (in the ideal case only to an imperceptible degree). A robust network would be expected to still end up with the same conclusion/ classification for the original input and a slightly modified input [184].

This section summarizes current techniques for the testing of neural networks with special focus on the robustness analysis and the adversarial input generation.

Quantitative coverage metrics. Testing techniques like DeepGauge by Ma et al. [174], DeepXplore by Pei et al. [181], and DeepConcolic by Sun et al. [184] propose the usage of neural network specific quantitative coverage metrics to assess the test inputs and guide the test input generation. They introduce coverage metrics based on neuron-level coverage like basic neuron coverage, MC/DC coverage, and neuron boundary coverage, but also layer-level coverage criteria like top-k neuron coverage.

Sun et al. [184] further propose concolic testing on neural networks to increase the proposed coverage metrics. Additionally they post-process the resulting test suite and check whether there are pairs of inputs that are close to each other and lead to a different output. Therefore, they search for adversarial inputs that can be used to assess the robustness of the neural network.

Odena and Goodfellow [178] and Xie et al. [196] both introduce coverage-based fuzzing tools for neural networks. Their frameworks TensorFuzz [178] and DeepHunter [196] show good performance in generating test inputs for neural networks.

Kim et al. [170] propose a test adequacy criterion called *Surprise Adequacy for Deep Learning Systems* (SADL). SADL measures the difference in network behavior between the test input and the training data. Their intuition is that the test input should *not* trigger a completely different behavior than the training data. They propose the usage of SADL metrics that can guide the input generation for a more effective retraining of neural networks.

Techniques guided by coverage metrics can effectively generate test data for neural networks, but are not guided to a differential exploration or adversarial input generation.

Differential testing. Pei et al. [181] and Xie et al. [197] propose the differential testing of neural networks by comparing the results for different neural networks. Therefore, these techniques represent the differential testing of two implementations that can lead to the identification of adversarial inputs, although it is not the main purpose.

Pei et al. [181] introduce the whitebox framework DeepXplore that aims to find inputs that maximize the coverage and differences between different networks for the same classification problem. The neural networks therefore have similar functionalities. They formulate an optimization problem, which can be solved by gradient-based search techniques.

Xie et al. [197] introduce their blackbox genetic algorithm DiffChaser to reveal differences between a neural network and several optimized variants. Their technique can be also seen as regression testing approach for evolving networks.

Adversarial attacks. Adversarial input generation tries to find inputs that lead to a misclassification in the neural network but only slightly differ from a correct classified input. In the ideal case this small difference is actually not perceptible to a human tester.

Goodfellow et al. [160] propose the alternating training of two neural networks. The discriminative network D is trained so that it can correctly classify data from the training set and data generated by the other network G. The generative network G is trained to maximize the error in network D and is called the generative adversarial net.

Guo et al. [165] propose DLFuzz to apply differential fuzzing on neural networks. The employed mutations aim to generate inputs that increase the neuron coverage and the prediction difference between original input and mutated input. In contrast to traditional fuzzing where the mutations represent random changes in the input, DLFuzz formulates an optimization problem to identify small changes in the input that lead to increased coverage and differences in the prediction. Based on the obtained gradient they generate perturbations of the selected seed input.

Tian et al. [186] present DeepTest for the directed, systematic testing of neural networks with a special focus on the image processing in autonomous driving. They aim at revealing erroneous behavior while maximizing the neuron coverage with the generated test inputs. Therefore, they apply realistic image transformations to the input like rotating, scaling, but also simulating different realistic driving conditions with blurring and adding rain or fog. Their mutation strategies are based on expert knowledge of erroneous root cause, and hence, highly effective for specific application scenarios.

Other techniques like [158, 161, 171, 176, 185] try to trigger missclassifications by applying hardly perceptible perturbations, which they formulate as an optimization problem. Such hard optimization problems are typically approximated with L-BFGS [177] or expensive matrix manipulations [179].

Furthermore, there are techniques that aim to detect adversarial inputs at runtime [189]. Wang et al. [189] apply the suspicious input on a slightly mutated neural network and on the original neural network. Then they check whether the networks disagree on the classification. Additionally, there a defensive mechanisms like [180], which apply defensive distillation as a technique to train neural networks specifically to be more robust on perturbations in the input. Such techniques help to mitigate the consequences of adversarial attacks, but do not provide an actual differential analysis of neural networks.

Classical program analysis on neural networks. So far there are only a few works that apply classic program analysis techniques to neural networks. Ma et al. [175] transfer the concept of mutation testing on neural networks. With their technique DeepMutation they mutate neural networks, on which they execute the test data to assess their quality.

Gopinath et al. [162] propose DeepSafe as a technique that identifies regions in the input space, for which the neural network is known to be robust. These areas are called *safe* regions and mean that all inputs in these regions are classified verifiably correct. For the verification they leverage Reluplex [169], a special SMT solver for neural networks. In a later work, Gopinath et al. [163, 164] introduce DeepCheck that applies a lightweight symbolic execution approach to identify pixels in an input image that have a high impact on the classification process inside the neural network. They further use these *important* pixels to create adversarial 1-/2-pixel attacks.

SUMMARY – ROBUSTNESS ANALYSIS OF NEURAL NETWORKS

The existing work on robustness analysis misses hybrid techniques that are specifically driven to expose differential behavior. Differential fuzzing techniques like [165] and [186] are very promising as they can be very effective in generating adversarial inputs for a specific context. However, they miss their general applicability. Symbolic execution based techniques like [163, 164] and [184] are computational expensive and need better guidance. The testing technique proposed later in this thesis combines fuzzing with symbolic execution to overcome these problems. Nevertheless, the analysis of neural networks is a challenging task and remains a stress-testing situation also for a hybrid technique.

3 RESEARCH METHOD & CONTRIBUTION

This chapter discusses the research methodology and the core technical and conceptual contributions of this thesis. In the beginning it presents the assumptions taken for the presented approach, followed by the concept of *hybrid differential software testing*. Afterwards the chapter discusses the core technical and conceptual contributions. Finally, the chapter introduces the research questions that are used to evaluate the contributions and discusses the evaluation strategy.

3.1 ASSUMPTIONS

The general need for differential testing techniques has been already motivated in Section 1.1. The overall goal of the presented techniques in this thesis is to generate test inputs that expose behavioral differences. The assumption thereby is that existing tests, like a regression test suite, are not sufficient to expose the behavioral differences. This assumption is valid because existing work has shown that regression test suites need augmentation to cover the changed behavior [87, 92] as well as prioritization or minimization to make its execution feasible [87, 103]. The assumption is also valid for other applications of differential software testing that aim to discover security vulnerabilities or to show robustness issues. They represent *unexpected* behavior that is challenging to avoid [107, 110, 146, 186] and there are usually no existing tests to discover them. Therefore, there is a need to generate these inputs. Furthermore, the presented techniques are designed to be effective with only one valid seed input that is used to initially execute the application under test. They do not assume a large test suite to derive any information. This increases the applicability of the presented techniques because they only make little assumptions about the availability of existing tests for the application. The presented techniques allow multiple seed inputs, which makes it possible to use an existing test suite (or at least some selected inputs) to guide the differential analysis. However, this is not necessary.

3.2 CONCEPT: HYDIFF

The concept of **Hy***brid* **Diff***erential Software Testing* (HyDiff) is to combine powerful techniques to tackle the problem of test input generation to reveal *behavioral differences* in software. As introduced in Chapter 1 differential program analysis aims to find behavioral differences. This is a difficult problem because it generally requires to reason about multiple program executions. The discussion of the existing work on differential analysis (cf. Section 2.4) shows that there is yet no hybrid differential software testing approach. Furthermore, the existing single approaches for a differential analysis have their own limitations.

Fuzzing. As representatives of inexpensive, random-based exploration *fuzzing* techniques (cf. Section 2.1) can generate a large number of inputs in a short time period due to their low overhead. Therefore, fuzzing is known for its input generation *speed* [77, 81]. However, fuzzing is based on random operators and is usually implemented as blackbox or greybox

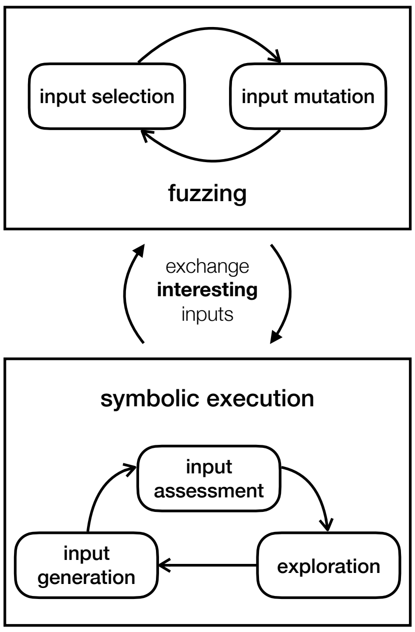


Figure 7: Conceptual overview of the parallel hybrid exploration.

technique. It lacks the knowledge about the program to go beyond complex constraints that guard deeper program behavior [81].

Symbolic Execution. Existing approaches based on *(dynamic) symbolic execution* (cf. Section 2.2) provide a systematic exploration of the state space, which can be guided by several heuristics. They have the full knowledge about the program, and hence, can unleash the full spectrum of program analysis techniques to, e.g., reach low-probability branches. However, the scope of symbolic execution is usually limited to smaller programs because its systematic exploration encounters the path explosion problem and expensive path constraints solving [77, 81]. Therefore, it does not scale to real-world applications.

Hybrid Analysis. This thesis proposes the usage of a *hybrid* approach, which combines random-based exploration (fuzzing) and systematic exploration (symbolic execution). The hybrid setup includes the exchange of interesting inputs between both single approaches that run in parallel. Therefore, both approaches can benefit from each other and the overall analysis can explore a larger state space, while quickly generating results.

In order to illustrate this idea, Figure 7 shows the overall concept of the proposed hybrid analysis. By running fuzzing and symbolic execution in parallel, both single techniques can perform their own exploration and can incorporate *interesting* inputs from the other component as well. This supports fuzzing to overcome narrow constraints in the program by importing inputs from symbolic execution, which is a whitebox technique that can analyze and solve these constraints with a constraint solver. Additionally, this hybrid concept also provides guidance for symbolic execution to focus on *interesting* program areas triggered by inputs from the fuzzing component.

The term *interesting* depends on the specific type of the differential analysis. For example in the context of regression analysis, an interesting input exposes a divergence between two program versions, while in the context of worst-case complexity analysis an interesting input maximizes the cost of the program execution. In order to enable a hybrid differential analysis, both single techniques need to be able to perform a differential analysis on their own. The existing techniques for fuzzing and symbolic execution (cf. Section 2.4) need to be adapted and extended.

Differential fuzzing (DF). The fuzzing process consists of a loop between *input selection* and *input mutation* (cf. the upper box in Figure 7). The input mutation applies various mutation operators to generate new inputs. The input selection determines which mutated inputs are reported as *interesting* and which are kept for following evolutions. In order to perform an differential analysis, the input selection is driven by *differential heuristics*. This general concept of *differential fuzzing* is not known yet, so that it is necessary to extend existing fuzzing techniques for a differential analysis. In contrary, the ability to synchronize with parallel fuzzing instances is already implemented in existing fuzzers (e.g., AFL [38]). This feature can be reused to exchange inputs with the symbolic execution component.

Differential dynamic symbolic execution (DDSE). The (dynamic) symbolic execution consists of a loop between *input assessment*, *exploration*, and *input generation* (cf. the lower box in Figure 7). The input assessment performs a concolic execution of concrete inputs, which includes the analysis of the executed branches based on differential heuristics. This analysis results in the identification and ranking of unexplored branches. The highest ranked, unexplored branch is used as starting point for some additional symbolic exploration. The resulting constraints of newly explored paths are extracted and used to generate new concrete inputs. These inputs are again assessed with concolic execution, which also reports interesting inputs for the fuzzing component. The concept of *differential symbolic execution* is already known (e.g., [85, 93, 96, 97, 102]), but does come with a couple of limitations and without the ability to incorporate new inputs during the analysis (cf. Section 2.4). The above presented concept of *differential* ***dynamic*** *symbolic execution* is a new dynamic symbolic execution technique for the differential analysis.

Contributions. Overall, this thesis makes the following contributions:

C1 The **concept of differential fuzzing** that incorporates various differential metrics to allow a general differential analysis. In particular it allows the search for side-channel vulnerabilities because it also uses cost metrics to determine the cost difference of two executions. Please note that the term *differential fuzzing* has been also used to describe the testing for output inconsistencies between two different implementations of the same application, e.g., two implementations of a PDF viewer, file parsers, or security libraries [34]. However, this represents a different type of analysis.

C2 The **concept of differential dynamic symbolic execution** as a technique to perform a dynamic symbolic exploration driven by differential heuristics, which allows to incorporate concrete inputs during the analysis. This allows the continuous guidance of the symbolic exploration to interesting program behaviors.

C3 The **concept of a general hybrid approach** in differential program analysis, which combines the strengths of single techniques in this research field. This concept closes a gap in the research of differential program analysis that is currently performed by specialized techniques with their own advantages and disadvantages. The hybrid concept allows to combine their advantages and compensate their disadvantages.

C4 The **concept of a hybrid setup** for applying fuzzing and symbolic execution **in parallel** as an alternative to already existing hybrid approaches [77, 81, 82] in test input generation. The parallel environment allows that both techniques can continue their own powerful differential analysis while being supported by the results of the other component.

3.3 RESEARCH QUESTIONS

The main research interest in hybrid differential software testing is to expose software bugs related to differential behavior. Finding such bugs is essential to improve the software quality in general. The four contributions C1-4, as described in the previous section, aim at providing efficient and effective techniques to contribute to this research interest. In particular, they aim at supporting software developers in creating reliable and secure software, and hence, also facilitate the main research idea in software engineering, namely to support software development with methods, techniques, and tools (cf. Chapter 1).

In order to validate the made contributions, the evaluation in this research investigates the following research questions:

RQ1 *How good is **differential fuzzing** and what are the limitations?*
This questions evaluates the contribution **C1**, the concept of differential fuzzing, and how it performs on differential software testing when applied in isolation.

RQ2 *How good is **differential dynamic symbolic execution** and what are the limitations?*
Analogous to RQ1, this question evaluates the contribution **C2**, the concept of differential dynamic symbolic execution, and how it performs on differential software testing when applied in isolation.

RQ3 *Can the **hybrid** approach outperform the single techniques?*
This questions evaluates **C3** and **C4** with regard to the basic intuition of the hybrid combination that it performs better than the single techniques (C1 and C2) on differential software testing.

RQ4 *Can the hybrid approach **not only combine** the results of fuzzing and symbolic execution, but also **amplify** the search itself and generate even better results than each approach on its own?*
This questions follows RQ3 to evaluate **C3** and **C4** in more detail. RQ4 investigates whether the components can benefit from each other and whether the hybrid approach can generate significantly better results.

RQ5 *Can the proposed hybrid differential software testing approach **reveal behavioral differences** in software?*
This question considers **all** contributions **C1-4** and evaluates whether the proposed techniques represent an important contribution to the overall research interest.

Summarized, the questions RQ1, RQ2, and RQ3 evaluate how the two components of HyDiff proceed and whether the combination of both is worthwhile. The fourth question, RQ4, evaluates whether the hybrid setup amplifies the exploration. The last question, RQ5, evaluates the general effectiveness of the proposed approach.

3.4 EVALUATION STRATEGY

The research questions are evaluated based on a quantitative analysis with experiments and benchmarks in specific application areas of differential program analysis:

A1 **Regression analysis** is one of the main applications of differential program analysis where the goal is to identify behavioral differences between two successive software versions. Regression analysis is also one of the "most extensively researched areas in [software] testing" [11] and is therefore the major focus of HyDiff.

A2 **Worst-case complexity analysis** searches for worst-case execution paths, which represent a serious threat to the system under test. Similar to the next application **A3** it is a highly relevant application in the security area. The current need for such techniques is also recognized by the US *Defense Advanced Research Projects Agency* (DARPA), which recently organized the *Space/Time Analysis for Cybersecurity* (STAC) program [204] that supported the development of new analysis techniques to identify algorithmic complexity and side-channel vulnerabilities. In comparison with the other applications worst-case complexity analysis is simpler because it does not require the reasoning about multiple execution paths at the same time.

A3 **Side-channel analysis** searches for information leakages that are caused by diverging cost-behaviors within the same application. As already mentioned for **A2**, side-channel analysis is highly relevant and in the focus of recent research projects. The popular *Meltdown* [118, 125] and *Spectre* [115] side-channel attacks also gave it some publicity outside the research community. Side-channel analysis is difficult because it requires the reasoning about multiple execution paths at the same time and additionally involves the handling of cost behaviors.

A4 **Robustness analysis of neural networks** is a relatively novel application of differential program analysis where the goal is to identify vulnerabilities in neural networks with regard to their robustness. It requires reasoning about multiple network executions, which makes it very expensive, and hence, serves as a stress testing application of HyDiff and its components.

State of the art in evaluation strategies. Various *fuzzing* techniques have been published in the recent years (e.g., [16, 17, 28, 32, 146, 35]) and the de facto standard evaluation approach is to perform experiments on a benchmark suite by comparing a baseline fuzzer with the new proposed solution. The evaluation subjects consist mostly of command-line tools with known vulnerabilities and the presentation of *newly* found vulnerabilities. The mostly used evaluation metric is the number of *crashes* found by the fuzzer. Fuzzing algorithms are based on random exploration, and hence, need some careful evaluation to make sure that measured effects are no coincidental behavior. Arcuri and Briand [14] present guidelines on how to evaluate randomized algorithms. A general concern addressed by their guidelines is to thoroughly describe the experiment setup. It is necessary to specify the numbers of conducted experiment runs as well as the employed statistical tests that have been used to draw conclusions from the data. They recommend to run each randomized algorithm at least 1000 times or to explain the reasons if this is not possible. "The objective is to strike a balance between generalization and statistical power" [14]. Therefore, it would be reasonable to have less experiment repetitions if the algorithms are evaluated on a large set of artifacts. The choice of artifacts thereby can have a large impact on the performance of the algorithms under test, and hence, requires a well-defined and well explained selection. Furthermore, Arcuri and Briand urge for the appropriate usage of statistical methods, like the Mann-Whitney U-test to detect statistical differences. Additionally, they recommend to report all available statistical data (means, standard deviations, p-values, etc.) to help the meta-analysis and to visualize them, e.g., with box plots.

Four years after the publication of Arcuri's and Briand's work, Klees et al. [27] studied recent publications of fuzzing techniques and assessed the conducted evaluations. They report that they have found problems in all evaluations, and hence, performed own experimental evaluations of fuzzers to gain more insights and deduce general guidelines for the assessment of fuzzing techniques. Important steps are amongst others to perform *multiple* runs with statistical tests to distinguish the distributions, to have benchmark subjects with known properties, the measurement of goal-specific metrics (like the number of identified crashes for most of the fuzzing techniques) instead of simple coverage metrics, and the plotting of the performance over time.

Symbolic execution techniques [54, 63, 93, 67, 97, 69, 102] are usually evaluated on several micro benchmarks and real-world applications. Typically different analysis types are compared by measuring code coverage metrics, the time to fail an assertion, the number of solver queries, or in general the time and memory cost for the exploration.

Hybrid approaches like [75, 77, 78, 81] focus as well on command-line tools. For example in the area of vulnerability detection hybrid approaches [81] are evaluated on subjects taken from the DARPA Cyber Grand Challenge (CGC) Qualifying Event data set [203], which already includes known vulnerabilities. The evaluation metrics combine usually the metrics from the components, i.e., in case of combining symbolic execution and fuzzing, it is often code coverage and the number of vulnerabilities/crashes found.

The state-of-the-art approaches in the specific application areas follow the evaluation approaches of the underlying techniques and focus their benchmark selection and evaluation metric design on the specifics of the application area. For regression testing there exists an regression benchmark for C programs, namely CoREBench [198], which includes subjects taken from repositories and bug reports of four open-source software projects: Make, Grep, Findutils, and Coreutils. The included subjects are known to have *real* regression errors. However, the related work in regression testing also uses subjects with seeded regression errors like in [85, 97, 102]. For worst-case complexity analysis, the fuzzing approach SlowFuzz [146] is evaluated one several textbook algorithms, like sorting algorithms or a hash table implementation, for which the worst-case complexity is known. Similarly, the symbolic execution approach SPF-WCA [145] focus on a micro-benchmark. As evaluation metric the actual slowdown of the application, i.e., measured execution time, is mainly used. For the side-channel analysis, the related work [104, 107, 110, 121] uses micro-benchmarks, textbook algorithms and subjects from the DARPA Space/Time Analysis for Cybersecurity (STAC) program [204], which are known to include side-channel vulnerabilities. For the robustness analysis of neural networks, the related works [157, 160, 181, 184, 187] use models generated based on popular data sets like MNIST [201] and CIFAR-10 [200]. The proposed techniques are evaluated based on coverage metrics like neuron coverage and the number of different identified classifications.

HyDiff's evaluation strategy. The evaluation strategy of HyDiff and its components aligns with the above described state of the art in evaluation strategies in the leveraged techniques and in the covered application areas. Therefore, for each application area the proposed hybrid differential analysis approach is evaluated on several micro- and macro-benchmarks, including real-world applications. Table 2 shows an overview of the data sets used in the benchmarks. For the evaluation of the application **A1 regression analysis** the subjects are taken from multiple versions of the Traffic collision avoidance system (TCAS) [208] as well as subjects from the Defects4J benchmark [199]. Additionally, HyDiff is evaluated on subjects from Apache Commons CLI library [202].

For the evaluation of application **A2 worst-case complexity analysis** the subjects are chosen based on the evaluation of the approach SlowFuzz [146], since it is the most related work in this context. The data set includes textbook algorithms like Insertion Sort and regular expression matching from the Java JDK. It also includes algorithms from the STAC program [204] and real-world applications, e.g., Apache Commons Compress [205].

The evaluation of application **A3 side-channel analysis** includes subjects from the evaluation of two state-of-the-art static analysis tools BLAZER [104] and THEMIS [110]. Their subjects consist smaller micro-benchmarks with JAVA programs up to 700 LOC as well as large, real-world applications up to 20K LOC. Additionally, the evaluation includes more subjects from the STAC program [204] and the modular exponentiation known from the RSA encryption technique [116].

The evaluation of application **A4 robustness analysis of neural networks** uses a neural network trained on the MNIST dataset [193]. In multiple experiment setups the model is investigated for its robustness with regard to changes in the input.

Table 2: Overview of the benchmark data sets.

Application	**Subjects**	**Source**
A1 Regression analysis	Traffic collision avoidance system (TCAS)	[208]
	Math-{10,46,60} and Time-1 from Defects4J	[199]
	APACHE CLI	[85, 202]
A2 Worst-case complexity analysis	Insertion Sort, QuickSort	JDK 1.5
	Regular expression matching	JDK 1.8, [207]
	HashTable, Image Processor	[204]
	APACHE COMMONS COMPRESS	[205]
A3 Side-channel analysis	BLAZER benchmark	[104]
	THEMIS benchmark	[110]
	Ibasys	[204]
	modular exponentiation: RSA_modpow	[121]
A4 Robustness analysis of neural networks	MNIST dataset	[193]

HyDiff's evaluation metrics. The evaluation of the research questions uses several metrics, which capture the interesting characteristics that differ with the specific application. The focus of applications A1 (regression analysis) and A4 (robustness analysis of neural networks) is the *difference* in the *output*. Therefore, the metrics for the experiments in these application contexts capture the number of obtained output differences as well as the time to the first identified output difference. The differences in the branching behavior (denoted as decision difference) is used as an additional indicator for behavioral differences. The following list summarizes the metrics used for the evaluation in A1 (cf. Section 7.2) and A4 (cf. Section 7.5):

- $\bar{t}$ +odiff: the average time to first output difference (lower is better)
- t_{min}: the minimum time (over all runs) needed to find the first output difference (lower is better)
- $\overline{\#\texttt{odiff}}$: the average number of identified output differences (higher is better)
- $\overline{\#\texttt{ddiff}}$: the average number of identified decision differences (higher is better)

The focus of application A2 (worst-case complexity analysis) is not on output differences or any kind of difference value because the analysis does not reason about multiple paths at once. The goal of the application is to maximize the *execution cost* with regard to a given cost function (e.g., number of executed instructions or memory usage). Therefore, the main characteristic to assess HyDiff and its components for A2 is the maximum execution cost triggered by the generated inputs. The following list summarizes the metrics used for the evaluation of A2 (cf. Section 7.3):

- $\bar{c}$: the average maximum cost obtained within the given time bound (higher is better)
- c_{max}: the maximum cost obtained over all runs (higher is better)
- $\bar{t} : c > 0$: the average time to find the first input, which improves the cost value with regard to the initial input as baseline (lower is better)

The focus of application A3 (side-channel analysis) is on different side-channel observations between execution paths. The side-channel information is measured as execution cost with regard to a given cost function, and hence, A3 focuses on the *cost difference*. The output and decision differences might be helpful to guide the search, but they cannot reveal a side-channel vulnerability. Similarly, the actual cost values of the execution paths are not as relevant as for A2 because the degree of difference (and not the single values) is essential. The following list summarizes the metrics used for the evaluation of A3 (cf. Section 7.4):

- $\bar{\delta}$: the average maximum cost difference obtained within the given time bound (higher is better)
- δ_{max}: the maximum cost difference obtained over all runs (higher is better)
- $\bar{t} : \delta > 0$: the average time to find the first input, which improves the δ value with regard to the initial input as baseline (lower is better)

Additionally, for each application the evaluation includes charts for the development over time of the measured *differential metrics*. This temporal perspective allows the discussion of the techniques with regard to the complete analysis time, in addition to the discussion at specific points in time.

3.5 SUMMARY

This thesis proposes HyDiff as a hybrid differential software testing approach, which combines random-based exploration with systematic exploration. The investigated research questions cover the investigation of HyDiff's components as well as of the hybrid combination. In particular this thesis investigates whether the combination of two differential exploration techniques does not only combine their results, but also amplifies their exploration. The proposed evaluation strategy focuses on the state of the art in the related work, and hence, follows a quantitative evaluation based on micro- and macro-benchmarks. Therefore, the wide field of differential analysis is separated into four specific application areas. The following chapters describe in detail the various components included in the proposed hybrid analysis, followed by the validation of the approach.

4 DIFFERENTIAL FUZZING

This chapter introduces *differential fuzzing* (DF) for the detection of behavioral differences and shows why state-of-the-art fuzzers cannot provide a differential analysis out of the box. The chapter starts with the explanation on how fuzzers can be guided for a differential analysis. It continues with a simple example and afterwards provides the discussion of the technical details. The chapter concludes with a preliminary evaluation of the presented differential fuzzing approach. Preliminary concepts and results for DF have been described in the following publications:

- Badger [3] proposes a differential fuzzing strategy for worst-case complexity analysis,
- DifFuzz [1] picks up the guided fuzzing idea by Badger and proposes a differential fuzzer for the identification of side-channel vulnerabilities, and
- HyDiff [6] improves a general differential fuzzer based on the gained insights in Badger and DifFuzz.

4.1 OVERVIEW

Fuzzing is a powerful technique to generate inputs that reveal errors (e.g., crashes) in programs. Recent fuzzing research efforts focus on optimizing the search process to find more crashes and cover more code [17, 28, 29, 32, 35]. The metrics to select new inputs from the mutated inputs for keeping in the mutation corpus is based only on the ability to increase the code coverage. Leveraging a fuzzer for differential program analysis appears to be interesting due to the fact that behavioral differences might be triggered by unexpected inputs, which is exactly what fuzzing is made for. Nevertheless, for a differential program analysis the current fuzzing approaches need to be significantly extended. The fuzzer should not only generate inputs for crashes or increased code coverage, but should specifically search for difference revealing inputs.

Example. An input that triggers a crash will be only interesting in differential program analysis if there is a difference recognizable, e.g., between two program versions: The old version does not crash, but the new version does crash for this input. In order to illustrate this problem the Listings 3 and 4 show two versions of a function with a fictive error. Listings 3 shows the function `abs_v1`, which fails for all values of `x > 3` (cf. line 5 and 6). Listing 4 shows an updated version of this function, now called `abs_v2`, in which the condition in line 5 changed so that the function fails for all values of `x > 2`. If a fuzzer identifies the input `x=4`, for which the function fails, then it would be in general interesting, but **not** for a differential analysis because both version would behave the same. The goal of a differential analysis would be to identify `x=3`, for which the old version does not fail, but the new version fails.

Listing 3: Crash Example v1.

```
int abs_v1(int x) {
   if (x < 0) {
      return -x;
   } else {
      if (x > 3)
         assert(false); // error
      return x;
   }
}
```

Listing 4: Crash Example v2.

```
int abs_v2(int x) {
   if (x < 0) {
      return -x;
   } else {
      if (x > 2) // changed condition
         assert(false); // error
      return x;
   }
}
```

In addition to coverage-based fuzzing, there are also efforts on directing fuzzing to specific program areas, e.g., with AFLGo [16]. However, such approaches cannot be explicitly targeted to differential behavior. Nevertheless, such guiding capabilities are crucial because hitting the areas of changed code is a key ability to find regression bugs.

The following meaning of *differential fuzzing* is used for this thesis, also to separate the following approach from the existing fuzzing techniques.

TERMINOLOGY – DIFFERENTIAL FUZZING

***Differential fuzzing* (DF)** means a guided fuzzing process, which aims at generating inputs that reveal behavioral differences.

The following approach focuses on coverage-guided, mutational fuzzing because the related work (see Section 2.1) has shown that such a search-based fuzzing approach is highly effective. Furthermore, it does not make strong assumptions about the existing testing artifacts, e.g., test suites or input grammars. Mutation-based fuzzing is built on a genetic algorithm, which belongs to the class of global search algorithms, known to be flexible, i.e., being able to overcome local maxima, and to scale up well to larger problems [47]. However, please note that the idea of differential fuzzing is not limited to this kind of fuzzing technique. In general, a mutation-based fuzzer can be guided by three parameters:

- the seed inputs,
- the applied mutation operators, and
- the selection mechanism.

Guidance by seed inputs. Guiding a fuzzer by its seed inputs requires existing inputs that already touch interesting areas of the program under test. Although in general an existing test suite can provide good seed inputs, it is a rather strong assumption that there are enough existing test inputs to sufficiently guide fuzzing. Additionally, a fuzzer will likely leave the areas touched by the seed inputs quite fast, based on the random mutation operators used in the fuzzing process. Therefore, a differential fuzzer should be able to make progress even without good seed inputs and at the same time it should have the possibility to incorporate new seed inputs during its fuzzing process.

Guidance by mutation operators. Guiding a fuzzer with its mutation operators is already implemented to some extend in the state-of-the-art greybox fuzzers. For example AFL implements mutation operators, which insert "known interesting integers" like 0, 1, or maximum values of data types like Integer.MAX_VALUE [39]. *Interesting* values means values that might lead to a crash or an exception in the program execution, e.g., the value Integer.MAX_VALUE might lead to an integer overflow during a calculation. Such mutation operators make sense when searching purely for crashes in programs, but might not be as efficient for a differential analysis. In differential analyses we need to guide the fuzzer first in interesting areas of the program, where actually a difference is occurring. Afterwards, it is of course interesting to trigger a crash. In best case the crashing behavior represents a difference, e.g., a crash in the new version while the old version terminates normally. Another way to use the mutation operators for fuzzer guidance is to select *which* parts of the input should be mutated with *which* operators. For example FairFuzz by Lemieux and Sen [28] first identifies rarely covered program branches, and then computes mutation masks to determine which input regions can be mutated with which mutation operators to still cover the rare branches. In particular, for each byte position in the input, they check whether the operators *flip byte*, *add random byte* and *delete byte* still produce inputs that cover the identified branch. Their experiments show that the computed mutation masks help to achieve a better program coverage. In a differential fuzzer it is also necessary to guide the fuzzing process to specific areas, e.g., where a change happened, but this is not enough because just reaching a changed area does not guarantee that a difference is found. There might be a long path left from the introduction of a change up to its observable difference.

Guidance by selection. Finally, the third option to guide the fuzzing process is the selection mechanism. In evolutionary algorithms, as they are used in fuzzing [17, 28, 38, 48], the mutant selection is the core guidance procedure. The selection procedure determines whether a mutant is kept in the mutation corpus, and hence, is reused for future mutations, or whether a mutant is eliminated. Therefore, a smart selection of mutants significantly helps to guide the fuzzer into interesting program behaviors. Typically, a fuzzer like AFL would keep inputs that produce program crashes, program hangs, or which cover new program branches because the goal is to find exactly these program behaviors. Inputs that produce crashes and hangs are moved into separate output folders and are not longer used for further mutations. They disturb the ongoing fuzzing process, which actually tries to maximize the program coverage. Only the inputs that increase the coverage are kept in the mutation corpus (cf. steps 5 and 6 in Figure 2). A differential fuzzer has slightly different needs: Inputs that lead to crashes and hangs should be still sorted out and not kept for further mutations. However, they should only be reported if they reveal behavioral differences. On the other hand, not only inputs that increase the branch coverage should be kept in the mutation corpus, but also inputs that, e.g., get closer to a change, show some output difference, or show some difference in its program exploration. Therefore, a differential fuzzer needs to be *guided* by a various set of differential metrics, like *output difference*, *decision difference*, *cost difference*, and the *patch distance*. Additionally, a differential fuzzer should also keep inputs that increase the program coverage, to further guide it into unexplored areas.

4.2 APPROACH

Figure 8 shows the overview of the proposed differential fuzzing technique. The overall workflow is similar to standard greybox fuzzing (cf. Figure 2 in Chapter 2.1). Similar as coverage-based, mutational fuzzing it starts with some initial seed inputs (see step 1 in Figure 8). It uses a queue (see step 2 in Figure 8) to store the current fuzzing corpus. In order to generate new mutants, it first trims the inputs (see step 3) and afterwards applies several mutation operators on the inputs (see step 4). The main difference to standard

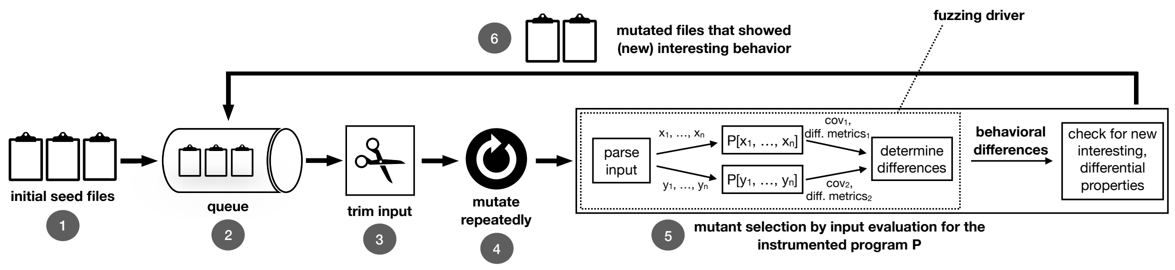


Figure 8: Conceptual overview of differential fuzzing.

greybox fuzzing is in the mutation selection mechanism (see step 5), which is specifically designed to select mutants that show new interesting behavioral properties. As shown in step 5 in Figure 8, it takes an input and parses it to extract the various parameters of the application, which are then used to evaluate the input on multiple program executions. Finally, the various observations are compared and the differences are determined. Overall, differential fuzzing keeps inputs that show new interesting behaviors for future mutant generation (see step 6).

4.2.1 *Mutant Selection Mechanism*

This section explains the introduced differential metrics and how they are used to select the mutants in the differential fuzzing process.

Output Difference. A program input, which triggers a difference in the output of a program, effectively forwards a change or a difference up to the end of the program execution, and hence, shows a clear observation for a behavioral difference. Therefore, a differential fuzzer needs to be able to compare the outcomes of the program under test depending on the result type. A crash represents a special output of a program, which needs also to be considered. In regression testing it is of particular interest, when the new version shows a crash while the old version does not show a crash for the same input. The output difference represents a binary value: there *is* a difference or there *is not* a difference. Inputs that show a difference are kept for further mutations. For some types of differential analysis (e.g., regression analysis) output differences already show the desired result of the overall analysis. However, they should be kept anyway for further mutations because some variations of these inputs may lead to other, different output differences. Additionally, the differential fuzzer needs to remember already observed output differences to eliminate duplicates.

Decision Difference. One can track the *decision history* of an input during program execution by collecting the branching decisions along the execution path. Two decision histories differ as soon as there is one decision pair, for which the choices do not match. Therefore, a *decision difference* is identified in a program location, where executions diverge, i.e., take different branches. This can be the case at for example `IF` conditions, `SWITCH` statements, or looping conditions. Such a difference does not necessarily mean a semantic divergence. For example in context of a program refactoring, the changed program might follow different branches, but at the end the same calculation can be executed. But still, a decision difference is an indication for a semantic difference, and hence, should be used as a differential metric. This information is stored as binary value and an encoded version of the difference is remembered to eliminate duplicates.

Listing 5: Input selection algorithm.

```
1  def boolean keep_if_interesting(Input i):
2
3     Result r = execute(i);
4
5     boolean hnb = r.has_new_bits();
6     boolean new_highscore = r.more_costly_than_highscore();
7     boolean output_diff = r.new_output_diff_found();
8     boolean decision_diff = r.new_decision_diff_found();
9     boolean patch_distance_improved = r.closer_to_patch();
10
11    if (hnb || new_highscore || output_diff || decision_diff || patch_distance_improved):
12       add_to_queue(i)
13       return true;
14
15    return false;
```

Cost Difference. In general, the *cost difference* means the difference in the resource consumption during program execution. The resource consumption can be also referred to as *execution cost* and can be measured by various types of metrics, e.g., execution time, memory consumption, power consumption, or any user-defined cost function. In the context of side-channel analysis, the cost difference is the key metric to identify side-channel vulnerabilities and the goal is to maximize the cost difference for a specific input configuration. In general, like the decision difference, the cost difference does not necessarily represent a semantic divergence, but similarly it is an indication.

Patch Distance. When having changes (also called patches) in a program (e.g., in regression analysis), these changes need to be touched by an input to trigger a difference because they actually introduce the difference, even if a divergence can be only observed later in the program execution. Therefore, the differential fuzzer should be able to guide its exploration to such changes, and hence, try to minimize the *patch distance* during fuzzing. This metric has no indication at all for a semantic divergence, but reaching a patch distance of zero is eventually necessary to hit a semantic divergence, and hence, this metric is an important guidance factor especially in the beginning of the fuzzing process.

In the general setting, the fuzzer would then keep an input for further mutations if one of these differential metrics is evaluated positively. Listing 5 shows the basic algorithm for the selection mechanism on an abstract level. It takes an input as parameter, which is evaluated in line 3, i.e., executed with the actual program. During program execution several metrics are collected by instrumentation and output observation. The assessment for the coverage and differential metrics in lines 5-9 are conducted *statefully*, i.e., the fuzzer stores the so far observed properties, so that no duplicates are stored in the fuzzing queue. Only inputs that show an *interesting*, *new* behavior will be kept (cf. line 11 to 13). Whereat it is enough to improve *one* of the differential metrics to be considered as interesting.

4.2.2 *Input Evaluation*

As described in the previous sections, the differential fuzzing is guided by the mutant selection mechanism. In order to select an input, the fuzzer needs to evaluate it by executing it (cf. line 3 in Listing 5). A so called *driver* is used by the fuzzer to handle the input execution, which can be generally summarized in three main activities (cf. also the dashed area in step 5 in Figure 8):

1. *Parsing*: In general it is necessary to prepare the input for some specific input format of the application under test. Therefore, the first step in the driver is to parse the input and extract or convert the contained values to match the requirements of the application.

2. *Execution*: After parsing, the driver runs the application with the prepared input starting at a specific entry point.

3. *Information Collection*: In order to characterize the input, the driver collects various properties of the input including the differential metrics and, e.g., the coverage information. Some of these values can be collected during program execution by some program instrumentation, but some of them need to be calculated after program execution based on the outcome, e.g., the output difference.

The concrete driver functionality is specific for every application, but the basic structure is always the same and can be reused.

4.3 EXAMPLE

In order to illustrate the differential fuzzing approach and its requirements, please take a look at the program `foo` in Listings 6 and 7. These are two variants of the same program taken from one of the preliminary papers [5]: Listing 6 represents the *old* version and Listing 7 represents the *new* version of the same program. The *old* version (cf. Listing 6) of program `foo` takes an integer value as parameter. It first checks whether `x` is greater than 0 and updates `y` accordingly (cf. line 4 to 8). Afterwards the value of `y` determines the result of the program: for a value greater than 1 it returns 0 (cf. line 10 to 11), for a value equal to 1 or smaller equal to -2 it throws an assertion error (cf. line 13 to 15), and otherwise it returns 1 (cf. line 17).

In the new version (cf. Listing 7) the statement in line 5 has been changed from `-x` to `x*x` and there is an additional statement in line 10. This example is an artificial program, but represents the fix of an assertion error for input `x=-1` in the old version, but introduces a new assertion error for input `x=0` in the new version. Such type of buggy fixes is also called *regression bug*.

In order to apply fuzzing, it is required to implement a fuzzing driver (see Listing 8). This driver parses the input and calls the two program versions, while monitoring the execution. For the differential analysis the goal is to find a different behavior between the two program versions with the *same* input. Therefore, the driver reads only one value from the file, in this case an integer value, and executes both versions with the same parameter. The lines 4 to 15 in Listing 8 reads one integer value in a byte-wise manner. The lines 17 to 27 execute and monitor the old version, and the lines 29 to 40 execute and monitor the new version. The lines 42 to 47 summarize the results and calculate the values for the differential metrics (cf. Section 4.4).

The results of the differential analysis are as follows (the results are averaged over 30 runs and report the 95% confidence interval): After 30 sec the differential fuzzer found on average 3.43 output differences (+/- 0.20 CI) and on average 4.10 decision differences (+/- 0.27 CI). On average it took the differential fuzzer 2.40 sec to find its first output difference

Listing 6: Differential fuzzing subject: old version [5].

```java
int foo_v1(int x) { /* OLD VERSION */
   int y;

   if (x < 0) {
      y = -x;
   } else {
      y = 2 * x;
   }

   if (y > 1) {
      return 0;
   } else {
      if (y == 1 || y <= -2) {
         throw new AssertionError("assert(false)");
      }
   }
   return 1;
}
```

Listing 7: Differential fuzzing subject: new version [5].

```java
int foo_v2(int x) { /* NEW VERSION */
   int y;

   if (x < 0) {
      y = x * x; // CHANGE: expression -x to x*x
   } else {
      y = 2 * x;
   }

   y = y + 1; // CHANGE: additional statement

   if (y > 1) {
      return 0;
   } else {
      if (y == 1 || y <= -2) {
         throw new AssertionError("assert(false)");
      }
   }
   return 1;
}
```

Listing 8: Driver for differential fuzzing example.

```java
public static void main(String[] args) {

   /* Read one input from the input file for both program versions. */
   int input;
   try (FileInputStream fis = new FileInputStream(args[0])) {
      byte[] bytes = new byte[Integer.BYTES];
      if ((fis.read(bytes)) == -1) {
         throw new RuntimeException("Not enough data!");
      }
      input = ByteBuffer.wrap(bytes).getInt();
   } catch (IOException e) {
      System.err.println("Error reading input");
      e.printStackTrace();
      throw new RuntimeException("Error reading input");
   }

   /* Execute old version. */
   Mem.clear();
   DecisionHistory.clear();
   Object res1 = null;
   try {
      res1 = Foo.foo_v1(input);
   } catch (Throwable e) {
      res1 = e;
   }
   boolean[] dec1 = DecisionHistory.getDecisions();
   long cost1 = Mem.instrCost;

   /* Execute new version. */
   Mem.clear();
   DecisionHistory.clear();
   CFGSummary.clear(); /* Only record the distances for the new version. */
   Object res2 = null;
   try {
      res2 = Foo.foo_v2(input);
   } catch (Throwable e) {
      res2 = e;
   }
   boolean[] dec2 = DecisionHistory.getDecisions();
   long cost2 = Mem.instrCost;

   /* Report differences. */
   DecisionHistoryDifference d = DecisionHistoryDifference
      .createDecisionHistoryDifference(dec1, dec2);
   Kelinci.setNewDecisionDifference(d);
   Kelinci.setNewOutputDifference(new OutputSummary(res1, res2));
   Kelinci.addCost(Math.abs(cost1 - cost2));
}
```

(+/- 0.34 CI), the fastest run needed 1 sec. Table 3 shows the generated inputs that reveal differences (the inputs are taken from one of the 30 experiments). All experiments have been started with the same seed input, which included the value 100, for which both program versions behave similarly and return the value 0.

Table 3: Results for differential fuzzing example (t = 30 sec).

ID	Input x	Output foo_v1	Output foo_v2
1	−956, 301, 312	0	java.lang.AssertionError
2	−1, 107, 038, 973	0	java.lang.AssertionError
3	0	1	java.lang.AssertionError
4	2, 147, 483, 647	java.lang.AssertionError	1

The input with id=1 has the value `x=-956301312`. Due to an integer overflow in the *new* version, the result of the statement in line 5 (cf. Listing 7) is `y=0`, and so, the expression in line 15 will be evaluated to `true`, which triggers the assertion error in the new version.

The input with id=2 has the value `x=-1107038973`. Similar as before, due to an integer overflow the multiplication in line 5 in the new version results in a negative value `y=-578777591`, which also triggers the assertion error in the new version.

The input with id=3 has the value `x=0`, which represents the newly introduced assertion error as mentioned earlier without having any integer overflow.

The input with id=4 has the value `x=2147483647` (i.e., exactly the maximum integer value). Due to the integer overflow the calculation in line 7 in the old version (cf. Listing 6) results in `y=-2`, which triggers the assertion in the old version, but not in the new version because of the additional increment in line 10.

In summary, the differential fuzzer was able to identify the intended behavioral change that fixes the assertion error with the input id=4. Note that `x=-1` and `x=2147483647` lead both to the assertion error in the old version. More importantly, it identified the newly introduced assertion error for `x=0` with the input id=3. Additionally, since the fuzzer does not assume any constraints on integer overflows, like static analysis could for example do, it can also find inputs that trigger more errors caused by such overflows in calculations (cf. input with id=1 and id=2).

4.4 TECHNICAL DETAILS

The presented differential fuzzing concept is generally applicable and not limited to specific programming languages. However, the implemented tools for BADGER, DIFFUZZ and HYDIFF aim at the analysis of JAVA bytecode for evaluation purposes. All fuzzing components are built on top of the fuzzer AFL [38], also by extending the KELINCI [26] framework for JAVA bytecode fuzzing. AFL is one of the state-of-the-art fuzzing tools (see Section 2.1) and is used to perform the actual mutation generation. The input evaluation, i.e., the execution of the generated mutant, is forwarded to the KELINCI framework, which takes the input and executes it with an instrumented version of the JAVA bytecode. After execution KELINCI sends the collected metric values back to the custom AFL implementation, where it decides whether to keep the input or not (cf. Listing 5).

As a greybox fuzzer, AFL in its original version only considers the coverage information for the input selection. Therefore, AFL needs to be extended to incorporate the differential metric values in its input selection mechanism. It needs to read the information sent from the KELINCI framework and it needs to be extended to perform an evaluation like presented in Listing 5. As already mentioned, this evaluation is stateful, meaning that a mutated input needs to be assessed based on the already observed coverage and differences. Therefore, AFL needs to be extended to store the already seen output differences, decision histories, etc. to enable this stateful comparison. KELINCI allows to apply AFL on JAVA bytecode and does initially support only the collecting of the coverage information. For a differential analysis KELINCI needs to also collect the information about the differential metrics.

4.4.1 *Measuring Execution Cost*

The analysis of execution costs includes three different cost models, similar as shown in KELINCIWCA, the cost-guided fuzzing procedure in BADGER [3]:

- *Execution Time*: Via some instrumentation of the JAVA bytecode, e.g., with the ASM bytecode manipulation framework [40], it is possible to measure the execution cost as the needed execution time calculated as the number of executed instructions. This way of measuring the execution time, instead of using the actual wallclock time, is more robust because it does not depend on other processes running on the same machine.
- *Memory Consumption*: The memory consumption can be measured by regularly polling the total memory consumption of the program execution during the input evaluation. This will lead to the maximum memory consumption during program execution. Another way would be to measure the memory allocation with some instrumentation, similar as for the execution time, but this would not incorporate, e.g., the garbage collection mechanism in JAVA.
- *User-defined Cost*: User-defined cost additions need to be supported, so that the developer can add costs by a program annotation. For example this is for particular interest for the analysis of smart contracts, in which it is necessary to observe the value of single variables [3]. Furthermore, the user-defined cost option allows to combine any other cost values to one overall cost. This is used in DIFFUZZ [1], where the driver first measures, e.g., the execution time of two individual program executions, and then calculates the cost difference and sets it as user-defined cost.

Furthermore, the cost metrics can be extended quite flexible by extending the custom KELINCI implementation.

4.4.2 *Measuring Patch Distance*

In order to calculate the patch distance, it is necessary to first construct the interprocedural control-flow graph (ICFG), e.g., with Apache Commons BCEL [42]. As mentioned in Section 2.2, the ICFG represents a graph that connects the control flow throughout various function calls. In such a graph it is possible to calculate the shortest distance from any node to a target node, e.g., by using Dijkstra's shortest path algorithm [211]. Therefore, the ICFG can be used to calculate for each instruction the shortest distance to some specified targets in the program (e.g., the changed areas in the context of regression analysis). The fuzzer can be guided by this distance so that the exploration gets closer to a patch. The

Listing 9: Decision difference calculation.

```
1  def Pair<int,String> calc_decision_difference(boolean[] dec1, boolean[] dec2):
2     String encoding = "";
3     int smallerLength = Math.min(dec1.length, dec2.length);
4     int distance = 0;
5
6     for (int i = 0; i < smallerLength; i++):
7        if (!dec1[i] & !dec2[i]):
8           encoding += "0"; // both false
9        elif (dec1[i] & dec2[i]):
10          encoding += "1"; // both true
11       elif (!dec1[i] & dec2[i]) {
12          encoding += "2"; // false -> true
13          distance++;
14       else:
15          encoding += "3"; // true -> false
16          distance++;
17
18    distance += Math.abs(dec1.length - dec2.length);
19    return <distance, encoding>;
```

construction of the ICFG and the distance calculation can be done offline. The distance information can be stored within the instrumentation of the bytecode, so that the information is instantly available during program execution.

4.4.3 *Measuring Decision Difference*

The decision history can be collected during program execution via the instrumentation as well, namely by collecting which decisions have been made during program execution. For each program execution the instrumentation produces a boolean array representing the performed decisions. The decision difference (`ddiff`) can be calculated after the program execution at the end of the fuzzing driver by comparing the decision histories of two program variants. Listing 9 shows how to calculate the difference and also how to generate an encoding of the difference, which can be stored on the fuzzer side to avoid duplicates and to allow various decision differences with the same distance but different decisions. The algorithm takes two boolean arrays with the decision histories of two program executions. The loop in line 6 to 16 compares decision by decision up to the shorter decision sequence and increases the distance value for each mismatch. If the decision histories have different lengths, then the length difference will be added as additional distance at the end of the algorithm (cf. line 18). At the same time the algorithm produces an encoding of the difference for each decision comparison: "`0`" denotes that both decisions have been `false`, "`1`" denotes that both decisions have been `true`, "`2`" denotes that the first decision was `false` and the second decision was `true`, and "`3`" denotes that the first decision was `true` and the second decision was `false`. The hashcode of the generated String encoding will be stored on the fuzzer side.

Listing 10: Output difference calculation.

```java
boolean isDifferent(Object o1, Object o2) {

   /* Check for different null values. */
   if (o1 == null && o2 != null) {
      return true;
   } else if (o1 != null && o2 == null) {
      return true;
   }
   /* Direct object comparison. */
   if (o1 == o2 || o1.equals(o2)) {
      return false;
   }

   /* Throwables: compare the String values of the messages. */
   if (o1 instanceof Throwable && o2 instanceof Throwable) {
      String o1str = ((Throwable) o1).toString();
      String o2str = ((Throwable) o2).toString();
      return !o1str.equals(o2str);
   }

   /* List: compare sizes and object in lists */
   if (o1 instanceof List && o2 instanceof List) {
      List<?> o1L = (List<?>) o1;
      List<?> o2L = (List<?>) o2;

      if (o1L.size() != o2L.size()) {
         return true;
      }
      if (o1L.isEmpty() && o2L.isEmpty()) {
         return false;
      }
      if (!o1L.get(0).getClass().equals(o2L.get(0).getClass())) {
         return true;
      }

      for (int i = 0; i < o1L.size(); i++) {
         Object o1LO = o1L.get(i);
         Object o2LO = o2L.get(i);

         if (o1LO == o2LO || o1LO.equals(o2LO)) {
            continue;
         }
         if (o1 == null && o2 != null) {
            return true;
         } else if (o1 != null && o2 == null) {
            return true;
         }

         /* Check String representations. */
         if (o1LO.toString().equals(o2LO.toString())) {
            continue;
         }

         return false;
      }
      return false;
   }

   /* For all other cases just compare the objects. */
   return !(o1.equals(o2));
}
```

4.4.4 *Measuring Output Difference*

After executing the program, the output difference (`odiff`) can be calculated by comparing the observed outputs. Listing 10 shows the algorithm for such a comparison, which is very Java specific. In general, it is necessary to check for all sort of output differences. In this example, the algorithm takes two Java objects and begins with null checks and a direct object comparison (cf. lines 3 to 12). Then it checks for different Exceptions/Throwables, i.e., different crashes (cf. lines 15 to 19). Afterwards it checks whether the outputs are lists and if yes, whether the content is different (cf. lines 22 to 57), which includes: checks for the length of the list (cf. line 26 to 31), check for the content type of the lists (cf. line 32 to 34), and a loop that compares every item in the list (cf. line 36 to 55). As soon as one of these specific checks reveals a difference, the algorithms will return the result. If no other check was able to determine the difference, the algorithm will finally return the result of the objects equals method (cf. line 60). The resulting boolean value will be send to the fuzzer. Additionally, the hashcode of each output object will be stored on the fuzzer side to remember the already seen combinations of output values.

Table 4: Results for the preliminary evaluation of differential fuzzing (DF) by comparing it with coverage-based fuzzing (CBF). The bold values represent significant differences to the other technique verified with the Wilcoxon rank-sum test ($\alpha = 0.05$).

Subject	**Coverage-Based Fuzzing (CBF)**				**Differential Fuzzing (DF)**			
(# changes)	$\overline{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$	$\overline{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$
TCAS-1 (1)	-	-	0.00 (±0.00)	0.00 (±0.00)	-	-	0.00 (±0.00)	0.00 (±0.00)
TCAS-2 (1)	-	-	0.00 (±0.00)	0.00 (±0.00)	**441.83 (±57.70)**	120	**0.70 (±0.23)**	**2.13 (±0.73)**
TCAS-3 (1)	596.87 (±6.04)	506	0.03 (±0.06)	8.10 (±0.32)	588.43 (±15.18)	392	0.10 (±0.11)	**38.63 (±1.96)**
TCAS-4 (1)	29.90 (±8.93)	5	1.00 (±0.00)	4.30 (±0.16)	28.47 (±10.42)	2	1.00 (±0.00)	**18.27 (±1.06)**
TCAS-5 (1)	241.93 (±79.35)	12	1.00 (±0.24)	8.70 (±0.32)	184.93 (±46.66)	24	**2.00 (±0.00)**	**31.97 (±1.06)**
TCAS-6 (1)	511.90 (±70.81)	9	0.17 (±0.13)	0.23 (±0.15)	**233.63 (±54.48)**	4	**0.97 (±0.06)**	**4.13 (±0.83)**
TCAS-7 (1)	-	-	0.00 (±0.00)	0.00 (±0.00)	-	-	0.00 (±0.00)	0.00 (±0.00)
TCAS-8 (1)	-	-	0.00 (±0.00)	0.00 (±0.00)	-	-	0.00 (±0.00)	0.00 (±0.00)
TCAS-9 (1)	-	-	0.00 (±0.00)	0.00 (±0.00)	**221.73 (±48.83)**	10	**1.00 (±0.00)**	**6.13 (±0.85)**
TCAS-10 (2)	337.50 (±80.72)	11	0.87 (±0.24)	1.60 (±0.39)	**173.47 (±46.27)**	1	**1.93 (±0.09)**	**12.27 (±1.69)**

4.5 PRELIMINARY EVALUATION

The baseline for the preliminary assessment is coverage-based fuzzing (CBF), i.e., fuzzing which is guided by coverage information and which is the current state-of-the-art approach for fuzzing (cf. Section 2.1). The goal is to show that the presented differential fuzzing approach performs significantly better in identifying behavioral differences than the baseline. The preliminary evaluation focuses on regression testing and uses ten subjects from the Traffic collision avoidance system (TCAS), which are originally taken from the SIR repository [208]. TCAS was used before in other regression testing work [102] and is available in several variants. The different versions are generated by mutation injection. The original program has 143 LOC and the used variants include 1-2 changes. In contrary to the differential fuzzing approach, coveraged-based fuzzing can only be applied on one version at once, therefore the experiments show the result for applying CBF on the new version. All experiments have been executed for 10 minutes and repeated 30 times. Table 4 presents the results for the conducted experiments. Each row represents the differential analysis between the original TCAS program and a generated variant. The first column declares the used generated variant and states the number of involved changes (cf. the number in the brackets). The other columns show the results for coverage-based fuzzing (CBF) and differential fuzzing (DF). The used metrics ($\bar{t}$ +odiff, t_{min}, $\overline{\#odiff}$, and $\overline{\#ddiff}$) have been already described in Section 3.4 and focus on the output difference (odiff) as well as the decision difference (ddiff). The highlighted values denote significant differences to the other technique verified with the Wilcoxon rank-sum test (with 5% significance level). The time values are presented in seconds and the values also report the 95% confidence intervals.

The results in Table 4 show that differential fuzzing (DF) performs mostly better or, in worst-case, similar as coverage-guided fuzzing (CBF). For the subjects 2 and 9 CBF cannot find any indication for a difference, while DF identifies output and decision differences. For the subjects 3, 4, 5, 6, and 10 both approaches identify the output differences. However, DF can find more output and decision differences and identifies the first output difference faster and more reliable. For the remaining subjects 1, 7, and 8, neither of the approaches can identify the injected changes. Overall, DF performs significantly better than CBF in finding behavioral differences. Nevertheless, the subjects 1, 7, and 8 show that also DF has its limitations and cannot identify all differences.

4.6 SUMMARY

This chapter introduced *differential fuzzing* (DF) as a method to identify behavioral differences with a guided mutational fuzzing approach. Standard coverage-guided fuzzing is extended by modifying the mutant selection mechanism (see Figure 8). The presented approach uses several differential metrics like output difference, decision difference, cost difference, and patch distance to assess the behavioral properties of the mutated inputs. The presented preliminary evaluation shows that differential fuzzing significantly outperforms coverage-guided fuzzing, although it still has its limitations. In the hybrid differential analysis these limitations will be mitigated by combining differential fuzzing with its counterpart: differential dynamic symbolic execution, which is introduced in the next chapter.

5 DIFFERENTIAL DYNAMIC SYMBOLIC EXECUTION

This chapter explains what *differential dynamic symbolic execution* (DDSE) means and why current symbolic execution techniques do not provide the necessary abilities for a general differential analysis. The chapter starts with the current advances on symbolic execution for regression analysis and their limitations. It continues with a detailed description of the differential dynamic symbolic execution approach followed by a simple example. Afterwards, the chapter continues with the technical details and concludes with a preliminary evaluation. Preliminary concepts and results for DDSE have been described in the following publications:

- Badger [3] proposes a dynamic symbolic execution framework specifically for worst-case complexity analysis that allows to incorporate concrete inputs during the exploration,
- Shadow$_{\text{JPF}}$ [4] and Shadow$_{\text{JPF+}}$ [5] provide implementations for shadow symbolic execution for Java, which are used as baseline for the implementation, and
- HyDiff [6] proposes a differential symbolic execution approach for the general usage of differential program analysis, which can be used in a hybrid setup.

5.1 OVERVIEW

Symbolic execution is well known for traversing the application in a systematic way. Under some assumption like that constraints can be solved in a reasonable time, or that third-party libraries calls can be analyzed or appropriate models are available, symbolic execution can efficiently generate test inputs to touch interesting program behavior. However, out-of-the-box symbolic execution has the limitation that it focuses on only one software version at once. A differential analysis, like regression analysis, is hence not possible with the standard symbolic execution approach.

TERMINOLOGY – DIFFERENTIAL DYNAMIC SYMBOLIC EXECUTION

***Differential Dynamic Symbolic Execution* (DDSE)** means a systematic exploration of the program's input space, characterized by symbolic values, which is specifically focused and guided on generating inputs that reveal behavioral differences.

In the last decade, a couple of approaches have been proposed to perform some sort of differential analysis with symbolic execution in the area of regression analysis [85, 93, 96, 97, 102]. Person et al. [97] proposed directed incremental symbolic execution (DiSE) to characterize the effect of program changes. DiSE guides the symbolic execution on the new program version by exploring only paths that can reach a changed location. Therefore, they use the notion of *decision difference*: If the same input follows different paths in the old and the new program respectively, and hence the resulting path conditions differ, then the path condition from the new program is called *affected*.

Yang et al. [102] introduced memoized symbolic execution for the efficient replay of symbolic execution. Similar to DiSE [97], it can be used for regression analysis, so that only the program parts gets re-executed, which are affected by a change. Both approaches consider only the guided execution of the new version, and hence, the identification of differences always needs a post-processing and comparison with the old version. Besides from the reachability of changes, there is no guidance in the direction of real divergences.

In a previous work, Person et al. [96] propose an approach called *Differential Symbolic Execution*. Instead of guiding symbolic execution by the reachability information of changes in the source code, it is based on equivalence checking of method summaries. With symbolic execution they generate summaries of the methods in two program versions and compare these summaries in terms of various notions of equivalence incorporating black-box behavior, or also white-box behavior. While being a general approach for computing program differences, the computation of method summaries can involve scalability issues.

Böhme et al. [85] propose partition-based regression verification (PRV) as an incremental way to verify the input space of two different program versions. PRV represents a specialized approach for patch verification and uses dynamic symbolic execution to calculate input partitions, which get classified as *equivalence-revealing* or *difference-revealing*. PRV represents an alternative to regression test generation techniques, while it also can get guarantees about the verified partitions when only a partial analysis is possible. Although using a dynamic analysis, PRV separately execute the different program versions, and hence, might miss chances to prioritize paths early. Similar as DiSE it also applies only on the new version, and hence, needs the re-execution of paths to check for divergences.

Palikareva et al. [93] propose *shadow symbolic execution* of a change-annotated program. They leverage dynamic symbolic execution based on concrete test inputs to identify divergences and bounded symbolic execution to further search for discrepancies starting from the divergence points. Shadow symbolic execution provides a crucial step into scalable regression analysis, but its results depend on the quality of the concrete inputs, i.e., it might miss divergences because the concrete inputs do not trigger them.

In summary, the recent advances in symbolic execution for regression analysis provide the basis for a general differential testing approach, although none of them provide all necessary aspects. They either perform their analysis only on the new version, rely too much on concrete inputs for its guidance, or suffer from scalability issues, which makes it hard for a practical application. Moreover, all of them focus on regression analysis and not on a general differential analysis.

For a scalable, general, and differential analysis it is necessary to incorporate concrete inputs, i.e., it needs a dynamic approach, to drive the exploration in interesting program areas. Furthermore the differential analysis should be able to analyze multiple program versions at the same time to simplify constraints and prioritize paths early that show the best chances to reveal divergences. Additionally, the analysis should be guided by syntactic information about the changes in the program, so that paths can be pruned efficiently.

5.2 APPROACH

Shadow symbolic execution [93] proposes the exploration of change-annotated programs (cf. Section 2.4.1), which represents an elegant way of combining multiple versions, or allowing multiple *differential* behaviors in one execution. As described in [5], shadow symbolic execution [93] uses concrete inputs to drive the differential analysis, which makes it scalable, but at the same time it strongly depends on these concrete inputs, which let

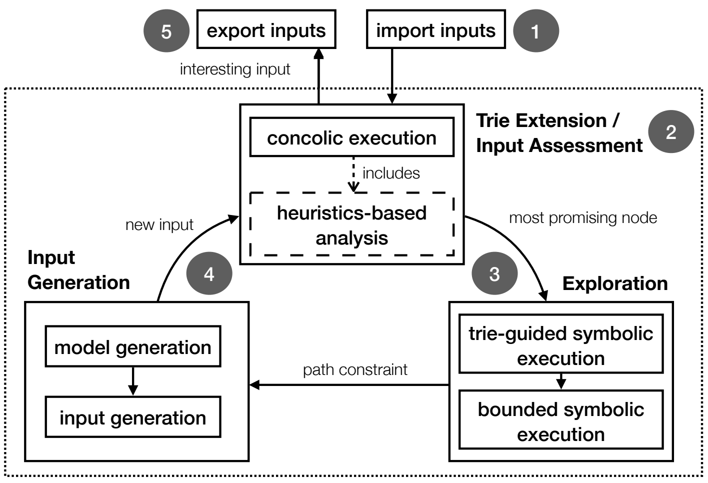


Figure 9: Conceptual overview of dynamic, heuristic-driven symbolic execution; based on [3].

shadow symbolic execution might miss important divergences. For example *deeper* divergences might be missed in the bounded symbolic execution phase because the concrete inputs imply constraints on the path conditions, which prevent the satisfiability of diff-paths. Additionally, the concrete inputs need to cover not only the *changed* program locations, but also the actual divergence points. Otherwise the shadow symbolic execution cannot detect them. The proposed approach in [5] *complete shadow symbolic execution* explores the usage of the four-way forking idea (cf. Section 2.4.1) in standard symbolic execution, without having any concrete inputs to drive the exploration. The experiments in [5] showed that indeed this analysis does not miss divergences, as long as the program can be explored exhaustively, but of course comes with its own scalability issues. Therefore, the presented approach in HyDiff [6], allows the usage of concrete inputs to drive the exploration, but still uses a complete four-way forking approach to detect all divergences in the search space. Figure 9 shows the overview of the proposed differential dynamic symbolic execution approach, consisting of five phases: (1) import of inputs, (2) input assessment, (3) exploration, (4) input generation, and (5) export of inputs.

All symbolic execution variants presented in Figure 9, i.e., concolic execution, trie-guided symbolic execution, and bounded symbolic execution, support the execution of a change-annotated program. All together, denoted with the dashed area in Figure 9, form the so called *differential dynamic symbolic execution*.

(1) Input Import. The process starts with *importing* initial seed inputs (cf. step 1 in Figure 9). Note: The term *importing* inputs refers to the fact that such an import can not only be performed in the beginning of the analysis but also periodically throughout the whole process. This functionality is crucial for the synchronization with another technique in a hybrid setup.

(2) Input Assessment. The given inputs are executed concolically, i.e., the symbolic execution follows only the path of the concrete values but collects all symbolic information (i.e., the value mapping and the path constraint) along this path (cf. step 2 in Figure 9). The execution is mapped to a simplified symbolic execution tree called *trie*, which stores nodes for all involved conditions with symbolic variables (similar as Yang et al. [102]). After the concolic execution of the given inputs, each node, which has unexplored branches, represents a potential entry point for further exploration. The nodes are analyzed and ranked with the defined *heuristics*, which is followed by the selection of the most promising node.

(3) Exploration. The idea behind the expanded exploration step (cf. step 3 in Figure 9) is to discover new, interesting parts of the state space. In order to reach the actual symbolic state at the selected node we start with a *trie-guided* symbolic execution and switch to a bounded symbolic execution as soon as we hit the selected node. *Trie-guided* means that the symbolic execution simply follows the choices stored in the trie without any invocation of a constraint solver. This step is very efficient and builds the symbolic state. As soon as hitting the selected node, the execution switches to a bounded symbolic execution mode, which will perform an exhaustive symbolic execution up to a pre-defined bound. For example in the later presented experiments, the bounds have been set to 1, i.e., the exploration will always look one choice further as the selected node. The selection of the bound value represents a trade-off between a deeper exploration from a specific node and the broader exploration from the heuristic perspective. Therefore, choosing a very large bound would generate a lot of inputs based on one promising node. On the other hand choosing a very small bound would generate less number of inputs but more focused on the given heuristics because after each input generation the nodes will be re-assessed. The exploration step results in a sequence of satisfiable path conditions.

(4) Input Generation. After generating the path constraints in the previous step, it is necessary to generate the actual inputs, which satisfy the path constraints (cf. step 4 in Figure 9). Therefore, an SMT solver is leveraged to generate a model for each path constraint. Afterwards, these models are used to constructs inputs. The input generation is application-specific since the path constraints and their model have no information about the actual input formats and requirements. For example, the application under test could need an image file in JPEG format, then the input generation would have to build a JPEG image based on the model, which would, e.g., contain the actual pixel values.

Note that the inputs have been generated based on the exploration of a promising node determined by heuristics. This means that after the generation the inputs need to be assessed for their actual usefulness for the current analysis. Therefore, they are executed concolically (cf. step 2 in Figure 9), and the trie is extended.

All together, the steps 2, 3, and 4 form an analysis loop (cf. dashed area in Figure 9), which can be paused for the import of further inputs (e.g., in the hybrid setup), or which can be stopped by a user-specified bound, or which finishes after the complete exploration.

(5) Input Export. As soon as a generated input is assessed as interesting, i.e., it shows some new behavior interesting for the current analysis, it is reported for the export (cf. step 5 in Figure 9). In a hybrid setup the exported inputs are made available for the other technique; in a single analysis setup, the exported inputs represent the output of the analysis.

5.2.1 *Central Data Structure: Trie*

As already mentioned, the central data structure in this dynamic symbolic execution is a so-called *trie*, which has been adapted from Yang et al. [102]. A trie represents a subset of the symbolic execution tree, where nodes represent the choices during symbolic execution that include symbolic variables. Therefore, a trie is a simplified variant of a symbolic execution tree, where only the components are included, which are interesting for the analysis and which are necessary to replay specific paths in the tree. A trie has a *root* node (illustrated in blue, see Figures 10 and 14), which represents the first decision point in the program. A *leaf* node in the trie represents the last choice that happened during the execution of a specific path. There are different leaf node types:

- *regular* leaf nodes (illustrated in gray), i.e., the execution path finishes without any error,
- *error* leaf nodes (illustrated in red), i.e., the execution path finishes with an error (e.g., an exception was thrown), and
- *unsatisfiable* leaf nodes (illustrated in yellow), i.e., choices that are not feasible.

In addition, there can be various *intermediate* nodes, which represent choices and new decision points at the same time. In Yang et al. [102] this data structure is used to replay parts of the trie, which have been changed between two program versions. In our approach it used to store the current state of the analysis (cf. step 2 in Figure 9) and to select a promising point to continue the exploration (cf. step 3 in Figure 9).

Example. Consider the program `foo` in Listing 6 from the previous chapter (also shown on the left side in Figure 10). There are three decision points: `(x < 0)` in line 4, `(y > 1)` in line 10 and `(y == 1 || y <= -2)` in line 13. The corresponding bytecode representation of program `foo` is presented in the middle of Figure 10. The directed graph on the right side in Figure 10 shows the trie for the concolic bytecode execution of program `foo` with `x=100`.

Note that the trie is constructed incrementally during concolic execution. Each choice, which includes symbolic values, is captured and a corresponding node is created. The ids represent therefore the node creation order.

The trie nodes with id=0 and id=1 show the decision points traversed during the concrete execution path, and show the information about the location in the source code (e.g., for id=0 it is line 4) and the specific bytecode opcode (e.g., for id=0 it is bc=156, which denotes the IFGE bytecode operation, cf. label `1` in the bytecode representation).

The trie nodes with id=1 and id=2 show the choices made at the decision points at the previous nodes. The choice=1 in the node with id=1 means that the bytecode operation IFGE (from line 4, bc=156) evaluated to `true`, i.e., $x \geqslant 0$ and the input takes the `false` branch in the source code. The choice=0 in the node with id=2 means that the bytecode operation IF_ICMPLE (from line 10, bc=164, cf. label `16` in the bytecode) evaluated to `false`, i.e., $y > 1$. The input takes the `true` branch in the source code and returns 0.

The complete trie for a standard symbolic execution of the program `foo` is presented in Figure 14 and includes three regular leaf nodes (id=2, 7, and 5), one error leaf node (id=9) and three unsatisfiable leaf nodes (id=10, 11, and 12).

In comparison to a common symbolic execution tree, the trie does only contain the information that are necessary to (1) select a most promising node for further exploration (if the trie is not complete yet), and (2) to be able to replay the path to a selected node as efficient as possible. This excludes all nodes from the symbolic execution tree which are no decision points or which handle no symbolic information. Especially, all assignments can be excluded because they do not influence the branching behavior.

Trie-guided symbolic execution. Yang et al. [102] defines a trie-guided symbolic execution, which is adapted in the presented approach (cf. step 3 in Figure 9). In order to reach a specific node for further exploration it is necessary to retrieve the necessary symbolic state. This can be done by replaying the path from the root node to the selected node. However, it is not necessary to actually re-execute the path because symbolic execution can be efficiently guided by the information in the trie.

As first step, the choices in the trie, which will lead from the root node to the selected node, need to be determined. This *path finding* phase, is performed backwards from the selected node to the root node by simply highlighting every parent node (cf. Listing 11).

```
1  int foo_v1(int x) {
2     int y;
3
4     if (x < 0) {
5        y = -x;
6     } else {
7        y = 2 * x;
8     }
9
10    if (y > 1) {
11       return 0;
12    } else {
13       if (y == 1 || y <= -2) {
14          throw new AssertionError("assert
              (false)");
15       }
16    }
17    return 1;
18 }
```

```
 0: iload_0
 1: ifge          10
 4: iload_0
 5: ineg
 6: istore_1
 7: goto          14
10: iconst_2
11: iload_0
12: imul
13: istore_1
14: iload_1
15: iconst_1
16: if_icmple     21
19: iconst_0
20: ireturn
21: iload_1
22: iconst_1
23: if_icmpeq     32
26: iload_1
27: bipush        -2
29: if_icmpgt     42
32: new           #16
35: dup
36: ldc           #18
38: invokespecial #20
41: athrow
42: iconst_1
43: ireturn
```

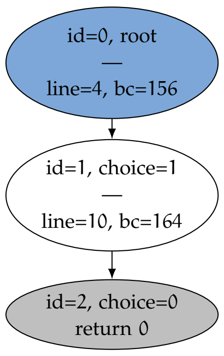


Figure 10: Left: program foo taken from Listing 6, Middle: corresponding bytecode, Right: corresponding trie representation for the initial input x=100.

Listing 11: Trie path finding procedure.

```
def void highlightPathToNode(TrieNode selectedNode):
   selectedNode.highlight();
   TrieNode currentNode = selectedNode;
   while (currentNode.hasParent()):
      currentNode = currentNode.getParent()
      currentNode.highlight()
```

This procedure generates a path consisting of highlighted nodes from the root node to the selected node. This means that for every decision point (i.e., an intermediate node in the trie) there is one determined choice (i.e., one highlighted child node).

Afterwards forward symbolic execution can be started from the root node and guided by the choices of the highlighted nodes. Whenever symbolic execution hits a decision point that includes symbolic values (i.e., the decision cannot determined concretely), then the concrete choice can be retrieved by probing the highlighted child of the current node. Note that the previously explained path finding algorithm ensures that there is at most one highlighted child per trie node. Therefore, the constraint solving can be turned off: only one choice is possible at every decision point and it is clear that this choice is feasible because it was already observed by a prior concolic execution run. Finally, the trie-guided symbolic execution will be aborted as soon as the selected node is reached.

5.2.2 *Differential Exploration Strategy*

The most crucial part in DDSE is to determine the exploration strategy of the unexplored choices in the trie, i.e., how to select the next node, ergo which is the currently most promising node for further exploration (cf. edge from step 2 to step 3 in Figure 9). The main advantage over the differential fuzzing part is that symbolic execution is applied on the

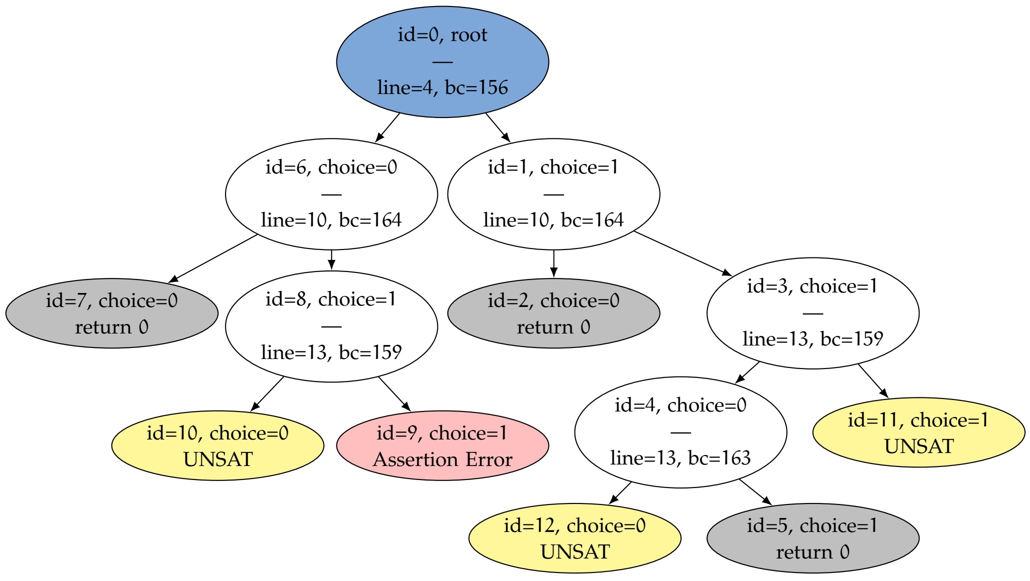


Figure 11: Complete trie for the program `foo` from Listing 6.

change-annotated program. The change-annotations can be used for path pruning based on the reachability information of the changed locations, but more important, each executed change-annotation introduces a so-called *differential expression*, or shorter: *diff expression*.

Differential Expression. These expressions are the key elements to handle two program executions at once; and furthermore, the *differential exploration strategy* is driven by these differential expressions to find paths where the control-flow diverges across executions, also called *diff paths*. A differential expression consists of four parts: the old symbolic value, the old concrete value, the new symbolic value, and the new concrete value. For example the statement `y=change(-x,x*x)` in line 5 in Listing 12 (with the assumption that `x` holds the symbolic value α and the concrete value 100) assigns the following differential expression to the variable `y`: { old_{sym}= -α, old_{con}= -100, new_{sym}= $\alpha * \alpha$, new_{con}= 10000 }.

Decision Difference. The information about whether the path condition at a trie node contains a *diff expression* is particular interesting because it is a good indication for a potential future divergence since having a *diff expression* is the requirement for finally reaching a *diff path*, i.e., to detect a *decision difference*. Therefore, this information is used to rank the nodes. Furthermore, it is interesting whether the node is already on a *diff path* because then there is already a concrete input, which triggers the decision difference, and it might be worthwhile to focus first on other trie nodes.

Although the focus of DDSE is on the decision difference because the change-annotations provide here a special benefit, the following *differential* metrics are sill collected for each trie node: *cost difference* and *patch distance*.

Cost Difference. As long as a node is not yet on a diff path, the node is assigned the difference in the *execution cost* for the changed behavior. For example for regression analysis, it is the cost difference between the old and the new version. Similar as for fuzzing (cf. Section 4.2.1), an increased cost difference is an indication for a differential behavior.

Patch Distance. As long as a node has not touched a patch yet, the *patch distance* is computed as the distance to the change-annotations in the program. Afterwards, a node is marked accordingly. The patch distance is mainly used to drive the exploration to change-annotations, and hence, a node with smaller distance will be prioritized. Additionally, these information are also used to prune all paths that cannot reach any change-annotation in the program, i.e., where the patch distance is not defined or infinity.

Output Difference. In contrast to differential fuzzing, the DDSE component cannot use the *output difference* as a search metric, since the execution of *diff paths* is limited to the new version, and hence, the full information about the output is not always available. However, the intrinsic goal of DDSE is efficiently explore diff paths, i.e., to identify *decision differences*.

Exploration Heuristics. To summarize the above presented ideas on differential metrics, the following *heuristics* are used to rank the nodes for exploration:

1. Prioritize nodes that contain a differential expression, but are not yet on a diff path.
2. Prioritize a node without differential expression before a node which is already on a diff path. (Note: here we only have nodes that can reach the changes).
3. Prioritize new branch coverage.
4. If two nodes have not yet touched any change, then prioritize the node with smaller distance.
5. Prioritize nodes that already have higher cost differences.
6. Prioritize higher trie nodes.

The highest priority is to find *decision differences*, i.e., divergences of the control-flow. Therefore, DDSE favors such potential nodes (points 1 and 2). It is the most valuable divergence metric, also because the output difference cannot be encoded. It can be simply detected by checking whether we are currently on a *diff path*.

The next priority is to support the fuzzing component in a hybrid setup during exploration, for which it is necessary to solve constraints corresponding to conditions that are difficult for the fuzzer (point 3). As further indications for a difference the information about the *patch distance* and the *cost difference* (point 4 and 5) are used. As last search parameter higher nodes in the trie are favored, which leads to a broader exploration of the search space, which also supports the hybrid exploration. These heuristics represent the default configuration setup for the presented differential analysis They can be easily modified, which might be necessary for different application scenarios. For example, the above set of heuristic make sense in the context of regression analysis, but for side-channel analysis a simpler exploration can be used, since there are no changes in the program itself (cf. Section 7.4).

5.3 EXAMPLE

In order to illustrate the workflow of differential dynamic symbolic execution, please take a look at the same example program as for differential fuzzing in the previous chapter: program `foo` in the Listings 6 and 7. As before, these are two variants of the same program taken from one of the preliminary papers [5]: Listing 6 represents the *old* version and Listing 7 represents the *new* version of the same program. The differential dynamic symbolic execution expects a change-annotated program, which combines the old and the new version, as shown in the Listing in Figure 12. The updated assignment in line 5 can be replaced with the change-annotation `change(-x, x*x)`, where `-x` is taken from the old version (cf. Listing 6 line 5) and `x*x` is taken from the new version (cf. Listing 7 line 5). The

```
1  int foo_change(int x) { /* CHANGE-ANNOTATED VERSION */
2     int y;
3
4     if (x < 0) {
5        y = change(-x, x*x);
6     } else {
7        y = 2 * x;
8     }
9
10    y = change(y, y+1);
11
12    if (y > 1) {
13       return 0;
14    } else {
15       if (y == 1 || y <= -2) {
16          throw new AssertionError("assert(false)");
17       }
18    }
19    return 1;
20 }
```

```
0: iload_0
1: ifge         16
4: iload_0
5: ineg
6: iload_0
7: iload_0
8: imul
9: invokestatic  #16
12: istore_1
13: goto         20
16: iconst_2
17: iload_0
18: imul
19: istore_1
20: iload_1
21: iload_1
22: iconst_1
23: iadd
24: invokestatic  #16
27: istore_1
28: iload_1
29: iconst_1
30: if_icmple     35
33: iconst_0
34: ireturn
35: iload_1
36: iconst_1
37: if_icmpeq     46
40: iload_1
41: bipush        -2
43: if_icmpgt     56
46: new           #22
49: dup
50: ldc           #24
52: invokespecial #26
55: athrow
56: iconst_1
57: ireturn
```

Figure 12: Left: change-annotated `foo` program [5], Right: corresponding bytecode.

additional statement in line 10 can be written as `change(y, y+1)`, where the first parameter `y` denotes the old version, in which the value is not changed, and second parameter `y+1` denotes the update of `y` from the new version (cf. Listing 7 line 10). The rest of the program remains unchanged.

In addition to the change-annotated program, the symbolic execution needs a driver, which provides the entry point for the symbolic analysis (cf. Listing 12). Similar as for fuzzing, this driver parses the input and executes the program under test, but there are some important differences. First of all, this driver is used for different symbolic execution variants (cf. Figure 9): concolic execution, trie-guided symbolic execution, and bounded symbolic execution. Concolic execution expects that the input is concolic, i.e., there is a concrete value as well as a symbolic value. Trie-guided and bounded symbolic execution expect a symbolic input, i.e., no concrete values. Therefore, the driver has two modes (cf. line 3 in Listing 12): If there is a concrete input, then the driver will read the input (line 6 to 16) and will add a symbolic value (cf. line 18). If there is no concrete input given, then the driver will simply add a symbolic value (cf. line 21). Afterwards the driver calls the change-annotated program with the constructed input. In contrast to fuzzing, the driver for symbolic execution calls the application only once because the two programs are already combined by the change-annotations.

Starting with the initial input `x=100`, Figure 13 shows the first couple of different stages during differential dynamic symbolic execution. Note that for the example differential dynamic symbolic execution is used and not standard symbolic execution, i.e., there will be in general four choices for every conditional statement instead of two.

Stage 1 in Figure 13. The initial input is imported and executed with concolic execution. Meanwhile the trie structure get initialized and extended for every choice made during

Listing 12: Driver for differential dynamic symbolic execution example.

```java
public static void main(String[] args) {
   /* Concolic execution or symbolic execution? */
   if (args.length == 1) {
      /* Read one input from the input file for both program versions. */
      int input;
      try (FileInputStream fis = new FileInputStream(args[0])) {
         byte[] bytes = new byte[Integer.BYTES];
         if ((fis.read(bytes)) == -1) {
            throw new RuntimeException("Not enough data!");
         }
         input = ByteBuffer.wrap(bytes).getInt();
      } catch (IOException e) {
         System.err.println("Error reading input");
         e.printStackTrace();
         throw new RuntimeException("Error reading input");
      }
      /* Insert concolic variables. */
      input = Debug.addSymbolicInt(input, "sym_0");
   } else {
      /* Insert pure symbolic variables. */
      input = Debug.makeSymbolicInteger("sym_0");
   }
   Object res = Foo.foo(input);
}
```

concolic execution. The resulting trie after this step includes three nodes: the root node (id=0), the intermediate node (id=1) for the choice happened in line 4, and the leaf node (id=2) for the choice happened in line 12. There is no diff path yet, but there are differential expression present. Based on the heuristics, the most promising node is the intermediate node with id=1 (therefore, presented in green color).

Stage 2 in Figure 13. As next step DDSE performs a trie-guided symbolic execution from the root node up to the selected node with id=1. In the current example this is only one step (= one guided choice). Afterwards DDSE starts bounded symbolic execution with bound 1, i.e., it makes one additional step ahead. The trie is extended with the node (id=3), which represents choice=1 (i.e., `false` branch in line 12). The bounded symbolic execution results in the path condition: $(x \geqslant 0 \wedge 2*x \leqslant 1 \wedge 2*x+1 \leqslant 1)$, for which Z3 generates the model `x=0`.

Stage 3 Figure 13. After generating an input for `x=0`, it is replayed with concolic execution to assess the input and extend the trie. During concolic execution the trie node with id=3 is updated and the error leaf node with id=4 is created. In the *changed/new* version the input `x=0` takes the `true` branch in line 15, which results in an assertion error, while the *old* version would take the `false` branch and return `1`. Therefore, node id=4 holds choice=3, which denotes the `diff`$_{\mathtt{TRUE}}$ path, i.e., the new version takes the `true` branch and the old version takes the `false` branch.

After this first iteration of all steps in DDSE (cf. Figure 9), the node with id=1 is still the most promising node for further exploration, as it still has two unexplored choices: choice=2 and choice=3, for which DDSE checks whether a decision difference is feasible in line 12 (as shown in Figure 14, these choices are actually not satisfiable).

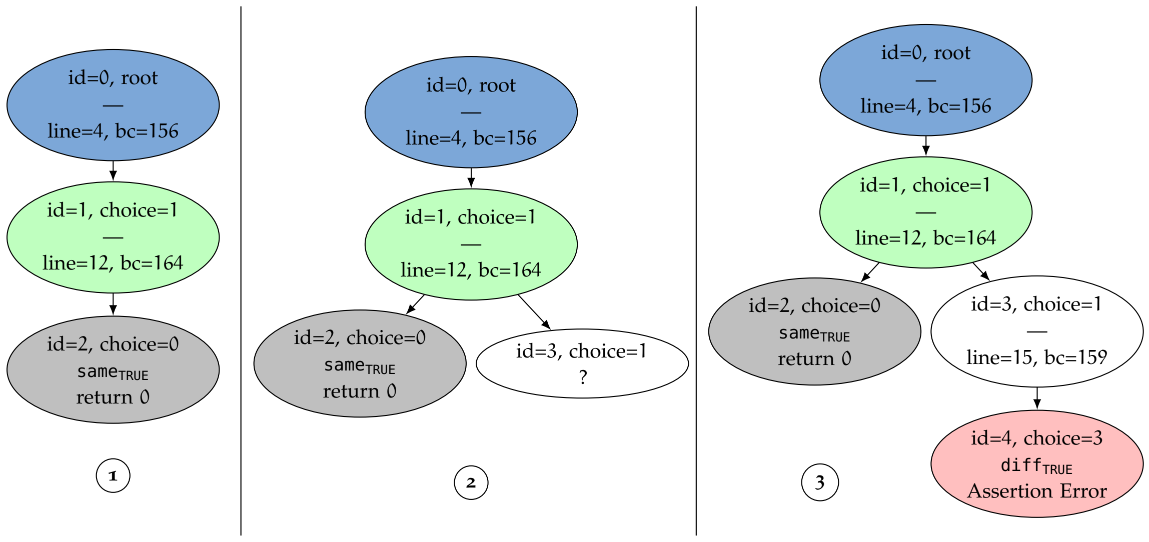


Figure 13: The first three of stages of the trie for the differential dynamic symbolic execution of the changed-annotated program from Listing in Figure 12.

Complete trie in Figure 14. After a complete exploration with DDSE the trie will have four leaf nodes as presented in Figure 14. The leaf node with id=2 is the leaf node from the initial input. The error leaf node with id=3 represents the already discussed assertion error for the new version with `x=0`. The leaf node with id=8 represents a `same`$_{\text{TRUE}}$ path, where both versions take the `true` branch in line 4. Finally, the leaf node with id=10 represents a `diff`$_{\text{TRUE}}$ path, where the new version takes the `true` branch in line 4, while the old version takes the `false` branch, representing the fix of the assertion error for `x=-1`.

5.4 TECHNICAL DETAILS

As for the differential fuzzing approach in the previous chapter, the here presented differential dynamic symbolic execution technique is universal applicable and not limited to specific programming languages. However, the implemented tools for BADGER, SHADOW$_{\text{JPF}}$, SHADOW$_{\text{JPF+}}$, and HYDIFF aim at the analysis of JAVA bytecode for evaluation purposes.

All symbolic execution components are built on top of Symbolic PathFinder (SPF), a symbolic execution tool for JAVA bytecode [67]. The path constraints are solved with the SMT solver Z3 [64]. In order to implement the shadow symbolic execution notions in SHADOW$_{\text{JPF}}$ [4], SHADOW$_{\text{JPF+}}$ [5], and HYDIFF [6], i.e., to handle change-annotations and apply four-way forking, it is necessary modify the interpretation of each bytecode operation. All bytecode operations need to able to detect and handle *differential expressions*, i.e., expression that hold values for both (the old and the new) versions. All bytecode branching operations, e.g., like IFEQ (check for equivalence), need to be able to detect the presence of differential expressions in the branching condition and apply four-way forking in such cases, instead of the standard two-way forking.

5.4.1 *Differential Expression*

The *differential expressions* enable the execution of multiple program versions at once. Listing 13 shows the JAVA class for a differential expression. It holds information about both, the old and the new version, and therefore combines both values in one expression. Returning

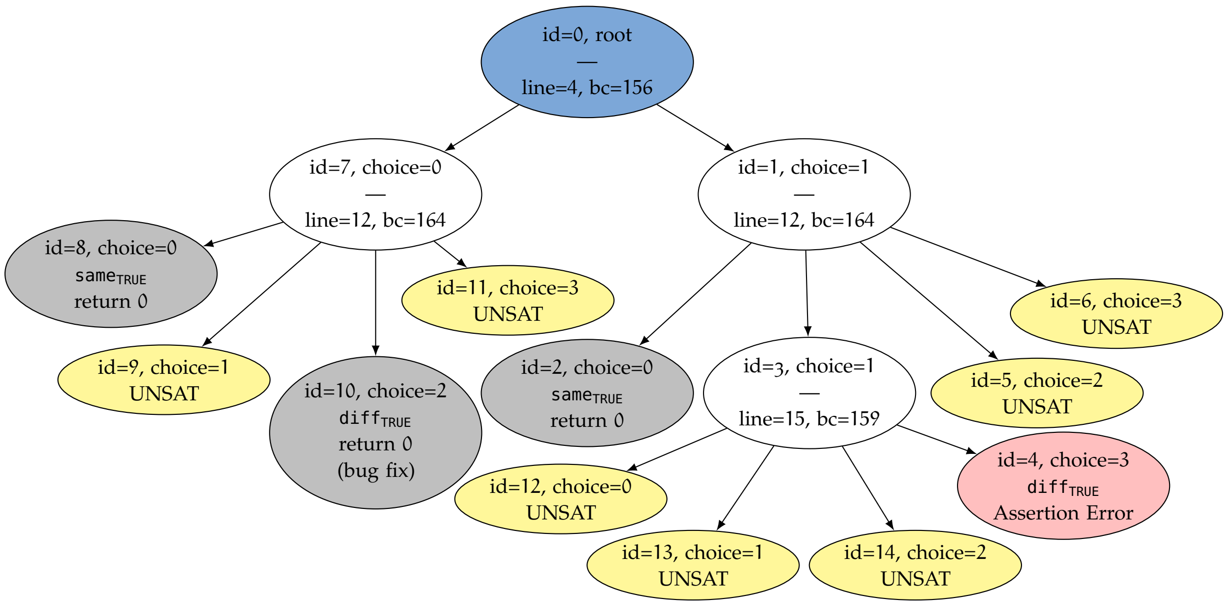


Figure 14: Final trie for differential dynamic symbolic execution of the change-annotated program from Listing in Figure 12.

to the example for the previous section, in SPF the statement `y=change(-x,x*x)` in line 5 in Listing 12 would usually cause that an `IntegerExpression` object will be mapped to the variable `y`. However, having a differential expression introduced by the change() method means that `y` is mapped to an `DiffExpression` object, which itself stores, in this case, four different `IntegerExpression` objects: two four the concrete values and two for the symbolic values for the old and new version respectively.

Listing 13: Java class for differential expressions.

```
public class DiffExpression {

   /* Symbolic Expressions */
   Expression oldSymbolicExpr;
   Expression newSymbolicExpr;

   /* Concrete Expressions. */
   Expression oldConcreteExpr;
   Expression newConcreteExpr;

   /* getter and setter */
   ...

}
```

5.4.2 *Measuring Execution Cost*

In the differential dynamic symbolic execution, the execution cost is measured inside the symbolic execution framework, which interpret every bytecode operation. By implementing listeners it is fairly simple to, e.g., count every statement when visited. Therefore, it is not necessary to instrument the bytecode as it is necessary for differential fuzzing (cf.

Section 4.4.1). Luckow et al. [145] demonstrate how to use SPF to calculate various types of execution cost metrics, e.g., the number executed instructions, the heap allocation and the maximum stack size, or the number of bytes written to an output stream. Basically, it is necessary to implement listeners, which collect the specific information during bytecode execution. In addition to these metrics, the presented approach also allows the handling of user-defined cost, which can be added as annotation to the program.

The leaf nodes in the trie will be associated with the final execution cost, which can be only determined at the end of an execution path. Each node holds two cost values, one for each observed behavior. In order to estimate the potential costs for intermediate nodes, their associated cost values represents the average cost values of their children nodes.

5.4.3 *Measuring Patch Distance*

Based on the interprocedural control flow graph (ICFG) the distances to the change-annotations and also the reachability information are pre-calculated and stored in memory. The ICFG is, similar as for differential fuzzing, created with Apache Commons BCEL [42]. The shortest distances are calculated by a backwards analysis starting from the target node, which are the represent the change-annotations in the program under test. For every statement in the program it is possible to retrieve the shortest distance to the change-annotations, and hence, it is possible to retrieve the distance for every trie node.

5.5 PRELIMINARY EVALUATION

The baseline for this preliminary assessment is coverage-based symbolic execution (CBSE), i.e., standard (traditional) symbolic execution that performs a deterministic exploration of the search space (cf. Section 2.2). The goal is to show that the presented differential dynamic symbolic execution (DDSE) performs significantly better in identifying behavioral differences than the baseline. Similar as for the preliminary assessment of differential fuzzing (cf. Section 4.5), this assessment focuses on regression testing and uses ten subjects from the Traffic collision avoidance system (TCAS), which are originally taken from the SIR repository [208]. In order to enable DDSE it is necessary to manually prepare the change-annotated programs by merging the several available variants. The original program has 143 LOC and the used variants include 1-2 changes, i.e., there will be 1-2 change-annotations per variant. In contrary to the differential dynamic symbolic execution approach, coverage-based symbolic execution can only be applied on one version at once, therefore the experiments show the result for applying CBSE on the new version. All experiments have been executed for 10 minutes and repeated 30 times. Table 5 presents the results for the conducted experiments. Each row represents the differential analysis between the original TCAS program and a generated variant. The first column declares the used generated variant and states the number of involved changes (cf. the number in the brackets). The other columns show the results for coverage-based symbolic execution (CBSE) and differential dynamic symbolic execution (DDSE). The used metrics ($\overline{t}$ +odiff, t_{min}, $\overline{\#odiff}$, and $\overline{\#ddiff}$) have been already described in Section 3.4 and focus on the output difference (odiff) as well as the decision difference (ddiff). The highlighted values represent significant differences to the other technique verified with the Wilcoxon rank-sum test (with 5% significance level). The time values are presented in seconds and the values also report the 95% confidence interval.

Table 5: Results for the preliminary evaluation of differential dynamic symbolic execution (DDSE) by comparing it with coverage-based (traditional) symbolic execution (CBSE). The bold values represent significant differences to the other technique verified with the Wilcoxon rank-sum test ($\alpha = 0.05$).

Subject	**Coverage-Based Symb. Exec. (CBSE)**				**Differential Dynamic Symb. Exec. (DDSE)**			
(# changes)	$\bar{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$	$\bar{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$
TCAS-1 (1)	**13.50 (±0.18)**	13	1.00 (±0.00)	3.00 (±0.00)	22.47 (±0.39)	21	1.00 (±0.00)	3.00 (±0.00)
TCAS-2 (1)	-	-	0.00 (±0.00)	3.00 (±0.00)	**182.37 (±1.96)**	177	**1.00 (±0.00)**	**9.00 (±0.00)**
TCAS-3 (1)	-	-	0.00 (±0.00)	**26.00 (±0.00)**	**239.07 (±2.57)**	232	**2.00 (±0.00)**	19.00 (±0.00)
TCAS-4 (1)	-	-	0.00 (±0.00)	**9.00 (±0.00)**	-	-	0.00 (±0.00)	3.00 (±0.00)
TCAS-5 (1)	**14.53 (±0.18)**	14	2.00 (±0.00)	19.00 (±0.00)	185.40 (±1.95)	180	2.00 (±0.00)	**24.00 (±0.00)**
TCAS-6 (1)	4.93 (±0.16)	4	1.00 (±0.00)	6.00 (±0.00)	5.30 (±0.23)	4	1.00 (±0.00)	6.00 (±0.00)
TCAS-7 (1)	-	-	0.00 (±0.00)	0.00 (±0.00)	**56.97 (±0.76)**	54	**2.00 (±0.00)**	**6.00 (±0.00)**
TCAS-8 (1)	**12.13 (±0.55)**	11	2.00 (±0.00)	6.00 (±0.00)	51.70 (±0.16)	51	2.00 (±0.00)	6.00 (±0.00)
TCAS-9 (1)	**30.10 (±0.28)**	29	1.00 (±0.00)	15.00 (±0.00)	184.20 (±0.57)	181	1.00 (±0.00)	15.00 (±0.00)
TCAS-10 (2)	**4.27 (±0.18)**	3	2.00 (±0.00)	12.00 (±0.00)	5.23 (±0.15)	5	2.00 (±0.00)	12.00 (±0.00)

The results show that coverage-based symbolic execution (CBSE) identifies only 6 of 10 subjects as output differential, where as differential dynamic symbolic execution (DDSE) identifies 9 of 10 subjects. Interestingly, in most of the cases, when both approaches find an output difference, CBSE usually finds the first output difference faster. The search space for DDSE is larger because it analyzes the change-annotated programs and uses four-way forking, whereas CBSE only analyzes a single version (the new one) and uses usual two-way forking. Overall, DDSE performs significantly better than CBSE in finding behavioral differences because the focus is on finding actual observable divergences, while the time differences are still in a manageable range. Nevertheless, the preliminary assessment shows that DDSE has its limitations in terms of exploration speed.

5.6 SUMMARY

This chapter introduced *differential dynamic symbolic execution (*DDSE*)* as a method to identify behavioral differences with a systematic, concolic exploration driven by differential heuristics. Existing solution ideas such as shadow symbolic execution [93] or differential symbolic execution [96] have their limitations, which have been discussed in this chapter. The proposed approach is driven by differential expressions introduced by change-annotations inside the program and is further guided by differential metrics like the cost difference and the patch distance. The presented preliminary evaluation shows that DDSE significantly outperforms coverage-guided symbolic execution in terms of effectiveness, although it still has its limitations. In a hybrid differential analysis setup, DDSE might be receptive for guidance based on concrete inputs from the fuzzing component, like discussed in the next chapter.

6 HYBRID DIFFERENTIAL ANALYSIS

This chapter introduces the *hybrid differential analysis* as a combination of *differential fuzzing* (see Chapter 4) and *differential dynamic symbolic execution* (see Chapter 5). The general concept of *hybrid differential analysis* has been already discussed in Chapter 3.2. This chapter thus implements this concept and explains each component and their synchronization in detail. Furthermore, it illustrates the strengths of the hybrid setup with an example. Preliminary concepts and results have been described in the following publications:

- Badger [3] proposes the combination of fuzzing and symbolic execution specifically for worst-case complexity analysis in a parallel setup, and
- HyDiff [6] extends the framework in Badger to enable a general differential program analysis.

6.1 OVERVIEW

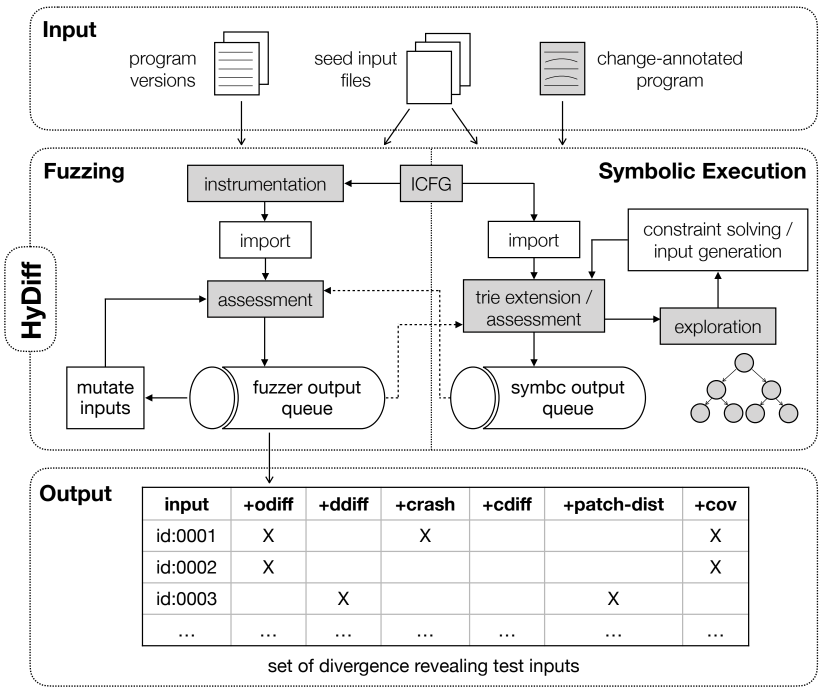
Chapter 4 introduced *differential fuzzing* and Chapter 5 introduced *differential dynamic symbolic execution*. As shown in their preliminary evaluations, both are effective techniques for a differential analysis. However, both also have their limitations.

Differential fuzzing is a very lightweight input generation technique, and hence, can generate a lot if inputs in a short period of time. Although it is guided by the differential metrics in the input selection mechanism and by the coverage information, it still suffers from the general problem in fuzzing: it is unlikely to hit low-probability branches with random mutations. The used random mutations are the key to an input generation that goes beyond what a developer might think is necessary to test, and at the same time they limit the search because branches, which have only a low-probability of being hit with random mutations, will not be tested at all.

Differential dynamic symbolic execution is a white-box input generation technique, and hence, represents a very powerful technique to generate targeted inputs for specific paths in the application. However, its analysis is very expensive and there are usually too many states for an exhaustive exploration. The state explosion, i.e., the exponential growth of unexplored branches, makes it necessary to focus on specific parts of the application.

Existing hybrid techniques. The general issue in fuzzing with hitting low-probability branches has been identified before [77, 78, 81]. Hybrid concolic testing by Majumdar and Sen [77] proposed the combination of random testing and concolic execution in a sequential setup. Starting the exploration with random testing, concolic execution takes over as soon as random testing gets *stuck*, i.e., it does not make any progress in terms of coverage. The concolic execution component tries to generate inputs that cover new branches. Afterwards random testing is started again and the process repeats. The techniques Driller by Stephens et al. [81] and Munch by Ognawala et al. [78] propose a similar sequential setup, in which fuzzing and concolic execution are executed in sequence, although they differ on how they guide the symbolic execution.

The state explosion in symbolic execution is usually addressed by a bounded symbolic execution, i.e., the exploration of a path will be aborted after a pre-defined bound on the



input	+odiff	+ddiff	+crash	+cdiff	+patch-dist	+cov
id:0001	X		X			X
id:0002	X					X
id:0003		X			X	
…	…	…	…	…	…	…

Figure 15: Overview of HyDiff's workflow [6].

number of visited states. Concolic testing [57, 69] was proposed to incorporate concrete inputs in order to focus the exploration and to simplify the constraint solving. Directed symbolic execution [63] further helps to guide the exploration to designated targets.

Inspired by the existing related work, this proposed technique combines differential fuzzing and differential dynamic symbolic execution in a *parallel*, *hybrid*, and *differential* analysis technique, as explained in the following sections.

6.2 APPROACH

The publications, which provide the basis for this thesis, represent the following hybrid analysis techniques: Badger [3] provides a framework combining fuzzing and concolic execution for a worst-case complexity analysis, and HyDiff [6] extends Badger to a general hybrid differential analysis framework. HyDiff's overview is presented in Figure 15. The setup allows to incorporate concrete inputs from the DF component in the concolic exploration of the DDSE component, as well as the further exchange interesting inputs between both components.

6.2.1 *Inputs*

Consider the upper compartment of Figure 15, as *input* HyDiff takes one or two program version(s) (used for fuzzing) and the change-annotated program (used for symbolic execution). Additionally, the approach expects one or more seed input files to drive the exploration, which are shared by both components.

6.2.2 *Collaborations between Components*

The middle part of Figure 15 shows the two components and their workflow. In contrast to the existing related work on hybrid analysis [77, 78, 81] the proposed approach executes fuzzing and symbolic execution in *parallel* and not in a sequential order. The intuition is that both techniques are highly effective on their own, but benefit from some guidance into certain areas of the search space. Therefore, they are executed in parallel, so that both can perform their own analysis. After a specified time bound both components synchronize with each other to incorporate interesting inputs, i.e., inputs that improve the differential metrics, from the other component. *Conceptually* this is performed by importing the inputs from the other's output queue (cf. the arrows from the output queues to the assessment nodes in both sides in the middle of Figure 15). *Technically* this is implemented as a synchronization on the file level. Both approaches generate input files and move them in specific output folders. After a specified time bound, both check the output folder of the other component for new and interesting input files. This check is done by replaying the new inputs with their own analysis.

For fuzzing this means to execute the new inputs generated by the symbolic execution component with the instrumented application. Whenever this check reveals that an input improves any of the employed differential metrics, fuzzing would copy the input in its fuzzing corpus to use it in the upcoming mutations (cf. left middle part of Figure 15). For symbolic execution the synchronization means to replay the dynamic symbolic execution with the new inputs generated by the fuzzing component (cf. right middle part of Figure 15). Whenever the inputs show some new behavior, the trie data structure is extended and the additional nodes are used to determine the trie node for the upcoming exploration.

As shown in Figure 15, both components use the information from the inter-procedural control flow graph (ICFG) to drive/prune their exploration. DF leverages the ICFG to calculate the distance values, which are used to determine whether inputs get closer to the changed location(s), and hence, to guide the DF towards the modification(s) (cf. Section 4.2.1). DDSE uses it to prune paths that cannot reach any changed area (cf. Section 5.2.2).

6.2.3 *Outputs*

The lower part of Figure 15 shows the expected output of the hybrid analysis, which is a list of generated inputs and their characteristics in terms of differential behavior. They are classified according the differential metrics presented in the Chapters 4 and 5. For example, in regression analysis it is of special interest when inputs reveal output differences or in particular crashes in the new version.

6.3 example

In order to illustrate the hybrid differential analysis and to demonstrate the performance of both components in comparison to the hybrid technique, please consider the following simple example for regression testing taken from one of the preliminary papers [6]. Listing 14 shows a change-annotated program, combining two versions of the program `calculate`. This program is an artificial example, which shows the strengths and drawbacks of fuzzing and symbolic execution. It processes two integer inputs, `x` and `y`, and calculates a division based on these two values. The large `switch` statement with cases from 0 to 250 (cf. lines 3 to 21) is a challenge for symbolic execution because there are a lot of branches to explore.

Listing 14: Sample program with annotated changes to illustrate hybrid approach [6].

```
1  int calculate(int x, int y) {
2     int div;
3     switch (x) {
4        case 0:
5           div = y + 1;
6           break;
7        case 1:
8           div = y + 2;
9           break;
10       ...
11       case 250:
12          div = y + 251;
13          break;
14       default:
15          if (x == 123456) {
16             // CHANGE: expression y + 123455 to y + 123456
17             div = change(y + 123455 , y + 123456);
18          } else {
19             div = x + 31;
20          }
21    }
22    int result = x / div;
23
24    // CHANGE: added conditional statement
25    if (change(false, result > 0))
26       result = result + 1;
27    return result;
28 }
```

In this example it is especially problematic because none of them can be pruned because all of them can reach the changed condition in line 25, and the interesting part is at the end of the switch statement, which will be reached late in the exploration (when having a deterministic exploration order). In the default case of the switch statement (cf. line 15), there is a check for the value `123456` representing a magic value, which guards the first change in line 17. There the developer changed the right-hand side expression from `y + 123455` to `y + 123456`, which fixed a division-by-zero error for `y=-123455`, but introduced another crash for `y=-123456`. In contrary to symbolic execution, fuzzing is expected to traverse the program quite fast, but it will have problems with handling the magic number. In line 23, the developer added a conditional statement `result = result + 1` if `result > 0`. This influences the output for all positive results. However, it does not directly fix or introduce any crash.

6.3.1 *Setup*

The differential fuzzing (DF) component works as described in Chapter 4. Similar to the example in Section 4.3, the fuzzing component needs a driver, in which each mutated input is executed on both successive versions of the `calculate`-program (cf. Listing 15).

The differential dynamic symbolic execution (DDSE) component works as described in Chapter 5. Also the symbolic execution component needs a driver (cf. Section 5.3). Listing 16 shows the symbolic execution driver for this example, which is slightly more complex

Listing 15: Fuzzing driver for the hybrid approach sample [6].

```java
public static void main(String[] args) {

   /* Read input file. */
   int x;
   int y;
   try (FileInputStream fis = new FileInputStream(args[0])) {
      byte[] bytes = new byte[Integer.BYTES];
      if ((fis.read(bytes)) == -1) {
         throw new RuntimeException("Not enough data!");
      }
      x = ByteBuffer.wrap(bytes).getInt();

      bytes = new byte[Integer.BYTES];
      if ((fis.read(bytes)) == -1) {
         throw new RuntimeException("Not enough data!");
      }
      y = ByteBuffer.wrap(bytes).getInt();
   } catch (IOException e) {
      System.err.println("Error reading input");
      e.printStackTrace();
      throw new RuntimeException("Error reading input");
   }

   /* Execute old version. */
   Mem.clear();
   DecisionHistory.clear();
   Object res1 = null;
   try {
      res1 = calculate_old(x, y);
   } catch (Throwable e) {
      res1 = e;
   }
   boolean[] dec1 = DecisionHistory.getDecisions();
   long cost1 = Mem.instrCost;

   /* Execute new version. */
   Mem.clear();
   DecisionHistory.clear();
   CFGSummary.clear();
   Object res2 = null;
   try {
      res2 = calculate_new(x, y);
   } catch (Throwable e) {
      res2 = e;
   }
   boolean[] dec2 = DecisionHistory.getDecisions();
   long cost2 = Mem.instrCost;

   /* Report differences. */
   DecisionHistoryDifference d = DecisionHistoryDifference
      .createDecisionHistoryDifference(dec1, dec2);
   Kelinci.setNewDecisionDifference(d);
   Kelinci.setNewOutputDifference(new OutputSummary(res1, res2));
   Kelinci.addCost(Math.abs(cost1 - cost2));
}
```

Listing 16: Symbolic execution driver for the hybrid approach sample [6].

```java
1  public static void main(String[] args) {
2     int x;
3     int y;
4  
5     /* Concolic execution or symbolic execution? */
6     if (args.length == 1) {
7        try (FileInputStream fis = new FileInputStream(args[0])) {
8           /* Read input and add symbolic value. */
9           byte[] bytes = new byte[Integer.BYTES];
10          if ((fis.read(bytes)) == -1) {
11             throw new RuntimeException("Not enough data!");
12          }
13          x = Debug.addSymbolicInt(ByteBuffer.wrap(bytes).getInt(), "sym_0");
14 
15          bytes = new byte[Integer.BYTES];
16          if ((fis.read(bytes)) == -1) {
17              throw new RuntimeException("Not enough data!");
18          }
19          y = Debug.addSymbolicInt(ByteBuffer.wrap(bytes).getInt(), "sym_1");
20 
21       } catch (IOException e) {
22          System.err.println("Error reading input");
23          e.printStackTrace();
24          throw new RuntimeException("Error reading input");
25       }
26 
27    } else {
28       /* Insert pure symbolic variables. */
29       x = Debug.makeSymbolicInteger("sym_0");
30       y = Debug.makeSymbolicInteger("sym_1");
31    }
32 
33    /* Execute change-annotated version. */
34    calculate(x, y);
35 }
```

than the fuzzing driver: it handles a concolic mode for the *input assessment* phase (cf. step 2 in Figure 9) and a symbolic mode for the *exploration* phase (cf. step 3 in Figure 9). During the *exploration* phase, the inputs are marked as symbolic (lines 13 and 19) and the change-annotated program is executed symbolically. During the *trie assessment* phase, the given concrete input is marked as symbolic (lines 29 and 30) and the change-annotated program is executed concollicaly, i.e., follows the concrete input (cf. line 34).

6.3.2 *Results*

To further illustrate the challenges of each individual component, the following paragraph first discusses the results for running both components in isolation, and afterwards together in the hybrid setup. The differential fuzzing component finds its first output difference after 5.07 (±0.99) sec (where the ± value denotes the 95% confidence interval). In total it finds 1.37 (±0.17) output differences and 1.00 (±0.00) decision differences. The *new* crash is not found within the time bound of 10 minutes. Therefore, fuzzing is very fast in findings an output difference (less than 5 seconds), but the narrow constraint at the end is difficult to reach for fuzzing: `(x=123456 & y=-123456)`.

In contrast, the differential dynamic symbolic execution component finds its first output difference after 135.27 (±0.66) seconds. In total, it finds 35.10 (±1.10) output differences and 2.00 (±0.00) decision differences. So it reveals much more output differences than fuzzing within the given time bound. In fact, the DDSE component can traverse all paths in 5 minutes. In contrast to fuzzing it also finds the new crash, after 135.80 (±0.64) seconds. Nonetheless, symbolic execution needs relatively long to find its first output difference.

In the hybrid setup, the differential fuzzing and symbolic execution components are started with the same seed input. Both run their analysis in parallel and exchange inputs that are deemed interesting according to the divergence metrics after a pre-specified time bound. The experimental results are as follows: first output difference after 4.73 (±0.78) seconds, in total 35.13 (±1.04) output differences and 2.00 (±0.00) decision differences. The hybrid technique finds the new crash already after 14.43 (±0.30) sec.

Figure 16 shows the temporal development of the results for the three techniques. Although DDSE and HyDiff come to similar conclusions after 10 minutes, HyDiff is significantly faster in finding the first output differences (as well as the crash). DF is fast in generating first results, but cannot achieve the same numbers as DDSE and HyDiff within the 10 minutes time bound.

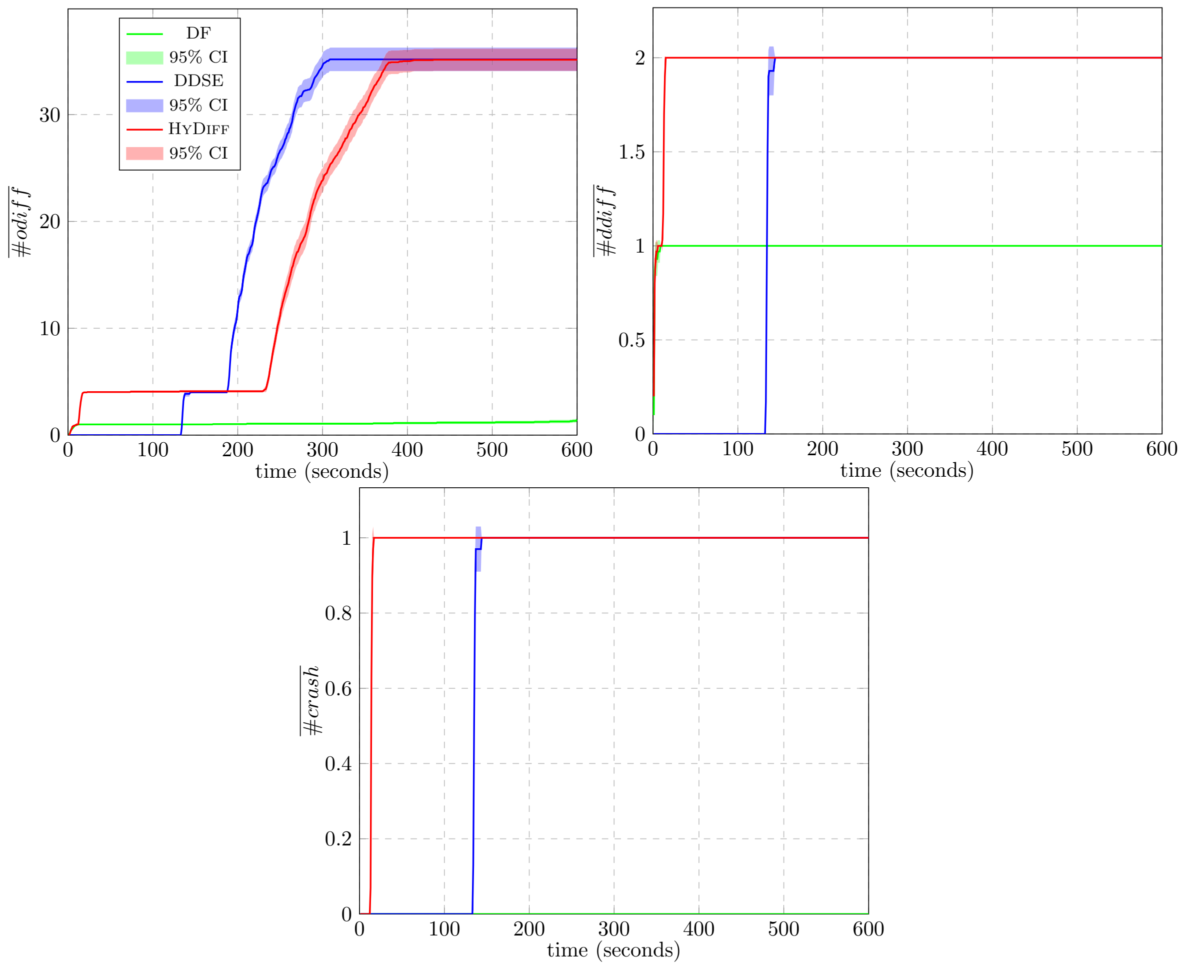


Figure 16: Results for DF, PDF, DDSE, and HyDiff on hybrid approach sample (lines and bands show averages and 95% confidence intervals across 30 repetitions).

SUMMARY – HYDIFF EXAMPLE

The hybrid approach detects the regression bug more than *nine times faster* than DDSE component in isolation. The DF component (in isolation) times out after ten minutes without detecting the regression bug. The hybrid setup can leverage the strengths of both techniques, so that it can get into many more paths by using symbolic execution and is very fast in finding its first output difference by using fuzzing.

6.4 SUMMARY

This chapter described the details of the hybrid differential analysis and presented an example that illustrates challenges for both single components. Furthermore, the example showed significantly better results with the hybrid setup. The following chapter shows the extensive validation of this hybrid approach in several application scenarios.

7 VALIDATION

This chapter presents the validation of the hybrid differential software testing approach. It starts with the discussion of the details about the experimental infrastructure and further continues with explaining four different application scenarios of hybrid differential testing: A1 regression analysis (cf. Section 7.2), A2 worst-case complexity analysis (cf. Section 7.3), A3 side-channel analysis (cf. Section 7.4), and A4 robustness analysis of neural networks (cf. Section 7.5). For each application the chapter presents examples, details about the drivers, the used evaluation metrics, the data sets, and the evaluation results. The results are discussed for each application with regard to the research questions (cf. Section 3.3). The chapter closes with a discussion of the obtained results and of the threats to validity.

7.1 EXPERIMENTAL INFRASTRUCTURE

Over all experiments HyDiff is compared to the differential fuzzing (DF) and the differential dynamic symbolic execution (DDSE) component. By comparing HyDiff with its components, it is possible to show how HyDiff combines the strengths of both components.

Since HyDiff runs fuzzing and symbolic execution in parallel, it is technically not fair to compare HyDiff directly with single runs of DF and DDSE. Therefore, the presented results also contain results for a parallel DF variant (PDF). The DF component can be *parallelized* quite simple: AFL already includes the functionality to execute multiple fuzzing instances in parallel, so that DF can be started with two fuzzing instances, which are started with the same seed inputs. For the DDSE component it is more difficult: just running the component in two parallel executed instances make no sense because the DDSE performs a deterministic exploration, so that both instances would produce the very same output. Parallelizing symbolic execution is an own research area [53, 70], in which it is important to generate a nicely balanced partitioning of the explored sub-trees between the instances (see Section 2.2.3). In order to still provide a technically *fairer* comparison between HyDiff and DDSE, the presented results also contain a DDSE variant, which gets *twice* the time budget of HyDiff (DDSEx2T), since HyDiff uses twice the processing time because it executes two instances in parallel.

All experiments have been conducted on a virtual machine with Ubuntu 18.04.1 LTS featuring 2x Intel(R) Xeon(R) CPU X5365 @ 3.00GHz with 8GB of memory, OpenJDK 1.8.0_191 and GCC 7.3.0. To incorporate the randomness in fuzzing, each experiment has been repeated 30 times. The reported numbers represent the averaged results together with the 95% confidence interval and the maximum/minimum value. Although symbolic execution is a deterministic process, the results showed some small variations between experiments, mostly in the time until the first observed difference. Such variations can be caused by the constraint solver and other activities on the machine. Therefore, also the experiments for symbolic execution has been repeated 30 times. The experiments for regression analysis use a timeout of 10 minutes (=600 seconds), the experiments for worst-case complexity analysis are performed for 1 hour (3, 600 seconds), the experiments for side-channel analysis use a timeout of 30 minutes (=1, 800 seconds), and the experiments with the neural network are executed for 1 hour (=3, 600 seconds) because of the long running program executions.

These time bounds have been chosen based on the experiments in the related work and pre-experimental executions to see when the techniques reach a plateau. The repeated experiments for each subject are started with the same (randomly generated) seed input file. The only constraint on the seed input was that it does not crash the application because this is a requirement by the underlying fuzzing engine AFL. The highlighted values in the presented tables represent significant differences to the closest other subject verified with the Wilcoxon rank-sum test (with 5% significance level).

7.2 REGRESSION ANALYSIS (A1)

Regression analysis aims to identify errors in software changes (also cf. Section 2.4.1), also called *regression errors*. Such errors represent an observable, semantic different behavior between the old and the new version. The most meaningful semantic different behaviors are differences in the actual output of an application like the result of a calculation or another output message. But also a difference in the runtime or the memory consumption can be something worthwhile to detect. Not all observable, semantic different behavior though is a regression error. It could be an expected change, or a change that is unexpected but not erroneous. Crashes that occur only in the new version tend to represent a regression error, but even these crashes might be intended, e.g., a new exception thrown for a missing command line parameter, which was not present in the old version. Therefore, in order to classify the inputs in expected and unexpected behavior, it is necessary to have a post-processing of the generated inputs (e.g., similar to [93] see Section 2.4.1), which should also include some user feedback. The presented techniques focus on finding all differences with regard to the leveraged differential metrics without performing such post-processing. The goal of the following evaluation is to assess the techniques for their ability to reveal behavioral divergences between two program versions independent whether these divergences are expected or unexpected. The evaluation on regression analysis extends the evaluation shown in one of the preliminary publications [6].

7.2.1 *Example & Drivers*

The previous chapters, which focused on explaining the general approach (Chapters 4, 5 and 6), already presented examples and the corresponding drivers for regression analysis. Therefore, this section will skip the examples and just briefly summarize the driver part. As already presented in Listings 8 and 15 the fuzzing drivers for regression analysis collect the input values and call the old and the new version separately. The driver collects the runtime information for both executions and reports the values to the fuzzer. Similarly, as presented in the Listings 12 and 16, the symbolic execution drivers for regression analysis collect the inputs, add symbolic values and call the change-annotated program, which already includes the information of both versions.

7.2.2 *Evaluation Metrics*

The most important metric for regression testing is the *output difference* because it represents an observable semantic difference between the two versions, which is exactly the goal of the analysis. Furthermore, a special variant of the output difference is the *crash* metric, which determines whether the analysis identified an input that triggers a crash in the new version, but not in the old version. In order to assess the experiments and answer the research questions, the results report the following metrics:

- $\bar{t}$ +odiff: the average time to first output difference (lower is better)
- t_{min}: the minimum time (over all runs) needed to find the first output difference (lower is better)
- $\overline{\#odiff}$: the average number of identified output differences incl. crashes (higher is better)
- $\overline{\#ddiff}$: the average number of identified decision differences (higher is better)

The *crash* metric is not included in the presentation of the results because for almost all subjects there was no such input identified. Section 7.2.5 discusses this in more detail. The number of identified output and decision differences ($\overline{\#odiff}$, $\overline{\#ddiff}$) help to assess the effectiveness of the compared techniques. The time to the first output difference ($\bar{t}$ +odiff) shows how focused the exploration technique is to reveal output differences, which is the primary goal of the analysis.

7.2.3 *Data Sets*

The preliminary evaluation section in the previous chapters already showed the results for the subjects from the Traffic collision avoidance system (TCAS). These subjects have been taken from the SIR repository [208]. The original TCAS program has 143 LOC and the regression analysis subjects have been generated by mutations injections, in this case 1-2 changes per version. As already mentioned, the first ten versions of TCAS are used as a preliminary assessment of the approach, but in order to show a complete picture the following evaluation discussion also includes these subjects. Furthermore, the evaluation includes regression bugs from the Defects4J benchmark [199], which contains a large set of Java bugs, but not necessarily regression bugs, and hence required some manual investigation. The evaluation contains four subjects from the projects Math (85 KLOC) and Time (28 KLOC): *Math-10*, *Math-46*, *Math-60* and *Time-1*. Each of them contain between 1 and 14 *changes* per version. A *change* represents a difference between the old and the new version, which can be annotated with the presented *change-annotation* (cf. Section 2.4.1). Additionally, the set of regression subjects also contain five versions from the Apache CLI [202] (4,966 LOC), which was also used before in other regression testing work [85] and contains between 8 and 21 changes per version.

7.2.4 *Evaluation Results*

Table 6 shows the results for the regression analysis. The used metrics ($\bar{t}$ +odiff, t_{min}, $\overline{\#odiff}$, and $\overline{\#ddiff}$) have been described in Section 7.2.2 and focus on the output difference (odiff) as well as the decision difference (ddiff). The highlighted values represent significant differences to the other technique verified with the Wilcoxon rank-sum test (with 5% significance level). The time values are presented in seconds and the values also report the 95% confidence intervals.

HyDiff vs. DF. The results in Table 6 show that DF miss-classifies 3 subjects, whereas HyDiff classifies all subjects correctly. In particular, HyDiff outperforms DF:

Table 6: Results for A1 regression analysis (t=600sec=10min, 30 runs). The bold values represent significant differences to the closest other technique verified with the Wilcoxon rank-sum test ($\alpha = 0.05$).

Subject	**Differential Fuzzing (DF)**				**Parallel Differential Fuzzing (PDF)**				**Differential Dynamic Sym. Exec. (DDSE)**				**DDSE double time budget (DDSEx2T)**				**HyDiff**			
(# changes)	$\bar{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$	$\bar{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$	$\bar{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$	$\bar{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$	$\bar{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$
TCAS-1 (1)	-	-	0.00 (±0.00)	0.00 (±0.00)	-	-	0.00 (±0.00)	0.00 (±0.00)	**20.10 (±0.14)**	19	1.00 (±0.00)	3.00 (±0.00)	**20.10 (±0.14)**	19	1.00 (±0.00)	3.00 (±0.00)	49.87 (±5.48)	29	1.00 (±0.00)	**4.67 (±0.40)**
TCAS-2 (1)	441.83 (±57.70)	120	0.70 (±0.23)	2.13 (±0.73)	335.93 (±58.24)	16	1.57 (±0.33)	5.40 (±1.29)	**170.07 (±0.32)**	168	1.00 (±0.00)	9.00 (±0.00)	**170.07 (±0.32)**	168	1.00 (±0.00)	9.00 (±0.00)	186.87 (±12.30)	92	1.23 (±0.18)	**13.83 (±0.37)**
TCAS-3 (1)	588.43 (±15.18)	392	0.10 (±0.11)	38.63 (±1.96)	531.87 (±30.90)	295	0.67 (±0.27)	55.53 (±2.18)	**230.37 (±0.52)**	228	2.00 (±0.00)	19.00 (±0.00)	**230.37 (±0.52)**	228	2.00 (±0.00)	19.00 (±0.00)	263.20 (±3.61)	236	2.00 (±0.00)	57.43 (±1.54)
TCAS-4 (1)	28.47 (±10.42)	2	1.00 (±0.00)	18.27 (±1.06)	**9.27 (±3.34)**	1	1.00 (±0.00)	24.10 (±1.24)	-	-	0.00 (±0.00)	3.00 (±0.00)	-	-	0.00 (±0.00)	3.00 (±0.00)	43.70 (±14.01)	3	1.00 (±0.00)	22.53 (±1.01)
TCAS-5 (1)	184.93 (±46.66)	24	2.00 (±0.00)	31.97 (±1.06)	79.77 (±21.40)	3	2.00 (±0.00)	40.00 (±1.73)	173.40 (±0.34)	171	2.00 (±0.00)	23.00 (±0.00)	173.40 (±0.34)	171	2.00 (±0.00)	23.00 (±0.00)	94.60 (±30.72)	1	2.00 (±0.00)	**49.83 (±1.27)**
TCAS-6 (1)	233.63 (±54.48)	4	0.97 (±0.06)	4.13 (±0.83)	114.63 (±37.12)	15	1.00 (±0.00)	9.50 (±0.98)	**4.73 (±0.16)**	4	1.00 (±0.00)	6.00 (±0.00)	**4.73 (±0.16)**	4	1.00 (±0.00)	6.00 (±0.00)	7.57 (±0.26)	6	1.00 (±0.00)	10.37 (±0.70)
TCAS-7 (1)	-	-	0.00 (±0.00)	0.00 (±0.00)	581.60 (±28.73)	164	0.07 (±0.09)	0.27 (±0.36)	73.50 (±0.20)	72	2.00 (±0.00)	6.00 (±0.00)	73.50 (±0.20)	72	2.00 (±0.00)	6.00 (±0.00)	**71.70 (±1.71)**	62	2.00 (±0.00)	**8.93 (±0.39)**
TCAS-8 (1)	-	-	0.00 (±0.00)	0.00 (±0.00)	-	-	0.00 (±0.00)	0.00 (±0.00)	78.73 (±1.24)	75	2.00 (±0.00)	6.00 (±0.00)	78.73 (±1.24)	75	2.00 (±0.00)	6.00 (±0.00)	**65.33 (±0.75)**	61	2.00 (±0.00)	**8.77 (±0.49)**
TCAS-9 (1)	221.73 (±48.83)	10	1.00 (±0.00)	6.13 (±0.85)	**109.73 (±28.35)**	4	1.00 (±0.00)	9.37 (±0.44)	148.57 (±1.76)	143	1.00 (±0.00)	15.00 (±0.00)	148.57 (±1.76)	143	1.00 (±0.00)	15.00 (±0.00)	185.53 (±18.42)	39	1.00 (±0.00)	**22.37 (±0.89)**
TCAS-10 (2)	173.47 (±46.27)	1	1.93 (±0.09)	12.27 (±1.69)	100.53 (±25.20)	3	2.00 (±0.00)	18.07 (±1.07)	**4.87 (±0.52)**	4	2.00 (±0.00)	12.00 (±0.00)	**4.87 (±0.52)**	4	2.00 (±0.00)	12.00 (±0.00)	7.63 (±0.22)	7	2.00 (±0.00)	**21.30 (±0.82)**
Math-10 (1)	221.13 (±56.26)	10	64.50 (±15.98)	15.50 (±2.35)	109.53 (±18.08)	13	**172.37 (±26.21)**	24.03 (±1.33)	**2.97 (±0.17)**	2	7.00 (±0.00)	10.00 (±0.00)	**2.97 (±0.17)**	2	7.00 (±0.00)	10.00 (±0.00)	3.87 (±0.20)	3	44.33 (±5.47)	**32.00 (±1.39)**
Math-46 (1)	377.87 (±63.43)	77	0.80 (±0.14)	36.33 (±1.07)	270.07 (±50.22)	8	1.00 (±0.00)	**43.03 (±0.78)**	**118.93 (±0.90)**	116	1.00 (±0.00)	5.60 (±0.18)	**118.93 (±0.90)**	116	1.00 (±0.00)	8.00 (±0.00)	122.00 (±8.34)	49	1.00 (±0.00)	38.17 (±0.82)
Math-60 (7)	6.93 (±0.63)	4	219.17 (±5.26)	92.90 (±1.64)	5.90 (±0.47)	4	**483.03 (±9.52)**	**138.10 (±3.56)**	**2.27 (±0.16)**	2	2.00 (±0.00)	3.00 (±0.00)	**2.27 (±0.16)**	2	2.00 (±0.00)	3.00 (±0.00)	4.77 (±0.15)	4	234.23 (±5.63)	94.20 (±2.67)
Time-1 (14)	5.17 (±1.20)	2	123.30 (±5.86)	170.63 (±3.43)	3.30 (±0.60)	2	**221.00 (±7.84)**	**249.10 (±4.29)**	5.23 (±0.18)	4	33.00 (±0.00)	32.00 (±0.00)	5.23 (±0.18)	4	33.00 (±0.00)	32.00 (±0.00)	3.80 (±0.69)	1	189.73 (±11.94)	225.33 (±5.62)
CLI1-2 (13)	-	-	0.00 (±0.00)	159.53 (±4.05)	-	-	0.00 (±0.00)	**202.17 (±3.48)**	-	-	0.00 (±0.00)	4.00 (±0.00)	-	-	0.00 (±0.00)	4.00 (±0.00)	-	-	0.00 (±0.00)	169.40 (±4.07)
CLI2-3 (13)	10.83 (±3.33)	2	82.30 (±3.98)	176.83 (±3.62)	**4.83 (±1.29)**	1	**161.60 (±6.62)**	242.53 (±6.92)	-	-	0.00 (±0.00)	37.00 (±0.00)	-	-	0.00 (±0.00)	37.00 (±0.00)	13.27 (±3.62)	2	84.63 (±4.24)	242.70 (±3.80)
CLI3-4 (8)	7.43 (±1.60)	1	96.73 (±4.54)	279.13 (±4.51)	7.20 (±1.85)	2	97.87 (±4.02)	467.27 (±5.05)	**4.07 (±0.36)**	3	1.00 (±0.00)	12.00 (±0.00)	**4.07 (±0.36)**	3	1.00 (±0.00)	12.00 (±0.00)	8.93 (±2.13)	2	**113.33 (±4.80)**	471.50 (±8.93)
CLI4-5 (13)	589.57 (±16.05)	358	0.07 (±0.09)	219.30 (±3.74)	-	-	0.00 (±0.00)	**274.43 (±4.22)**	-	-	0.00 (±0.00)	4.00 (±0.00)	-	-	0.00 (±0.00)	4.00 (±0.00)	**551.97 (±45.65)**	125	**0.13 (±0.12)**	235.17 (±5.73)
CLI5-6 (21)	**4.13 (±1.04)**	1	143.87 (±4.99)	182.00 (±5.54)	**3.43 (±0.72)**	1	**277.17 (±6.81)**	**272.17 (±7.32)**	-	-	0.00 (±0.00)	5.00 (±0.00)	-	-	0.00 (±0.00)	5.00 (±0.00)	6.17 (±1.31)	2	177.80 (±4.39)	214.47 (±6.38)

- for the time to the first output difference ($\bar{t}$ +odiff) in 11 of 19 subjects (for another 6 subjects both achieve similar numbers),
- for the number of output differences ($\overline{\texttt{\#odiff}}$) in 10 of 19 subjects (for another 8 subjects both achieve similar numbers), and
- for the number of decision differences ($\overline{\texttt{\#ddiff}}$) in 17 of 19 subjects (for the remaining 2 subjects both achieve similar numbers).

Moreover, the results show that parallel differential fuzzing (PDF) performs better than DF. It is usually faster in revealing the first output difference and it can significantly improve the number of output and decision differences for some subjects. However, PDF still miss-classifies 3 subjects, which means the parallelization of DF can boost the performance, but cannot solve the actual problems of DF. The results for comparing HyDiff with PDF are as follows, HyDiff outperforms PDF:

- for the time to the first output difference ($\bar{t}$ +odiff) in 11 of 19 subjects (for another 4 subjects both achieve similar numbers),
- for the number of output differences ($\overline{\texttt{\#odiff}}$) in 6 of 19 subjects (for another 8 subjects both achieve similar numbers), and
- for the number of decision differences ($\overline{\texttt{\#ddiff}}$) in 8 of 19 subjects (for the remaining 5 subjects both achieve similar numbers).

HyDiff vs. DDSE. The results in Table 6 show that there is no significant difference when giving DDSE twice the time budget (cf. columns for DDSE and DDSEx2T). In fact only for the subject *Math-46* there is a small improvement for the number of decision differences ($\overline{\texttt{\#ddiff}}$). Therefore, the following result discussion will ignore the DDSEx2T experiments and focus on DDSE. DDSE miss-classifies 4 subjects, i.e., it misses to identify at least one output difference for these subjects, whereas HyDiff classifies all subjects correctly. In particular, HyDiff outperforms DDSE:

- for the time to the first output difference ($\bar{t}$ +odiff) in 8 of 19 subjects,
- for the number of output differences ($\overline{\texttt{\#odiff}}$) in 9 of 19 subjects (for the remaining 10 subjects both achieve similar numbers), and
- for the number of decision differences ($\overline{\texttt{\#ddiff}}$) in all 19 of 19 subjects.

Temporal development of the results. To further illustrate the different performances of the approaches Figure 17, 18, and 19 show the temporal development of the presented results for *TCAS-7*, *CLI3-4* and *CLI5-6*. Instead of showing the Figures for all subjects, these subjects have been selected because they provide a good overview. The graphs on the left side of the Figures show the results for the number of output differences ($\overline{\texttt{\#odiff}}$) and the graphs on the right side show the results for the number of decision differences ($\overline{\texttt{\#ddiff}}$).

For *TCAS-7* (cf. Figure 17) DF and PDF perform poorly: they cannot reliably identify any output or decision differences in the given time bound of 10 minutes. HyDiff can rely on its symbolic execution component to quickly generate some meaningful results. For $\overline{\texttt{\#odiff}}$ HyDiff and DDSE both remain in a plateau after approximately 2.1 minutes. For $\overline{\texttt{\#ddiff}}$ DDSE remains in a plateau after approximately 2.4 minutes, but HyDiff can leverage its fuzzing component to still make progress and continuously improve the score.

For *CLI3-4* (cf. Figure 18) DDSE performs poorly with identifying only one output difference and 12 decision differences, whereas DF, PDF, and HyDiff perform much better. More surprisingly is the result for $\overline{\texttt{\#odiff}}$, for which HyDiff significantly outperforms DF and PDF. All three techniques start with a similarly behavior, but HyDiff can break away, while PDF shows no significant benefit compared to DF. The results for $\overline{\texttt{\#ddiff}}$ show that HyDiff performs similarly as PDF. For both metrics DDSE performs poorly as standalone technique, but still, as component of HyDiff it can contribute.

For *CLI5-6* (cf. Figure 19) DDSE again performs poorly without identifying any output difference and only 4 decision differences. PDF performs significantly better than DF and HyDiff, however HyDiff significantly outperforms DF in both $\overline{\texttt{\#odiff}}$ and $\overline{\texttt{\#ddiff}}$. As already mentioned, the standalone DDSE cannot find any output difference, but still it can support HyDiff and help to make it identify more output differences than standalone DF.

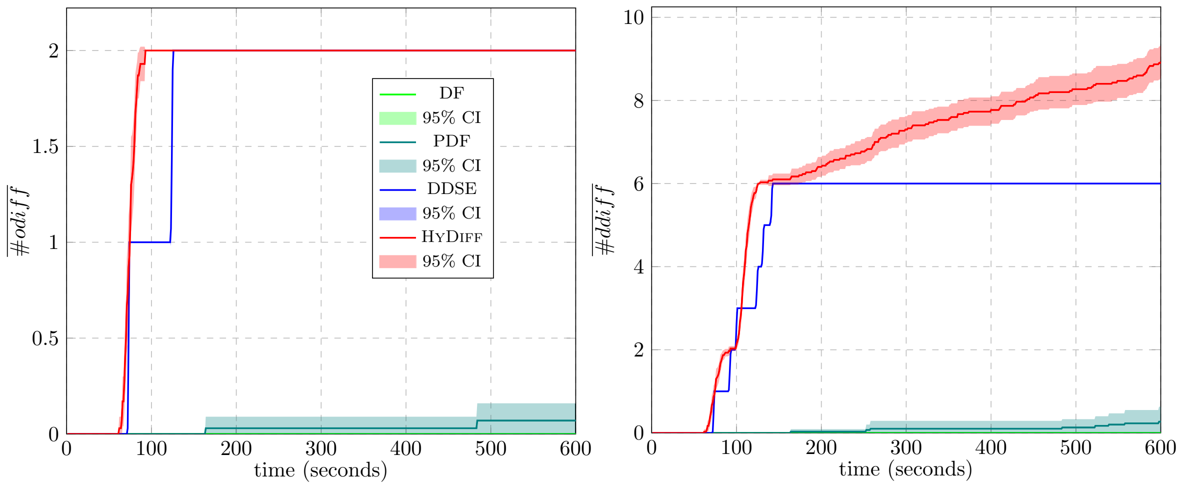


Figure 17: Results for DF, PDF, DDSE, and HyDiff on *TCAS-7* (lines and bands show averages and 95% confidence intervals across 30 repetitions).

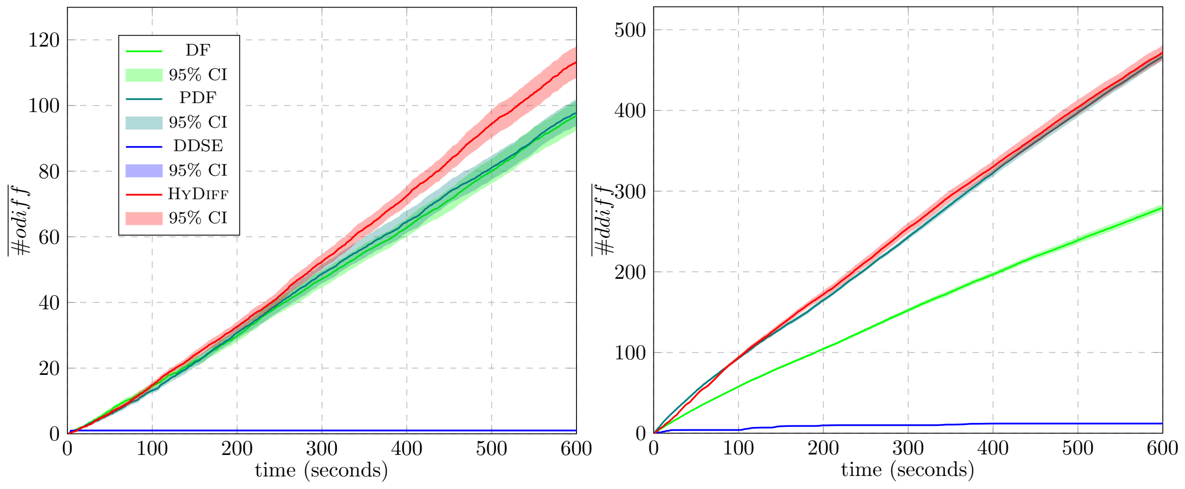


Figure 18: Results for DF, PDF, DDSE, and HyDiff on *CLI3-4* (lines and bands show averages and 95% confidence intervals across 30 repetitions).

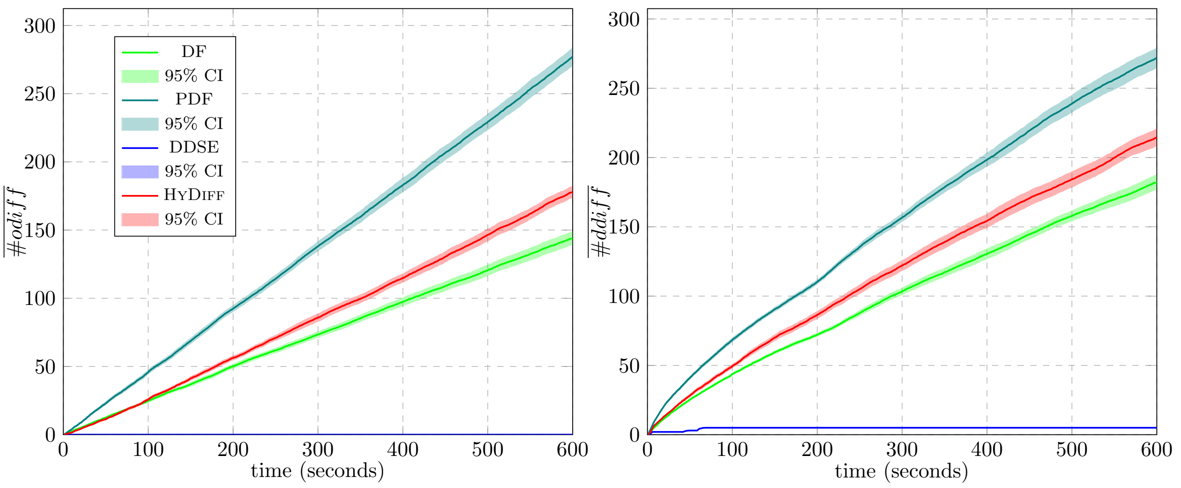


Figure 19: Results for DF, PDF, DDSE, and HyDiff on *CLI5-6* (lines and bands show averages and 95% confidence intervals across 30 repetitions).

7.2.5 *Discussion*

RQ1-A1 Differential fuzzing (DF). Differential fuzzing appears to be a well fitted technique for the identification of output and decision differences, as it can usually identify more differences than differential dynamic symbolic execution. The parallelization of DF (PDF) shows some great performance benefits. Although it cannot solve the classification problems of DF, it can boost its performance significantly: PDF usually improves the number of output and decision differences, and also for some cases it can reduce the time to find the first output difference. The increased number of differences is expected because two fuzzing instances will in general also generate twice as many inputs. However, DF and PDF both miss the output differences in *TCAS-1* and *TCAS-8*, which can be found by DDSE and HyDiff (cf. Table 6). The changes in these subjects are very narrow, like a condition change from > to >= or a small increase in a constant number. This supports the intuition that it is hard to identifying such narrow differences with random mutations.

RQ2-A1 Differential dynamic symbolic execution (DDSE). DDSE is usually faster in finding the first output difference ($\overline{t}$ +odiff) than the other techniques. However, it cannot find as many output differences and decision differences. It also misses to identify the output difference for *TCAS-4*, *CLI2-3*, *CLI4-5*, and *CLI5-6* (cf. Table 6). The temporal development in Figure 17, 18, and 19 shows that DDSE rarely can continuously improve over time. It mostly makes progress in jumps and stays in plateaus over long time periods. Especially for the *CLI* subjects DDSE does not perform well. The code base is much larger than for *TCAS* and the subjects contain many more changes (cf. Section 7.2.3). This increases the search space and makes it more complex and hard for the systematic exploration to identify differences in the output.

RQ3+4-A1 HyDiff vs. DF and DDSE. The results in Table 6 show that HyDiff outperforms DF in all three categories: $\overline{t}$ +odiff, $\overline{\texttt{\#odiff}}$ and $\overline{\texttt{\#ddiff}}$, which shows the strength of the hybrid combination. Also for the stronger parallel variant PDF, HyDiff still outperforms it with regard to $\overline{t}$ +odiff. Additionally, HyDiff outperforms DDSE in both categories $\overline{\texttt{\#odiff}}$ and $\overline{\texttt{\#ddiff}}$. Although DDSE is usually faster in identifying its first output difference ($\overline{t}$ +odiff), the absolute time differences are just in the range of seconds. Since HyDiff can classify all subjects correctly in terms of the output difference, it represents a good combination of both approaches, which in isolation cannot identify all subjects correctly. Furthermore, the results in Table 6 as well as in the Figures 17, 18, and 19 show that HyDiff also outperforms both approaches, i.e., the hybrid approach amplifies the exploration.

RQ5-A1 HyDiff for differential testing. A detailed analysis of the identified output differences showed that for most of the subjects the presented techniques cannot identify any crash in the new version. This is expected because the subjects have not been specifically selected to include such crashes. For *Math-60* HyDiff, DF, and PDF identify two crashes in the new version. All three techniques detect the first crash after approximately 5 seconds, and hence, there is no clear benefit in one of these approaches. For *CLI5-6* only PDF identifies a crash in one of the 30 experiment runs, which gives no statistical evidence about PDF's ability to identify this crash reliably. Nevertheless, HyDiff was not able to identify the crash for *CLI5-6*. Apart from crashes, HyDiff identifies all expected output differences in the subjects.

SUMMARY – REGRESSION ANALYSIS (A1)

The conducted experiments indicate that the symbolic execution component of HyDiff can greatly benefit from the combination with fuzzing and vice versa. The performance of differential fuzzing can be improved by running it in a parallel setup, but still it cannot classify all subjects correctly. In contrast, HyDiff does classify all subjects correctly, i.e., it identifies for all subjects output differences, for which they actually exist. Furthermore, the overall evaluation shows that HyDiff still outperforms DDSE, DF, and PDF for the presented subjects.

7.3 WORST-CASE COMPLEXITY ANALYSIS (A2)

This section describes the application scenario *Worst-Case Complexity Analysis* (WCA), which represents the search for an input, which triggers a worst-case execution behavior. The basics and related work on worst-case complexity analysis can be found in Section 2.4.2. This section starts with the explanation of how HyDiff can be applied for the worst-case complexity analysis, and further shows an example and presents the necessary drivers. Afterwards the section shows the evaluation of HyDiff on this application scenario and discusses the results. The evaluation on worst-case complexity analysis extends the evaluation shown in one of the preliminary publications [3].

7.3.1 *Approach*

The main objective of the worst-case complexity analysis is not to identify the theoretical complexity of an algorithm, but to identify inputs that trigger a worst-case execution behavior. Such a characterization of an algorithm's complexity can help for example to identify worst-case complexity vulnerabilities [146]. Due to the simpler nature of the worst-case complexity analysis (compared to the other application scenarios), this approach is quite straightforward: the basic goal is to maximize the observed execution cost.

HyDiff's fuzzing component reduces in this setup to a *cost-guided* fuzzer without differential metrics and focuses only on maximizing the execution cost. The fuzzing driver therefore reads the input, executes the application and reports the observed cost. The mutant selection mechanism only focuses on maximizing the cost value and on increasing the

Listing 17: Sample program for WCA: *Insertion Sort*.

```java
public static void sort(int[] a) {
   final int N = a.length;
   for (int i = 1; i < N; i++) {
      int j = i - 1;
      int x = a[i];
      while ((j >= 0) && (a[j] > x)) {
         a[j + 1] = a[j];
         j--;
      }
      a[j + 1] = x;
   }
}
```

coverage, and hence, selects inputs that increase the so far observed cost *highscore* or that improve the program coverage.

HyDiff's symbolic execution component distinguishes between the concolic and symbolic mode and adds symbolic values accordingly. In contrast to the other application scenarios, the driver adds no change-annotations and there are also no changes in the program. The heuristic for the symbolic exploration (i.e., the ranking strategy for the trie nodes) can be stated as follows:

1. Prioritize new branch coverage.
2. Prioritize higher cost.
3. Prioritize higher/lower nodes in the trie.

The first point in this exploration heuristic is to prioritize the branch coverage, which aims at supporting the fuzzing component. The second goal is to find inputs that increase the execution cost, as the primary goal of the overall analysis. Finally the heuristic prefers nodes that are located higher in the trie (i.e., closer to the root node) because this likely leads to a broader exploration of the search space (point 3). Please note that the last point can also be changed to focus on lower nodes in the trie to push the exploration deeper in the execution path, which might make sense for the worst-case complexity analysis. For the presented evaluation subjects this last point has been varied based on the outcome of the preliminary experiments, in which both variants have been applied.

7.3.2 *Example & Drivers*

Listings 17 shows an implementation of the sorting algorithm *Insertion Sort* taken from JDK 1.5, for which the worst-case execution behavior (in terms of the runtime) is known: N^2, where N denotes the length of the input array. The worst-case would be triggered by a reverse-ordered array.

Listing 18 shows the driver for HyDiff's fuzzing component. For this example the length of the array was fixed to $N = 64$ (cf. line 2 in Listing 18). After reading the input (cf. line 5 to 12), the driver executes the sorting algorithm and measures the execution cost. The cost metric in this case is defined as the number of executed Java bytecode instructions, which is measured by using the instrumented bytecode. At the end the driver reports the obtained cost value (cf. line 16).

Listing 18: Fuzzing driver for *Insertion Sort*.

```java
public static void main(String[] args) {
   int N = 64;
   int a[] = new int[N];

   try (FileInputStream fis = new FileInputStream(args[0])) {
      int b;
      int i = 0;
      while (((b = fis.read()) != -1) && (i < N)) {
         a[i] = b;
         i++;
      }
   } catch (IOException e) {..}

   Mem.clear();
   sort(a);
   Kelinci.addCost(Mem.instrCost);
}
```

Listing 19 shows the driver for HyDiff's symbolic execution component. Similar to the fuzzing driver, the symbolic execution driver works with a fixed array size of $N = 64$ (cf. line 2 in Listing 19). The concolic mode is shown in line 7 to 20, and the symbolic mode is shown in 22 to 24. In the concolic mode the driver first reads the input (cf. line 7 to 14) and then adds symbolic values (cf. line 17 to 19). In the symbolic mode the driver simply adds symbolic values for all array elements (cf. line 22 to 24). Afterwards the driver executes the algorithm (cf. line 27).

Table 7 shows the results for applying DF, PDF, DDSE, and HyDiff on the presented *Insertion Sort* subject. The experiments have been executed for 1 hour and repeated for 30 times. The columns in this table show the average maximum cost obtained within the given time bound ($\bar{c}$), the maximum cost value over all runs (c_{max}), and the time in seconds until the first cost improvement with regard to the cost value of the initial input ($\bar{t} : c > 0$), which had a cost value of 509 bytecode instructions. The numbers in Table 7 show that HyDiff can generate inputs with significantly higher costs. However, within the one hour time bound, none of the techniques has been able to identify **the** worst-case input. The maximum cost value generated by HyDiff was 9,923 and the actual worst-case cost value would be 10,526 for a totally reverse ordered array with $N = 64$. Nonetheless, the input by HyDiff with 9,923 gets very close to this worst-case (cf. Listing 20). Therefore, HyDiff achieves on average a *slowdown* of ca. 19.04x, i.e., that the identified cost value as 19.04x more expensive than the cost value of the initial input. Parallel differential fuzzing (PDF) takes the second position with a slowdown of 18.38x followed by single differential fuzzing (DF) with a slowdown of 17.78x. Far behind is differential dynamic symbolic execution (DDSE) with a slowdown of 2.27x, which cannot achieve high cost values. However, with regard to the time to the first cost improvement, DDSE is the best followed by HyDiff. Please note that these differences are quite small and also the fuzzing techniques show very similar behavior.

More interesting is the comparison of the techniques over the analysis time like shown in Figure 20. During the first 2 minutes DF, PDF, and HyDiff perform very similar, but afterwards HyDiff can break away and can generate an average cost value of 9000 executed bytecode instructions within 8.2 minutes, for which PDF needs 27.5 and DF 54.6 minutes.

Listing 19: Symbolic execution driver for *Insertion Sort*.

```java
public static void main(String[] args) {
   int N = 64;
   int a[] = new int[N];

   if (args.length == 1) {

      try (FileInputStream fis = new FileInputStream(args[0])) {
         int b;
         int i = 0;
         while (((b = fis.read()) != -1) && (i < N)) {
            a[i] = b;
            i++;
         }
      } catch (IOException e) {...}

      // Insert symbolic variables.
      for (int i = 0; i < N; i++) {
         a[i] = Debug.addSymbolicInt(a[i], "sym_" + i);
      }

   } else {
      for (int i = 0; i < N; i++) {
         a[i] = Debug.makeSymbolicInteger("sym_" + i);
      }
   }

   sort(a);
}
```

Listing 20: HyDiff's worst-performing input for *Insertion Sort* N=64 (t=60min).

```
a=[22, 23, 22, 21, 20, 20, 19, 19, 18, 17, 17, 17, 17, 13, 16, 16, 16, 15, 13, 14, 16, 16,
   15, 13, 14, 15, 12, 12, 12, 11, 9, 10, 11, 11, 9, 9, 10, 7, 7, 7, 8, 7, 8, 5, 6, 6, 6,
   6, 4, 6, 5, 5, 5, 5, 4, 3, 2, 2, 2, 2, 2, 2, 1, 0]
```

Table 7: Results for the *Insertion Sort* with N=64 (t=3600sec=60min, 30 runs). The execution cost c is measured as the number of executed Java bytecode instructions.

Technique	$\bar{c}$	c_{max}	$\bar{t} : c > 0$
DF	9,048.40 (±85.51)	9,567	5.70 (±0.16)
PDF	9,355.03 (±41.53)	9,571	5.10 (±0.11)
DDSE	1,157.00 (±0.00)	1,157	2.13 (±0.15)
HyDiff	9,693.77 (±42.44)	9,923	2.93 (±0.16)

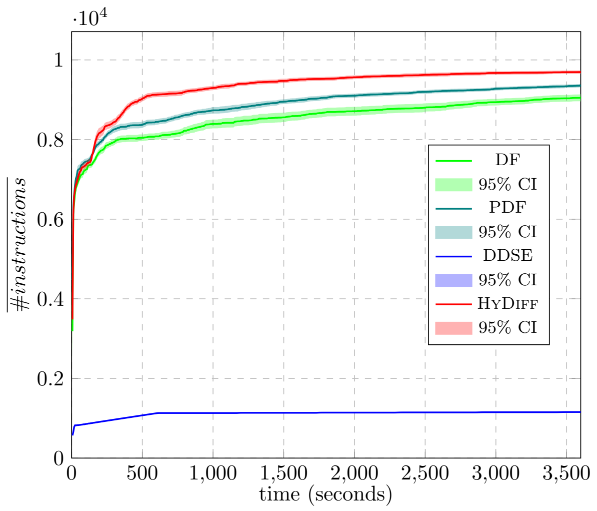


Figure 20: Results for DF, PDF, DDSE, and HyDiff on the *Insertion Sort* Example with $N = 64$ (lines and bands show averages and 95% confidence intervals across 30 repetitions).

7.3.3 *Evaluation Metrics*

The nature of the worst-case complexity analysis (WCA) does not know metrics like *output difference*, *decision difference*, or *cost difference* because the analysis does not reason about multiple paths at once and thus there are no direct *differences* to detect. The main metric for the WCA is the measured *cost* for the execution, which is maximized. In order to assess the experiments and answer the research questions, the results report the following metrics:

- $\bar{c}$: the average maximum cost obtained within the given time bound (higher is better)
- c_{max}: the maximum cost obtained over all runs (higher is better)
- $\bar{t} : c > 0$: the average time to find the first input, which improves the cost value with regard to the initial input as baseline (lower is better)

7.3.4 *Data Sets*

The evaluation subjects have been chosen to match the evaluation of the approach SlowFuzz [146], which is the most related work for this area.

The first two evaluation subjects are two textbook examples: *Insertion Sort* (cf. example in Section 7.3.2) and *Quicksort*. Both implementations are taken from the JDK 1.5 and are executed with $N = 64$, i.e., the input is an array of 64 integers. The initial cost values have been 509 for Insertion Sort and $2,829$ for QuickSort.

The next 12 subjects consider regular expression matching, which can be vulnerable to so called ReDoS (Regular expression DoS) attacks [146]. These subjects are taken from the `java.util.regex` JDK 1.8 package. In the experiments the first 11 subjects (*Regex 1, 2, 3, 4, 5, 6, 7a, 7b, 8, 9, 10*) are set up with fixed regular expressions so that the goal is to identify a text with which the regular expression matching performs poor. The initial input represents some *lorem ipsum* text and is limited to 100 characters. Table 8 shows the selected regular expressions, which have been taken based on ten popular examples [207]. In the last regex experiment (*Regex Lorem*), the text is fixed to the *lorem ipsum* filler text and the regular expression is varied. The regular expression $[\backslash s \backslash S]^*$ is used as initial input and

Table 8: List of regular expression used in the WCA evaluation.

Subject	Regular Expression	Initial Cost
Regex 1-username	^[a-z0-9_-]{3, 15}$	1, 349
Regex 2-password	((?=.*\d)(?=.*[a - z])(?=.*[A - Z])(?=.*[@#$%]).{6, 20})	8, 443
Regex 3-hexcolor	^#([A-Fa-f0-9]{6}| [A-Fa-f0-9]{3})$	2, 235
Regex 4-email	^[_A-Za-z0-9-]+(\\. [_A-Za-z0-9-]+)*@ [A-Za-z0-9]+	6, 282
	(\\. [A-Za-z0-9]+)*(\\. [A-Za-z]{2, })$	
Regex 5-imageext	([^\s]+(\.(?i)(jpg | png | gif | bmp))$)	3, 231
Regex 6-ipaddress	^([01]?\\d\\d?|2 [0-4] \\d|25 [0-5])\\.	9, 823
	([01]?\\d\\d?|2 [0-4] \\d|25 [0-5])\\.	
	([01]?\\d\\d?|2 [0-4] \\d|25 [0-5])\\.	
	([01]?\\d\\d?|2 [0-4] \\d|25 [0-5])$	
Regex 7a-time12hour	(1 [012] | [1-9]) : [0-5] [0-9] (\\s)?(?i)(am|pm)	3, 463
Regex 7b-time24hour	([01]? [0-9] |2 [0-3]) : [0-5] [0-9]	2, 357
Regex 8-date	(0? [1-9] | [12] [0-9] |3 [01])/(0? [1-9] |1 [012])/((19|20)\\d\\d)	4, 861
Regex 9-html	<("[^"]*"|' [^']*'| [^'" >])* >	2, 624
Regex 10-htmllink	(?i) < a([^ >]+) > (.+?) < /a >	2, 018

is limited to 10 characters during the analysis. The cost value for the initial input is already quite large with 68, 101.

The next evaluation subject *HashTable* is an implementation of a hash table taken from the STAC program [204] and modified to match the hash function by SlowFuzz [146], which was taken from a vulnerable PHP implementation [205]. The size of the hash table is fixed 64, each key in the hash table has a length of 8 characters. The hash table is filled by reading the first 64 · 8 characters from an input file. The worst-case of a hash table implementation can be triggered by generating hash collisions. Therefore, besides the costs, we also report the number of hash collisions in Section 7.3.5. The initial cost is 2, 282 with 8 collisions.

The evaluation subject *Compress* is taken from Apache Commons Compress. The experiments for this subject use BZIP2 to compress files up to 250 bytes. The initial cost value is 1, 505, 039. The last subject *Image Processor* is another application taken from the STAC program [204], which includes a vulnerability related to particular pixel values in the input image causing a significantly increased runtime for the program. For the sake of simplicity, the image size is limited 2x2 pixels. The initial cost value is 8, 706.

For all presented subjects the Java bytecode has been instrumented to count the number of executed bytecode *jump* instructions, i.e., a different instrumentation compared to the side-channel analysis (cf. Section 7.4.4). Counting the number of jump instructions means less additional bytecode instruction generated by the instrumentation, but still provides a good estimate about the time for the program execution. In the worst-case complexity analysis, the program execution will, in general, eventually degrade to a very heavy execution because the input is expected to trigger a very long or very resource-consuming execution. Therefore, it is important that the impact of the instrumentation on the actual program execution is as low as possible. Please note that HyDiff allows the usage of any kind of cost definition, for example, a developer could also add some user-defined cost annotations in the code with `Kelinci.addCost(...)`, where the parameter can be any constant integer number or any integer variable. If the variable contains a symbolic value, the symbolic execution component will instruct the constraint solver to maximize the cost expression.

7.3.5 *Evaluation Results*

Table 9 shows the results for the worst-case complexity analysis. The used metrics ($\bar{c}$, c_{max}, and $\bar{t} : c > 0$) have been described in Section 7.3.3 and focus on the execution cost. The highlighted values represent significant differences to the other technique verified with the Wilcoxon rank-sum test (with 5% significance level). The time values are presented in seconds and the values also report the 95% confidence intervals.

HyDiff vs. DF. The results in Table 9 show that DF usually is outperformed by HyDiff or they perform similar in both categories. Only for the subject *Regex 2-password* DF is significantly faster in improving the cost value, although the absolute difference is again only some seconds. In particular, HyDiff outperforms DF:

- for the time to the first cost improvement in 10 of 17 subjects (for another 6 subjects both achieve similar numbers), and
- for the average cost value in 13 of 17 subjects (for the remaining 4 subjects both achieve similar numbers).

Similar to the previous application scenario, the experiments show that the parallel differential fuzzing (PDF) performs better than DF: PDF usually can identify a larger cost value and is also faster in improving the initial cost value. This means that PDF cannot only improve the $\bar{c}$ values, but is also faster than DF in identifying the first $c > 0$. The results in Table 9 show that HyDiff still outperforms PDF in the majority of the cases, but PDF significantly improves over DF and also can outperform HyDiff in some subjects. In particular, HyDiff outperforms PDF:

- for the time to the first cost improvement in 10 of 17 subjects (for another 2 subjects both achieve similar numbers), and
- for the average cost value in 9 of 17 subjects (for another 5 subjects both achieve similar numbers).

HyDiff vs. DDSE. Similar as for the regression analysis, there was no significant difference between the DDSE and DDSEx2T experiments for the subjects in the worst-case complexity analysis. There have been some very minor improvement for the average cost value for the subjects *Regex 1-username*, *Regex 3-hexcolor*, and *Image Processor*, and two larger improvements for *Regex 4-email* and *HashTable*, but which had no impact on the relative comparison to HyDiff. Therefore, the following result discussion will ignore the DDSEx2T experiments and focus on DDSE.

The results in Table 9 show that DDSE usually cannot reveal cost values as high as HyDiff, but is in more subjects significantly faster in improving the initial score (although the absolute differences in time are just seconds). In particular, HyDiff outperforms DDSE:

- for the time to the first cost improvement in 5 of 17 subjects (for one other subject both achieve similar numbers), and
- for the average cost value in 10 of 17 subjects (for another 5 subjects both achieve similar numbers).

Table 9: Results for Worst-Case Complexity Analysis (t=3600sec=60min, 30 runs). The bold values represent significant differences to the closest other technique verified with the Wilcoxon rank-sum test ($\alpha = 0.05$).

Subject	**Differential Fuzzing (DF)**			**Parallel Differential Fuzzing (PDF)**			**Differential Dynamic Sym. Exec. (DDSE)**			**DDSE double time budget (DDSEx2T)**			**HyDiff**		
	$\overline{c}$	c_{max}	$\overline{t}: c > 0$	$\overline{c}$	c_{max}	$\overline{t}: c > 0$	$\overline{c}$	c_{max}	$\overline{t}: c > 0$	$\overline{c}$	c_{max}	$\overline{t}: c > 0$	$\overline{c}$	c_{max}	$\overline{t}: c > 0$
Insertion Sort	9, 048.40 (±85.51)	9, 567	5.70 (±0.16)	9, 355.03 (±41.53)	9, 571	5.10 (±0.11)	1, 157.00 (±0.00)	1, 157	**2.13 (±0.15)**	1, 187.00 (±0.00)	1, 187	**2.13 (±0.15)**	**9,693.77 (±42.44)**	9, 923	2.93 (±0.16)
Quicksort	3, 501.40 (±56.95)	3, 913	17.43 (±3.57)	3, 593.10 (±70.30)	4, 099	**9.80 (±1.26)**	2, 970.00 (±0.00)	2, 970	53.63 (±0.30)	2, 970.00 (±0.00)	2, 970	53.63 (±0.30)	3, 463.97 (±53.18)	3, 915	15.43 (±2.91)
Regex 1-username	2, 203.97 (±12.36)	2, 261	30.83 (±9.75)	2, 240.80 (±16.84)	2, 340	17.73 (±3.85)	2, 373.00 (±0.00)	2, 373	**3.37 (±0.17)**	2, 374.00 (±0.00)	2, 374	**3.37 (±0.17)**	2, 330.23 (±9.81)	2, 372	4.23 (±0.15)
Regex 2-password	22, 887.93 (±60.13)	23, 207	20.43 (±2.69)	**23,113.07 (±65.69)**	23, 380	16.90 (±2.26)	13, 530.37 (±178.51)	13, 623	850.83 (±31.00)	13, 530.37 (±178.51)	13, 623	850.83 (±31.00)	22, 880.60 (±71.71)	23, 334	30.30 (±4.27)
Regex 3-hexcolor	2, 599.43 (±82.36)	2, 843	2, 220.37 (±353.52)	2, 688.30 (±73.73)	2, 843	1, 970.87 (±374.13)	2, 742.00 (±0.00)	2, 742	**2.80 (±0.17)**	2, 764.00 (±0.00)	2, 764	**2.80 (±0.17)**	**2,827.17 (±4.69)**	2, 843	4.10 (±0.54)
Regex 4-email	11, 941.37 (±403.32)	14, 335	25.03 (±6.55)	15, 536.60 (±698.34)	17, 674	17.10 (±2.71)	10, 725.00 (±33.76)	10, 872	44.17 (±1.12)	12, 129.00 (±26.92)	12, 222	44.17 (±1.12)	15, 530.73 (±163.97)	16, 509	32.77 (±7.28)
Regex 5-imageext	9, 252.87 (±109.14)	9, 980	6.70 (±0.16)	9, 837.10 (±163.92)	10, 605	6.77 (±0.15)	8, 706.00 (±0.00)	8, 706	**3.03 (±0.11)**	8, 706.00 (±0.00)	8, 706	**3.03 (±0.11)**	9, 813.93 (±247.53)	11, 945	3.97 (±0.15)
Regex 6-ipaddress	9, 844.83 (±3.99)	9, 889	728.83 (±314.69)	9, 848.17 (±1.76)	9, 863	315.77 (±182.32)	10, 531.00 (±20.00)	10, 531	**31.77 (±0.64)**	10, 531.00 (±20.00)	10, 531	**31.77 (±0.64)**	10, 522.10 (±29.24)	10, 531	38.33 (±0.68)
Regex 7a-time12hour	3, 549.27 (±15.35)	3, 678	311.80 (±97.24)	3, 553.10 (±15.99)	3, 662	147.93 (±57.40)	3, 780.00 (±0.00)	3, 780	**3.07 (±0.13)**	3, 780.00 (±0.00)	3, 780	**3.07 (±0.13)**	3, 780.00 (±0.00)	3, 780	3.90 (±0.11)
Regex 7b-time24hour	2, 409.70 (±3.95)	2, 452	387.57 (±190.67)	2, 419.70 (±6.32)	2, 470	168.60 (±64.43)	2, 470.00 (±0.00)	2, 470	**24.30 (±0.21)**	2, 470.00 (±0.00)	2, 470	**24.30 (±0.21)**	2, 471.63 (±1.06)	2, 477	30.90 (±1.62)
Regex 8-date	4, 954.03 (±16.54)	5, 089	392.53 (±236.12)	4, 967.50 (±16.77)	5, 091	205.90 (±107.65)	5, 175.00 (±0.00)	5, 175	**3.10 (±0.17)**	5, 175.00 (±0.00)	5, 175	**3.10 (±0.17)**	5, 175.00 (±0.00)	5, 175	3.93 (±0.18)
Regex 9-html	13, 741.27 (±1, 560.40)	15, 453	1, 581.90 (±444.86)	15, 453.00 (±0.00)	15, 453	686.50 (±270.07)	15, 453.00 (±0.00)	15, 453	**3.67 (±0.17)**	15, 453.00 (±0.00)	15, 453	**3.67 (±0.17)**	15, 453.00 (±0.00)	15, 453	4.93 (±0.18)
Regex 10-htmllink	5, 728.60 (±1, 009.02)	7, 911	1, 340.43 (±356.25)	7, 892.33 (±10.73)	7, 927	653.13 (±195.53)	**8,421.0 (±0.00)**	8, 421	**2.80 (±0.14)**	**8,421.0 (±0.00)**	8, 421	**2.80 (±0.14)**	7, 941.03 (±37.31)	8, 366	3.53 (±0.18)
HashTable	5, 214.43 (±216.76)	6, 997	3.93 (±0.66)	5, 996.80 (±285.34)	7, 502	2.83 (±0.49)	3, 481.80 (±1.09)	3, 483	3.30 (±0.16)	3, 711.07 (±134.84)	4, 902	3.30 (±0.16)	5, 796.43 (±345.75)	8, 337	3.23 (±0.47)
Image Processor	138, 933.60 (±14, 010.11)	232, 374	5.37 (±1.17)	167, 635.20 (±13, 231.49)	235, 146	**3.93 (±0.52)**	80, 238.00 (±0.00)	80, 238	69.13 (±0.94)	80, 299.20 (±19.15)	80, 346	69.13 (±0.94)	**216,992.40 (±19,600.09)**	434, 082	5.70 (±1.13)

	Regex Lorem			Compress		
	$\overline{c}$	c_{max}	$\overline{t}: c > 0$	$\overline{c}$	c_{max}	$\overline{t}: c > 0$
Differential Fuzzing (DF)	102, 086.07 (±8, 025.79)	156, 896	1, 690.20 (±445.00)	1, 699, 242.83 (±13, 537.78)	1, 779, 871	12.83 (±1.99)
Parallel Differential Fuzzing (PDF)	203, 839, 935.67 (±392, 592, 842.34)	6, 111, 914, 017	**783.77 (±342.45)**	**1,734,081.90 (±11,552.57)**	1, 789, 608	10.30 (±1.32)
Differential Dynamic Sym. Exec. (DDSE)	68, 101 (±0.00)	68, 101	-	1, 509, 791.00 (±0.00)	1, 509, 791	**8.20 (±0.14)**
DDSE double time budget (DDSEx2T)	68, 101 (±0.00)	68, 101	-	1, 509, 791.00 (±0.00)	1, 509, 791	**8.20 (±0.14)**
HyDiff	190, 958, 572.77 (±366, 073, 662.23)	5, 699, 892, 712	1, 559.03 (±458.24)	1, 686, 551.67 (±12, 242.49)	1, 762, 605	10.63 (±0.25)

Temporal development of the results. In order to compare the efficiency of the techniques, the Figures 21, 22, 23, and 24 show the temporal development for the subjects *Quicksort*, *Regex 3-hexcolor*, *Regex 4-email*, and *Image Processor*. The temporal development of *Insertion Sort* has been already shown in Figure 20 in Section 7.3.2.

For *Quicksort* (cf. Figure 21) DF, PDF, and HyDiff perform very similar as they continuously improve the cost value. DDSE stays in a plateau for a long time and makes only minor improvements compared to the initial input. Therefore, HyDiff is mostly driven by its fuzzing component.

For *Regex 3-hexcolor* (cf. Figure 22), HyDiff shows a similar development as DDSE because DDSE performs much better than fuzzing. Both jump relatively fast to a high cost value and later improve only slightly. DF and PDF can improve their values over time, but cannot reach the cost value of HyDiff.

For *Regex 4-email* (cf. Figure 23), HyDiff finishes with a similar cost value as PDF after 30 minutes, but performs significantly better in the meantime. Standalone DDSE also can continuously improve the score, but at the end performs worse than DF. PDF shows its benefits over DF as it is faster in generating higher scores.

For *Image Processor* (cf. Figure 24), HyDiff totally outperforms the other techniques. It is very fast in generating a high cost value and also finishes with a significantly higher cost value than the other three techniques. DDSE does not perform as good as the other techniques, which shows that HyDiff can use both components to achieve a better value.

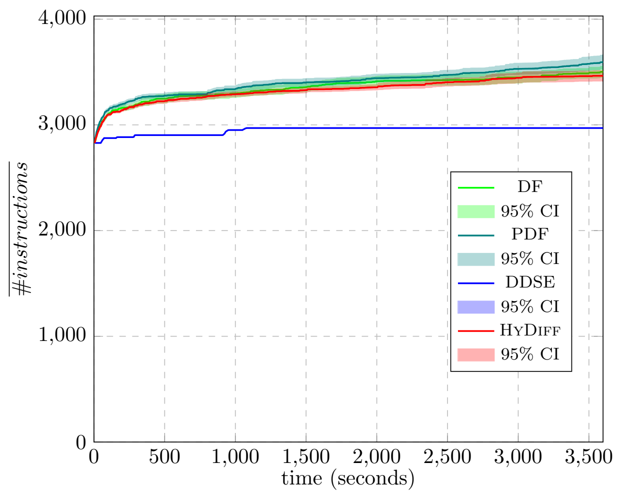


Figure 21: Results for DF, PDF, DDSE, and HyDiff on the *Quicksort* subject with N = 64 (lines and bands show averages and 95% confidence intervals across 30 repetitions).

For a better illustration of the results Table 10 shows the slowdown rates for each subject. The goal of this analysis is to generate an input that performs very low in terms of execution cost, therefore the slowdown is an important assessment metric. The slowdown rates are calculated by the quotient of the initial cost and the average observed cost value. The bold values in this table show the largest slowdown for the specific subject, although it does not need to be a significant difference. The results show that HyDiff generates in the majority of the cases the best slowdown. The number of collisions for the *HashTable* subject are calculated based on the maximum observed cost difference.

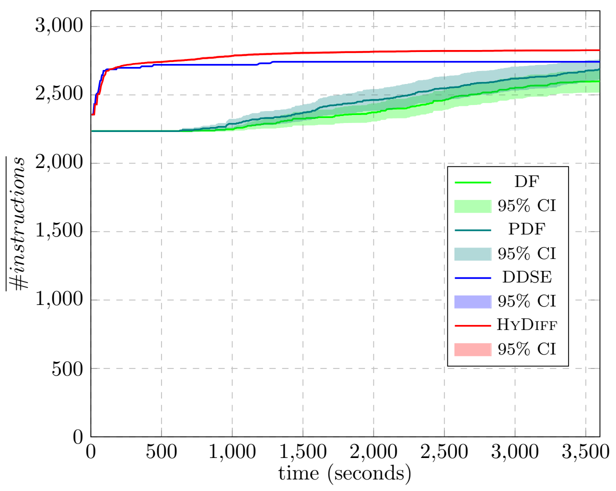


Figure 22: Results for DF, PDF, DDSE, and HyDiff on the *Regex 3-hexcolor* subject (lines and bands show averages and 95% confidence intervals across 30 repetitions).

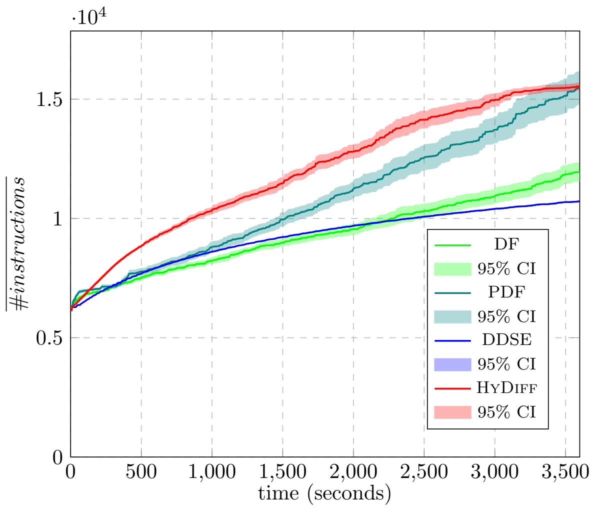


Figure 23: Results for DF, PDF, DDSE, and HyDiff on the *Regex 4-email* subject (lines and bands show averages and 95% confidence intervals across 30 repetitions).

7.3.6 *Discussion*

RQ1-A2 Differential fuzzing (DF). The results in Table 9 show that differential fuzzing is an effective technique for the generation of low performing inputs as for all subjects it can generate a slowdown. Additionally, the temporal development characterized in Figures 21, 22, 23, and 24 shows that DF does improve over time. Nevertheless, the results also show the other techniques can significantly outperform the slowdowns by DF as well as the time to generate its first slowdown. The main disadvantage is that DF needs relatively long to achieve comparable slowdown rates. Parallel differential fuzzing (PDF) shows its advantage over a single differential fuzzing instance (DF), so that PDF can constantly show better results than DF. Although there are 3 subjects, where PDF results in a better cost value than HyDiff, for most of the cases differential fuzzing is outperformed by the other techniques. Due to the fact that the mutations in DF are applied randomly it takes time to identify a low performing input.

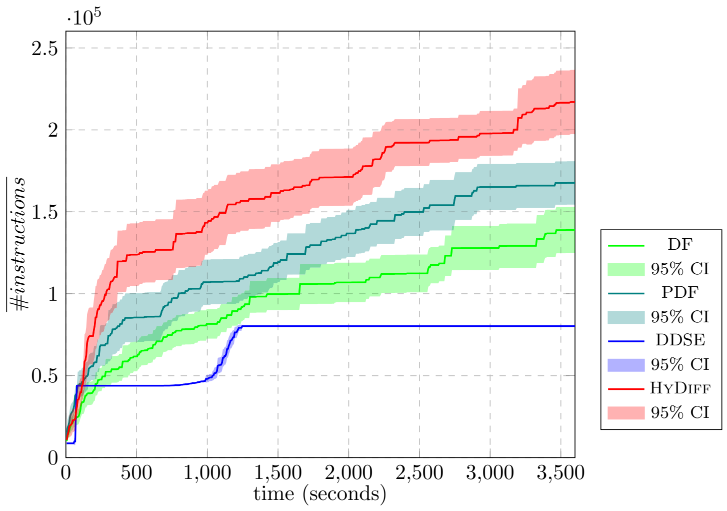


Figure 24: Results for DF, PDF, DDSE, and HyDiff on the *Image Processor* for a 2x2 JPEG image (lines and bands show averages and 95% confidence intervals across 30 repetitions).

RQ2-A2 Differential dynamic symbolic execution (DDSE). For the majority of the subjects DDSE is significantly faster than DF and HyDiff in generating its first improvement for the cost value, although the difference is in the range of seconds. On the other hand HyDiff can outperform the cost value of DDSE in most of the cases. Only in 3 of 17 cases DDSE achieves a better cost value. The Figures 21, 22, 23, and 24 show that DDSE often makes jumps to larger slowdowns and does not improve continuously over time. Its advantage is often in a high jump straight in the beginning because its exploration strategy selects branches with already large cost values so that it can dig even deeper into the program execution. Nevertheless, it often remains in plateaus, while the other techniques can make progress over time. The systematic exploration obviously also explores branches that do not directly produce larger slowdowns.

RQ3+4-A2 HyDiff vs. DF and DDSE. The results in Table 9 show that HyDiff is in the large majority of the cases better than DF and the experiments never show that DF is better than HyDiff. HyDiff often can leverage the power of symbolic execution to quickly get to a high value, from where it can improve the cost value over time. Therefore, HyDiff outperforms DF in both categories: generating a higher cost value and being faster in doing so. Even when PDF can improve the performance of DF, it still cannot reach the same results as HyDiff. As DDSE often remains in plateaus, HyDiff can leverage its fuzzing component to continuously improve the cost value. As the numbers in Table 9 as well as the Figures 21, 22, 23, and 24 show, HyDiff combines both techniques very well and even produces better results. HyDiff is significantly faster in producing high slowdowns and even after 1 hour analysis time HyDiff often remains in larger slowdowns.

RQ5-A2 HyDiff for differential testing. The obtained results show (also see Table 10) HyDiff and its components are effective in generating high slowdowns for the considered subjects. The slowdowns represent very large runtimes of the inputs in these applications (cf. the c_{max} column in Table 9), which represent relevant algorithmic complexity vulnerabilities.

Table 10: List of generated slowdowns in the WCA evaluation (based on the average cost value at the end of the analysis). The highlighted values show the largest numbers (which does not mean a statistical significance).

Subject	**Initial Cost**	**Slowdown**				
		DF	PDF	DDSE	DDSEx2T	HyDiff
Insertion Sort	509	17.78	18.38	2.27	2.33	**19.04**
Quicksort	2,829	**1.24**	**1.27**	1.05	1.05	1.22
Regex 1-username	1,349	1.63	1.66	**1.76**	**1.76**	1.73
Regex 2-password	8,443	2.71	**2.74**	1.60	1.60	2.71
Regex 3-hexcolor	2,235	1.16	1.20	1.23	1.24	**1.26**
Regex 4-email	6,282	1.90	**2.47**	1.71	1.93	**2.47**
Regex 5-imageext	3,231	2.86	**3.04**	2.69	2.69	**3.04**
Regex 6-ipaddress	9,823	1.00	1.00	**1.07**	**1.07**	**1.07**
Regex 7a-time12hour	3,463	1.02	1.03	**1.09**	**1.09**	**1.09**
Regex 7b-time24hour	2,357	1.02	1.03	**1.05**	**1.05**	**1.05**
Regex 8-date	4,861	1.02	1.02	**1.06**	**1.06**	**1.06**
Regex 9-html	2,624	1.89	1.89	**1.97**	**1.97**	**1.97**
Regex 10-htmllink	2,018	2.84	3.91	**4.17**	**4.17**	3.94
Regex Lorem	68,101	1.50	**2,993.20**	1.00	1.00	2,804.05
HashTable	2,282	2.29	**2.63**	1.53	1.63	2.54
HashTable collisions	8	24	30	0	21	**35**
Compress	1,505,039	1.13	**1.15**	1.00	1.00	1.12
Image Processor	8,706	15.96	19.26	9.22	9.22	**24.92**

summary – worst-case complexity analysis (a2)

HyDiff successfully combines the strengths of DDSE and DF. Symbolic execution helps HyDiff to quickly make progress and fuzzing supports by continuously improving the score. Therefore, in the majority of the cases HyDiff is as good as its components or even outperforms them. In particular HyDiff's strength is to quickly generate a high cost value, for which the other techniques take quite long. Great examples are *InsertionSort* and *ImageProcessor*.

7.4 side-channel analysis (a3)

This section describes the application scenario *side-channel analysis*, which aims at finding leakages of secret information by observing the non-functional system behavior, like execution time, memory consumption, or response message sizes. The basics on information theory and side-channel analysis are explained in the background Section 2.4.3. This section starts with the explanation on how to use HyDiff and its components for the side-channel analysis. For further illustration the section presents a simple example for a side-channel vulnerability and how to implement the HyDiff's drivers for this example. Finally, it presents the conducted evaluation of HyDiff on the side-channel subjects and discusses the results. The evaluation on side-channel analysis extends the evaluation discussed in two of the preliminary publications [1] and [6].

7.4.1 *Approach*

The key idea for the implementation of side-channel analysis with HyDiff is to use the idea of self-composition [106] and consider two execution paths, which both are initialized with the same *public* input but different *secret* inputs. The goal is to maximize the cost difference (δ) between these two execution paths. The higher the cost difference can be identified, the more severe is a side-channel vulnerability. This idea is described by the following formula:

$$\underset{pub, sec_1, sec_2}{\text{maximize:}} \ \delta = |c(P[\![pub, sec_1]\!]) - c(P[\![pub, sec_2]\!])|$$

In this formula pub denotes the public value, sec_1 and sec_2 denote the two secret values. $P[\![pub, sec_1]\!]$ denotes the execution of program P with the public value pub and the secret value sec_1. $c(P[\![..]\!])$ denotes the cost measurement of the execution of program P. δ denotes the cost difference of both program executions.

For HyDiff's fuzzing component this means to fuzz three values: the public value and two secret values. Note that this approach naturally extends to tuples of values. The driver for differential fuzzing needs to take these three values and perform two program executions: both with the same public input, but with different secret inputs. The most important metric to detect the side-channel vulnerability is the cost difference between these two executions, but the driver will also collect the information about decision differences and output differences. They are still important metrics to drive the fuzzing process, although they cannot directly measure the severity of a side-channel vulnerability.

For HyDiff's symbolic execution component the variation in the secret input can be realized by the usage of change-annotations:

$$secret = change(secret_1, secret_2)$$

The driver for differential dynamic symbolic execution does also read three inputs (one public and two secret values), but does combine the two secret values in one change-annotated expression. This means that for symbolic execution there is only one program execution necessary. Since this change-annotation happens directly in the driver, the program itself does not contain any change-annotation, and hence, the patch distance metric is not relevant for side-channel analysis. Also the control-flow information can not help to prune any trie node because the differential expression is introduced straight in the beginning. This also means that the heuristic for the symbolic exploration (i.e., the ranking strategy for the trie nodes) can be simplified as follows:

1. Prioritize new branch coverage.
2. Prioritize higher cost difference.
3. Prioritize higher nodes in the trie.

The primary goal of the symbolic execution in the hybrid setup is to support the fuzzing component by solving complex branching conditions, which are infeasible for fuzzing. Therefore, the first point in this exploration heuristic is to prioritize the branch coverage. The second goal is to find inputs that increase the cost difference, since they signal side-channel vulnerabilities (point 2). Finally the heuristic prefers nodes that are located higher in the trie (i.e., closer to the root node) because this likely leads to a broader exploration of the search space (point 3). Please note that technically such an exploration strategy can be easily modified so that different analysis types can be incorporated. Some preliminary assessment of this heuristic showed no improvement with a changed prioritization order.

7.4.2 *Example & Drivers*

To illustrate the analysis approach, please consider the *unsafe* password comparison algorithm from Section 2.4.3, which is shown again in Listing 21.

Listing 21: Side-channel analysis example password checking algorithm (see also Listing 2).

```java
boolean pwcheck_unsafe(byte[] pub, byte[] sec) {
   if (pub.length != sec.length) {
      return false;
   }
   for (int i = 0; i < pub.length; i++) {
      if (pub[i] != sec[i]) {
         return false;
      }
   }
   return true;
}
```

The Listings 22, 23, and 24 show the drivers for this example. The driver for differential fuzzing (cf. Listings 22) parses the input and creates three arrays with the same length: two arrays for the secret input and one array for the public input (cf. Listing 22 lines 4 to 29). Afterwards the driver calls the application first with the public input and the first secret input (cf. lines 32 to 41), and afterwards with the same public input but with the second secret input (lines 44 to 53). At the end the driver reports the observed differences to the fuzzer (cf. lines 56 to 60). The main difference to the drivers in the regression analysis is the way on how to call the applications: for the regression analysis the driver executes the old and the new version with the same input, for side-channel analysis the driver executes the same program but varies the secret input.

The driver for the differential dynamic symbolic execution (cf. Listing 23) distinguishes two executions modes (similar as for regression analysis): one for the concolic execution phase, which reads a concrete input and adds symbolic variables (lines 8 to 35), and one for the symbolic execution phase, which simply generates symbolic variables (lines 37 to 47). Afterwards the driver calls a helper function (cf. Listing 24) to handle arbitrary input sizes up to a defined maximum size and to introduce the changes. This special handling is necessary because the observed side-channels represent non-functional characteristics of the program, which in general can be affected by the input size. Therefore, the analysis should incorporate multiple input sizes. The fuzzing driver simply reads the input up to a maximum input size. In symbolic execution the handling of arbitrary input sizes is more sophisticated because, e.g., it is difficult to handle arrays with a symbolic length. In order to solve this problem the driver helper introduces a decision straight in beginning, i.e., before the actual application execution. This decision determines the current size of the input. This initial decision allows the remaining symbolic execution to perform the analysis in this sub-tree with a fixed size array.

Technically, the decision introduction works as follows: the process always starts with a concrete input, which is concollicaly executed. The driver recognizes this input size, which is concrete, and the driver helper inserts a decision straight in beginning by checking the input size n (cf. Listing 24 line 3 to 16). In the trie this *dummy* decision will be represented as a trie node in the very beginning. During the trie extension, which happens during the concolic execution, the current input size is assigned to the trie nodes. Therefore, each trie

Listing 22: Driver for differential fuzzing on password checking example.

```java
public static void main(String[] args) {

   // Read all inputs up to 16x3 bytes (simplified illustrated).
   List<Byte> values = ... ;

   if (values.size() < 3) {
      throw new RuntimeException("Not enough input data...");
   }
   int m = values.size() / 3;

   byte[] secret1_pw, secret2_pw, public_guess;

   // Read public.
   public_guess = new byte[m];
   for (int i = 0; i < m; i++) {
      public_guess[i] = values.get(i);
   }

   // Read secret1.
   secret1_pw = new byte[m];
   for (int i = 0; i < m; i++) {
      secret1_pw[i] = values.get(i + m);
   }

   // Read secret2.
   secret2_pw = new byte[m];
   for (int i = 0; i < m; i++) {
      secret2_pw[i] = values.get(i + 2 * m);
   }

   /* Execute with secret1. */
   Mem.clear();
   DecisionHistory.clear();
   Object res1 = null;
   try {
      res1 = PWCheck.pwcheck_unsafe(public_guess, secret1_pw);
   } catch (Throwable e) {
      res1 = e;
   }
   boolean[] dec1 = DecisionHistory.getDecisions();
   long cost1 = Mem.instrCost;

   /* Execute with secret2. */
   Mem.clear();
   DecisionHistory.clear();
   Object res2 = null;
   try {
      res2 = PWCheck.pwcheck_unsafe(public_guess, secret2_pw);
   } catch (Throwable e) {
      res2 = e;
   }
   boolean[] dec2 = DecisionHistory.getDecisions();
   long cost2 = Mem.instrCost;

   /* Report differences. */
   DecisionHistoryDifference d = DecisionHistoryDifference
      .createDecisionHistoryDifference(dec1, dec2);
   Kelinci.setNewDecisionDifference(d);
   Kelinci.setNewOutputDifference(new OutputSummary(res1, res2));
   Kelinci.addCost(Math.abs(cost1 - cost2));
}
```

Listing 23: Driver for differential dynamic symbolic execution on password checking example.

```java
public static void main(String[] args) {

   byte[] secret1_pw, secret2_pw, public_guess;
   int n;

   if (args.length == 1) {

      /* Read all data up to 16x3 bytes (simplified illustrated). */
      List<Byte> values = ... ;

      if (values.size() < 3) {
         throw new RuntimeException("Not enough input data...");
      }

      int n_concrete = values.size() / 3;
      n = Debug.addSymbolicInt(values.size() / 3, "sym_n");

      // Read public and insert symbolic variables.
      public_guess = new byte[n_concrete];
      for (i = 0; i < n_concrete; i++) {
         public_guess[i] = Debug.addSymbolicByte(values.get(i), "sym_0_" + i);
      }

      // Read secret1 and insert symbolic variables.
      secret1_pw = new byte[n_concrete];
      for (i = 0; i < n_concrete; i++) {
         secret1_pw[i] = Debug.addSymbolicByte(values.get(i + n_concrete), "sym_1_" + i);
      }

      // Read secret2 and insert symbolic variables.
      secret2_pw = new byte[n_concrete];
      for (i = 0; i < n_concrete; i++) {
         secret2_pw[i] = Debug.addSymbolicByte(values.get(i + 2 * n_concrete), "sym_2_" +
             i);
      }

   } else {
      int currentN = Debug.getLastObservedInputSizes()[0];
      public_guess = new byte[currentN];
      secret1_pw = new byte[currentN];
      secret2_pw = new byte[currentN];

      n = Debug.makeSymbolicInteger("sym_n");
      for (int i = 0; i < currentN; i++) {
         public_guess[i] = Debug.makeSymbolicByte("sym_0_" + i);
         secret1_pw[i] = Debug.makeSymbolicByte("sym_1_" + i);
         secret2_pw[i] = Debug.makeSymbolicByte("sym_2_" + i);
      }
   }

   driver_helper(n, public_guess, secret1_pw, secret2_pw);
}
```

Listing 24: Driver helper function for differential dynamic symbolic execution on password checking example.

```
1  public static void driver_helper(int n, byte[] guess, byte[] secret_1, byte[] secret_2) {
2     int[] sizes = new int[1];
3     switch (n) {
4        case 0:
5           sizes[0] = 0;
6           break;
7        case 1:
8           sizes[0] = 1;
9           break;
10       ...
11       case 16:
12          sizes[0] = 16;
13          break;
14       default:
15          throw new RuntimeException("unintended input size");
16    }
17
18    byte[] secret = new byte[secret_1.length];
19    for (int i = 0; i < secret.length; i++) {
20       secret[i] = (byte) change(secret_1[i], secret_2[i]);
21    }
22
23    Object res = PWCheck.pwcheck_unsafe(guess, secret);
24 }
```

Table 11: Results for the password checking example (t=300sec=5min, 30 runs).

Technique	$\bar{\delta}$	δ_{max}	$\bar{t} : \delta > 0$
Differential Fuzzing (DF)	34.30 (±3.11)	47	4.20 (±1.53)
Parallel Differential Fuzzing (PDF)	40.93 (±1.84)	47	2.33 (±0.63)
Differential Dynamic Symbolic Execution (DDSE)	47.00 (±0.00)	47	13.27 (±0.24)
HyDiff	47.00 (±0.00)	47	4.43 (±1.00)

node knows the expected input size. In the symbolic execution phase, the input size, which is assigned to the identified trie node (cf. Figure 9 step 3), is used to initialize the symbolic variables (cf. Listing 23 line 37). Please note that such special handling of multiple input sizes would be also necessary when incorporating multiple input sizes in other analysis types like regression analysis. But the evaluation subjects used for regression analysis (cf. Section 7.2.3) consider simple input types, for which the input size is not relevant.

Before calling the application the driver helper introduces the previously mentioned change-annotations to combine the two secret inputs (cf. Listing 24 line 18 to 21).

Table 11 shows the experiment results for applying DF, PDF, DDSE, and HyDiff on the password checking example. The experiments have been executed with a time bound of 5 minutes and have been repeated 30 times. The number of executed bytecode instructions are used as cost metric, which is an alternative to measure the real runtime of the algorithm. Other processes running on the machine are expected to influence the actual real time measurement, and hence, counting the executed bytecode instruction is a more robust metric.

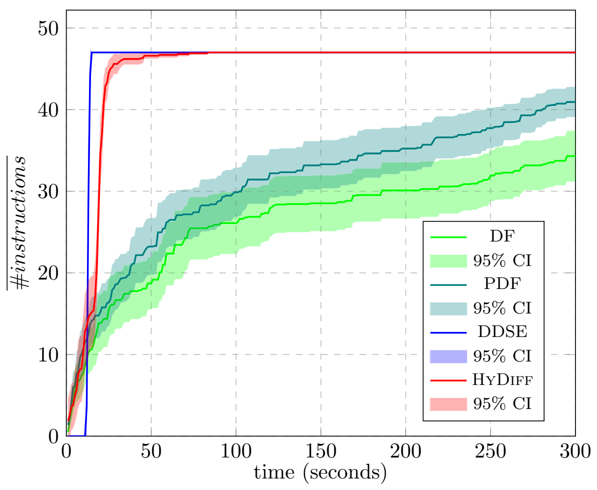


Figure 25: Results for DF, PDF, DDSE, and HyDiff on the Password Checking Example (lines and bands show averages and 95% confidence intervals across 30 repetitions).

Listing 25: Input for maximum cost difference after 5 min.

```
pub=[16, 0, 108, 108, 111, 32, 67, 97, 72, 101, 108, 108, 111, 32, 67, 97]
sec_1=[114, 110, 101, 103, 105, 101, 32, 77, 101, 108, 114, 110, 101, 103, 105, 101]
sec_2=[16, 0, 108, 108, 111, 32, 67, 97, 72, 101, 108, 108, 111, 32, 67, 97]
```

For this experiment, the maximum input size was set to 16 bytes, which allows a maximum cost difference of 47 bytecode instructions. Listing 25 shows an input, which triggers the maximum cost difference.

All of the four experiment setups (DF, PDF, DDSE, and HyDiff) have been able to reach the maximum value at least once (cf. δ_{max} column in Table 11). The values in the table also show that DF and PDF cannot reliably generate this maximum value within the time bound of 5 minutes. However, the fuzzing techniques show a better performance in identifying the first input for an improved δ value (cf. column $\bar{t} : \delta > 0$). Figure 25 shows the temporal development. DF, PDF, and HyDiff perform quite similar in the beginning, whereas DDSE takes longer to generate an interesting input. After approximately 13 seconds, DDSE jumps almost directly to the actual maximum cost difference. HyDiff follows quickly, whereas DF and PDF take longer to get to this maximum value.

7.4.3 *Evaluation Metrics*

The evaluation metrics are related to the metrics used in the worst-case complexity analysis, which handles the maximization of execution cost. However, the most important metric for the side-channel analysis is the *cost difference* because it reveals a side-channel vulnerability with respect to the leveraged cost metric. The metrics *output difference* and *decision difference* might be helpful during the search process, i.e., these metrics should be used to drive the differential exploration in fuzzing (cf. Section 7.4.1), but they cannot assess the ability to reveal side-channel vulnerabilities. Especially the metric *output difference* is not interesting to report because it represents information in the main-channel of the application and not in any side-channel. In order to assess the experiments and answer the research questions, the results report the following metrics:

- $\overline{\delta}$: the average maximum cost difference obtained within the given time bound (higher is better)
- δ_{max}: the maximum cost difference obtained over all runs (higher is better)
- $\overline{t} : \delta > 0$: the average time to find the first input, which improves the δ value with regard to the initial input as baseline (lower is better)

7.4.4 *Data Sets*

The evaluation for the side-channel analysis is two-fold: (1) it first compares the proposed differential fuzzing strategy with state-of-the-art static analysis tools Blazer [104] and Themis [110], and (2) applies HyDiff on a selected benchmark to show the strengths of the hybrid combination.

The benchmarks for the direct comparison with Blazer and Themis are taken from their publications and contain programs with known time and space side-channel vulnerabilities. These subjects provide an *unsafe* and a *safe* variant. The *safe* variant usually means that there is no information leakage. Blazer's subjects are small applications with up to a hundred lines of code. Themis' subjects are larger Java programs with up to 20K LOC and are taken from complex-real world applications, such as Tomcat, Spring-Security, and Eclipse Jetty HTTP web server. Please note that although these applications are quite large, the analysis is usually focused on suspicious components, so that not the complete application is covered by the analysis. All benchmarks except *DynaTable*, *Advanced_table*, *OpenMRS*, and *OACC* come with a repaired version. The subjects for *Tomcat* and *pac4j* require interactions with a database. Therefore, these experiments needed the prior setup of a database, for which the H2 database engine [206] was used to create an SQL database accessible via the JDBC API.

For the side-channel analysis differential fuzzing already performs quite well, therefore, only a subset of subjects is selected to evaluate the hybrid combination: *Blazer_login* (25 LOC) and *Themis_Jetty* (17 LOC), and a sophisticated authentication procedure *STAC_ibasys* (707 LOC) taken from the STAC program [204], which handles complex image manipulations. These subjects provide an *unsafe* and a *safe* variant. The subject *Themis_Jetty Safe* is known to still leak information (but the difference in cost is small). Additionally the subjects include an implementation of modular exponentiation, *RSA_modpow* (30 LOC), from [121]. This implementation is known to have a timing side-channel due to an optimized step in the exponentiation procedure. Paul Kocher [116] has shown how a similar vulnerability can be exploited to break RSA encryption/decryption. The subjects include three variants of *RSA_modpow* with different values for modulo: 1717, 834443, and 1964903306.

For all presented subjects the Java bytecode has been instrumented to count the number of executed bytecode instructions, so that this number can be used as cost value. Therefore, the cost difference means the difference in the number of executed bytecode instructions. Please note that counting all bytecode instructions might result in a lower performance with regard to the execution of the bytecode, but provides the highest precision, which is desirable when searching for side-channel vulnerabilities. Usually, the bytecode is also instrumented to track the decision history, but for three subjects this was not feasible because the Java heap space would have been exceeded during the bytecode execution. Therefore, for *LoopAndbranch*, *Sanity*, and *unixlogin* the instrumentation does not keep track of the decisions made during program execution.

Table 12: The results of applying differential fuzzing to the BLAZER subjects. Discrepancies are highlighted in red and italics.

Benchmark	Version	Differential Fuzzing (DF)			Time (s)	
		$\bar{\delta}$	δ_{max}	$\bar{t} : \delta > 0$	BLAZER	THEMIS
MicroBench						
Array	Safe	1.00 (±0.00)	1	4.83 (±1.18)	1.60	0.28
Array	Unsafe	195.00 (±0.00)	195	3.53 (±0.71)	0.16	0.23
LoopAndbranch	*Safe*	*2.05×10^9 (±6.63×10^8)*	*4.29×10^9*	*18.33 (±6.64)*	*0.23*	*0.33*
LoopAndbranch	Unsafe	4.20×10^9 (±8.17×10^7)	4.29×10^9	15.67 (±15.19)	0.65	0.16
Sanity	Safe	0.00 (±0.00)	0	-	0.63	0.41
Sanity	Unsafe	4.29×10^9 (±1.08×10^6)	4.29×10^9	18.57 (±6.25)	0.30	0.17
Straightline	Safe	0.00 (±0.00)	0	-	0.21	0.49
Straightline	Unsafe	8.00 (±0.00)	8	5.57 (±1.50)	22.20	5.30
unixlogin	Safe	2.00 (±0.00)	2	438.50 (±101.64)	0.86	-
unixlogin	Unsafe	2.69×10^9 (±2.28×10^8)	3.20×10^9	313.03 (±78.50)	0.77	-
STAC						
modPow1	Safe	0.00 (±0.00)	0	-	1.47	0.61
modPow1	Unsafe	2,301.90 (±130.90)	3,630	2.70 (±0.51)	218.54	14.16
modPow2	Safe	0.00 (±0.00)	0	-	1.62	0.75
modPow2	Unsafe	59.73 (±18.01)	135	78.70 (±18.01)	7,813.68	141.36
passwordEq	Safe	0.00 (±0.00)	0	-	2.70	1.10
passwordEq	Unsafe	122.87 (±4.27)	127	7.17 (±2.39)	1.30	0.39
Literature						
k96	Safe	0.00 (±0.00)	0	-	0.70	0.61
k96	Unsafe	38,799.63 (±7,455.08)	112,557	13.47 (±4.13)	1.29	0.54
gpt14	*Safe*	*504.50 (±106.86)*	*1,048*	*1.83 (±0.19)*	*1.43*	*0.46*
gpt14	Unsafe	84,833.23 (±11,605.25)	147,465	22.80 (±8.86)	219.30	1.25
login	Safe	0.00 (±0.00)	0	-	1.77	0.54
login	Unsafe	132.87 (±14.87)	238	5.07 (±1.18)	1.79	0.70

7.4.5 *Evaluation Results*

Differential Fuzzing (DF) on Blazer benchmark. Table 12 shows the results for the BLAZER subjects. THEMIS also evaluated their approach with BLAZER so that Table 12 shows also the results of THEMIS. For differential fuzzing (DF) the table reports the metrics $\bar{\delta}$, δ_{max}, and $\bar{t} : \delta > 0$ as described in Section 7.4.3. The metrics focus on the cost difference (δ) as well as time to identify the first cost difference. The time values are presented in seconds and the values also report the 95% confidence intervals.

BLAZER and THEMIS report both the time in seconds until their static analysis came to a conclusion. Both always agree with the categories mentioned in the *Version* column. For the majority of the subjects differential fuzzing (DF) also agrees with the categories. As the δ values in Table 12 show: for *safe* the average δ is 0 or very small and for *unsafe* the average δ is quite large. In particular, DF identified all vulnerabilities in the *unsafe* versions. Nevertheless, the evaluation results also indicate some discrepancies (see Table 12 in red and italics). For the safe variants of the subjects *Array* and *unixlogin* DF identified values slightly greater than zero. These discrepancies may be attributed to differences between

Table 13: The results of applying differential fuzzing to the Themis subjects. Discrepancies are highlighted in red and italics.

Benchmark	Version	Differential Fuzzing (DF)			Themis		
		$\bar{\delta}$	δ_{max}	$\bar{t} : \delta > 0$	$\epsilon = 64$	$\epsilon = 0$	Time (s)
Spring-Security	Safe	1.00 (±0.00)	1	4.77 (±1.07)	✓	✓	1.70
Spring-Security	Unsafe	149.00 (±0.00)	149	4.17 (±0.90)	✓	✓	1.09
JDK7-MsgDigest	Safe	1.00 (±0.00)	1	10.77 (±2.12)	✓	✓	1.27
JDK6-MsgDigest	Unsafe	140.03 (±20.39)	263	3.20 (±0.81)	✓	✓	1.33
Picketbox	Safe	1.00 (±0.00)	1	16.90 (±3.89)	✓	✗	1.79
Picketbox	Unsafe	363.70 (±562.18)	8,822	5.13 (±1.83)	✓	✓	1.55
Tomcat	Safe	25.07 (±0.36)	26	19.90 (±9.29)	✓	✗	9.93
Tomcat	*Unsafe*	49.00 (±0.36)	50	23.53 (±9.73)	✓	✓	8.64
Jetty	*Safe*	11.77 (±0.60)	15	3.77 (±0.72)	✓	✓	2.50
Jetty	Unsafe	70.87 (±6.12)	105	6.83 (±1.62)	✓	✓	2.07
orientdb	Safe	1.00 (±0.00)	1	16.60 (±5.14)	✓	✗	37.99
orientdb	Unsafe	458.93 (±685.64)	10,776	4.77 (±1.06)	✓	✓	38.09
pac4j	Safe	10.00 (±0.00)	10	1.10 (±0.11)	✓	✗	3.97
pac4j	*Unsafe*	11.00 (±0.00)	11	1.13 (±0.12)	✓	✓	1.85
pac4j	*Unsafe**	39.00 (±0.00)	39	1.10 (±0.11)	-	-	-
boot-auth	Safe	5.00 (±0.00)	5	1.00 (±0.00)	✓	✗	9.12
boot-auth	Unsafe	101.00 (±0.00)	101	1.00 (±0.00)	✓	✓	8.31
tourPlanner	Safe	0.00 (±0.00)	0	-	✓	✓	22.22
tourPlanner	Unsafe	238.00 (±21.78)	353	57.07 (±6.47)	✓	✓	22.01
DynaTable	Unsafe	75.40 (±3.83)	94	3.90 (±0.97)	✓	✓	1.165
Advanced_table	Unsafe	23.03 (±8.08)	73	783.80 (±318.00)	✓	✓	2.01
OpenMRS	Unsafe	206.00 (±0.00)	206	14.03 (±3.60)	✓	✓	9.71
OACC	*Unsafe*	49.90 (±0.19)	50	3.07 (±0.77)	✓	✓	1.83

the intermediate representations of the different analysis types, and can thus be considered negligible. However, for the safe variants of the subjects *LoopAndbranch* and *gpt14* DF identified large δ values, which indicates that the repaired versions are in fact *not* safe.

The large δ value for *LoopAndbranch* is caused by an integer overflow inside the calculation. Blazer and Themis cannot identify this δ because they do not handle overflow in their analysis. In contrast, DF does execute the actual Java bytecode that allows overflows.

Although the δ value for the *safe* version of the subject *gpt14* shows some great improvement over the *unsafe* version, it is still quite high. It is triggered by a vulnerability where the secret information depends on an extra `if` statement, which has been confirmed by the Themis developers.

Differential Fuzzing (DF) on Themis benchmark. In addition to the subjects by Blazer [104], Table 13 shows the results for the subjects by Themis [110], for which Themis reported their results with regard to an $\epsilon = 64$ and $\epsilon = 0$. This can be read like follows: for a *safe* version with $\epsilon = 0$ Themis reports that there is absolutely no vulnerability and for a *safe* version with $\epsilon = 64$ Themis reports that there might be a vulnerability, which is hardly exploitable. Similarly as for the Blazer subjects, differential fuzzing (DF) successfully identified vulnerabilities in the unsafe versions of the Themis subjects and for the majority of the repaired versions, DF identified only small differences, as expected. But also here, the evaluation results of DF show some discrepancies (see Table 13 in red and italics).

The vulnerability in the unsafe version of *pac4j* is due to the password encoding, which is performed during user authentication. In the static analysis by Themis the encoding procedure is assumed to be expensive, which is represented in a separate model. However, the actual code provided by Themis does not include an expensive implementation, and hence, DF only identified a maximum δ value of 11 bytecode instructions. Since DF does not use any models, it could not find a noteworthy cost difference between the provided safe and unsafe versions (cf. Table 13 subject *pac4j Safe* and *pac4j Unsafe*). In order to further analyze the problem, DF was applied on a more expensive password encoding method, denoted with a star (*) in Table 13, which iterates over the password, to get a stronger indication that there is an actual timing side-channel vulnerability. The discrepancies for the unsafe version of *Tomcat* and *OACC* appear to be similar to *pac4j*, as some manually built models have been used for the static analysis.

The discrepancy in the safe version of the *Jetty* subject indicates the safe version is still vulnerable. A closer look at the bytecode reveals that there is indeed one bytecode instruction that depends on the secret input. Amplified inside a loop, this difference can lead to a large cost difference, which in this case depends on the input size. Therefore, the version can be considered safe for small input sizes but not for large input sizes. Themis did not find this vulnerability because it analyzes an intermediate bytecode representation called Jimple [214], which indeed does not contain the extra instruction.

In addition to the Blazer and Themis subjects, Table 14 shows the results for the hybrid side-channel analysis. The highlighted values represent significant differences to the other technique verified with the Wilcoxon rank-sum test (with 5% significance level).

HyDiff vs. DF. DF usually identifies some improved δ values relatively fast, but often cannot reach similar maximum δ values as HyDiff. In particular, HyDiff outperforms DF:

- for the time to the first $\delta > 0$ in only 1 of 8 subjects (for the remaining 7 subjects both achieve similar numbers), and
- for the average δ value in 4 of 8 subjects (for the remaining 4 subjects both achieve similar numbers).

The experiments reported in Table 14 show that the parallel differential fuzzing (PDF) performs better than DF: PDF cannot only improve the δ values, but is also faster than DF in identifying the first $\delta > 0$. The results for comparing HyDiff with PDF are as follows, HyDiff outperforms PDF:

- for the time to the first $\delta > 0$ in none of the 8 subjects (for another 5 subjects both achieve similar numbers), and
- for the average δ value in 3 of 8 subjects (for another 4 subjects both achieve similar numbers).

Please note that the performance benefit of PDF is in absolute numbers in 2 of the 3 cases below one second.

HyDiff vs. DDSE. Similar as for the regression analysis, there is no significant difference between the DDSE and DDSEx2T experiments for the subjects in the side-channel analysis. There is some small improvement for the average δ for the subjects *Themis_Jetty Safe*, *RSA 1717*, and *RSA 834443*. Therefore, the following result discussion will ignore the DDSEx2T

Table 14: Results for side-channel analysis (t=1800sec=30min, 30 runs). The bold values represent significant differences to the closest other technique verified with the Wilcoxon rank-sum test ($\alpha = 0.05$).

Benchmark		Differential Fuzzing (DF)			Parallel Differential Fuzzing (PDF)			Differential Dynamic Sym. Exec. (DDSE)			DDSE double time budget (DDSEx2T)			HyDiff		
Subject	Version	$\bar{\delta}$	δ_{max}	$\bar{t}: \delta > 0$	$\bar{\delta}$	δ_{max}	$\bar{t}: \delta > 0$	$\bar{\delta}$	δ_{max}	$\bar{t}: \delta > 0$	$\bar{\delta}$	δ_{max}	$\bar{t}: \delta > 0$	$\bar{\delta}$	δ_{max}	$\bar{t}: \delta > 0$
Blazer_login	Safe	0.00 (±0.00)	0	-	0.00 (±0.00)	0	-	0.00 (±0.00)	0	-	0.00 (±0.00)	0	-	0.00 (±0.00)	0	-
Blazer_login	Unsafe	132.87 (±14.87)	238	5.07 (±1.18)	141.13 (±13.22)	232	2.67 (±0.49)	254.00 (±0.00)	254	43.03 (±2.58)	254.00 (±0.00)	254	43.03 (±2.58)	254.00 (±0.00)	254	3.47 (±0.74)
Themis_Jetty	Safe	11.77 (±0.60)	15	3.77 (±0.72)	13.63 (±0.47)	18	3.23 (±0.47)	13.13 (±0.71)	18	58.63 (±0.86)	13.17 (±0.69)	18	58.63 (±0.86)	13.80 (±1.02)	23	4.20 (±0.96)
Themis_Jetty	Unsafe	70.87 (±6.12)	105	6.83 (±1.62)	77.97 (±6.74)	128	4.93 (±1.40)	98.00 (±0.00)	98	58.07 (±0.79)	98.00 (±0.00)	98	58.07 (±0.79)	**100.53 (±1.37)**	111	5.90 (±1.12)
STAC_ibasys	Unsafe	129.40 (±19.52)	280	41.60 (±3.17)	155.50 (±16.07)	271	**35.43 (±1.25)**	280.00 (±0.00)	280	66.13 (±1.81)	280.00 (±0.00)	280	66.13 (±1.81)	280.00 (±0.00)	280	45.63 (±3.11)
RSA	1717	107.27 (±1.22)	112	2.20 (±0.23)	108.40 (±1.16)	116	**1.43 (±0.18)**	108.50 (±0.82)	116	3.53 (±0.18)	109.87 (±0.90)	116	3.53 (±0.18)	108.17 (±1.10)	116	2.33 (±0.38)
RSA	834443	186.77 (±1.75)	196	1.87 (±0.18)	188.10 (±1.26)	197	1.57 (±0.20)	184.37 (±0.88)	190	3.47 (±0.18)	185.67 (±0.84)	190	3.47 (±0.18)	184.50 (±1.24)	191	1.73 (±0.23)
RSA	1964903306	272.77 (±2.47)	286	1.90 (±0.28)	277.20 (±2.39)	301	**1.53 (±0.20)**	252.00 (±0.00)	252	3.17 (±0.16)	252.00 (±0.00)	252	3.17 (±0.16)	275.93 (±3.17)	307	1.93 (±0.26)
RSA (30s)	1717	85.00 (±6.14)	104	2.20 (±0.23)	93.60 (±3.36)	104	**1.43 (±0.18)**	**102.00 (±0.00)**	102	3.53 (±0.20)	**102.00 (±0.00)**	102	3.53 (±0.20)	97.80 (±0.74)	104	2.33 (±0.38)
RSA (30s)	834443	152.93 (±6.58)	187	1.87 (±0.18)	162.43 (±5.27)	193	1.57 (±0.20)	**175.70 (±1.71)**	184	3.47 (±0.18)	**175.70 (±1.71)**	184	3.47 (±0.18)	172.80 (±0.80)	181	1.73 (±0.23)
RSA (30s)	1964903306	226.67 (±7.94)	262	1.90 (±0.28)	233.93 (±6.84)	266	**1.53 (±0.20)**	252.00 (±0.00)	252	3.17 (±0.16)	252.00 (±0.00)	252	3.17 (±0.16)	**254.27 (±1.75)**	269	1.93 (±0.26)

experiments and focus on DDSE. DDSE usually needs more time to identify the first $\delta > 0$, but is very powerful in generating high δ values. In particular, HyDiff outperforms DDSE:

- for the time to the first $\delta > 0$ in 7 of 8 subjects (for the remaining subject both achieve similar numbers), and
- for the average δ value in 3 of 8 subjects (for the remaining 5 subjects both achieve similar numbers).

Temporal development of the results. Since the overall results in Table 14 are quite similar after the time bound of 30 minutes (1,800 seconds), especially for the RSA subjects, the Figures 26 and 27 show exemplary the temporal development for the subjects *Themis_Jetty unsafe* and *RSA 1964903306*.

Figure 26 shows the development for *Themis_Jetty unsafe* in the complete time bound in the graph on the left side and the development in the first 120 seconds in the graph on the right side. This subject represents the typical case, in which DDSE takes some time to identify a good input, but afterwards jumps quite fast to a very high δ value. As long as DDSE performs very low, HyDiff orientates itself towards its fuzzing component. As soon as DDSE makes the jump after 60 seconds, HyDiff joins this development and overtakes DF and PDF. Over time, PDF and DF can improve, but will not get to the same high δ value. Interestingly, even when the standalone DDSE cannot make progress anymore, HyDiff still can leverage its components to continuously improve the δ value.

The considered techniques (DF, PDF, DDSE, and HyDiff) produce very similar results for the RSA subjects after the 30 minutes time bound (cf. Table 14 and the left part of Figure 27). But in particular the beginning of the analysis looks quite different (cf. the right part in Figure 27). DDSE and HyDiff can achieve a δ value of 252 after approximately 10 seconds, for which PDF needs around 60 seconds, and DF needs around 138 seconds.

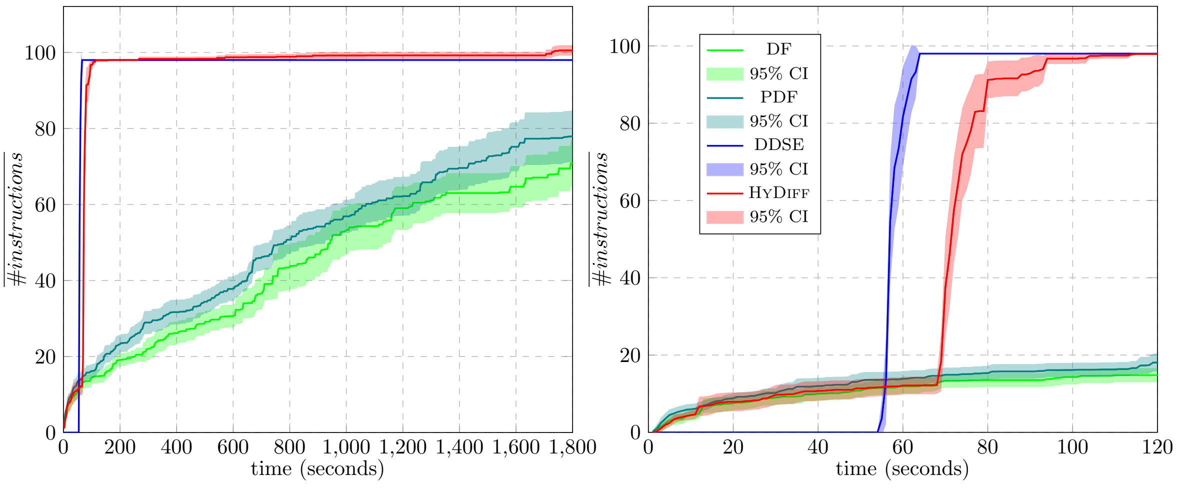


Figure 26: Results for DF, PDF, DDSE, and HyDiff on *Themis_Jetty unsafe* for t=1,800 sec and t=120 sec (lines and bands show averages and 95% confidence intervals across 30 repetitions).

7.4.6 *Discussion*

RQ1-A3 Differential fuzzing (DF). The comparison of differential fuzzing (DF) with state-of-the-art static analysis tools in side-channel vulnerability detection shows that the dy-

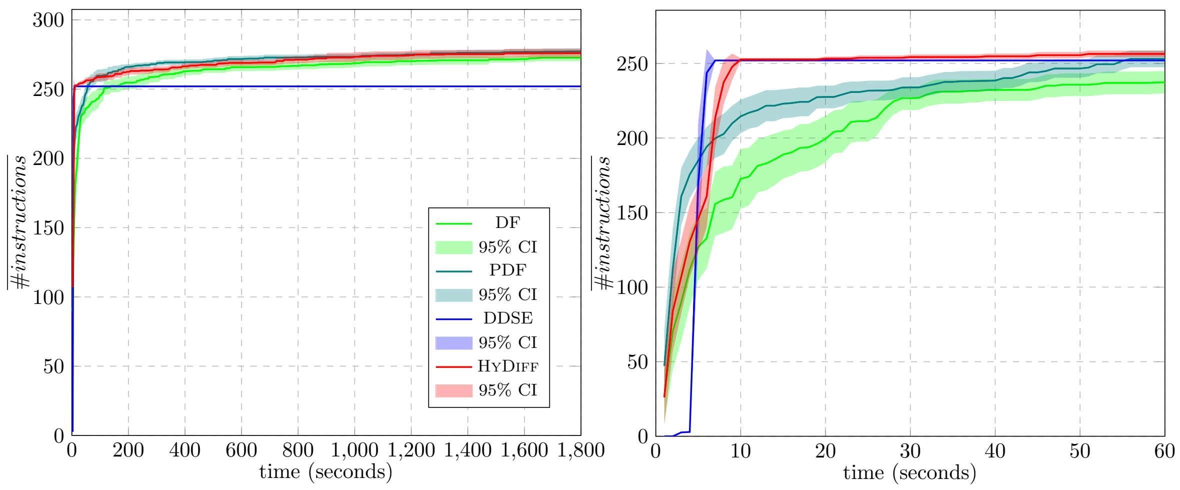


Figure 27: Results for DF, PDF, DDSE, and HyDiff on *RSA 1964903306* for t=1,800 sec and t=60 sec (lines and bands show averages and 95% confidence intervals across 30 repetitions).

namic analysis by DF can find the same vulnerabilities as the static analysis tools. Furthermore, since DF does apply on the actual bytecode and does not use any models, it is employed in a more realistic environment and identified unknown vulnerabilities. However, considering the analysis time, DF often took longer to report its findings (cf. Table 12 and 13). However, the absolute differences in the analysis time usually is in the range of seconds. Similarly, DF performs very well on the presented subjects in the comparison to HyDiff and DDSE. It is very fast in identifying the first improvement ($\bar{t} : \delta > 0$) and identifies all vulnerabilities. The parallelization of DF (PDF) can improve the results of DF, but the performance benefit is not so strong compared to the results in the regression analysis (cf. Section 7.2.5). PDF can improve especially for the time to the first δ improvement ($\bar{t} : \delta > 0$). In fact HyDiff cannot identify a first δ improvement faster than PDF in any of the subjects, although the absolute values are still very close. However, the temporal development shows that PDF performs only slightly better than DF in identifying large δ values, for which HyDiff clearly performs better. Therefore, a disadvantage of DF is the relatively long analysis time needed to identify very large δ values.

RQ2-A3 Differential dynamic symbolic execution (DDSE). While DDSE is very fast in identifying its best δ value, the results in Table 14 indicate that it needs longer than the other techniques to identify any $\delta > 0$. The temporal development in Figures 26 and 27 show that DDSE takes some time and then directly jumps to a very high δ value. In contrast to DF and HyDiff, DDSE does not improve its result continuously over time.

RQ3+4-A3 HyDiff vs. DF and DDSE. The results in Table 14 show that HyDiff, as the combination of DF and DDSE, always matches the best final δ value compared to DF and DDSE. In only two subjects HyDiff can achieve a significantly higher value than its components in isolation. Similarly, the temporal developments in Figures 26 and 27 show that HyDiff always performs on the better curve from DF and DDSE. Therefore, HyDiff represents a well-balanced combination of DF and DDSE, while it cannot significantly amplify the exploration.

RQ5-A3 HyDiff for differential testing. For all subjects (cf. Tables 12, 13, and 14) HyDiff and its components can efficiently detect side-channel vulnerabilities. Furthermore, the comparison between differential fuzzing and Blazer/Themis (cf. Tables 12 and 13) shows that HyDiff can keep up with state-of-the-art static analysis tools for the detection of side-channel vulnerabilities.

SUMMARY – SIDE-CHANNEL ANALYSIS (A3)

Although HyDiff shows some peaks in the evaluation, it is usually not much better than the naive combination of the results from both components. Nevertheless, HyDiff represents a well balanced combination of both: differential fuzzing and differential dynamic symbolic execution, which can be well observed in the graphics about the temporal development. As it combines both techniques, it can identify a cost difference very fast and is able to assess the severity of side-channel vulnerabilities very well because it can quickly identify large δ values. The comparison with Blazer and Themis has shown that differential fuzzing, and hence also HyDiff, can keep up with state-of-the-art static analysis tools and even outperform them in the detection in side-channel vulnerabilities.

7.5 ROBUSTNESS ANALYSIS OF NEURAL NETWORKS (A4)

This section describes the application of HyDiff on the *robustness analysis of neural networks*. The basics and related work on neural networks and their analysis can be found in Section 2.4.4. The following paragraphs explain how HyDiff and its components can be applied on neural networks. Afterwards they discuss an extract of the analyzed neural network and show how to implement HyDiff's drivers. Finally, this section presents the conducted evaluation of HyDiff and discusses the results. Please note that the analysis of neural networks is very expensive: there are many paths involved and the neurons are usually highly connected. Therefore, this kind of analysis represents a stress testing scenario for HyDiff in the domain of high complexity. The evaluation on robustness analysis of neural networks extends the evaluation shown in one of the preliminary publications [6].

7.5.1 *Approach*

The goal of the proposed analysis is to check the robustness of neural networks, i.e., whether a small change in the input can already lead to a change in the output of the network. Specifically, in this application scenario HyDiff is used to find adversarial inputs for an image classification network. Since HyDiff performs the analysis of Java bytecode, the first step is to re-write a given neural network model into a Java program [191]. In the below presented experiments, the neural network model has been built with KERAS [192], a high-level neural networks API. In order to generate a Java program, one can iterate over the layers of the resulting KERAS model and, depending on the layer type (e.g., convolution or activation layer), can generate loops which perform the necessary calculations, i.e., linear combinations of the input at the neurons, the learned weights and biases. As the test set of the learning process showed, the resulting Java translation preserves the prediction ability of the original learned neural network. The translation from the original neural network to the Java program is based on the toolset by Youcheng Sun [191].

Similarly to the side-channel analysis (cf. Section 7.4), the idea for the differential analysis is to allow changes in the input and observe differences in the network's behavior. More precisely, the proposed analysis changes up to x% of the pixels in the input image, and checks whether there can be any difference in the output of the network.

HyDiff's fuzzing component needs to fuzz (1) values for the complete image, and then it needs to fuzz (2) values and positions of the pixels to change to reach the x% boundary. Therefore, the driver will build two images that differ only in up to x% of the pixels. Afterwards the driver executes the *transformed* neural network model once with each image and measure the differential metrics similar as for the regression analysis. The metric *output difference* is of particular interest.

HyDiff's symbolic execution component needs to introduces the changes, similar as for the side-channel analysis, directly in the input and not in the program. Therefore, the driver reads a complete image and the values and pixels to change, but does introduce change-annotations in the input. Then the driver executes the *transformed* neural network model with the change-annotated input. Since the goal of the neural network analysis is to identify differences in the classification, i.e., output differences similar to regression, and not cost differences like the side-channel analysis, the symbolic exploration uses the same heuristics as for regression analysis.

The analysis approach for neural networks represents a combination of the ideas from the regression analysis and the side-channel analysis: (1) changes happen in the input and not in the program (cf. side-channel analysis), and (2) the goal is to detect an output difference (cf. regression analysis).

Some preliminary experiments have shown that with HyDiff in its default setup (i.e., differential fuzzing and differential dynamic symbolic execution start at the same time), both components do not synchronize with each other because they are busy with their own analysis due to the expensive program execution. Therefore, the execution setup for the neural network analysis is different: the experiments start with differential symbolic execution for 10 minutes. After this time bound the differential fuzzing component is started with the already generated inputs by the differential symbolic execution component as additional seed inputs and both component run in parallel for the remaining time. The 10 minutes delay provides sufficient time to the DDSE component to generate a first interesting input.

7.5.2 *Example & Drivers*

Listing 26 shows an extract of the Java program, which represents the *transformed* neural network model for the experiments for this case study. The program takes a double array as input, which represents a normalized 28x28 greyscale image. The image is expected to include a handwritten digit and the network has been trained to recognize such digits.

Each compartment of this program denotes a layer of the neural network. For example layer 0 is implemented in the lines 3 to 12. Firstly, the resulting array after the convolution is declared in line 3. Afterwards the value for each neuron gets initialized with the stored biases (cf. line 7). Afterwards the linear combination is performed in the lines 8 to 11, in which the multiplication of the weights and the input values is shown in line 11. The resulting array represents the input for the next layer.

The final layer shows the final classification of the hand-written digits into the numbers 0 to 9. The total size of the program is 81 LOC.

Drivers. Listings 27 and 28 show the drivers for the experiment with x=1% changed pixels. The fuzzing driver (cf. Listing 27) starts with reading the input image (cf. lines 7 to 20), which includes the normalization of the pixel values from the bytes in the range [-128,127] to double values in the range [0, 1] in line 18. Afterwards the driver introduces changes by

Listing 26: Extract of the transformed neural network model as Java program.

```java
int runDNN(double[][][] input) { // input image is of shape 28x28x1
   // layer 0: convolution
   double[][][] layer0 = new double[26][26][2];
   for (int i = 0; i < 26; i++)
      for (int j = 0; j < 26; j++)
         for (int k = 0; k < 2; k++) {
            layer0[i][j][k] = biases0[k];
            for (int I = 0; I < 3; I++)
               for (int J = 0; J < 3; J++)
                  for (int K = 0; K < 1; K++)
                     layer0[i][j][k] += weights0[I][J][K][k] * input[i + I][j + J][K];
         }

   // layer 1: activation
   double[][][] layer1 = new double[26][26][2];
   for (int i = 0; i < 26; i++)
      for (int j = 0; j < 26; j++)
         for (int k = 0; k < 2; k++)
            if (layer0[i][j][k] > 0)
               layer1[i][j][k] = layer0[i][j][k];
            else
               layer1[i][j][k] = 0;

   // layer 2: convolution
   double[][][] layer2 = new double[24][24][4];
   for (int i = 0; i < 24; i++)
      for (int j = 0; j < 24; j++)
         for (int k = 0; k < 4; k++) {
            layer2[i][j][k] = internal.biases2[k];
            for (int I = 0; I < 3; I++)
               for (int J = 0; J < 3; J++)
                  for (int K = 0; K < 2; K++)
                     layer2[i][j][k] += weights2[I][J][K][k] * layer1[i + I][j + J][K];
         }

   // layer 3: activation
   double[][][] layer3 = new double[24][24][4];
   for (int i = 0; i < 24; i++)
      for (int j = 0; j < 24; j++)
         for (int k = 0; k < 4; k++)
            if (layer2[i][j][k] > 0)
               layer3[i][j][k] = layer2[i][j][k];
            else
               layer3[i][j][k] = 0;

   ... // layer 4 - 8

   // layer 9: activation
   int ret = 0;
   double res = -100000;
   for (int i = 0; i < 10; i++) {
      if (layer8[i] > res) {
         res = layer8[i];
         ret = i;
      }
   }
   return ret;
}
```

Listing 27: Fuzzing driver for the neural network experiment with 1% pixel change.

```java
public static void main(String[] args) {
   final int IMG_HEIGHT = 28;
   final int IMG_WIDTH = 28;
   final int NUMBER_OF_PIXEL_CHANGE = 7; // 1%

   // Reading input from fuzzed file.
   double[][][] input1 = new double[28][28][1];
   double[][][] input2 = new double[28][28][1];
   try (FileInputStream fis = new FileInputStream(args[0])) {
      byte[] bytes = new byte[1];
      for (int i = 0; i < 28; i++)
         for (int j = 0; j < 28; j++)
            for (int k = 0; k < 1; k++) {
               if (fis.read(bytes) == -1)
                  throw new RuntimeException("Not enough data to read input!");

               /* Normalize value from [-128,127] to be in range [0, 1] */
               input1[i][j][k] = (bytes[0] + 128) / 255.0;
               input2[i][j][k] = input1[i][j][k];
            }

      // Introduce change for second input: in total 784 pixels, 1% means 7 pixel values.
      for (int i = 0; i < NUMBER_OF_PIXEL_CHANGE; i++) {
         bytes = new byte[3];
         if (fis.read(bytes) == -1)
            throw new RuntimeException("Not enough data to read input!");

         int i_pos = Math.floorMod(bytes[0], 28);
         int j_pos = Math.floorMod(bytes[1], 28);
         input2[i_pos][j_pos][0] = (bytes[2] + 128) / 255.0;
      }
   } catch (IOException e) {...}

   Mem.clear();
   DecisionHistory.clear();
   Object res1 = null;
   try {
      res1 = runDNN(input1);
   } catch (Throwable e) {
      res1 = e;
   }
   boolean[] dec1 = DecisionHistory.getDecisions();
   long cost1 = Mem.instrCost;

   Mem.clear();
   DecisionHistory.clear();
   Object res2 = null;
   try {
      res2 = runDNN(input2);
   } catch (Throwable e) {
      res2 = e;
   }
   boolean[] dec2 = DecisionHistory.getDecisions();
   long cost2 = Mem.instrCost;

   DecisionHistoryDifference d = DecisionHistoryDifference
      .createDecisionHistoryDifference(dec1, dec2);
   Kelinci.setNewDecisionDifference(d);
   Kelinci.setNewOutputDifference(new OutputSummary(res1, res2));
   Kelinci.addCost(Math.abs(cost1 - cost2));
}
```

modifying the pixel value at specific locations (cf. lines 28 to 30). Note that only `input2` is changed, and that `input1` remains unchanged. Therefore, `input1` denotes the original image and `input2` the changed image. After reading the images and introducing the changes, the driver executes the neural network program with `input1` (cf. lines 34 to 43) and with `input2` (cf. lines 45 to 54). Finally, the driver reports the observed differences.

The symbolic execution driver (cf. Listing 28) again distinguishes two modes: a concolic mode (lines 10 to 36) and a pure symbolic execution mode (lines 37 to 47). The driver follows the same general idea as the fuzzing driver: first read the image pixels and then modify the pixels at specific locations. Furthermore, the symbolic execution driver adds symbolic values for each pixel (cf. lines 21 and 22) and for the changes (cf. lines 29 to 34). For the symbolic execution mode it simply generates these symbolic values (cf. lines 41 and 43 to 45). Before executing the neural network in line 57, the driver adds the change-annotations to the input to combine two images into one (cf. lines 50 to 54).

Example. The implemented analysis in the fuzzing and symbolic execution driver tries to identify two images that differ only up to a given threshold and that are differently classified by the neural network. In order to illustrate the effect of this approach, please consider Figure 28 and Figure 29. Figure 28 shows two inputs that have been generated by differential fuzzing after 1,375 seconds. The image on the left side is classified as the digit 6 and the image on the right side is classified as the digit 5. Figure 29 shows two inputs that have been generated by differential dynamic symbolic execution after 295 seconds. The image on the left side is classified as the digit 5 and the image on the right side is classified as the digit 1. For both pairs the images differ only in 7 pixel values, i.e., 1% of the pixels in the image. The generated images are not necessarily representative as adversarial inputs as they do not fulfill the assumption of the neural network, namely that it expects handwritten digits. Due to the driver implementation, the search process can change the complete image, and so, these images look randomly generated. Note that this case study is used as stress testing for HyDiff and its components. Therefore, it is fair enough to identify images that differ slightly and lead to different classifications in order to assess the robustness of the neural network.

However, it is also possible to update the drivers to perform a more realistic scenario. If the original input is kept and the process searches only the pixel locations and values, then the result can be used as adversarial inputs. This analysis would be much harder because there are not so many ways to change an existing image. Figure 30 shows an exemplary result of performing such an analysis with differential fuzzing. The left side shows the original image that is classified as 6. The right side shows an image with 50% changed pixel that is classified as 8. In the experiment, differential fuzzing needed more than 60 hours to generate the adversarial input.

7.5.3 *Evaluation Metrics*

The evaluation metrics for the neural network analysis are the same as for the regression analysis. The most important metric is the *output difference* because it shows a difference in the resulting classification of the image. The *crash* metric is not of interest in this analysis since the goal is not to produce a crash, which should actually also not be possible. Errors in the network should lead to a miss-classification; real crashes in the program would represent an error in the transformation from neural network model to the Java program or an error in the data processing in the driver.

Listing 28: Symbolic execution driver for the neural network experiment with 1% pixel change.

```java
public static void main(String[] args) {
   int IMG_HEIGHT = 28;
   int IMG_WIDTH = 28;
   int NUMBER_OF_PIXEL_CHANGE = 7; // 1%
   int[] i_pos_changes = new int[NUMBER_OF_PIXEL_CHANGE];
   int[] j_pos_changes = new int[NUMBER_OF_PIXEL_CHANGE];
   double[] value_changes = new double[NUMBER_OF_PIXEL_CHANGE];
   double[][][] input = new double[IMG_HEIGHT][IMG_WIDTH][1];

   if (args.length == 1) {
      try (FileInputStream fis = new FileInputStream(fileName)) {
         byte[] bytes = new byte[1];
         for (int i = 0; i < IMG_HEIGHT; i++)
            for (int j = 0; j < IMG_WIDTH; j++)
               for (int k = 0; k < 1; k++) {
                  if (fis.read(bytes) == -1)
                     throw new RuntimeException("Not enough data to read input!");

                  /* Normalize value from [-128,127] to be in range [0, 1] */
                  double value = (bytes[0] + 128) / 255.0;
                  input[i][j][k] = Debug.addSymbolicDouble(value,
                     "sym_" + i + "_" + j + "_" + k);
               }
         for (int i = 0; i < NUMBER_OF_PIXEL_CHANGE; i++) {
            bytes = new byte[3];
            if (fis.read(bytes) == -1)
               throw new RuntimeException("Not enough data to read input!");

            i_pos_changes[i] = Debug.addConstrainedSymbolicInt(
               Math.floorMod(bytes[0], 28), "sym_ipos_" + i, 0, 27);
            j_pos_changes[i] = Debug.addConstrainedSymbolicInt(
               Math.floorMod(bytes[1], 28), "sym_jpos_" + i, 0, 27);
            value_changes[i] = Debug.addSymbolicDouble(
               (bytes[2] + 128) / 255.0, "sym_change_" + i);
         }
      } catch (IOException e) {...}
   } else {
      for (int i = 0; i < IMG_HEIGHT; i++)
         for (int j = 0; j < IMG_WIDTH; j++)
            for (int k = 0; k < 1; k++)
               input[i][j][k] = Debug.makeSymbolicDouble("sym_" + i + "_" + j + "_" + k);
      for (int i = 0; i < NUMBER_OF_PIXEL_CHANGE; i++) {
         i_pos_changes[i] = Debug.makeConstrainedSymbolicInteger("sym_ipos_" + i, 0, 27);
         j_pos_changes[i] = Debug.makeConstrainedSymbolicInteger("sym_jpos_" + i, 0, 27);
         value_changes[i] = Debug.makeSymbolicDouble("sym_change_" + i);
      }
   }

   // Insert changes.
   for (int i = 0; i < NUMBER_OF_PIXEL_CHANGE; i++) {
      int i_pos = i_pos_changes[i];
      int j_pos = j_pos_changes[i];
      double value = value_changes[i];
      input[i_pos][j_pos][0] = change(input[i_pos][j_pos][0], value);
   }

   runDNN(a);
}
```

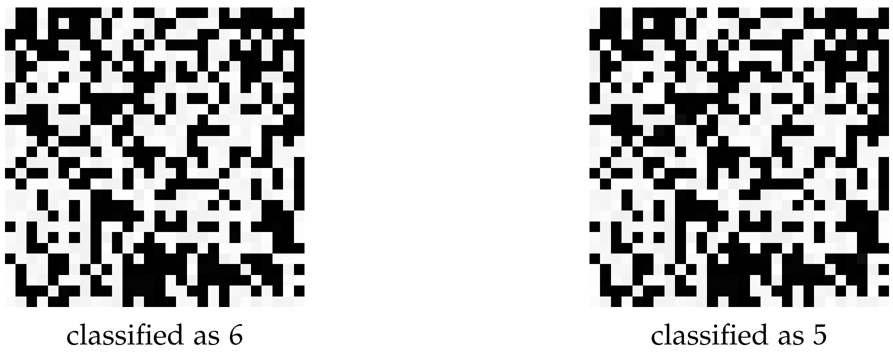


Figure 28: Inputs identified by DF after 1,375 seconds by fuzzing two images that differ in up to 1% of the pixels. The learned model classifies the left image as a 6, while it classifies the right image as a 5. The images differ only in 7 pixels.

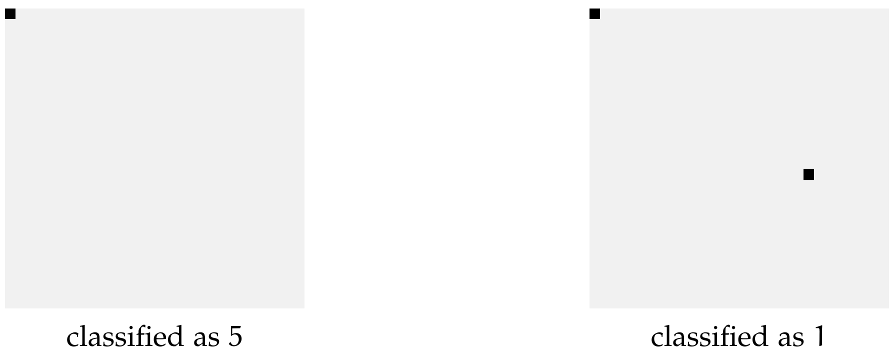


Figure 29: Inputs identified by DDSE after 295 seconds by synthesizing two images that differ in up to 1% of the pixels. The learned model classifies the left image as a 5, while it classifies the right image as a 1. The images differ only in 1 pixel.

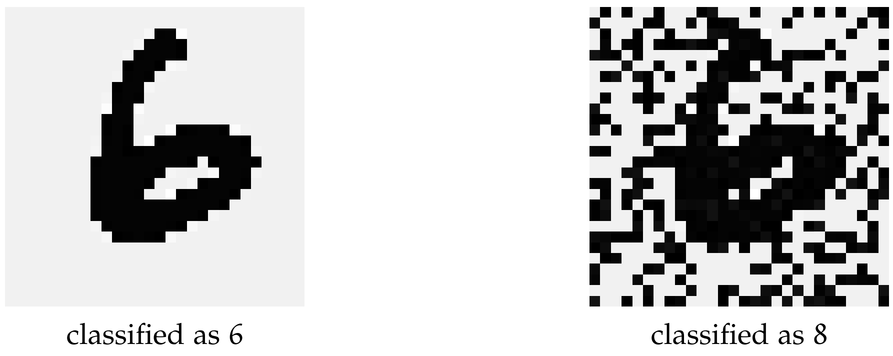


Figure 30: Adversarial changes identified by DF after 60.89 hours by fuzzing up to 50% pixel changes for a fixed image. The learned model classifies the left (original) image as a 6, while it classifies the right image as an 8. The images differ in 314 pixels.

The results report the following metrics:

- $\overline{t}$ +odiff: the average time to first output difference (lower is better)
- t_{min}: the minimum time (over all runs) needed to find the first output difference (lower is better)
- $\overline{\texttt{\#odiff}}$: the average number of identified output differences (higher is better)
- $\overline{\texttt{\#ddiff}}$: the average number of identified decision differences (higher is better)

Table 15: Results for robustness analysis of neural networks (t=3600sec=60min, 30 runs). The bold values represent significant differences to the closest other technique verified with the Wilcoxon rank-sum test ($\alpha = 0.05$). DDSEx2T produces the exact same results as DDSE.

Subject	**Differential Fuzzing (DF)**				**Parallel Differential Fuzzing (PDF)**				**Differential Dynamic Sym. Exec. (DDSE)**				**HyDiff**			
(% change)	$\bar{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$	$\bar{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$	$\bar{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$	$\bar{t}$ +odiff	t_{min}	$\overline{\#odiff}$	$\overline{\#ddiff}$
1	2,725.40 (±341.09)	1,074	0.57 (±0.20)	7.73 (±0.18)	2,928.60 (±289.44)	1,202	1.00 (±0.31)	**12.00 (±0.48)**	296.03 (±1.49)	289	1.00 (±0.00)	1.00 (±0.00)	297.10 (±2.38)	267	**1.20 (±0.14)**	6.10 (±0.11)
2	2,581.47 (±326.21)	1,032	0.93 (±0.28)	7.93 (±0.13)	2.509.20 (±289.37)	1,117	1.23 (±0.33)	**12.63 (±0.48)**	309.77 (±7.04)	293	1.00 (±0.00)	1.00 (±0.00)	**297.93 (±1.29)**	292	1.53 (±0.20)	6.93 (±0.13)
5	2,402.97 (±329.59)	1,189	1.23 (±0.37)	6.47 (±0.18)	2,501.43 (±285.86)	1,429	1.70 (±0.44)	**10.33 (±0.43)**	304.53 (±1.06)	300	1.00 (±0.00)	1.00 (±0.00)	**301.83 (±1.16)**	296	2.07 (±0.29)	6.90 (±0.17)
10	2,155.40 (±343.76)	996	1.57 (±0.34)	8.10 (±0.17)	2,127.70 (±229.21)	1,418	2.20 (±0.33)	**11.23 (±0.40)**	311.90 (±0.74)	308	1.00 (±0.00)	1.00 (±0.00)	311.07 (±1.01)	306	2.37 (±0.31)	7.00 (±0.13)
20	1,695.83 (±228.18)	953	2.70 (±0.37)	9.13 (±0.12)	1,897.67 (±219.97)	1,340	3.30 (±0.49)	**11.57 (±0.50)**	346.87 (±1.98)	339	1.00 (±0.00)	1.00 (±0.00)	**341.83 (±1.27)**	336	3.13 (±0.34)	7.20 (±0.14)
50	1,830.83 (±259.79)	1,220	2.43 (±0.42)	6.33 (±0.21)	1,696.10 (±86.20)	1,423	3.80 (±0.35)	**12.00 (±0.39)**	455.03 (±1.62)	449	1.00 (±0.00)	1.00 (±0.00)	452.63 (±2.06)	434	3.77 (±0.34)	7.27 (±0.16)
100	1,479.17 (±231.25)	960	2.47 (±0.37)	9.37 (±0.20)	1,790.87 (±270.10)	1,109	3.03 (±0.54)	**13.97 (±0.68)**	583.33 (±2.83)	571	1.00 (±0.00)	1.00 (±0.00)	**575.13 (±2.65)**	564	3.10 (±0.35)	7.60 (±0.18)

7.5.4 *Data Sets*

The neural network model used in this evaluation has been trained for handwritten digit recognition using the MNIST dataset [193]. The data set comes with a training set of 60,000 examples and a test set of 10,000 examples. The trained model has an accuracy of 97.95% on the test set. It consists of 11 layers including convolutional/max-pooling/flatten/dense layers with Rectified Linear Unit (ReLU) activations, contains 10,000 neurons, and uses the max function in the final classification layer. The Listing 26 shows an extract of the *transformed* JAVA program.

7.5.5 *Evaluation Results*

Table 15 shows the results for the robustness analysis of neural networks. The used metrics ($\bar{t}$ +odiff, t_{min}, $\overline{\#odiff}$, and $\overline{\#ddiff}$) have been described in Section 7.5.3 and focus on the output difference (odiff) as well as the decision difference (ddiff). The highlighted values represent significant differences to the closest other technique verified with the Wilcoxon rank-sum test (with 5% significance level). The time values are presented in seconds and the values also report the 95% confidence intervals.

HyDiff vs. DF. As the results in Table 15 show differential fuzzing takes very long to identify its first interesting input: depending on the number of pixels changed between 24 and 45 minutes. Due to the contribution by DDSE, HYDIFF can be much faster. However, for the number of decision differences, DF outperforms HYDIFF for the majority of the subjects. In particular, HYDIFF outperforms DF:

- for the time to the first output difference ($\bar{t}$ +odiff) in 7 of 7 subjects,
- for the number of output differences ($\overline{\#odiff}$) in 6 of 7 subjects (for the remaining subject both achieve similar numbers), and
- for the number of decision differences ($\overline{\#ddiff}$) in 2 of 7 subjects.

Although the parallel differential fuzzing (PDF) cannot improve the time to the first output difference compared to DF, it can significantly improve the number of decision differences. As HYDIFF already performs poorly in this category ($\overline{\#ddiff}$) compared to DF, PDF outperforms HYDIFF there in all subjects. Table 15 shows that PDF can find on average between 4 to 6 more inputs that reveal a decision difference compared to HYDIFF. Note that HYDIFF already finds 6 to 7 inputs for such a difference. Although PDF performs very well for this metric, it cannot solve the problem of DF in this context, which is the time to the first output difference. PDF is never faster than 28 minutes on average, whereas HYDIFF can find output differences already after approximately 5 minutes. In particular, HYDIFF outperforms PDF:

- for the time to the first output difference ($\bar{t}$ +odiff) in 7 of 7 subjects,
- for the number of output differences ($\overline{\#odiff}$) in 1 of 7 subjects (for the remaining 6 subjects both achieve similar numbers), and
- for the number of decision differences ($\overline{\#ddiff}$) in none of 7 the subjects.

HyDiff vs. DDSE. Similar to the other applications, the results for the DDSE and DDSEx2T experiments show that for the analyzed subjects it makes no difference when giving DDSE twice the time budget. Therefore, the following discussion ignores the DDSEx2T experiments and focus on DDSE. Although DDSE performs relatively well for the time to the first output difference, it can only reveal one difference in the beginning of the analysis. With increasing number of changed pixels, also the time to first output difference increases. In particular, HyDiff outperforms DDSE:

- for the time to the first output difference ($\overline{t}$ +odiff) in 4 of 7 subjects (for the remaining 3 subjects both achieved similar numbers),
- for the number of output differences ($\overline{\texttt{\#odiff}}$) in 7 of 7 subjects, and
- for the number of decision differences ($\overline{\texttt{\#ddiff}}$) in all 7 of 7 subjects.

Temporal development of the results. In order to illustrate the input generation, Figure 31 shows the temporal development for the neural network subject with 1% changes. The Figures for the remaining subjects look very similar, although the curves for the techniques get closer together as the numbers in Table 15 indicate. The left part in Figure 31 shows

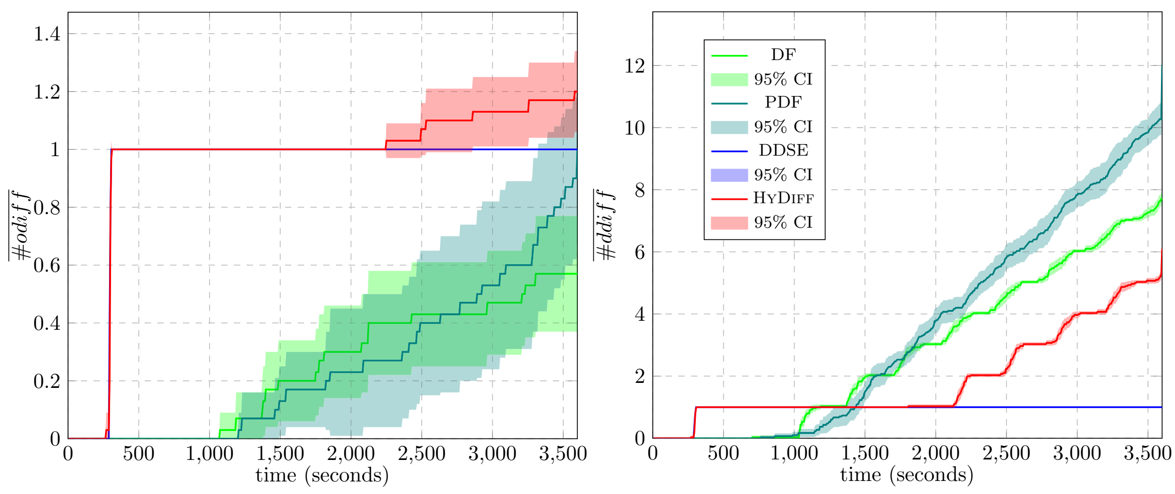


Figure 31: Results for DF, PDF, DDSE, and HyDiff on the neural network subject with 1% changes (lines and bands show averages and 95% confidence intervals across 30 repetitions).

the input generation in terms of output differences, the right part of Figure 31 shows the input generation in terms of decision differences. DDSE identifies its first output difference after approximately 5 minutes, which is the same as for HyDiff because the setup configuration of HyDiff only allows DDSE in the first 10 minutes. Afterwards HyDiff's fuzzing component is started and initialized with the initial input and the inputs that has been generated so far by HyDiff's symbolic execution component. After this first input, DDSE cannot improve anymore: no further output differences and no further decision differences are identified. However, HyDiff can leverage its fuzzing component and still improves its result.

Both graphs in Figure 31 show a delay in the fuzzing behavior between HyDiff and DF. For the output difference (cf. the graph on the left side of Figure 31) DF begins to make progress after approximately 18 minutes, HyDiff's fuzzing component makes progress after approximately 37.5 minutes, i.e., 27.5 minutes after it is started. There is a difference

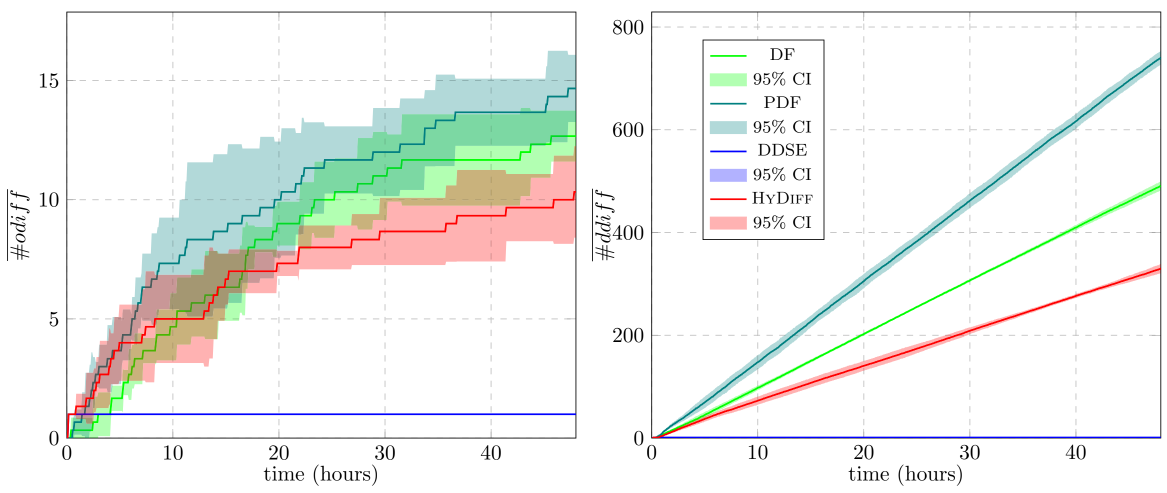


Figure 32: Results for DF, PDF, DDSE, and HyDiff on the neural network subject with 1% changes after 48 hours (lines and bands show averages and 95% confidence intervals across 3 repetitions).

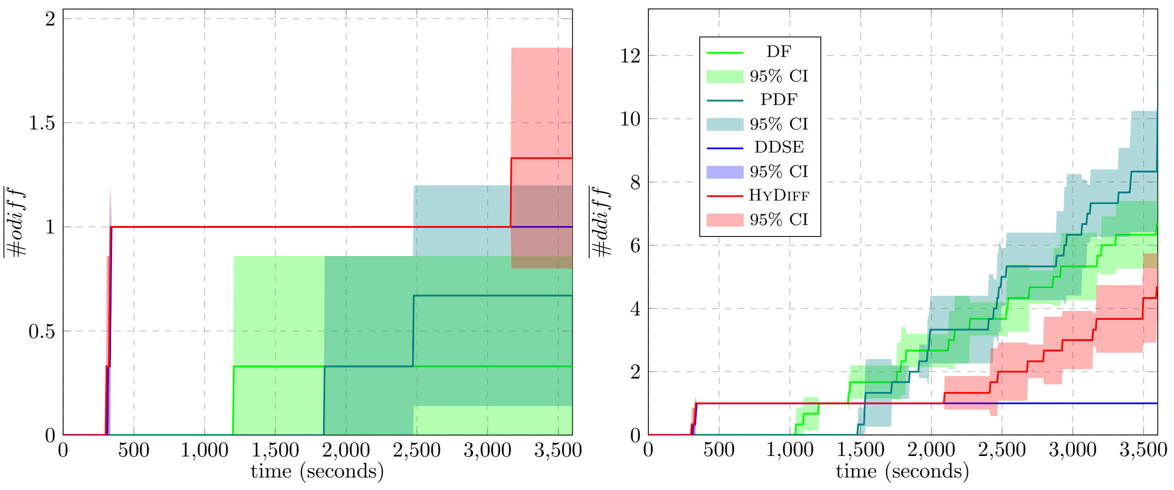


Figure 33: Results for DF, PDF, DDSE, and HyDiff on the neural network subject with 1% changes within the first hour of the 48 hours experiment (lines and bands show averages and 95% confidence intervals across 3 repetitions).

of 9.5 minutes between DF and HyDiff. This represents the additional time, fuzzing needs to replay and assess the extra initial input by DDSE, and hence, it shows how expensive the execution of the neural network is. A similar behavior can be seen for the decision difference in the graph on the right side of Figure 31. Similarly to DF, parallel DF (PDF) needs long to actually identify a difference, but afterwards it quickly outperforms DF.

In order to further investigate the performance of HyDiff under this stress test, an additional experiment has been conducted to explore the results after a 1% change analysis of 48 hours. The expensive experiment has been repeated 3 times and the results are shown in Figure 32. The results for the first hour are shown in Figure 33. First of all, the results for the 3 experiments within the first hour of the analysis (cf. Figure 33) look very similar to the results observed with the 30 repetitions in the earlier experiments (cf. Figure 31): the confidence intervals are higher, but the trend looks the same. Therefore, although the 48 hours experiment has only 3 repetitions, the trends look reasonable.

The left part of Figure 32 shows the results for the number of identified output differences. HyDiff starts good and outperforms DF and PDF within the first hour. Within the first 5 hours, HyDiff and PDF performs quite similar, but afterwards PDF improves much better over time. Also the numbers of DF passes HyDiff after roughly 17 hours. Note that the absolute difference is not very large, but still single DF or PDF appear to be more effective for longer time periods. HyDiff's symbolic execution component does not contribute, since it can only identify one output difference in the beginning. Nevertheless, it still generates inputs that need to be synchronized with fuzzing. Therefore, the effectiveness of HyDiff in contrast to single fuzzing will degrade if the inputs of symbolic execution are not useful and the synchronization effort is too large. This can happen when DDSE generates too many useless inputs or when the input execution is very expensive.

The right part of Figure 32 shows the results for the number of decision differences. The trend, which have been observable already within the first hour, continues for the complete 48 hours, as PDF performs better than DF and better than HyDiff. This development has the same reasons as for the output difference.

7.5.6 *Discussion*

RQ1-A4 Differential fuzzing (DF). The results in Table 15 show that differential fuzzing takes very long to make some initial progress. This time decreases with increasing number pixels to change, which make sense: the more fuzzing is allowed to change in the image, the more likely it should be to find a difference in the classification. The parallel setup of differential fuzzing significantly improves its output.

RQ2-A4 Differential dynamic symbolic execution (DDSE). The complexity of the neural network is challenging for symbolic execution. The subject with 1% change introduces 7 x 3 = 21 additional symbolic variables: 7 for each change, and 3 for the x and y position of the pixel and the updated pixel value. In total this means the handling of 784 (one symbolic variable per image pixel) + 21 = 405 symbolic variables. This large number of symbolic variables represents quite an effort for symbolic execution. In particular the constraint solving gets expensive because of the complex constraints due to the highly connected neurons in the network and the high number of symbolic variables. All these factors contribute to the low performance of DDSE. Throughout the different subjects, the results show that with increasing number of changed pixels (1% to 100%), DDSE takes longer to reveal the first output difference. For 1% it is approximately 5 minutes, and for 100% (i.e., 784 x 3 = 2,352 additional symbolic variables) it is more than 9 minutes. Therefore, for DDSE it becomes more and more difficult to reason about the neural network when the analysis includes more symbolic variables.

RQ3+4-A4 HyDiff vs. DF and DDSE. The results in Table 15 as well as the temporal development in Figure 31 show that HyDiff is significantly faster than DF with its initial progress and also identifies more output differences. The temporal development for the decision difference for DF and HyDiff (cf. Figure 31) show that they perform very similar but with a delay of approximately 13 minutes. HyDiff can only outperform PDF in terms of the time to the first output difference; for the total number of output differences both achieve similar numbers, and for the decision difference PDF outperforms HyDiff. Please note that the focus of HyDiff is to quickly generate an output difference. In this context HyDiff outperforms PDF with the help of symbolic execution. HyDiff benefits from DDSE due to the fast input generation in the beginning, but cannot leverage its capabilities for the remaining analysis time. Therefore, HyDiff clearly outperforms standalone DDSE. HyDiff is also able to amplify the exploration (cf. left graph in Figure 31), however, the 48 hour experiment (cf. Figure 32) shows that this is only valid for analysis times within 1 hour.

RQ5-A4 HyDiff for differential testing. The evaluation for the neural network subjects has shown that all three techniques (DF, DDSE, and their hybrid combination HyDiff) can be used to find adversarial inputs for neural networks. Although this scenario shows the limitation of all approaches, output differences have been generated. The fact that 1% change in the pixel values of an image can be used to produce a different classification is an indication that the model learned on the MNIST dataset might not be very robust. Similar results have been obtained in previous works [157, 187].

SUMMARY – NEURAL NETWORK ANALYSIS (A4)

The results for this experimental setup show the different benefits of the two different approaches (fuzzing and symbolic execution) and why it is important to combine them! HyDiff leverages its differential symbolic execution component to quickly generate a first output difference, and further leverages differential fuzzing to identify even more output differences. HyDiff does not only combine the results of both components, but the components can benefit from each others' inputs to further improve the outcome.

7.6 DISCUSSION

This section summarizes the evaluation results with regard to the research questions from Section 3.3 and further summarizes the contributions made by this thesis.

7.6.1 *General Answers to Research Questions*

RQ1: Differential fuzzing (DF) and its limitations. Differential fuzzing (DF) has been quite effective in the presented applications. For regression analysis DF has been able to identify the majority of the output differences. However, it missed some of them. Even the more powerful parallel DF (PDF) was not able to improve this significantly, and therefore it indicates that there are several constraints that fuzzing cannot overcome in the given time bound. In the worst-case complexity analysis DF showed its ability to continuously improve the high-score. Eventually it identifies high cost values, but it takes time. For the detection of side-channel vulnerabilities DF performed great and even outperformed the state-of-the-art static analysis techniques Blazer [104] and Themis [110]. DF does not depend on models or abstractions as it applies on the actual Java bytecode of the application under test. With regard to the analysis time, DF was slower than the static analysis techniques. For the robustness analysis of neural networks DF is effective, as it finds an adversarial input for a small pixel change ratio, but it is very slow also because the network execution is very expensive. Fuzzing gets faster with a higher pixel change ratio, but it is still not efficient. This case study shows that DF reaches its limitations when the program execution is taking too long.

In conclusion, DF is effective and continuously improves its results over time. The parallel variant of DF shows even better results and also outperforms HyDiff for some subjects.

RQ2: Differential dynamic symbolic execution (DDSE) and its limitations. Similarly as DF, differential dynamic symbolic execution (DDSE) performed quite well for all applications. For regression analysis DDSE identifies output differences for subjects, for which DF cannot find them. However, DDSE also misses other output differences, which have been identified by DF. DDSE with twice the time budget (i.e., DDSEx2T) also did not help to identify these output differences. Therefore, both techniques have their own advantages. In the worst-case complexity analysis DDSE remains in plateaus and cannot improve a lot over time. However, DDSE is very fast in detecting a first slowdown and performed significantly better for the regular expression subjects than the other techniques. For the detection of side-channel vulnerabilities DDSE is usually not faster than fuzzing to identify the first $\delta > 0$, but it is significantly faster in identifying its maximum value (see the graphs in Figure 26 and 27). DDSE appears to not make fast progress, but then eventually produces a better value than fuzzing. For the robustness analysis of neural networks DDSE is relatively fast in generating first output differences, which indicates that the exploration is well directed. However, DDSE cannot reveal more than one difference due to the expensive constraint solving.

In conclusion, DDSE is a very effective and efficient technique. It can leverage the power of constraint solving to complement the results of DF. DDSE is not always slower than fuzzing and represents itself an effective technique. In contrast to fuzzing, DDSE does not improve the results continuously over time, but develops in jumps. Giving DDSE twice the time budget does not improve the result in the considered experiments. This indicates that DDSE already identifies interesting results in the beginning of the exploration and then only slowly improves over time due to the path explosion. A parallel variant of DDSE might significantly improve the behavior (cf. Section 8.2).

RQ3+4: HyDiff vs. DF and DDSE. For regression analysis HyDiff can identify all output differences and often generates higher values for the number of decision differences in a shorter period of time. In the worst-case complexity analysis HyDiff clearly outperforms the single components and can further increase the slowdown of the applications under test. It shows great performance in quickly identifying slow-performing inputs. For the detection of side-channel vulnerabilities HyDiff does not show a clear benefit over the combination of both components, meaning that there is no significant amplification of the exploration. Nevertheless, HyDiff represents a well balanced combination and sometimes performs better. For the robustness analysis of neural networks DF and DDSE reach their limits, and hence, also HyDiff shows its bottlenecks. In the standard setup there is no synchronization happening between the two components because they are busy with their own analysis. However, the concept of HyDiff allows delays between the starting points of the components, which significantly helps HyDiff to make progress. Therefore, HyDiff represents a good combination of DF and DDSE.

In conclusion, HyDiff does not only combine the results but also can amplify the exploration of the components (e.g., see the results on *Math-60*, *CLI3-4*, *CLI5-6*, *Themis_Jetty*, *Insertion Sort* and *Image Processor*). The results of the conducted experiments also indicate that parallel differential fuzzing (PDF) might be an alternative when the application of symbolic execution is not reasonable due to issues with constraint solving (e.g., complex string constraints) or unsupported programming languages.

RQ5: HyDiff for differential testing. For regression analysis HyDiff identified all output differences. HyDiff also identified crashes as regression bugs, although the data set was not chosen to contain this kind of bugs specifically. In the worst-case complexity analysis

HyDiff triggers algorithmic complexity (AC) vulnerabilities. Especially the ability of HyDiff to also incorporate user-defined cost specifications makes it generally applicable. For the detection of side-channel (SC) vulnerabilities HyDiff identified all SC vulnerabilities in the data set. Furthermore, HyDiff can find the same SC vulnerabilities as DF and therefore it also outperforms the state-of-the-art static analysis tools. For the robustness analysis of neural networks HyDiff has been shown to be effective for the generation of adversarial inputs. However, the application was used to stress test the technique and the results should be not be overrated.

In conclusion, HyDiff can be used for differential testing. It effectively and efficiently combines fuzzing and symbolic execution to generate inputs that reveal differential behaviors including vulnerabilities in software.

7.6.2 *Contribution Results*

The investigation of the research questions RQ1-5 showed that the contributions C1-4 provide efficient and effective techniques to contribute to the research interest of differential software testing. Differential Fuzzing (C1) is effective for all considered differential testing applications and shows good performance over time as it can improve the outcomes continuously. Although Differential Dynamic Symbolic Execution (C2) is driven by costly program analysis and input reasoning it can effectively be used for differential testing as well and it can leverage constraint solving to quickly make progress. The parallel employment of fuzzing and symbolic execution in HyDiff (C3 and C4) successfully combines the strengths of both techniques and amplifies the exploration of behavioral differences.

The evaluation showed that HyDiff and its components can reveal differences that are relevant for numerous testing purposes. In regression analysis (A1) the presented techniques identified output differences between two program versions that potentially represent regression bugs. In worst-case complexity analysis (A2) serious algorithmic complexity vulnerabilities have been identified, which can be exploited, e.g., for denial of service attacks. In side-channel analysis (A3) the techniques showed to be effective to reveal potential side-channel vulnerabilities. Both, A2 and A3, are highly relevant analysis techniques in the context of software security. Finally, HyDiff has been stress-tested on the robustness analysis of neural networks. Despite its limitations, it still has been able to identify output differences to show the existence of adversarial inputs.

Overall, the proposed hybrid differential software testing technique and its components are valuable contributions in the area of software testing.

7.7 LIMITATIONS & THREATS TO VALIDITY

This section discusses the limitations and the threats to internal, external, and construct validity of the conducted research.

7.7.1 *Limitations of the Proposed Approaches*

In addition to the benefits of the proposed hybrid differential testing approach, it is also relevant to discuss the potential practical limitations. The fuzzing and the symbolic execution components need drivers to parse the input and call the application under test at a specific entry point. Although other studies have shown how to automate this step [57, 92], HyDiff requires this as input. Furthermore, HyDiff needs the information about syntactic changes (only for regression analysis) to define the fuzzing targets and the change-annotations for symbolic execution. Also these tasks could be automated in future work [91, 95].

Especially for regression analysis the output of HyDiff needs some post-processing to identify intended and unintended behavioral changes (cf. [93]).

In contrast to the worst-case complexity analysis, for which HyDiff generates an input that shows and triggers a vulnerability, HyDiff cannot automatically retrieve inputs that exploit side-channel vulnerabilities. For this application HyDiff solely shows a potential vulnerability, which still needs to be proven to be exploitable in a practical environment. Therefore, it needs some attack synthesis techniques for side-channels [122, 123].

7.7.2 *Threats to Validity*

In addition to the discussed practical limitations in the previous paragraph, it is also important to discuss the potential limitations of the conducted evaluation. Therefore, this section discusses the threats to internal, external, and construct validity.

Internal Validity. The main threat to internal validity is that systematic errors in the evaluation lead to invalid conclusions. Due to the fact that the evaluation of HyDiff handles randomized algorithms it is crucial to mitigate the risk of reporting randomly occurring phenomena. Fuzzing is mainly based on random mutations, and hence, it requires a sound statistical inspection. Therefore, the presented evaluation follows the guidelines by Arcuri and Briand [14] and Klees et al. [27]. All experiments have been repeated 30 times and the reported average values are augmented with the maximum/minimum values and the 95%-confidence intervals. The seed inputs used for fuzzing and symbolic execution have been randomly generated and have been the same for all repetitions of an experiment. The only requirement for the seed inputs is that they do not trigger any crash or timeout inside the application due to the requirements of the underlying fuzzing framework. A potential threat regarding the seed inputs is that fuzzing might perform different with other inputs. Therefore, the inputs have been generated randomly and are not chosen based on existing test suites. Additionally, the parameters like the time bounds of the multiple analysis types and the leveraged exploration heuristics might not have been selected adequately. Therefore, all parameters have been chosen based on preliminary assessments on their effectiveness. Furthermore, the collection of the data and the statistical evaluation is automated to avoid any careless error by humans.

External Validity. The main threat to external validity is that the evaluation may not generalize to other software projects or other types of differential analysis. The selected application scenarios, namely regression analysis, worst-case complexity analysis, side-channel analysis, and robustness analysis of neural networks, represent a wide range of differential analysis types, and therefore show the general applicability of the proposed testing framework. The subjects for the specific benchmarks are chosen based on the state-of-the-art benchmarks in the specific fields, including micro-benchmarks and real-world applications to show the practicality of HyDiff.

Construct Validity. The main threat to construct validity is that the used evaluation metrics may not represent adequate views to answer the research questions. Informally one can ask: Do the evaluation metrics measures the *right* thing? In the case of differential analysis we overall goal is to search *differences*. Therefore, the evaluation metrics need to focus on the various notions of *difference*, which depends on the specific application scenarios (cf. Section 3.4). Regression analysis (A1) and the robustness analysis of neural networks (A4) search for differences in the output, whereas worst-case complexity analysis (A2) and side-channel analysis (A3) search for differences in the execution cost.

Research question RQ1 and RQ2 ask *how good* the single techniques are in order to reveal differences. Therefore, the evaluation metrics measure how many differences can be found by the investigated techniques. The metrics follow that intuition that revealing more differences than another techniques is better. Depending on the application this can be measured best by counting the *output differences* (A1 and A4), by collecting the maximum execution cost (A2), and by calculating the execution cost difference (A3). The number of *decision differences* further helps to detect diverging execution behavior (relevant for A1 and A4). The same metrics are relevant for RQ3 and RQ4, which ask how good the hybrid approach performs compared to its single components in isolation. RQ5 asks whether HyDiff can be used for differential software testing. Also this question can be answered with the used metrics because all of them depend on the actual differences.

7.8 SUMMARY

This chapter discussed the validation of HyDiff and its components. The evaluation on four types of differential analysis techniques showed that HyDiff can reveal behavioral differences in real-world applications. Furthermore, it showed that HyDiff can outperform its components. Therefore, the hybrid combination is not only a combination of two techniques, but also amplifies the differential exploration. The chapter also showed the limitations of differential fuzzing and differential dynamic symbolic execution, which can affect the effectiveness of HyDiff. The following chapter represents the last chapter in this thesis and summarizes the contributions, their impact, and the future work.

8 CONCLUSION

This chapter concludes the results and contributions of this PhD work and discusses the future work.

8.1 SUMMARY & IMPACT

This thesis contributes the concept of hybrid differential software testing (HyDiff) as a combination of differential fuzzing (DF) and differential dynamic symbolic execution (DDSE). HyDiff's fuzzing component employs a search-based differential exploration implemented by a genetic algorithm. Its benefit is the inexpensive generation of inputs as well as the generation of unexpected inputs due to the random mutation strategies. HyDiff's symbolic execution component performs a systematic exploration guided by several differential heuristics. Because it can incorporate concrete inputs at run-time, it also can be driven by the inputs of the fuzzing component. It further can overcome specific constraints due to its constraint solving capabilities. This supports fuzzing, which might not reach deep program behaviors due to its random nature. Overall, HyDiff strengthens the presented differential fuzzing technique by combining it with the heuristic-based, systematic exploration in symbolic execution. As combination this supports a wide spectrum of differential testing applications and contributes a generally usable testing technique.

In order to evaluate these contributions, HyDiff, DF, and DDSE have been applied on several application scenarios: regression analysis, worst-case complexity analysis, side-channel analysis, and robustness analysis of neural networks. The results of the evaluation show that HyDiff can reveal behavioral differences in software and that HyDiff outperforms its components in isolation. This multifaceted evaluation shows that HyDiff can be applied in numerous testing disciplines, and so, contributes to the overall research interest of software testing. Additionally, the application of fuzzing for side-channel vulnerability detection, as proposed in one of the preliminary papers [1], already had a direct impact on the development of more fuzzing tools, namely SideFuzz [124].

In 2018 Willem Visser stated in his ACM interview: “I believe the combination of fuzzing and symbolic execution is where the next big breakthroughs are going to come from.” [13]. Therefore, HyDiff aligns well with the current research expectations of powerful hybrid testing techniques. HyDiff complements the existing work on hybrid testing and provides a baseline for future research. In summary, the contributions made by this thesis (especially the technical abilities to reveal behavioral differences) represent an important step in the direction of better (i.e., more secure and more reliable) software, and hence, support the overall goal of software engineering.

8.2 FUTURE WORK

The conducted research also has revealed interesting future research directions. First of all, the powerful employment of parallel differential fuzzing asks for a parallel variant of differential dynamic symbolic execution (DDSE). The experiments show that DDSE with double time budget does not significantly improve the results. A parallel DDSE approach could select n promising trie nodes for further exploration, which could result in n parallel instances of bounded symbolic execution. The approach of DDSE could therefore be split up in parallel exploration and input generation instances.

Furthermore, it could be examined in more detail why parallel fuzzing performs so much better than a single fuzzing instance and corresponding fuzzing guidelines could then be elaborated in a future study.

Another interesting study could be a comparison of different strategies for the combination of two (or more) components in a hybrid setup. HyDiff proposes the employment of a parallel setup, whereas for example Driller [81] follows a sequential combination. The proposed concept of hybrid differential software testing (HyDiff) can be also used to simply generate inputs to increase the program coverage. Therefore, these strategies could be assessed in a future empirical evaluation.

As discussed in Section 7.7.1, the proposed approach is not yet fully automated. Therefore, an interesting future work would be to automate the change-annotations as well as the driver synthesis to offer a fully automated differential testing approach. Since the different kinds of differential testing need different types of inputs, this automation probably will be specific for certain analysis types.

Additionally, the research around HyDiff aims at the generation of test inputs. The next step is to further automate the actual debugging and repair of the identified errors and vulnerabilities. Therefore, an interesting future work could be automated repair in the areas of software evolution and security vulnerabilities and their combination with the techniques proposed in this thesis.

BIBLIOGRAPHY

OWN PUBLICATIONS

[1] Shirin Nilizadeh, Yannic Noller, and Corina S. Păsăreanu. "DifFuzz: Differential Fuzzing for Side-channel Analysis." In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE '19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 176–187. DOI: 10.1109/ICSE.2019.00034.

[2] Yannic Noller. "Differential Program Analysis with Fuzzing and Symbolic Execution." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. Montpellier, France: ACM, 2018, pp. 944–947. ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3241537.

[3] Yannic Noller, Rody Kersten, and Corina S. Păsăreanu. "Badger: Complexity Analysis with Fuzzing and Symbolic Execution." In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: ACM, 2018, pp. 322–332. ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213868.

[4] Yannic Noller, Hoang Lam Nguyen, Minxing Tang, and Timo Kehrer. "Shadow Symbolic Execution with Java PathFinder." In: *SIGSOFT Softw. Eng. Notes* 42.4 (Jan. 2018), pp. 1–5. ISSN: 0163-5948. DOI: 10.1145/3149485.3149492.

[5] Yannic Noller, Hoang Lam Nguyen, Minxing Tang, Timo Kehrer, and Lars Grunske. "Complete Shadow Symbolic Execution with Java PathFinder." In: *SIGSOFT Softw. Eng. Notes* 44.4 (Dec. 2019), pp. 15–16. ISSN: 0163-5948. DOI: 10.1145/3364452.33644558.

[6] Yannic Noller, Corina S. Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske. "HyDiff: Hybrid Differential Software Analysis." In: *Will appear in: Proceedings of the 42nd International Conference on Software Engineering*. ICSE '20. Seoul, Korea, 2020.

GENERAL SOFTWARE TESTING REFERENCES

[7] Gordon Fraser and Andrea Arcuri. "A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite." In: *ACM Trans. Softw. Eng. Methodol.* 24.2 (Dec. 2014). ISSN: 1049-331X. DOI: 10.1145/2685612.

[8] Gordon Fraser and Andrea Arcuri. "Evolutionary Generation of Whole Test Suites." In: *2011 11th International Conference on Quality Software*. July 2011, pp. 31–40. DOI: 10.1109/QSIC.2011.19.

[9] Gordon Fraser and Andrea Arcuri. "EvoSuite: Automatic Test Suite Generation for Object-Oriented Software." In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. ESEC/FSE '11. Szeged, Hungary: Association for Computing Machinery, 2011, pp. 416–419. ISBN: 9781450304436. DOI: 10.1145/2025113.2025179.

[10] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming.” In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259.

[11] Alessandro Orso and Gregg Rothermel. “Software Testing: A Research Travelogue (2000-2014).” In: *Proceedings of the on Future of Software Engineering*. FOSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 117–132. ISBN: 9781450328654. DOI: 10.1145/2593882.2593885.

[12] Website. *Automated JUnit Generation*. http://www.agitar.com/solutions/products/automated_junit_generation.html. 2008.

[13] Website. *People of ACM - Willem Visser*. https://www.acm.org/articles/people-of-acm/2018/willem-visser. 2018.

FUZZING REFERENCES

[14] Andrea Arcuri and Lionel Briand. “A Hitchhiker’s guide to statistical tests for assessing randomized algorithms in software engineering.” In: *Software Testing, Verification and Reliability* 24.3 (2014), pp. 219–250. DOI: 10.1002/stvr.1486.

[15] Marcel Böhme. “STADS: Software Testing as Species Discovery.” In: *ACM Trans. Softw. Eng. Methodol.* 27.2 (June 2018). ISSN: 1049-331X. DOI: 10.1145/3210309.

[16] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. “Directed Greybox Fuzzing.” In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’17. Dallas, Texas, USA: ACM, 2017, pp. 2329–2344. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134020.

[17] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. “Coverage-based Greybox Fuzzing As Markov Chain.” In: *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’16. Vienna, Austria: ACM, 2016, pp. 1032–1043. ISBN: 978-1-4503-4139-4. DOI: 10.1145/2976749.2978428.

[18] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. “S2E: A Platform for in-Vivo Multi-Path Analysis of Software Systems.” In: *SIGPLAN Not.* 46.3 (Mar. 2011), pp. 265–278. ISSN: 0362-1340. DOI: 10.1145/1961296.1950396.

[19] Vijay Ganesh, Tim Leek, and Martin Rinard. “Taint-based directed whitebox fuzzing.” In: *2009 IEEE 31st International Conference on Software Engineering*. May 2009, pp. 474–484. DOI: 10.1109/ICSE.2009.5070546.

[20] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. “Grammar-Based Whitebox Fuzzing.” In: *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’08. Tucson, AZ, USA: Association for Computing Machinery, 2008, pp. 206–215. ISBN: 9781595938602. DOI: 10.1145/1375581.1375607.

[21] Patrice Godefroid, Michael Y. Levin, and David Molnar. “SAGE: Whitebox Fuzzing for Security Testing.” In: *Commun. ACM* 55.3 (Mar. 2012), pp. 40–44. ISSN: 0001-0782. DOI: 10.1145/2093548.2093564.

[22] Mark Harman, Phil McMinn, Jerffeson Teixeira de Souza, and Shin Yoo. "Search Based Software Engineering: Techniques, Taxonomy, Tutorial." In: *Empirical Software Engineering and Verification: International Summer Schools, LASER 2008-2010, Elba Island, Italy, Revised Tutorial Lectures*. Ed. by Bertrand Meyer and Martin Nordio. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 1–59. ISBN: 978-3-642-25231-0. DOI: 10.1007/978-3-642-25231-0_1.

[23] Christian Holler, Kim Herzig, and Andreas Zeller. "Fuzzing with Code Fragments." In: *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 445–458. ISBN: 978-931971-95-9.

[24] Matthias Höschele and Andreas Zeller. "Mining input grammars from dynamic taints." In: *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. Sept. 2016, pp. 720–725.

[25] Matthias Höschele and Andreas Zeller. "Mining Input Grammars with AUTOGRAM." In: *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. May 2017, pp. 31–34. DOI: 10.1109/ICSE-C.2017.14.

[26] Rody Kersten, Kasper Luckow, and Corina S. Păsăreanu. "POSTER: AFL-based Fuzzing for Java with Kelinci." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: ACM, 2017, pp. 2511–2513. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3138820.

[27] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. "Evaluating Fuzz Testing." In: *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. CCS '18. Toronto, Canada: ACM, 2018, pp. 2123–2138. ISBN: 978-1-4503-5693-0. DOI: 10.1145/3243734.3243804.

[28] Caroline Lemieux and Koushik Sen. "FairFuzz: A Targeted Mutation Strategy for Increasing Greybox Fuzz Testing Coverage." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. Montpellier, France: ACM, 2018, pp. 475–485. ISBN: 978-1-4503-5937-5. DOI: 10.1145/3238147.3238176.

[29] Yuekang Li, Bihuan Chen, Mahinthan Chandramohan, Shang-Wei Lin, Yang Liu, and Alwen Tiu. "Steelix: Program-state Based Binary Fuzzing." In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: ACM, 2017, pp. 627–637. ISBN: 978-1-4503-5105-8. DOI: 10.1145/3106237.3106295.

[30] Barton P. Miller, Louis Fredriksen, and Bryan So. "An Empirical Study of the Reliability of UNIX Utilities." In: *Commun. ACM* 33.12 (Dec. 1990), pp. 32–44. ISSN: 0001-0782. DOI: 10.1145/96267.96279.

[31] Carlos Pacheco and Michael D. Ernst. "Randoop: Feedback-Directed Random Testing for Java." In: *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*. OOPSLA '07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007, pp. 815–816. ISBN: 9781595938657. DOI: 10.1145/1297846.1297902.

[32] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. "Semantic Fuzzing with Zest." In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2019. Beijing, China: ACM, 2019, pp. 329–340. ISBN: 978-1-4503-6224-5. DOI: 10.1145/3293882.3330576.

[33] Esteban Pavese, Ezekiel Soremekun, Nikolas Havrikov, Lars Grunske, and Andreas Zeller. “Inputs from Hell: Generating Uncommon Inputs from Common Samples.” In: *arXiv preprint arXiv:1812.07525* (2018).

[34] Theofilos Petsios, Adrian Tang, Salvatore Stolfo, Angelos D. Keromytis, and Suman Jana. “NEZHA: Efficient Domain-Independent Differential Testing.” In: *2017 IEEE Symposium on Security and Privacy (SP)*. May 2017, pp. 615–632. DOI: 10.1109/SP.2017.27.

[35] Van-Thuan Pham, Marcel Böhme, Andrew E. Santosa, Alexandru R. Căciulescu, and Abhik Roychoudhury. “Smart Greybox Fuzzing.” In: *IEEE Transactions on Software Engineering* (2019), pp. 1–17.

[36] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. “Superion: Grammar-Aware Greybox Fuzzing.” In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. May 2019, pp. 724–735. DOI: 10.1109/ICSE.2019.00081.

[37] Website. *A library for coverage-guided fuzz testing - LLVM 3.9 documentation*. http://llvm.org/docs/LibFuzzer.html.

[38] Website. *American Fuzzy Lop (AFL) - a security-oriented fuzzer*. http://lcamtuf.coredump.cx/afl/. 2014.

[39] Website. *American Fuzzy Lop (AFL) - Technical Whitepaper*. http://lcamtuf.coredump.cx/afl/technical_details.txt. 2014.

[40] Website. *ASM: a very small and fast Java bytecode manipulation framework*. https://asm.ow2.io. 2019.

[41] Website. *Binary fuzzing strategies: what works, what doesn't*. http://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html. 2014.

[42] Website. *Commons BCEL*. https://commons.apache.org/proper/commons-bcel/. 2019.

[43] Website. *Introducing jsfunfuzz*. http://www.squarefree.com/2007/08/02/introducing-jsfunfuzz/. 2007.

[44] Website. *Microsoft Security Risk Detection*. https://www.microsoft.com/en-us/security-risk-detection/. 2016.

[45] Website. *OSS-Fuzz - continuous fuzzing of open source software*. https://google.github.io/oss-fuzz/. 2016.

[46] Website. *Peach Fuzzer Platform*. https://www.peach.tech/products/peach-fuzzer/peach-platform/.

[47] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. “Search-Based Fuzzing.” In: *The Fuzzing Book*. Saarland University, 2019. URL: https://www.fuzzingbook.org/html/SearchBasedFuzzer.html.

[48] Andreas Zeller, Rahul Gopinath, Marcel Böhme, Gordon Fraser, and Christian Holler. “The Fuzzing Book.” In: *The Fuzzing Book*. Saarland University, 2019. URL: https://www.fuzzingbook.org/.

SYMBOLIC EXECUTION REFERENCES

[49] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. "Demand-Driven Compositional Symbolic Execution." In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 367–381. ISBN: 978-3-540-78800-3.

[50] Abdulbaki Aydin, Lucas Bang, and Tevfik Bultan. "Automata-Based Model Counting for String Constraints." In: *Computer Aided Verification*. Ed. by Danielw Kroening and Corina S. Păsăreanu. Cham: Springer International Publishing, 2015, pp. 255–272. ISBN: 978-3-319-21690-4.

[51] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. "CVC4." In: *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*. Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. Lecture Notes in Computer Science. Snowbird, Utah. Springer, July 2011, pp. 171–177.

[52] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. "SELECT - a Formal System for Testing and Debugging Programs by Symbolic Execution." In: *SIGPLAN Not.* 10.6 (Apr. 1975), pp. 234–245. ISSN: 0362-1340. DOI: 10.1145/390016.808445.

[53] Stefan Bucur, Vlad Ureche, Cristian Zamfir, and George Candea. "Parallel Symbolic Execution for Automated Real-World Software Testing." In: *Proceedings of the Sixth Conference on Computer Systems*. EuroSys '11. Salzburg, Austria: Association for Computing Machinery, 2011, pp. 183–198. ISBN: 9781450306348. DOI: 10.1145/1966445.1966463.

[54] Cristian Cadar, Daniel Dunbar, and Dawson Engler. "KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs." In: *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. OSDI'08. San Diego, California: USENIX Association, 2008, pp. 209–224.

[55] Lori A. Clarke. "A Program Testing System." In: *Proceedings of the 1976 Annual Conference*. ACM '76. Houston, Texas, USA: ACM, 1976, pp. 488–491. DOI: 10.1145/800191.805647.

[56] Lori A. Clarke and Debra J. Richardson. "Applications of symbolic evaluation." In: *Journal of Systems and Software* 5.1 (1985), pp. 15–35. ISSN: 0164-1212. DOI: https://doi.org/10.1016/0164-1212(85)90004-4.

[57] Patrice Godefroid, Nils Klarlund, and Koushik Sen. "DART: Directed Automated Random Testing." In: *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '05. Chicago, IL, USA: Association for Computing Machinery, 2005, pp. 213–223. ISBN: 1595930566. DOI: 10.1145/1065010.1065036.

[58] Patrice Godefroid and Daniel Luchaup. "Automatic Partial Loop Summarization in Dynamic Test Generation." In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA '11. Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 23–33. ISBN: 9781450305624. DOI: 10.1145/2001420.2001424.

[59] Klaus Havelund and Thomas Pressburger. "Model checking JAVA programs using JAVA PathFinder." In: *International Journal on Software Tools for Technology Transfer* 2.4 (2000), pp. 366–381. DOI: 10.1007/s100090050043.

[60] Sarfraz Khurshid, Corina S. Păsăreanu, and Willem Visser. “Generalized Symbolic Execution for Model Checking and Testing.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by Hubert Garavel and John Hatcliff. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 553–568. ISBN: 978-3-540-36577-8.

[61] James C. King. “A New Approach to Program Testing.” In: *SIGPLAN Not.* 10.6 (Apr. 1975), pp. 228–233. ISSN: 0362-1340. DOI: 10.1145/390016.808444.

[62] James C. King. “Symbolic Execution and Program Testing.” In: *Commun. ACM* 19.7 (July 1976), pp. 385–394. ISSN: 0001-0782. DOI: 10.1145/360248.360252.

[63] Kin-Keung Ma, Khoo Yit Phang, Jeffrey S. Foster, and Michael Hicks. “Directed Symbolic Execution.” In: *Static Analysis*. Ed. by Eran Yahav. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 95–111. ISBN: 978-3-642-23702-7.

[64] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver.” In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[65] Saahil Ognawala, Martín Ochoa, Alexander Pretschner, and Tobias Limmer. “MACKE: Compositional Analysis of Low-Level Vulnerabilities with Symbolic Execution.” In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. ASE 2016. Singapore, Singapore: Association for Computing Machinery, 2016, pp. 780–785. ISBN: 9781450338455. DOI: 10.1145/2970276.2970281.

[66] Corina S. Păsăreanu, Neha Rungta, and Willem Visser. “Symbolic Execution with Mixed Concrete-Symbolic Solving.” In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA ’11. Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 34–44. ISBN: 9781450305624. DOI: 10.1145/2001420.2001425.

[67] Corina S. Păsăreanu, Willem Visser, David Bushnell, Jaco Geldenhuys, Peter Mehlitz, and Neha Rungta. “Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis.” In: *Automated Software Engineering* 20.3 (2013), pp. 391–425. DOI: 10.1007/s10515-013-0122-2.

[68] Prateek Saxena, Pongsin Poosankam, Stephen McCamant, and Dawn Song. “Loop-Extended Symbolic Execution on Binary Programs.” In: *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*. ISSTA ’09. Chicago, IL, USA: Association for Computing Machinery, 2009, pp. 225–236. ISBN: 9781605583389. DOI: 10.1145/1572272.1572299.

[69] Koushik Sen, Darko Marinov, and Gul Agha. “CUTE: A Concolic Unit Testing Engine for C.” In: *SIGSOFT Softw. Eng. Notes* 30.5 (Sept. 2005), pp. 263–272. ISSN: 0163-5948. DOI: 10.1145/1095430.1081750.

[70] Matt Staats and Corina Păsăreanu. “Parallel Symbolic Execution for Structural Test Generation.” In: *Proceedings of the 19th International Symposium on Software Testing and Analysis*. ISSTA ’10. Trento, Italy: Association for Computing Machinery, 2010, pp. 183–194. ISBN: 9781605588230. DOI: 10.1145/1831708.1831732.

[71] Jan Strejček and Marek Trtík. “Abstracting Path Conditions.” In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA 2012. Minneapolis, MN, USA: Association for Computing Machinery, 2012, pp. 155–165. ISBN: 9781450314541. DOI: 10.1145/2338965.2336772.

[72] Willem Visser, Klaus Havelund, Guillaume Brat, SeungJoon Park, and Flavio Lerda. "Model Checking Programs." In: *Automated Software Engineering* 10.2 (2003), pp. 203–232. DOI: 10.1023/A:1022920129859.

[73] Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. "Test Input Generation for Java Containers Using State Matching." In: *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. ISSTA '06. Portland, Maine, USA: Association for Computing Machinery, 2006, pp. 37–48. ISBN: 1595932631. DOI: 10.1145/1146238.1146243.

[74] Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. "Fitness-guided path exploration in dynamic symbolic execution." In: *2009 IEEE/IFIP International Conference on Dependable Systems Networks*. June 2009, pp. 359–368. DOI: 10.1109/DSN.2009.5270315.

HYBRID TESTING REFERENCES

[75] Sang K. Cha, Thanassis Avgerinos, Alexandre Rebert, and David Brumley. "Unleashing Mayhem on Binary Code." In: *2012 IEEE Symposium on Security and Privacy*. May 2012, pp. 380–394. DOI: 10.1109/SP.2012.31.

[76] Juan P. Galeotti, Gordon Fraser, and Andrea Arcuri. "Improving search-based test suite generation with dynamic symbolic execution." In: *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*. Nov. 2013, pp. 360–369. DOI: 10.1109/ISSRE.2013.6698889.

[77] Rupak Majumdar and Koushik Sen. "Hybrid Concolic Testing." In: *29th International Conference on Software Engineering (ICSE'07)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2007, pp. 416–426. DOI: 10.1109/ICSE.2007.41.

[78] Saahil Ognawala, Thomas Hutzelmann, Eirini Psallida, and Alexander Pretschner. "Improving Function Coverage with Munch: A Hybrid Fuzzing and Directed Symbolic Execution Approach." In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*. SAC '18. Pau, France: ACM, 2018, pp. 1475–1482. ISBN: 978-1-4503-5191-1. DOI: 10.1145/3167132.3167289.

[79] Saahil Ognawala, Fabian Kilger, and Alexander Pretschner. "Compositional fuzzing aided by targeted symbolic execution." In: *arXiv preprint arXiv:1903.02981* (2019).

[80] Brian S. Pak. "Hybrid Fuzz Testing: Discovering Software Bugs via Fuzzing and Symbolic Execution." MA thesis. Pittsburgh, PA, USA: Carnegie Mellon University, 2012.

[81] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution." In: *23nd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. 2016. DOI: https://doi.org/10.14722/ndss.2016.23368.

[82] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. "QSYM : A Practical Concolic Execution Engine Tailored for Hybrid Fuzzing." In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 745–761. ISBN: 978-1-939133-04-5.

REGRESSION ANALYSIS REFERENCES

[83] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. "Boogie: A Modular Reusable Verifier for Object-Oriented Programs." In: *Formal Methods for Components and Objects*. Ed. by Frank S. de Boer, Marcello M. Bonsangue, Susanne Graf, and Willem-Paul de Roever. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 364–387. ISBN: 978-3-540-36750-5.

[84] Marcel Boehme. "Automated Regression Testing and Verification of Complex Code Changes." PhD thesis. Singapore: National University of Singapore, 2014.

[85] Marcel Böhme, Bruno C. d. S. Oliveira, and Abhik Roychoudhury. "Partition-based Regression Verification." In: *Proceedings of the 2013 International Conference on Software Engineering*. ICSE '13. San Francisco, CA, USA: IEEE Press, 2013, pp. 302–311. ISBN: 978-1-4673-3076-3.

[86] Zhongxian Gu, Earl T. Barr, David J. Hamilton, and Zhendong Su. "Has the Bug Really Been Fixed?" In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE 2010. Cape Town, South Africa: Association for Computing Machinery, 2010, pp. 55–64. ISBN: 9781605587196. DOI: 10.1145/1806799.1806812.

[87] Mary Jean Harrold. "Testing evolving software." In: *Journal of Systems and Software* 47.2 (1999), pp. 173–181. ISSN: 0164-1212. DOI: https://doi.org/10.1016/S0164-1212(99)00037-0.

[88] Chris Hawblitzel, Jon Howell, Jacob R. Lorch, Arjun Narayan, Bryan Parno, Danfeng Zhang, and Brian Zill. "Ironclad Apps: End-to-End Security via Automated Full-System Verification." In: *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. Broomfield, CO: USENIX Association, Oct. 2014, pp. 165–181. ISBN: 978-1-931971-16-4.

[89] Wei Jin, Alessandro Orso, and Tao Xie. "Automated Behavioral Regression Testing." In: *2010 Third International Conference on Software Testing, Verification and Validation*. Apr. 2010, pp. 137–146. DOI: 10.1109/ICST.2010.64.

[90] Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. "SYMDIFF: A Language-Agnostic Semantic Diff Tool for Imperative Programs." In: *Computer Aided Verification*. Ed. by P. Madhusudan and Sanjit A. Seshia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 712–717. ISBN: 978-3-642-31424-7.

[91] Wei Le and Shannon D. Pattison. "Patch Verification via Multiversion Interprocedural Control Flow Graphs." In: *Proceedings of the 36th International Conference on Software Engineering*. ICSE 2014. Hyderabad, India: Association for Computing Machinery, 2014, pp. 1047–1058. ISBN: 9781450327565. DOI: 10.1145/2568225.2568304.

[92] Alessandro Orso and Tao Xie. "BERT: BEhavioral Regression Testing." In: *Proceedings of the 2008 International Workshop on Dynamic Analysis: Held in Conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*. WODA '08. Seattle, Washington: Association for Computing Machinery, 2008, pp. 36–42. ISBN: 9781605580548. DOI: 10.1145/1401827.1401835.

[93] Hristina Palikareva, Tomasz Kuchta, and Cristian Cadar. "Shadow of a Doubt: Testing for Divergences between Software Versions." In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. May 2016, pp. 1181–1192. DOI: 10.1145/2884781.2884845.

[94] Jihun Park, Miryung Kim, Baishakhi Ray, and Doo-Hwan Bae. "An empirical study of supplementary bug fixes." In: *2012 9th IEEE Working Conference on Mining Software Repositories (MSR)*. June 2012, pp. 40–49. DOI: 10.1109/MSR.2012.6224298.

[95] Nimrod Partush and Eran Yahav. "Abstract Semantic Differencing via Speculative Correlation." In: *SIGPLAN Not.* 49.10 (Oct. 2014), pp. 811–828. ISSN: 0362-1340. DOI: 10.1145/2714064.2660245.

[96] Suzette Person, Matthew B. Dwyer, Sebastian Elbaum, and Corina S. Păsăreanu. "Differential Symbolic Execution." In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. SIGSOFT '08/FSE-16. Atlanta, Georgia: ACM, 2008, pp. 226–237. ISBN: 978-1-59593-995-1. DOI: 10.1145/1453101.1453131.

[97] Suzette Person, Guowei Yang, Neha Rungta, and Sarfraz Khurshid. "Directed Incremental Symbolic Execution." In: *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI '11. San Jose, California, USA: ACM, 2011, pp. 504–515. ISBN: 978-1-4503-0663-8. DOI: 10.1145/1993498.1993558.

[98] Hyunmin Seo and Sunghun Kim. "Predicting Recurring Crash Stacks." In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. ASE 2012. Essen, Germany: Association for Computing Machinery, 2012, pp. 180–189. ISBN: 9781450312042. DOI: 10.1145/2351676.2351702.

[99] Sina Shamshiri, Gordon Fraser, Phil Mcminn, and Alessandro Orso. "Search-Based Propagation of Regression Faults in Automated Regression Testing." In: *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops*. Mar. 2013, pp. 396–399. DOI: 10.1109/ICSTW.2013.51.

[100] Kunal Taneja and Tao Xie. "DiffGen: Automated Regression Unit-Test Generation." In: *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*. Sept. 2008, pp. 407–410. DOI: 10.1109/ASE.2008.60.

[101] Zhihong Xu, Yunho Kim, Moonzoo Kim, Myra B. Cohen, and Gregg Rothermel. "Directed test suite augmentation: an empirical investigation." In: *Software Testing, Verification and Reliability* 25.2 (2015), pp. 77–114. DOI: 10.1002/stvr.1562.

[102] Guowei Yang, Corina S. Păsăreanu, and Sarfraz Khurshid. "Memoized Symbolic Execution." In: *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ISSTA 2012. Minneapolis, MN, USA: ACM, 2012, pp. 144–154. ISBN: 978-1-4503-1454-1. DOI: 10.1145/2338965.2336771.

[103] Shin Yoo and Mark Harman. "Regression testing minimization, selection and prioritization: a survey." In: *Software Testing, Verification and Reliability* 22.2 (2012), pp. 67–120. DOI: 10.1002/stvr.430.

SIDE-CHANNEL ANALYSIS REFERENCES

[104] Timos Antonopoulos, Paul Gazzillo, Michael Hicks, Eric Koskinen, Tachio Terauchi, and Shiyi Wei. "Decomposition Instead of Self-Composition for Proving the Absence of Timing Channels." In: *SIGPLAN Not.* 52.6 (June 2017), pp. 362–375. ISSN: 0362-1340. DOI: 10.1145/3140587.3062378.

[105] Lucas Bang, Abdulbaki Aydin, Quoc-Sang Phan, Corina S. Păsăreanu, and Tevfik Bultan. "String Analysis for Side Channels with Segmented Oracles." In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. FSE 2016. Seattle, WA, USA: Association for Computing Machinery, 2016, pp. 193–204. ISBN: 9781450342186. DOI: 10.1145/2950290.2950362.

[106] Gilles Barthe, Pedro R. D'Argenio, and Tamara Rezk. "Secure information flow by self-composition." In: *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.* June 2004, pp. 100–114. DOI: 10.1109/CSFW.2004.1310735.

[107] Tegan Brennan, Seemanta Saha, Tevfik Bultan, and Corina S. Păsăreanu. "Symbolic Path Cost Analysis for Side-channel Detection." In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: ACM, 2018, pp. 27–37. ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213867.

[108] Billy Bob Brumley and Nicola Tuveri. "Remote Timing Attacks Are Still Practical." In: *Computer Security – ESORICS 2011*. Ed. by Vijay Atluri and Claudia Diaz. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 355–371. ISBN: 978-3-642-23822-2.

[109] David Brumley and Dan Boneh. "Remote timing attacks are practical." In: *Computer Networks* 48.5 (2005). Web Security, pp. 701–716. ISSN: 1389-1286. DOI: https://doi.org/10.1016/j.comnet.2005.01.010.

[110] Jia Chen, Yu Feng, and Isil Dillig. "Precise Detection of Side-Channel Vulnerabilities Using Quantitative Cartesian Hoare Logic." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: Association for Computing Machinery, 2017, pp. 875–890. ISBN: 9781450349468. DOI: 10.1145/3133956.3134058.

[111] Tom Chothia, Yusuke Kawamoto, and Chris Novakovic. "LeakWatch: Estimating Information Leakage from Java Programs." In: *Computer Security - ESORICS 2014*. Ed. by Mirosław Kutyłowski and Jaideep Vaidya. Cham: Springer International Publishing, 2014, pp. 219–236. ISBN: 978-3-319-11212-1.

[112] Goran Doychev, Boris Köpf, Laurent Mauborgne, and Jan Reineke. "CacheAudit: A Tool for the Static Analysis of Cache Side Channels." In: *ACM Trans. Inf. Syst. Secur.* 18.1 (June 2015). ISSN: 1094-9224. DOI: 10.1145/2756550.

[113] Ralf Hund, Carsten Willems, and Thorsten Holz. "Practical Timing Side Channel Attacks against Kernel Space ASLR." In: *2013 IEEE Symposium on Security and Privacy*. May 2013, pp. 191–205. DOI: 10.1109/SP.2013.23.

[114] Yusuke Kawamoto, Fabrizio Biondi, and Axel Legay. "Hybrid Statistical Estimation of Mutual Information for Quantifying Information Flow." In: *FM 2016: Formal Methods*. Ed. by John Fitzgerald, Constance Heitmeyer, Stefania Gnesi, and Anna Philippou. Cham: Springer International Publishing, 2016, pp. 406–425. ISBN: 978-3-319-48989-6.

[115] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. "Spectre Attacks: Exploiting Speculative Execution." In: *2019 IEEE Symposium on Security and Privacy (SP)*. May 2019, pp. 1–19. DOI: 10.1109/SP.2019.00002.

[116] Paul C. Kocher. "Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and Other Systems." In: *Advances in Cryptology — CRYPTO '96*. Ed. by Neal Koblitz. Berlin, Heidelberg: Springer Berlin Heidelberg, 1996, pp. 104–113.

[117] Boris Köpf, Laurent Mauborgne, and Martín Ochoa. "Automatic Quantification of Cache Side-Channels." In: *Computer Aided Verification*. Ed. by P. Madhusudan and Sanjit A. Seshia. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 564–580. ISBN: 978-3-642-31424-7.

[118] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. "Meltdown: Reading Kernel Memory from User Space." In: *27th USENIX Security Symposium (USENIX Security 18)*. Baltimore, MD: USENIX Association, Aug. 2018, pp. 973–990. ISBN: 978-1-939133-04-5.

[119] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. "Last-Level Cache Side-Channel Attacks are Practical." In: *2015 IEEE Symposium on Security and Privacy*. May 2015, pp. 605–622. DOI: 10.1109/SP.2015.43.

[120] Heiko Mantel, Alexandra Weber, and Boris Köpf. "A Systematic Study of Cache Side Channels Across AES Implementations." In: *Engineering Secure Software and Systems*. Ed. by Eric Bodden, Mathias Payer, and Elias Athanasopoulos. Cham: Springer International Publishing, 2017, pp. 213–230. ISBN: 978-3-319-62105-0.

[121] Corina S. Păsăreanu, Quoc-Sang Phan, and Pasquale Malacaria. "Multi-run Side-Channel Analysis Using Symbolic Execution and Max-SMT." In: *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*. June 2016, pp. 387–400. DOI: 10.1109/CSF.2016.34.

[122] Quoc-Sang Phan, Lucas Bang, Corina S. Păsăreanu, Pasquale Malacaria, and Tevfik Bultan. "Synthesis of Adaptive Side-Channel Attacks." In: *2017 IEEE 30th Computer Security Foundations Symposium (CSF)*. Aug. 2017, pp. 328–342. DOI: 10.1109/CSF.2017.8.

[123] Seemanta Saha, William Eiers, Ismet Burak Kadron, Lucas Bang, and Tevfik Bultan. "Incremental Attack Synthesis." In: *SIGSOFT Softw. Eng. Notes* 44.4 (Dec. 2019), p. 16. ISSN: 0163-5948. DOI: 10.1145/3364452.336445759.

[124] Website. *SideFuzz: Fuzzing for side-channel vulnerabilities*. https://github.com/phayes/sidefuzz. 2018.

[125] Website. *The Meltdown Attack*. https://meltdownattack.com/.

[126] Website. *Timing attack in Google Keyczar library*. https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/. 2009.

[127] Website. *Xbox 360 Timing Attack*. http://beta.ivc.no/wiki/index.php/Xbox_360_Timing_Attack.

[128] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. "STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: ACM, 2017, pp. 859–874. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134016.

[129] Elvira Albert, Puri Arenas, Samir Genaim, and Germán Puebla. "Closed-Form Upper Bounds in Static Cost Analysis." In: *Journal of Automated Reasoning* 46.2 (2011), pp. 161–203.

[130] Glenn Ammons, Jong-Deok Choi, Manish Gupta, and Nikhil Swamy. "Finding and Removing Performance Bottlenecks in Large Systems." In: *ECOOP 2004 – Object-Oriented Programming*. Ed. by Martin Odersky. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 172–196. ISBN: 978-3-540-24851-4.

[131] Jacob Burnim, Sudeep Juvekar, and Koushik Sen. "WISE: Automated test generation for worst-case complexity." In: *2009 IEEE 31st International Conference on Software Engineering*. May 2009, pp. 463–473. DOI: 10.1109/ICSE.2009.5070545.

[132] Xiang Cai, Yuwei Gui, and Rob Johnson. "Exploiting Unix File-System Races via Algorithmic Complexity Attacks." In: *2009 30th IEEE Symposium on Security and Privacy*. May 2009, pp. 27–41. DOI: 10.1109/SP.2009.10.

[133] Bihuan Chen, Yang Liu, and Wei Le. "Generating Performance Distributions via Probabilistic Symbolic Execution." In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. Austin, Texas: Association for Computing Machinery, 2016, pp. 49–60. ISBN: 9781450339001. DOI: 10.1145/2884781.2884794.

[134] Scott A. Crosby and Dan S. Wallach. "Denial of Service via Algorithmic Complexity Attacks." In: *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*. SSYM'03. Washington, DC: USENIX Association, 2003, p. 3.

[135] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. "Gprof: A Call Graph Execution Profiler." In: *SIGPLAN Not.* 17.6 (June 1982), pp. 120–126. ISSN: 0362-1340. DOI: 10.1145/872726.806987.

[136] Bhargav S. Gulavani and Sumit Gulwani. "A Numerical Abstract Domain Based on Expression Abstraction and Max Operator with Application in Timing Analysis." In: *Computer Aided Verification*. Ed. by Aarti Gupta and Sharad Malik. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 370–384. ISBN: 978-3-540-70545-1.

[137] Sumit Gulwani. "SPEED: Symbolic Complexity Bound Analysis." In: *Computer Aided Verification*. Ed. by Ahmed Bouajjani and Oded Maler. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 51–62. ISBN: 978-3-642-02658-4.

[138] Jan Gustafsson, Andreas Ermedahl, Christer Sandberg, and Björn Lisper. "Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution." In: *2006 27th IEEE International Real-Time Systems Symposium (RTSS'06)*. Dec. 2006, pp. 57–66. DOI: 10.1109/RTSS.2006.12.

[139] Christopher Healy, Mikael Sjödin, Viresh Rustagi, David Whalley, and Robert van Engelen. "Supporting Timing Analysis by Automatic Bounding of Loop Iterations." In: *Real-Time Systems* 18.2 (2000), pp. 129–156. DOI: 10.1023/A:1008189014032.

[140] Benjamin Holland, Ganesh R. Santhanam, Payas Awadhutkar, and Suresh Kothari. "Statically-Informed Dynamic Analysis Tools to Detect Algorithmic Complexity Vulnerabilities." In: *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*. Oct. 2016, pp. 79–84. DOI: 10.1109/SCAM.2016.23.

[141] Guoliang Jin, Linhai Song, Xiaoming Shi, Joel Scherpelz, and Shan Lu. "Understanding and Detecting Real-World Performance Bugs." In: *SIGPLAN Not.* 47.6 (June 2012), pp. 77–88. ISSN: 0362-1340. DOI: 10.1145/2345156.2254075.

[142] Xuan-Bach D. Le, Corina S. Păsăreanu, Rohan Padhye, David Lo, Willem Visser, and Koushik Sen. "Saffron: Adaptive Grammar-Based Fuzzing for Worst-Case Analysis." In: *SIGSOFT Softw. Eng. Notes* 44.4 (Dec. 2019), p. 14. ISSN: 0163-5948. DOI: 10.1145/3364452.3364455.

[143] Caroline Lemieux, Rohan Padhye, Koushik Sen, and Dawn Song. "PerfFuzz: Automatically Generating Pathological Inputs." In: *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2018. Amsterdam, Netherlands: ACM, 2018, pp. 254–265. ISBN: 978-1-4503-5699-2. DOI: 10.1145/3213846.3213874.

[144] Yau-Tsun Steven Li, Sharad Malik, and Benjamin Ehrenberg. *Performance Analysis of Real-Time Embeded Software*. Norwell, MA, USA: Kluwer Academic Publishers, 1998. ISBN: 0792383826.

[145] Kasper Luckow, Rody Kersten, and Corina S. Păsăreanu. "Symbolic Complexity Analysis Using Context-Preserving Histories." In: *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*. Mar. 2017, pp. 58–68. DOI: 10.1109/ICST.2017.13.

[146] Theofilos Petsios, Jason Zhao, Angelos D. Keromytis, and Suman Jana. "SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities." In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. CCS '17. Dallas, Texas, USA: ACM, 2017, pp. 2155–2168. ISBN: 978-1-4503-4946-8. DOI: 10.1145/3133956.3134073.

[147] Gary Sevitsky, Wim de Pauw, and Ravi Konuru. "An information exploration tool for performance analysis of Java programs." In: *Proceedings Technology of Object-Oriented Languages and Systems. TOOLS 38*. Mar. 2001, pp. 85–101. DOI: 10.1109/TOOLS.2001.911758.

[148] Govind S. Shenoy, Jordi Tubella, and Antonio González. "Hardware/Software Mechanisms for Protecting an IDS against Algorithmic Complexity Attacks." In: *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops PhD Forum*. May 2012, pp. 1190–1196. DOI: 10.1109/IPDPSW.2012.145.

[149] Govind S. Shenoy, Jordi Tubella, and Antonio González. "Improving the Resilience of an IDS against Performance Throttling Attacks." In: *Security and Privacy in Communication Networks*. Ed. by Angelos D. Keromytis and Roberto Di Pietro. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 167–184. ISBN: 978-3-642-36883-7.

[150] Olha Shkaravska, Rody Kersten, and Marko van Eekelen. "Test-Based Inference of Polynomial Loop-Bound Functions." In: *Proceedings of the 8th International Conference on the Principles and Practice of Programming in Java*. PPPJ '10. Vienna, Austria: Association for Computing Machinery, 2010, pp. 99–108. ISBN: 9781450302692. DOI: 10.1145/1852761.1852776.

[151] Randy Smith, Cristian Estan, and Somesh Jha. "Backtracking Algorithmic Complexity Attacks against a NIDS." In: *2006 22nd Annual Computer Security Applications Conference (ACSAC'06)*. Dec. 2006, pp. 89–98. DOI: 10.1109/ACSAC.2006.17.

[152] Xiaoshan Sun, Liang Cheng, and Yang Zhang. "A Covert Timing Channel via Algorithmic Complexity Attacks: Design and Analysis." In: *2011 IEEE International Conference on Communications (ICC)*. June 2011, pp. 1–5. DOI: 10.1109/icc.2011.5962718.

[153] Jiayi Wei, Jia Chen, Yu Feng, Kostas Ferles, and Isil Dillig. "Singularity: Pattern Fuzzing for Worst Case Complexity." In: *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: ACM, 2018. DOI: 10.1145/3236024.3236039.

[154] Reinhard Wilhelm. "Determining Bounds on Execution Times." In: *Embedded Systems Design and Verification - Volume 1 of the Embedded Systems Handbook*. 2009, p. 9. DOI: 10.1201/9781439807637.ch9.

[155] Pingyu Zhang, Sebastian Elbaum, and Matthew B. Dwyer. "Automatic generation of load tests." In: *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*. Nov. 2011, pp. 43–52. DOI: 10.1109/ASE.2011.6100093.

NEURAL NETWORK REFERENCES

[156] Elie Aljalbout, Vladimir Golkov, Yawar Siddiqui, Maximilian Strobel, and Daniel Cremers. "Clustering with deep learning: Taxonomy and new methods." In: *arXiv preprint arXiv:1801.07648* (2018).

[157] Osbert Bastani, Yani Ioannou, Leonidas Lampropoulos, Dimitrios Vytiniotis, Aditya Nori, and Antonio Criminisi. "Measuring Neural Net Robustness with Constraints." In: *Advances in Neural Information Processing Systems 29*. Ed. by D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett. Curran Associates, Inc., 2016, pp. 2613–2621.

[158] Nicholas Carlini and David Wagner. "Towards Evaluating the Robustness of Neural Networks." In: *2017 IEEE Symposium on Security and Privacy (SP)*. May 2017, pp. 39–57. DOI: 10.1109/SP.2017.49.

[159] Ronan Collobert and Jason Weston. "A Unified Architecture for Natural Language Processing: Deep Neural Networks with Multitask Learning." In: *Proceedings of the 25th International Conference on Machine Learning*. ICML '08. Helsinki, Finland: Association for Computing Machinery, 2008, pp. 160–167. ISBN: 9781605582054. DOI: 10.1145/1390156.1390177.

[160] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. "Generative Adversarial Nets." In: *Advances in Neural Information Processing Systems 27*. Ed. by Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger. Curran Associates, Inc., 2014, pp. 2672–2680.

[161] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and harnessing adversarial examples." In: *arXiv preprint arXiv:1412.6572* (2014).

[162] Divya Gopinath, Guy Katz, Corina S. Păsăreanu, and Clark Barrett. "DeepSafe: A Data-Driven Approach for Assessing Robustness of Neural Networks." In: *Automated Technology for Verification and Analysis*. Ed. by Shuvendu K. Lahiri and Chao Wang. Cham: Springer International Publishing, 2018, pp. 3–19. ISBN: 978-3-030-01090-4.

[163] Divya Gopinath, Corina S. Păsăreanu, Kaiyuan Wang, Mengshi Zhang, and Sarfraz Khurshid. "Symbolic Execution for Attribution and Attack Synthesis in Neural Networks." In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. May 2019, pp. 282–283. DOI: 10.1109/ICSE-Companion.2019.00115.

[164] Divya Gopinath, Kaiyuan Wang, Mengshi Zhang, Corina S. Păsăreanu, and Sarfraz Khurshid. "Symbolic execution for deep neural networks." In: *arXiv preprint arXiv:1807.10439* (2018).

[165] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. "DLFuzz: Differential Fuzzing Testing of Deep Learning Systems." In: *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ESEC/FSE 2018. Lake Buena Vista, FL, USA: Association for Computing Machinery, 2018, pp. 739–743. ISBN: 9781450355735. DOI: 10.1145/3236024.3264835.

[166] John R. Hershey, Zhuo Chen, Jonathan Le Roux, and Shinji Watanabe. "Deep clustering: Discriminative embeddings for segmentation and separation." In: *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. Mar. 2016, pp. 31–35. DOI: 10.1109/ICASSP.2016.7471631.

[167] "Introduction to artificial neural networks." In: *Proceedings Electronic Technology Directions to the Year 2000*. Los Alamitos, CA, USA: IEEE Computer Society, May 1995, pp. 36–62. DOI: 10.1109/ETD.1995.403491.

[168] S. Jothilakshmi and V.N. Gudivada. "Chapter 10 - Large Scale Data Enabled Evolution of Spoken Language Research and Applications." In: *Cognitive Computing: Theory and Applications*. Ed. by Venkat N. Gudivada, Vijay V. Raghavan, Venu Govindaraju, and C.R. Rao. Vol. 35. Handbook of Statistics. Elsevier, 2016, pp. 301–340. DOI: https://doi.org/10.1016/bs.host.2016.07.005.

[169] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. "Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks." In: *Computer Aided Verification*. Ed. by Rupak Majumdar and Viktor Kunčak. Cham: Springer International Publishing, 2017, pp. 97–117. ISBN: 978-3-319-63387-9.

[170] Jinhan Kim, Robert Feldt, and Shin Yoo. "Guiding Deep Learning System Testing Using Surprise Adequacy." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. May 2019, pp. 1039–1049. DOI: 10.1109/ICSE.2019.00108.

[171] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. "Adversarial machine learning at scale." In: *arXiv preprint arXiv:1611.01236* (2016).

[172] Richard Lippmann. "An introduction to computing with neural nets." In: *IEEE ASSP Magazine* 4.2 (Apr. 1987), pp. 4–22. ISSN: 1558-1284. DOI: 10.1109/MASSP.1987.1165576.

[173] Siqi Liu, Sidong Liu, Weidong Cai, Sonia Pujol, Ron Kikinis, and Dagan Feng. "Early diagnosis of Alzheimer's disease with deep learning." In: *2014 IEEE 11th International Symposium on Biomedical Imaging (ISBI)*. Apr. 2014, pp. 1015–1018. DOI: 10.1109/ISBI.2014.6868045.

[174] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, and et al. "DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. Montpellier, France: Association for Computing Machinery, 2018, pp. 120–131. ISBN: 9781450359375. DOI: 10.1145/3238147.3238202.

[175] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, and Yadong Wang. "DeepMutation: Mutation Testing of Deep Learning Systems." In: *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. Oct. 2018, pp. 100–111. DOI: 10.1109/ISSRE.2018.00021.

[176] Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, and Pascal Frossard. "DeepFool: A Simple and Accurate Method to Fool Deep Neural Networks." In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[177] Jorge Nocedal and Stephen Wright. *Numerical optimization*. Springer Science & Business Media, 2006.

[178] Augustus Odena and Ian Goodfellow. "Tensorfuzz: Debugging neural networks with coverage-guided fuzzing." In: *arXiv preprint arXiv:1807.10875* (2018).

[179] Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z. Berkay Celik, and Ananthram Swami. "The Limitations of Deep Learning in Adversarial Settings." In: *2016 IEEE European Symposium on Security and Privacy (EuroS P)*. Mar. 2016, pp. 372–387. DOI: 10.1109/EuroSP.2016.36.

[180] Nicolas Papernot, Patrick McDaniel, Xi Wu, Somesh Jha, and Ananthram Swami. "Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks." In: *2016 IEEE Symposium on Security and Privacy (SP)*. May 2016, pp. 582–597. DOI: 10.1109/SP.2016.41.

[181] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. "DeepXplore: Automated Whitebox Testing of Deep Learning Systems." In: *Proceedings of the 26th Symposium on Operating Systems Principles*. SOSP 2017. Shanghai, China: Association for Computing Machinery, 2017, pp. 1–18. ISBN: 9781450350853. DOI: 10.1145/3132747.3132785.

[182] Alun Preece. "Asking Why in AI: Explainability of intelligent systems - perspectives and challenges." In: *Intelligent Systems in Accounting, Finance and Management* 25.2 (2018), pp. 63–72. DOI: 10.1002/isaf.1422.

[183] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. "Mastering the game of Go with deep neural networks and tree search." In: *Nature* 529.7587 (2016), pp. 484–489. DOI: 10.1038/nature16961.

[184] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. "Concolic Testing for Deep Neural Networks." In: *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ASE 2018. Montpellier, France: Association for Computing Machinery, 2018, pp. 109–119. ISBN: 9781450359375. DOI: 10.1145/3238147.3238172.

[185] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. "Intriguing properties of neural networks." In: *arXiv preprint arXiv:1312.6199* (2013).

[186] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. "DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars." In: *Proceedings of the 40th International Conference on Software Engineering*. ICSE 2018. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 303–314. ISBN: 9781450356381. DOI: 10.1145/3180155.3180220.

[187] Florian Tramer and Dan Boneh. "Adversarial Training and Robustness for Multiple Perturbations." In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett. Curran Associates, Inc., 2019, pp. 5858–5868.

[188] Avraam Tsantekidis, Nikolaos Passalis, Anastasios Tefas, Juho Kanniainen, Moncef Gabbouj, and Alexandros Iosifidis. "Forecasting Stock Prices from the Limit Order Book Using Convolutional Neural Networks." In: *2017 IEEE 19th Conference on Business Informatics (CBI)*. Vol. 01. July 2017, pp. 7–12. DOI: 10.1109/CBI.2017.23.

[189] Jingyi Wang, Guoliang Dong, Jun Sun, Xinyu Wang, and Peixin Zhang. "Adversarial Sample Detection for Deep Neural Network through Model Mutation Testing." In: *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. May 2019, pp. 1245–1256. DOI: 10.1109/ICSE.2019.00126.

[190] Website. *A Google self-driving car caused a crash for the first timet*. https://www.theverge.com/2016/2/29/11134344/google-self-driving-car-crash-report. 2016.

[191] Website. *Deep Learning Test Toolset*. https://github.com/theyoucheng/DLTT. 2020.

[192] Website. *Keras: The Python Deep Learning library*. https://keras.io. 2019.

[193] Website. *MNIST database*. http://yann.lecun.com/exdb/mnist/. 2013.

[194] Website. *Understanding the fatal Tesla accident on Autopilot and the NHTSA probe*. https://electrek.co/2016/07/01/understanding-fatal-tesla-accident-autopilot-nhtsa-probe/. 2016.

[195] Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. "Chapter 10 - Deep learning." In: *Data Mining (Fourth Edition)*. Ed. by Ian H. Witten, Eibe Frank, Mark A. Hall, and Christopher J. Pal. Fourth Edition. Morgan Kaufmann, 2017, pp. 417–466. ISBN: 978-0-12-804291-5. DOI: https://doi.org/10.1016/B978-0-12-804291-5.00010-6.

[196] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. "DeepHunter: A Coverage-Guided Fuzz Testing Framework for Deep Neural Networks." In: *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA 2019. Beijing, China: Association for Computing Machinery, 2019, pp. 146–157. ISBN: 9781450362245. DOI: 10.1145/3293882.3330579.

[197] Xiaofei Xie, Lei Ma, Haijun Wang, Yuekang Li, Yang Liu, and Xiaohong Li. "Diffchaser: Detecting disagreements for deep neural networks." In: *Proceedings of the 28th International Joint Conference on Artificial Intelligence*. AAAI Press. 2019, pp. 5772–5778.

BENCHMARKS / DATA SETS REFERENCES

[198] Marcel Böhme and Abhik Roychoudhury. “CoREBench: Studying Complexity of Regression Errors.” In: *Proceedings of the 23rd ACM/SIGSOFT International Symposium on Software Testing and Analysis*. ISSTA. San Jose, California, USA, 2014, pp. 105–115.

[199] René Just, Darioush Jalali, and Michael D. Ernst. “Defects4J: A Database of Existing Faults to Enable Controlled Testing Studies for Java Programs.” In: *Proceedings of the 2014 International Symposium on Software Testing and Analysis*. ISSTA 2014. San Jose, CA, USA: ACM, 2014, pp. 437–440. ISBN: 978-1-4503-2645-2. DOI: 10.1145/2610384.2628055.

[200] Alex Krizhevsky, Geoffrey Hinton, et al. *Learning multiple layers of features from tiny images*. Tech. rep. 2009.

[201] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. “Gradient-based learning applied to document recognition.” In: *Proceedings of the IEEE* 86.11 (Nov. 1998), pp. 2278–2324. ISSN: 1558-2256. DOI: 10.1109/5.726791.

[202] Website. *Commons CLI*. https://commons.apache.org/proper/commons-cli/. 2019.

[203] Website. *Cyber Grand Challenge (CGC)*. https://www.darpa.mil/program/cyber-grand-challenge. 2015.

[204] Website. *DARPA's Space/Time Analysis for Cybersecurity (STAC) program*. https://www.darpa.mil/program/space-time-analysis-for-cybersecurity. 2015.

[205] Website. *Debian Bug report log 800564 – php5: trivial hash complexity DoS attack*. https://bugs.debian.org/cgi-bin/bugreport.cgi?bug=800564. 2015.

[206] Website. *H2 Database Engine*. http://www.h2database.com/html/main.html.

[207] Website. *Regular Expressions*. http://www.mkyong.com/regular-expressions/10-java-regular-expression-examples-you-should-know/. 2012.

[208] Website. *Software-artifact Infrastructure Repository*. http://sir.unl.edu. 2019.

OTHER REFERENCES

[209] Ole-Johan Dahl, Edsger W. Dijkstra, and C. A. R. Hoare, eds. *Structured Programming*. GBR: Academic Press Ltd., 1972. ISBN: 0122005503.

[210] Frank DeRemer and Hans H. Kron. “Programming-in-the-Large Versus Programming-in-the-Small.” In: *IEEE Transactions on Software Engineering* 2.02 (Apr. 1976), pp. 80–86. ISSN: 1939-3520. DOI: 10.1109/TSE.1976.233534.

[211] Edsger W. Dijkstra. “A note on two problems in connexion with graphs.” In: *Numerische Mathematik* 1.1 (1959), pp. 269–271.

[212] “ISO/IEC/IEEE International Standard - Systems and software engineering–Vocabulary.” In: *ISO/IEC/IEEE 24765:2017(E)* (Aug. 2017), pp. 1–541. DOI: 10.1109/IEEESTD.2017.8016712.

[213] Ian Sommerville. *Software Engineering*. 9th. Pearson Education, Inc., 2011. ISBN: 137035152.

[214] Raja Vallee-Rai and Laurie J. Hendren. *Jimple: Simplifying Java Bytecode for Analyses and Transformations*. Tech. rep. 1998.

[215] Hans van Vliet. *Software Engineering: Principles and Practice*. 3rd. Wiley Publishing, 2008. ISBN: 0470031468.

All links were last followed on June 18, 2020.

OVERVIEW OF PUBLICATIONS

This section lists the publications that have been created along this PhD work and that served as basis for this thesis. Each publication is followed by a brief contribution statement.

JPF'2017 (Workshop)
Shadow Symbolic Execution with Java PathFinder [4]
Yannic Noller, Hoang Lam Nguyen, Minxing Tang, and Timo Kehrer
Discusses the implementation of shadow symbolic execution in the framework of Java PathFinder.

Contribution Statement: *Large parts of the implementation have been done by Hoang Lam Nguyen. My contributions are mostly on the conceptual level and the evaluation.*

ISSTA'2018
Badger: Complexity Analysis with Fuzzing and Symbolic Execution [3]
Yannic Noller, Rody Kersten, and Corina S. Păsăreanu
Badger proposes a hybrid testing framework for worst-case complexity analysis that combines cost-guided fuzzing and dynamic symbolic execution.

Contribution Statement: *My contribution is mainly the hybrid concept, the implementation of the symbolic execution component and its combination with fuzzing, and the evaluation.*

ASE'2018 (Doctoral Symposium)
Differential Program Analysis with Fuzzing and Symbolic Execution [2]
Yannic Noller
Proposes the idea of a general hybrid differential testing framework that combines differential fuzzing and differential symbolic execution.

Contribution Statement: *Completely my contribution.*

ICSE'2019
DifFuzz: Differential Fuzzing for Side-Channel Analysis [1]
Shirin Nilizadeh, Yannic Noller*, and Corina S. Păsăreanu (* joint first authors)*
DifFuzz proposes the concept of differential fuzzing for the detection of side-channel vulnerabilities. It is based on the cost-guided fuzzing approach by Badger and refines the concept to the maximization of the cost difference between two program executions.

Contribution Statement: *I mainly contributed to the concept, the implementation, and the evaluation. Shirin Nilizadeh and me shared the largest work package (the evaluation), which is the reason that we are both mentioned as first authors.*

JPF'2019 (Workshop)
Complete Shadow Symbolic Execution with Java PathFinder [5]
Yannic Noller, Hoang Lam Nguyen, Minxing Tang, Timo Kehrer, and Lars Grunske
Discusses the limitations of shadow symbolic execution and the concept of exhaustive symbolic execution driven by four-way forking.

Contribution Statement: *Large parts of the implementation have been done by Hoang Lam Nguyen and Minxing Tang. My contributions are mostly on the conceptual level and the evaluation.*

ICSE'2020
HyDiff: Hybrid Differential Software Analysis [6]
Yannic Noller, Corina S. Păsăreanu, Marcel Böhme, Youcheng Sun, Hoang Lam Nguyen, and Lars Grunske
HyDiff proposes the concept of hybrid differential software testing. Based on the gained insights in Badger and DifFuzz, HyDiff proposes a general differential fuzzer and combines it with a differential dynamic symbolic execution.

Contribution Statement: *The implementation of* HyDiff *is partially based on the work by Hoang Lam Nguyen, although largely extended by myself. The concept, implementation, and the evaluation are my main contributions.*

ERKLÄRUNG / DECLARATION

Hiermit erkläre ich, die Dissertation selbstständig und nur unter Verwendung der angegebenen Hilfen und Hilfsmittel angefertigt zu haben. Ich habe mich nicht anderwärts um einen Doktorgrad in dem Promotionsfach beworben und besitze keinen entsprechenden Doktorgrad. Die Promotionsordnung der Mathematisch-Naturwissenschaftlichen Fakultät, veröffentlicht im Amtlichen Mitteilungsblatt der Humboldt-Universität zu Berlin Nr. 42 am 11. Juli 2018, habe ich zur Kenntnis genommen.

I declare that I have completed the thesis independently using only the aids and tools specified. I have not applied for a doctor's degree in the doctoral subject elsewhere and do not hold a corresponding doctor's degree. I have taken due note of the Faculty of Mathematics and Natural Sciences PhD Regulations, published in the Official Gazette of Humboldt-Universität zu Berlin no. 42 on July 11, 2018.

Berlin, Germany, June 2020

Yannic Noller