Automatically Detecting and Mitigating Issues in Program Analyzers

Thesis approved by
the Department of Computer Science
University of Kaiserslautern-Landau
for the award of the Doctoral Degree
Doctor of Engineering (Dr.-Ing.)

to

Muhammad Numair Mansur

Date of Defense: March 31, 2023

Dean: Christoph Garth

Reviewer: Maria Christakis

Reviewer: Rupak Majumdar

Reviewer: Michael Pradel

DE-386

Abstract

In recent years, the formal methods community has made significant progress towards the development of industrial-strength static analysis tools that can check properties of real-world production code. Such tools can help developers detect potential bugs and security vulnerabilities in critical software before deployment. While the potential benefits of static analysis tools are clear, their usability and effectiveness in mainstream software development workflows often comes into question and can prevent software developers from using these tools to their full potential. In this dissertation, we focus on two major challenges that can limit their ability to be incorporated into software development workflows.

The first challenge is *unintentional unsoundness*. Static program analyzers are complicated tools, implementing sophisticated algorithms and performance heuristics. This makes them highly susceptible to undetected unintentional soundness issues. These issues in program analyzers can cause false negatives and have disastrous consequences e.g., when analyzing safety critical software. In this dissertation, we present novel techniques to detect unintentional unsoundness bugs in two foundational program analysis tools namely SMT solvers and Datalog engines. These tools are used extensively by the formal methods community, for instance, in software verification, systematic testing, and program synthesis. We implemented these techniques as easy-to-use open source tools that are publicly available on Github. With the proposed techniques, we were able to detect more than 55 unique and confirmed critical soundness bugs in popular and widely used SMT solvers and Datalog engines in only a few months of testing.

The second challenge is finding the right balance between *soundness*, *precision*, and *performance*. In an ideal world, a static analyzer should be as precise as possible while maintaining soundness and being sufficiently fast. However, to overcome undecidability issues, these tools have to employ a variety of techniques to be practical for example, compromising on the soundness of the analysis or approximating code behavior. Static analyzers therefore are not trivial to integrate into any usage scenario with different program sizes, resource constraints and SLAs. Most of the times, these tools also don't scale to large industrial code bases containing millions of lines of code. This makes it extremely challenging to get the most out of these analyzers and integrate them into everyday development activities, especially for average software development teams with little to no knowledge or understanding of advanced static analysis techniques. In this dissertation we present an approach to automatically tailor an abstract interpreter to the code under analysis and any given resource constraints. We implemented our technique as an open source framework, which is publicly available on Github. The second contribution of this dissertation in this challenge area is a technique to horizontally scale analysis tools in cloud-based static analysis platforms by splitting the input to the analyzer into partitions and analyzing the partitions independently. The technique was developed in collaboration with Amazon Web Services and is now being used in production in their CodeGuru service.

Acknowledgements

During the past few years, I met many people who, directly or indirectly, supported me through this time. I would like to thank them all.

First and foremost, I am very grateful to my advisor Maria Christakis for her continuous support, kindness, patience, and generous feedback. Without her, this thesis would not have been possible. Thank you Maria for giving me the opportunity of being your first Ph.D. student and for being an amazing mentor, colleague, teacher, and friend. I hope you are proud of this work. It was a pleasure working with you and I hope we will continue to collaborate in the future.

I am also very grateful to Valentin Wüstholz, who was a close collaborator during my Ph.D. years. This work would not have been the same without him. Thank you Valentin for your insightful comments, feedback, and ideas.

I also want to express my gratitude towards Nico Rosner and Martin Schäf, with whom I had a great internship experience at Amazon Web Services.

I would also like to thank my co-authors: Thomas Cottenier, Antonio Filieri, Matthias Heizmann, Linghui Luo, Benjamin Mariano, Jorge A. Navas, Nico Rosner, Martin Schäf, Christian Schilling, Aritra Sengupta, Willem Visser, and Fuyuan Zhang. I am also thankful to Irmak Saglam for all the help and support and to all my friends, colleagues, and ex-colleagues at MPI-SWS, especially Aman, Ana, Anne, Ashwani, Azalea, Burcu, Cedric, Clothilde, Damien, Daniel, Felix, Filip, Hasan, Iason, Ivan G., James, Khushraj, Kimaya, Laura, Leo, Lovro, Mariam, Mahmoud, Marco M., Marco P., Marko, Mehrdad, Michalis, Murat, Nastaran, Nina, Rajarshi, Ram, Rosa, Satya, Simin, Sofia, Soham, Stratis, Utkarsh, and Xuan.

I also want to thank the reviewers Maria Christakis, Rupak Majumdar, and Michael Pradel for taking the time to review this dissertation and to the head of my PhD committee Annette Bieniusa.

Many thanks to our amazing administrative staff at MPI-SWS, including Corinna, Geraldine, Mouna, Roslyn, Susanne, Vera, Mary-Lou, Christian, Pascal, Tobias, Torsten, and Andy.

Lastly, I would like to thank my parents, my brother, and my sister for their eternal love, prayers, and support. Wherever I am today, it's because of them.

This dissertation is dedicated to my late grandfather Sheikh M. Amin. He is one of the biggest inspirations of my life. I hope he is proud of me.

List of Publications

Related Publications

This dissertation is based on the following publications.

1. *Muhammad Numair Mansur*, Benjamin Mariano, Maria Christakis, Jorge A. Navas, and Valentin Wüstholz. Automatically tailoring abstract interpretation to custom usage scenarios. In *CAV*, volume 12760 of LNCS, pages 777–800. Springer, 2021.

2. *Muhammad Numair Mansur*, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *ESEC/FSE*, pages 701–712. ACM, 2020.

3. *Muhammad Numair Mansur*, Maria Christakis, and Valentin Wüstholz. Metamorphic testing of datalog engines. In *ESEC/FSE*, pages 639–650. ACM, 2021.

4. *Muhammad Numair Mansur*, Maria Christakis, and Valentin Wüstholz. Dependency-Aware Metamorphic Testing of Datalog Engines. Accepted in *ISSTA*, ACM, 2023.

5. Maria Christakis, Thomas Cottenier, Antonio Filieri, Linghui Luo, *Muhammad Numair Mansur*, Lee Pike, Nico Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. Input splitting for cloud-based static application security testing platforms. In *ESEC/FSE*, pages 1367–1378. ACM, 2022.

Other Publications

The following paper was published during my PhD studies, but is not included in this dissertation.

1. Maria Christakis, Matthias Heizmann, *Muhammad Numair Mansur*, Christian Schilling, and Valentin Wüstholz. Semantic fault localization and suspiciousness ranking. In *TACAS*, volume 11427 of LNCS, pages 226–243. Springer, 2019.

Contents

1 Introduction **2**
1.1 Detecting Unintentional Unsoundness Bugs in Program Analyzers 5
1.1.1 Detecting Soundness Bugs in SMT Solvers 7
1.1.2 Detecting Soundness Bugs in Datalog Engines 8
1.2 Balancing Soundness, Precision, and Performance in Static Analysis 10
1.2.1 Maximizing Precision while Adhering to Time Constraints 11
1.2.2 Splitting Analysis Inputs to Balance Soundness, Precision, and Performance 12
1.3 Outline and Publication Details 14

I Detecting Unintentional Unsoundness **15**

2 Detecting Critical Soundness Bugs in SMT Solvers **17**
2.1 Introduction 17
2.2 Overview 19
2.3 Our Approach 23
2.3.1 Fuzzing Technique 25
2.3.2 Instance Minimization 26
2.4 Implementation 27
2.5 Experimental Evaluation 28
2.5.1 Solver Selection 28
2.5.2 Logic Selection 29
2.5.3 Benchmark Selection 29
2.5.4 Experimental Setup 31
2.5.5 Experimental Results 31
2.6 Threats to Validity 36
2.7 Related Work 37
2.8 Summary and Remarks 39

3 Metamorphic Testing of Datalog Engines Using Conjunctive Queries **40**
3.1 Introduction 40
3.2 Overview 42
3.3 Background 45

3.4 Metamorphic Transformations 48
3.5 ADD Transformations 48
3.6 MOD Transformations 52
3.7 REM Transformations 53
3.7.1 Transformation Sequences 55
3.8 Beyond Conjunctive Queries 56
3.8.1 NEG Transformation 57
3.9 Implementation 58
3.10 Experimental Evaluation 58
3.10.1 Experimental Setup 59
3.10.2 Experimental Results 59
3.11 Threats to Validity 64
3.12 Related Work 64
3.13 Summary and Remarks 66

4 Dependency-Aware Metamorphic Testing of Datalog Engines 67
4.1 Introduction 67
4.2 Background 68
4.2.1 Datalog Programs 69
4.2.2 Precedence Graphs 70
4.3 Overview 71
4.4 Graph Annotator 73
4.5 Graph Transformer 75
4.5.1 Graph Rewrite Rules 75
4.5.2 Specifying Metamorphic Transformations 77
4.5.3 Example Metamorphic Transformations 79
4.6 Implementation 84
4.7 Experimental Evaluation 85
4.7.1 Setup 86
4.7.2 Results 86
4.7.3 Threats to Validity 89
4.8 Related Work 90
4.9 Summary and Remarks 90

II Balancing Soundness, Precision, and Performance 91

5 Automatically Tailoring Abstract Interpretation to Custom Usage Scenarios 93
5.1 Introduction 93
5.2 Overview 95
5.3 A Generic Abstract Interpreter 97
5.4 Our Technique 99
5.4.1 Recipe Optimization 99

5.4.2 Recipe Evaluation 100
5.4.3 Recipe Generation 101
5.5 Experimental Evaluation 103
5.5.1 Implementation 104
5.5.2 Benchmark Selection 104
5.5.3 Results 105
5.5.4 Threats to Validity 113
5.6 Related Work 113
5.7 Summary and Remarks 114

6 Input Splitting for Cloud-Based Static Application Security Testing Platforms 116
6.1 Introduction 116
6.2 Motivating Example 119
6.3 The SPLITMERGE Strategy 122
6.4 Experimental Evaluation 126
6.4.1 Experimental Settings 127
6.4.2 Experimental Results 129
6.5 Related Work 134
6.6 Summary and Remarks 136

7 Conclusion and Future Work 138

References 141

List of Algorithms

1 Core fuzzing procedure in STORM. 24
2 Depth-minimization procedure in STORM. 26
3 ADD transformations 50
4 MOD transformations 52
5 REM transformations 55
6 Optimization engine. 100
7 A recipe-generator instantiation. 102
8 Generate initial partitions 124
9 Split 125
10 Merge partitions 126

List of Figures

1.1 A simple program demonstrating the undecidability of live variable analysis. 11

2.1 Overview of the three STORM phases. 20
2.2 Original seed instance from SMT-COMP 2019 on the top, and simplified instance revealing critical bug in Z3's Z3str3 string solver on the bottom. 21
2.3 Simplified instance revealing critical bug in Z3's `dom-simplify` tactic on the top, and logically equivalent instance not revealing the bug on the bottom. 22
2.4 Median number of iterations to find bugs with different configurations of STORM. Each bar corresponds to a configuration with a certain depth and assertion bound. 33
2.5 Median time (in seconds) to find bugs with different configurations of STORM. Each bar corresponds to a configuration with a certain depth and assertion bound. 34

3.1 A simple Datalog program. 42
3.2 Pictorial view and transitive closure of `edge`. 43
3.3 Overview of our approach. 44
3.4 Generated program snippet for testing μZ. 45
3.5 Generated program snippet for testing Soufflé. 45
3.6 Containment mapping θ from Q_1 to Q_2 induces a mapping of subgoals. No mapping exists from Q_2 to Q_1. 47
3.7 Example of ADDEQU transformation. 49
3.8 Example of ADDCON transformation. 51
3.9 Example of MODEXP transformation. 51
3.10 Example of MODCON transformation. 53
3.11 Example of REMEXP transformation. 54
3.12 Example of REMEQU transformation. 56
3.13 All bugs reported in the three Datalog engines from May 1, 2020 to Feb 15, 2021. 62
3.14 Generated program snippet for testing DDlog. 63

4.1 Precedence graph (a) for a simple Datalog program (b). 70

4.2 Overview of our approach. 72
4.3 Annotated precedence graph for generating the program of Fig. 4.1. 74
4.4 An example EQU-AddRelNode transformation. 79
4.5 An example EQU-AddRelEdges transformation. 80
4.6 An example EQU-AddSelfEdge transformation. 81
4.7 An example CON-AddRelEdge transformation. 82
4.8 An example EXP-AddRelEdge transformation. 84

5.1 Overview of our framework. 96
5.2 Generic architecture of an abstract interpreter. 98
5.3 Comparing logico-numerical domains in CRAB. A domain d_1 is less precise than d_2 if there is a path from d_1 to d_2 going upward, otherwise d_1 and d_2 are incomparable. 103
5.4 Comparison of the number of assertions verified with the best recipe generated by each optimization algorithm and with the default recipe, for varying timeouts. 107
5.5 Comparison of the number of assertions verified by a tailored vs. the default recipe. 108
5.6 Comparison of the total time (in sec) that each algorithm requires for all iterations, for varying timeouts. 108
5.7 Comparison of the number of assertions verified with the best recipe generated by the different optimization algorithms, for different numbers of iterations. 109
5.8 Effect of different settings on the precision and performance of the abstract interpreter. (DW: `NUM_DELAY_WIDEN`, NI: `NUM_NARROW_ITERATIONS`, WT: `NUM_WIDEN_THRESHOLDS`, AS: array smashing, B: backward analysis, D: abstract domain, O: ingredient ordering). 110
5.9 Occurrence of domains (in %) in the best recipes for all assertion types. 111
5.10 Difference in the safe assertions across commits. 112

6.1 Simplified version of an OWASP test that uses a shared class. The method *doSomething* is referenced 347 times in different OWASP tests. 121
6.2 Dependency graphs of P. 123
6.3 LOC of the 17 open source Python projects with dependency analysis and analysis time used by Bandit. 132
6.4 Time used by dependency analysis compared to time used by Bandit. 134

List of Tables

2.1 Classes of bugs in SMT solvers. **GT** stands for ground truth and **SR** for solver result. 19
2.2 The tested logics per solver and the number of seed instances per logic. 30
2.3 Previously unknown, unique, and confirmed critical bugs found by STORM in the tested SMT solvers. 32
2.4 Size of original and minimized bug-revealing instances. Instance size is shown in terms of the number of bytes / number of assertions / maximum formula depth. 35
2.5 Code coverage increase as more instances are generated by STORM. 36

3.1 Query bugs detected by queryFuzz. 59
3.2 By-product bugs detected by queryFuzz. 61
3.3 Categorization of Soufflé bugs into the components in which they were found. 62

4.1 Remaining metamorphic transformations implemented in DLSmith (grouped by oracles EQU, CON, and EXP). 78
4.2 Query bugs detected by DLSmith. 87
4.3 Code coverage achieved by seeds alone, queryFuzz, DLSmith with empty seeds, and DLSmith. L represents line coverage, and F function coverage. 89
4.4 Average running time (in seconds) of DLSmith when executing its first two phases. 89

5.1 CRAB settings and their possible values as used in our experiments. Default settings are shown in bold. 104
5.2 Overview of projects. 105
5.3 Benchmark characteristics (20 files per project). The last three columns show the number of functions, assertions, and LLVM instructions in the analyzed files. 106

6.1 Score (based on precision and recall) and analysis time for several SAST tools on the OWASP Benchmark v1.1. Data taken from the OWASP Benchmark public repository. 120

6.2 Comparing both SIZELIMITING and SPLITMERGE to baselines on OWASP, Juliet and Maven. The percentages in the rows for SIZELIMITING and SPLITMERGE strategies correspond to the reduction (or gain) in the number of findings, total time and memory usage when compared with NoSplit-UT. **Best Possible Latency** column shows the speedup achieved with SIZELIMITING and SPLITMERGE strategies. In the cases where NoSplit-UT failed to give a result within 24 hours, we report the speedup as ∞x and the number of findings as N/A. 130
6.3 Timeout, crash and success rates of analysis runs. 131

Chapter 1

Introduction

The importance of correct, robust, and secure software systems is increasingly vital for the smooth functioning of our society. From remote working and online education to autonomous vehicles and space travel, software plays a profound role in our modern way of life. Although these systems are incredibly effective in understanding, computing, and manipulating complex data, they do have one weakness: they are developed by human beings. And humans make mistakes. Since developing these software systems is a notoriously difficult task, any reasonably complex piece of software inevitably contains bugs. Depending on the application domain, these bugs can have serious consequences. For example, in 2022 Tesla Motors had to recall 130,000 vehicles due to a software issue in its infotainment system which could cause the CPU to overheat and affect critical functions, increasing the risk of a crash. In 2016, anonymous hackers were able to exploit a combination of vulnerabilities in The DAO's (Decentralized Autonomous Organization) smart contracts [182] and were able to steal over $50 million USD worth of Ether from the Ethereum blockchain. Finding bugs in modern software systems is, therefore, a question of safeguarding life and property.

Recent years have seen tremendous progress in the development and industrial adoption of rigorous techniques and tools to automatically detect bugs in software and ensure its correctness. *Static program analysis* is one such technique to automatically reason about the runtime behavior of a program without actually running it. It is a powerful approach that can be used to detect a wide range of security vulnerabilities and enables program optimization in compilers, automatic parallelization and, if accurate enough, correctness verification. *Static program analyzers* are tools that implement program analysis techniques and are used to analyze other programs for flaws.

Over the past couple of years, the formal methods community has made significant progress towards the development of industrial-strength static program analyzers that can check properties of real-world production code. These tools can help developers detect potential bugs and security vulnerabilities in critical software systems before deployment without executing them. Tools such as Coverity

Scan [2], FindBugs [21], Klocwork [7], fault prediction [158] and Infer [59, 60] analyze hundreds of thousands of lines of open-source and industrial code every day. Since 2014, using Infer, developers at Meta were able to catch and fix over 100,000 issues before they ever reached production code [87]. Infer participates as a bot during the code review process for Android and iOS apps for Facebook, Instagram, Messenger, and WhatsApp, as well as on their backend C++ and Java code.

While the potential benefits of analyzers are obvious, their usability and effectiveness in mainstream software development workflows still often comes into question and can prevent average software development teams from using these tools to their full potential. This is because the undecidability of the halting problem makes formal reasoning about program properties difficult. In fact, Rice has shown that all non-trivial questions about the behavior of a program are undecidable [219]. One common theme in static analysis is that to remain computable, one can only provide *approximate answers*. In practice, this is achieved with a sacrifice in terms of *soundness* or *completeness*. This means that a static analysis technique might either miss a bug (*false negative*) or report correct code as having a bug (*false positive*), might not handle certain language features or can only report certain types of bugs. Developers, however, expect static program analyzers to satisfy a particularly challenging set of requirements [68]:

– *Soundness*: The tool should not report any false negatives.

– *Precision*: The tool should be as precise as possible and return a minimum number of false positives.

– *Scalability*: The tool should scale to millions of lines of code.

– *Automation*: The tool should be push-button, i.e., it should be able to configure itself automatically to the code under analysis and any given resource constraints.

– *Efficiency/performance*: The tool should not get in the way of the development cycle, i.e., it should report the results within the SLA (service-level agreement), which is typically minutes.

Decades of research have yielded the discovery of novel algorithms, data structures, and design principles that make static program analysis more precise, scalable, and faster than ever before. Practical program analyzers have to make *tradeoffs* (e.g., in soundness, precision, or performance) to maintain a delicate balance between returning the minimum number of false negatives/positives, scaling to very large industrial codebases, being highly automatic and having minimum overhead for the developers. Designing, implementing and deploying program analyzers, therefore, is an extremely challenging task. This makes them extremely complicated pieces of software with a high likelihood of having bugs themselves (challenge 1) or tradeoffs not suitable for every piece of code under every usage

scenario (challenge 2). In this dissertation, we focus our attention on these two challenges that can limit the ability of static program analyzers to be incorporated into mainstream software development workflows.

Challenge 1 is *unintentional unsoundness*. Program analyzers are complicated tools, implementing sophisticated algorithms and performance heuristics. This makes them highly susceptible to undetected unintentional soundness issues. These issues in program analyzers can cause false negatives and have disastrous consequences e.g., when analyzing software used for electronic voting, financial systems, transportation, or secure communications. For example, Astrée was used to verify the absence of runtime errors in flight control software for Airbus A340 and A380 [45, 80]. Unintentional soundness bugs in Astree's implementation could result in the verification of buggy flight control software. Bugs in program analyzers may also hamper developer productivity and trust. Rigorous testing of program analyzers before they are used to verify properties of real-world production code is, therefore, an important but still largely unexplored area of research in the testing and formal methods communities.

Challenge 2 is finding the right balance between soundness, precision, and performance. In an ideal world, static analysis should be as precise as possible while maintaining soundness and being sufficiently fast. However, to overcome undecidability issues, static analysis tools have to employ a variety of techniques to be practical, for example, compromising on the soundness of the analysis or approximating code behavior. If compromising soundness is not an option, we have to rely on code approximation techniques to improve performance. Typically, the closer the approximation is to the actual code behavior, the less efficient and the more precise the analysis is, that is, the fewer false positives it reports. For less tight approximations, the analysis tends to become more efficient but less precise. For very large codebases, approximation might still not be a viable solution and in order to improve performance, many analysis techniques therefore also often trade soundness in order to improve performance, making them efficient and effective bug and vulnerability detection tools. Finding the right balance between the precision, soundness, and performance of the analysis results for a particular code base under certain resource constraints and different usage scenarios to get the most out of a program analyzer can be a challenging task for software developers, most of whom lack an advanced understanding of these analyzers.

1.1. Detecting Unintentional Unsoundness Bugs in Program Analyzers

Program analyzers are complex tools that implement complex algorithms and data structures. Typically, analyzers are implemented as a pipeline of interacting components e.g., lexical analysis of the source code, parsing the source code and generating an intermediate representation, applying optimizations and transformations to the intermediate representation and passing it along to often several self-contained core analysis components which are already very complex by themselves. This means that program analyzers are all the more likely to contain correctness issues themselves. The most dangerous kind of correctness issue in an analyzer is a *critical bug*, which we define as a bug leading to a wrong analysis result, e.g., returning 'correct' for an 'incorrect' program. Critical bugs in program analyzers can have disastrous consequences for the security and safety of our modern digital infrastructure. For example, Amazon Web Services (AWS) developed and uses Zelkova [22], a static Access Control Policy (ACP) analyzer. Users specify ACPs to define access permissions to IT resources in the cloud. An ACP expressively specifies what resources can be accessed, by whom, and under what conditions. A policy misconfiguration can result in unauthorized access to critical resources, posing a serious security risk to any organization. Zelkova works by encoding ACPs into SMT formulas and then uses off-the-shelf SMT solvers Z3 [84] and CVC4 [32] (we use program-analyzer components, like SMT solvers, as analyzers in this dissertation) to verify their properties. Zelkova is extensively used both within AWS and by its customers to verify ACPs across a wide range of AWS services and is invoked many millions of times daily. A critical bug in Z3 (returning 'SAT' for an 'UNSAT' formula) can result in Zelkova verifying a misconfigured ACP. Therefore, it is vitally important to guarantee the reliability of program analyzers.

The difficulties in correctly implementing program analyzers lead to several challenges in proving the absence of critical bugs in their implementation. In the past, efforts to fully verify large critical software systems have proven to be quite expensive. For example, CompCert [156], a verified high-assurance compiler for a subset of C language, which is about 15K lines of code, required 6 person-years to write 100K lines of specifications. Different components in modern program analysis toolchains, cumulatively, are already surpassing a million lines of code [30, 84, 120]. Fully verifying a program analysis toolchain is, therefore, practically infeasible. Automated test-generation techniques, on the other hand, can be effectively used to find hard-to-detect critical bugs in modern program analyzers without providing absolute correctness guarantees. Over the years, many techniques have been proposed to facilitate the automated testing of complex program analyzers. However, each of these techniques comes with its own limitations.

Random fuzzing [187] is a black box software testing technique that works by

generating massive amounts of normal and abnormal inputs and then detecting exceptions by feeding the generated inputs and monitoring the execution states. Compared to other techniques, fuzzing is easier to quickly deploy and requires little to no knowledge of the target application. However, the semantic richness of the input that analyzers have to deal with puts random fuzzing at a major disadvantage. Generating input programs with non-trivial behavior with random fuzzing is very difficult. However, to reach deep into the analyzer's pipeline, such non-trivial programs are often required. For example, an input program must first pass all the syntactic, semantic, and type checks before it can reach the transformation and analysis phase. Another limitation is that this approach can only detect *crash bugs* where the analyzer does not return any result. Such bugs are far less serious than critical bugs, since they can, for instance, result in verifying incorrect safety-critical code. Fortunately, the semantics of the source language taken as input by program analyzers is usually specified, either informally in a language specification or formally, e.g., SMT-LIB. *Grammar based testing* [73,176,243] techniques use these language specifications to generate syntactically and semantically valid inputs, for example, Csmith [266], which generates random C programs to stress-test compilers. Csmith has also been applied to find bugs in static analyzers, for example, Farma-C [82].

Specification-based testing (e.g., [56, 185]) involves specifying a precise and detailed description of an analyzer's functionality and testing it against the provided specification. Similar to verification, fully specifying large software systems is highly non-trivial and can require a prohibitive amount of time and effort.

Differential testing of program analyzers [141, 148, 207, 262] involves running multiple analyzers that are expected to produce the same output on a single input program. If disagreement exists on the analysis results, then a bug in at least one of the analyzers has been detected. Differential testing-based approaches have proven to be very effective in detecting bugs in program analyzers. This approach has the advantage that no oracle for test results is needed. The key idea this technique exploit is that multiple deterministic implementations of the same specification must all produce the same result for the same input. If two implementations produce different outputs, one of them must be faulty. In case of three or more implementations, majority voting can be used to determine which implementations are wrong. However, differential testing is not well suited for emerging domains where multiple implementations of a new analysis technique do not exist yet or for domains without standardization where no two analyzers accept the same input. This is the case with Datalog engines [112]. For Datalog, there is no unified syntax and each engine supports a different dialect of the language.

Metamorphic testing [55, 168, 169, 263, 271] works by producing test cases after applying some kind of metamorphosis to the existing test cases, using known properties of the program analyzer under test to infer a relationship between the output of the original and the transformed test case. The relationship between the outputs is characterized with metamorphic relations and these relations are

used to circumvent the oracle problem. Metamorphic testing lies in the middle of the spectrum between specification-based and differential testing. It relies on the idea that it is easier to reason about the relations between the results of an analyzer than to fully formalize its input-output behavior. For example, consider an SMT solver S and two logical formulas φ and ϕ. Then running the solver on φ i.e., $S(\varphi)$ should yield the same satisfiability result as $S(\varphi \vee \phi)$ if ϕ is known to be unsatisfiable.

1.1.1. Detecting Soundness Bugs in SMT Solvers

SMT solvers are extensively used in formal methods, most notably in software verification (e.g., Boogie [30] and Dafny [155]), systematic test case generation (e.g., KLEE [58] and Sage [107]) and program synthesis (e.g., Alive [165]). Solvers, such as CVC4 [32] and Z3 [84], evaluate the satisfiability of SMT instances and are extremely complex in their implementation. Due to their high degree of complexity, it is all the more likely that SMT solvers contain correctness issues, and due to their wide applicability in software reliability, these issues may be catastrophic.

A bug in an SMT solver is called a *refutational soundness bug* if the solver returns `unsat` (i.e., unsatisfiable) for a satisfiable (`sat`) SMT instance. A bug is called a *solution soundness bug* if the solver returns `sat` (i.e., satisfiable) for unsatisfiable instances. We call refutational soundness bugs critical for two main reasons. First, such bugs may cause unsoundness in program analyzers that rely on SMT solvers. Second, it is much harder to safeguard against these bugs. Specifically, consider that, when an instance is found to be `sat`, the solver typically provides a model, that is, an assignment to all free variables in the instance such that it is satisfiable. Therefore, such bugs could be detected by simply evaluating the instance under the model generated by the solver (assuming that the model is correct). If this evaluation returns false, then there is a solution soundness bug.

In the dissertation, we present a technique for detecting critical bugs in any SMT solver [169]. Our technique does not require a grammar to synthesize instances from scratch. Instead, it takes inspiration from state-of-the-art mutational fuzzers (e.g., AFL [11]) and generates new SMT instances by mutating existing ones, called seeds. The key novelty is that our approach generates satisfiable instances from any given seed. As a result, our technique detects a critical bug whenever an SMT solver returns `unsat` for one of our generated instances. We implement our technique in a tool called STORM[1], which has the additional ability to effectively minimize the size of bug-revealing instances to facilitate debugging. The tool is open source and publicly available on Github.

[1] `https://github.com/Practical-Formal-Methods/storm`

1.1.2. Detecting Soundness Bugs in Datalog Engines

Datalog [112] is a declarative, logic-based query language that is syntactically a subset of Prolog. Datalog is expressive, yet concise, and as a result, it is used as a domain-specific language in several application domains, such as natural-language processing [194], bio-informatics [145, 232], big-data analytics [118, 134], networking [164], program analysis [48, 83, 110, 195, 261], robotics [213], generic graph databases [239], and security [49, 50, 111, 251].

Datalog queries are evaluated by Datalog engines e.g., Soufflé [136], bddbddb [260], DDlog [228], μZ [125]. These engines are complex, especially since they typically employ advanced query transformation, optimization, and compilation techniques to improve their performance and scalability. As a result of this complexity, Datalog engines are prone to bugs. Such bugs, called *query bugs*, may compromise the soundness of upstream program analyzers, having potentially detrimental consequences in safety-critical settings. As an example, imagine a static analyzer that uses Datalog to implement a may-alias (or must-alias) analysis. A query bug that results in computing fewer (or more) aliases could lead to missing critical bugs in the analyzed software.

Finding such bugs, however, is impossible without an oracle, that is, a specification of the expected results. Differential testing [180] could overcome the oracle problem by running multiple Datalog engines on the same input programs and looking for disagreement in the results. In our context, this would be extremely difficult since there is no unified syntax for Datalog, and each engine understands a (very) different dialect; for instance, Soufflé [136] enables large-scale, logic-oriented programming, whereas Formulog [37] provides support for constructing and reasoning about SMT formulas.

In this dissertation, we present the *first* two automatic test-case generation approaches for detecting query bugs in Datalog engines. The proposed techniques use metamorphic testing [66] to circumvent the lack of an oracle. In our context, metamorphic testing would transform a Datalog program such that the result of the transformed program has an a-priori known relationship to the result of the original program. For example, the new result could be equivalent to the original result, it could be contained in the original result, or it could contain the original result.

The first approach is based on concepts in database theory, and in particular, formal properties of *conjunctive queries*. Despite their simplicity, conjunctive queries constitute an important class of database queries due to their theoretical properties. Specifically, while many fundamental problems in query optimization and minimization are computationally hard—or even undecidable—for general forms of queries, they are feasible for conjunctive queries. An example of such a problem is *query containment*. The key insight behind our approach is to leverage properties of conjunctive queries to develop metamorphic transformations for full-blown Datalog programs.

In the second approach, we propose a general metamorphic testing technique that leverages rich precedence information stored in the *precedence graph* of a Datalog program. Every Datalog program has an associated precedence graph that captures dependencies between relations in the program. Precedence graphs are used, for example, to determine if the Datalog program is stratifiable. Stratification allows us to provide well-defined semantics to evaluate a Datalog program and hence most Datalog engines only support stratifiable Datalog programs.

We implemented both approaches in two different tools and were able to detect 29 query bugs in mature Datalog implementations. The first approach is implemented in a tool called queryFuzz[2] and the second approach is implemented in a tool called DLSmith[3]. Both tools are open-source and publicly available on Github.

[2] https://github.com/Practical-Formal-Methods/queryFuzz

[3] https://github.com/Practical-Formal-Methods/dlsmith

1.2. Balancing Soundness, Precision, and Performance in Static Analysis

Static analysis is the process of checking interesting program properties without actually running the program, in order to improve code quality or detect errors. In contrast to dynamic analysis, where we reason about a specific execution or a set of executions, in sound static analysis, we reason about all executions of the program. This allows us to reason about the universal properties of a program that hold for all inputs. Static analysis is used to detect errors ranging from a simple crash bug to severe security vulnerabilities. Following is a non-exhaustive list of errors that can be detected using static analysis:

- *Null pointer dereference*, i.e., a pointer with NULL value is used as though it contains a valid memory address.
- *Buffer overflow*, i.e., the amount of data in a memory location exceeds its storage capacity.
- *Array out of bounds*, i.e., accessing an array index that is negative, greater than, or equal to the size of the array.
- *Memory leaks*, i.e., program fails to return memory which is no longer needed.
- *Invalid arithmetic operation*, e.g., division by zero.
- *Non-terminating loop*, i.e., the exit condition of a loop will never evaluate to false.
- *SQL injection*, i.e., an insertion of a malicious executable SQL query is possible via the input data from an untrusted client.

Static analysis, however, has fundamental limitations. Consider for example a *live variable analysis*. A variable x is *live* at a statement s iff on some execution of the program, x is used after s is executed without being redefined. For the program in Fig. 1.1, it might seem obvious that the variable x is live at line 2, but suppose that function `f()` never returns. In that case, the value of x is not needed. In other words, x is live if `f()` halts. Since the halting problem is undecidable, so is the live variable problem, at least if we want precise results. In fact, Rice's theorem [219] shows that all interesting or non-trivial questions about the behavior of a program are undecidable.

To be practical, program analyzers have to employ a number of techniques, for example, compromising on the soundness or completeness of the analysis. That means that the analyzer might miss a bug (false negative) or report the correct code as having a bug (false positive). Some analyzers approximate code behavior. Performance is another reason for this approximation. As mentioned before,

```
1 read(y);
2 x = y;
3 f();
4 return x;
```

Figure 1.1: A simple program demonstrating the undecidability of live variable analysis.

software developers expect program analyzers to return a minimum number of false positives while scaling to millions of lines of code and returning the results in minutes. Program analyzers, therefore, use sophisticated heuristics that take into account, for example, the size of the codebase, the allowed resources (e.g., time and memory), or the tolerance to false positives, etc. But these heuristics can and sometimes do fail, hampering developer's productivity and damaging trust in the tool.

As the size, complexity, and importance of software systems grow, so does the need for practical and easy-to-use tools that can provide us with a mechanism to check software correctness and increase the confidence of both the developers and the users. Large software companies are making significant efforts in integrating automatic static analysis tools in their software development life cycle. Infer is a static analysis tool developed at Meta that reports bugs ranging from memory safety, to concurrency, to security, and many more specialized errors suggested by Meta developers [87]. Infer is integrated into the continuous integration system at Meta and participates as a bot during the code review process.

Static analysis tools however are not trivial to integrate into any usage scenario with different program sizes, resource constraints, and SLAs. Most of the time, these tools also don't scale to large industrial code bases containing millions of lines of code. This makes it extremely challenging to get the most out of these analyzers and integrate them into everyday development activities, especially for average software development teams with little to no knowledge or understanding of advanced static analysis techniques.

1.2.1. Maximizing Precision while Adhering to Time Constraints

Recent years have seen tremendous progress in the development and industrial adoption of static analyzers. Notable successes include Meta's Infer [59, 60] and AbsInt's Astrée [45]. Many analyzers, such as these, are based on *abstract interpretation* [76], a technique that abstracts the concrete program semantics and reasons about its abstraction. Most abstract interpreters offer a wide range of abstract domains that impact the precision and performance of the analysis.

In addition to the domains, abstract interpreters usually provide a large number

of options, for instance, whether backward analysis should be enabled or how quickly a fixpoint should be reached. In fact, the sheer number of option combinations is bound to overwhelm users, especially non-expert ones. To make matters worse, the best option combinations may vary significantly depending on the code under analysis and the resources, such as time or memory, that users are willing to spend.

In light of this, we suspect that most users resort to using the default options that the analysis designer pre-selected for them. However, these are definitely not suitable for all code. Moreover, they do not adjust to different stages of software development, e.g., running the analysis in the editor should be much faster than running it in a continuous integration (CI) pipeline, which in turn should be much faster than running it prior to a major release. As a result, the widespread adoption of abstract interpreters is severely hindered, which is unfortunate since they constitute an important class of practical analyzers.

To address this issue, in this dissertation, we present the first technique that automatically tailors a generic abstract interpreter to a particular piece of code and specific resource constraints. The key idea behind our technique is to phrase the problem of customizing the abstract-interpretation configuration to a given usage scenario as an optimization problem. Specifically, different configurations are compared using a cost function that penalizes those that prove fewer properties or require more resources. The cost function can guide the configuration search of a wide range of existing optimization algorithms.

We implement our technique in a framework called TAILOR, which configures a generic abstract interpreter for a given usage scenario using a given optimization algorithm. This enables the abstract interpreter to prove as many properties as possible within the resource limit without requiring any expertise on behalf of the user. TAILOR is open-source and publicly available on Github[4].

1.2.2. Splitting Analysis Inputs to Balance Soundness, Precision, and Performance

With the increasing popularity of DevSecOps development practices, Static Application Security Testing (SAST) is increasingly becoming the responsibility of developers rather than security experts. Many development teams do not have the expertise to configure and maintain their own static analysis infrastructure and prefer SAST platforms that offer a variety of static analyses on demand. This is creating a growing demand for easy-to-use cloud-based Static Application Security Testing (SAST) platforms such as the Software Assurance Marketplace (SWAMP) [151] or ShipShape [229], that provide a simple interface through which developers can submit their code and receive recommendations on how to improve their code.

[4] https://github.com/Practical-Formal-Methods/tailor.

Internally, such platforms may employ a variety of static analysis tools, such as [10, 13, 59, 60, 71, 214, 220] and are typically run as a cloud-based service, in which individual analysis tools are containerized and instantiated on-demand on cloud-based machines. Developers expect such platforms to handle inputs (codebases) of arbitrary size and complexity, and still deliver results within a certain time window. This is especially true for customers that integrate SAST platforms in their continuous integration and deployment (CI/CD) pipelines.

To maintain a predictable response time, SAST platforms face the challenge that they need to scale to different input sizes, and that, every time they add a new analysis tool, they have to ensure that the new tool does not slow down the response time for existing customers. Vertical scaling by adding more memory or faster machines is not a cost-effective solution to the risk of running out of time or space when analyzing complex inputs. Provisioning machines large enough to handle the most complex analysis inputs would make the service unnecessarily expensive for customers that analyze smaller and simpler codebases.

Much research has been conducted on various optimization strategies to improve the scalability of specific analysis engines, such as summarization of method calls [20, 217, 226], caching and reuse of partial results from prior analyses [5, 19], and incremental analysis [72, 253]. However, when operating a SAST platform, modifying the individual tools may not be an option because the tools might be proprietary or maintaining forks with custom modifications may be too costly.

In this dissertation, we propose a technique to horizontally scale analysis tools in a static analysis platform by splitting the input codebase into partitions such that the amount of code in each partition is below a provided bound. The different partitions can then be analyzed on parallel cloud instances of a given analysis tool. Depending on the complexity of the static analyzer, the partition size can be adjusted to curtail the overall response time. Such a horizontal scaling strategy can be configured per analysis tool, but without modifying the tool itself. More complex tools can be configured to handle smaller pieces of code than lightweight tools to ensure that the overall latency of the platform does not change when a new complex tool gets added.

We evaluate how this splitting process affects the precision and soundness of different static analysis tools and how the computational cost of analyzing partitions in parallel relates to the cost of analyzing the entire input program. The experimental results show that simple splitting strategies can effectively reduce the running time and memory usage per partition without significantly affecting the findings produced by the tool. The technique was developed in collaboration with Amazon Web Services and is now being used in production in their CodeGuru service[5].

[5] https://aws.amazon.com/codeguru/

1.3. Outline and Publication Details

This dissertation is based on publications that I authored with my collaborators over the course of my Ph.D. studies. The following list presents an outline of the dissertation and the publication upon which each chapter is based on:

1. Chapter 2 presents STORM, a novel blackbox mutational fuzzing technique for detecting critical soundness bugs in SMT solvers. In three months of testing, STORM detected 29 critical soundness bugs in three mature SMT solvers and 15 different logics. This work was published in *ESEC/FSE'20* under the title *Detecting Critical bugs in SMT Solvers Using Black Box Mutational Fuzzing* [169].

2. Chapter 3 presents queryFuzz, the first metamorphic-testing approach for detecting critical bugs in Datalog engines. queryFuzz detected 13 previously unknown critical bugs in three popular Datalog engines. This work was published in *ESEC/FSE'21* under the title *Metamorphic Testing of Datalog Engines* [168].

3. Chapter 4 presents DLSmith, a powerful and general metamorphic testing framework for Datalog engines. DLSmith overcomes the limitations in queryFuzz and is the most comprehensive and effective metamorphic testing approach for detecting critical bugs in Datalog engines to date. DLSmith detected 16 previously unknown soundness bugs in four Datalog engines. Accepted in *ISSTA'23* under the title *Dependency-Aware Metamorphic Testing of Datalog Engines* [171].

4. Chapter 5 presents TAILOR, a tool that automatically tailors a generic abstract interpreter to the code under analysis and any given resource constraints. This work was published in *CAV'21* under the title *Automatically Tailoring Abstract Interpretation to Custom Usage Scenarios* [170].

5. Chapter 6 presents an approach to scale static analysis tools in cloud-based static analysis platforms. The approach is currently being used at Amazon Web Services in their CodeGuru service. This work was published in *ESEC/FSE'22* under the title *Input Splitting for Cloud-Based Static Application Security Testing Platforms* [69].

Part I

Detecting Unintentional Unsoundness

Chapter 2

Detecting Critical Soundness Bugs in SMT Solvers

Formal methods use SMT solvers extensively for deciding formula satisfiability, for instance, in software verification, systematic test generation, and program synthesis. However, due to their complex implementations, solvers may contain critical bugs that lead to unsound results. Given the wide applicability of solvers in software reliability, relying on such unsound results may have detrimental consequences. In this chapter, we present STORM, a novel blackbox mutational fuzzing technique for detecting critical bugs in SMT solvers. We ran our fuzzer on seven mature solvers and found 29 previously unknown critical bugs. STORM was also used in testing new features of popular solvers before deployment.

2.1. Introduction

The *Satisfiability Modulo Theories* (SMT) problem [33] is the decision problem of determining whether logical formulas are satisfiable with respect to a variety of background theories. More specifically, an SMT *formula* generalizes a Boolean SAT formula by supplementing Boolean variables with predicates from a set of theories. As an example, a predicate could express a linear inequality over real variables, in which case its satisfiability is determined with the theory of linear real arithmetic. Other theories include bitvectors, arrays, and integers [101], to name a few.

SMT solvers, such as CVC4 [32] and Z3 [84], are complex tools for evaluating the satisfiability of SMT instances. A typical SMT *instance* contains assertions of SMT formulas and a satisfiability check (see Figs. 2.2 and 2.3 for examples). SMT solvers are extensively used in formal methods, most notably in software verification (e.g., Boogie [30] and Dafny [155]), systematic test case generation (e.g., KLEE [58] and Sage [107]), and program synthesis (e.g., Alive [165]). Due

to their high degree of complexity, it is all the more likely that SMT solvers contain correctness issues, and due to their wide applicability in software reliability, these issues may be detrimental.

Tab. 2.1 shows classes of bugs that may occur in SMT solvers. We restrict the classification to bugs that manifest themselves as an incorrect solver result. For bugs in class A, the solver is *unsound* and returns `unsat` (i.e., unsatisfiable) for instances that are satisfiable. These bugs are known as *refutational soundness bugs* in the SMT community. Class B refers to bugs where the solver returns `sat` (i.e., satisfiable) for unsatisfiable instances. These bugs are known as *solution soundness bugs*. A solver is *incomplete* when it returns `unknown` for an instance that lies in a decidable theory fragment. We categorize such bugs in class C. Finally, bugs in class D indicate crashes, where the solver does not return any result, for example, in case of an assertion failure or a segmentation fault.

We call bugs in class A *critical* for two main reasons. First, such bugs may cause unsoundness in program analyzers that rely on SMT solvers. As an example, consider a software verifier (e.g., Dafny [155]) or a test case generator (e.g., KLEE [58]) that checks reachability of an error location by querying an SMT solver. If the solver unsoundly proves that the error is unreachable (e.g., returns `unsat` for the path condition to the error), then the verifier will verify incorrect code and the testing tool will not generate inputs that exercise the error.

Second, it is much harder to safeguard against bugs in class A than bugs in other classes. Specifically, consider that, when an instance is found to be `sat`, the solver typically provides a *model*, that is, an assignment to all free variables in the instance such that it is satisfiable. Therefore, bugs in class B could be detected by simply evaluating the instance under the model generated by the solver (assuming that the model is correct). If this evaluation returns false, then there is a B bug. Bugs in class C are detected whenever the solver returns `unknown` for an instance that lies in a decidable theory fragment, and bugs in class D are detected when the solver crashes.

In this chapter, we present a general blackbox fuzzing technique for detecting critical bugs in any SMT solver. Our technique does not require a grammar to synthesize instances from scratch. Instead, it takes inspiration from state-of-the-art mutational fuzzers (e.g., AFL [11]) and generates new SMT instances by mutating existing ones, called *seeds*. The key novelty is that our approach generates satisfiable instances from any given seed. As a result, our fuzzer detects a critical bug whenever an SMT solver returns `unsat` for one of our generated instances. We implement our technique in an open-source tool called STORM, which has the additional ability to effectively minimize the size of bug-revealing instances to facilitate debugging.

Contributions. This chapter makes the following contributions:

1. We present a novel blackbox mutational fuzzing technique for detecting critical bugs in SMT solvers.

Table 2.1: Classes of bugs in SMT solvers. GT stands for ground truth and SR for solver result.

GT \ SR	`sat`	`unsat`	`unknown`	Crash
`sat`		A	C	D
`unsat`	B		C	D

2. We implement our technique in an open-source fuzzer[1] that was used for testing new features of solvers before deployment.

3. We evaluate the effectiveness of our fuzzer on seven mature solvers and 43 logics. Over a three months testing phase, we found 29 previously unknown critical bugs in three solvers (or nine solver variants) and 15 different logics.

Outline. The rest of this chapter is organized as follows. The next section gives an overview of our approach. Sect. 2.3 explains the technical details, and Sect. 2.4 describes our implementation. We present our experimental evaluation in Sect. 2.5, discuss related work in Sect. 2.7, and give concluding remarks in Sect. 2.8.

2.2. Overview

To give an overview of our fuzzing technique for SMT solvers, we first describe a few interesting examples of STORM in action and then explain what happens under the hood on a high level.

In action. One of the critical bugs[2] found by STORM was in Z3's `QF_LIA` logic, which stands for quantifier-free linear integer arithmetic. We opened a GitHub issue to report this bug, which resulted in an eight-comment discussion between two Z3 developers on how to resolve it. Note that eight comments (or in fact any discussion) on how to fix a bug is typically uncommon. From the GitHub issues we have seen, developers simply acknowledge an issue or additionally ask for a minimized SMT instance. The issue was closed but re-opened a day later with more comments on what still needs to be fixed. The issue was closed for the last time three days after that. Based on our understanding and explanations by the developers, this bug was triggered by applying Gomory's cut on an input that did not satisfy a fundamental assumption of the cut. STORM was able to generate an instance that violated this assumption and led to misapplying Gomory's cut. The fix in Z3 included changing the implementation of the cut.

[1] https://github.com/Practical-Formal-Methods/storm
[2] https://github.com/Z3Prover/z3/issues/2871

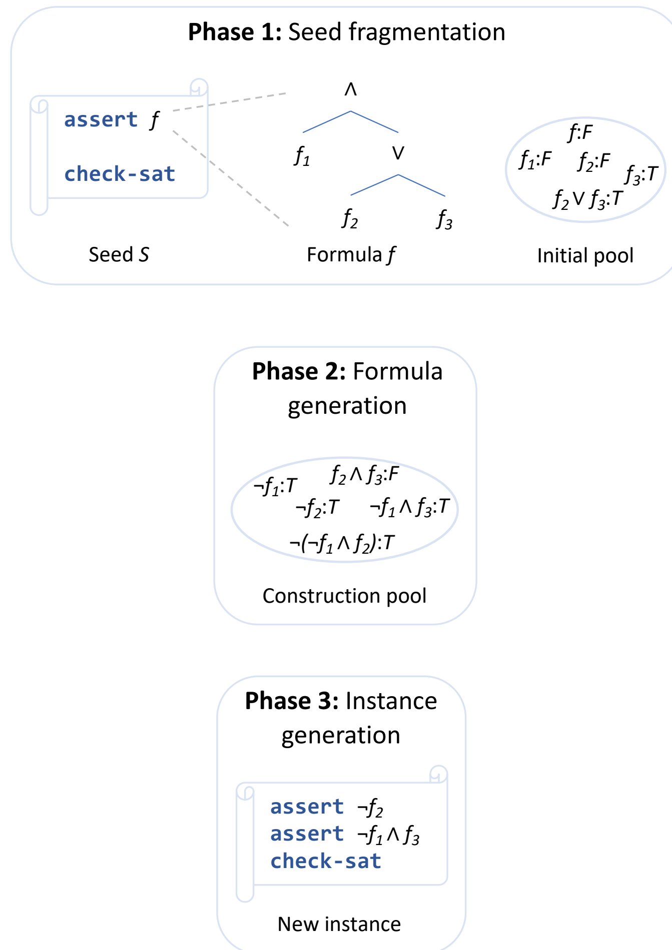


Figure 2.1: Overview of the three STORM phases.

STORM detected another critical bug[3] in Z3's Z3str3 string solver [40]. According to a developer of Z3str3, the bug existed for a long time before STORM found it. During this time, it remained undetected even though Z3str3 was being tested with fuzzers exclusively targeting string solvers [46, 55]. A simplified version of the SMT instance that revealed the bug is shown in Fig. 2.2b. (We will discuss it in detail later in this section.)

A third critical bug[4] was found in Z3's tactic for applying dominator simplification rules. The instance that was generated by STORM and revealed the

[3] https://github.com/Z3Prover/z3/issues/2994
[4] https://github.com/Z3Prover/z3/issues/3052

```
1 (declare-const S String)
2 (assert (str.in.re S (re.++ re.allchar (re.++
3   (str.to.re "7;") (re.++ re.allchar
4     (str.to.re "aa"))))))
5 (assert (not (str.in.re S (re.union re.allchar
6   (str.to.re "X'jafa")))))
7 (check-sat)
```

(a) Original instance.

```
1 (declare-const S String)
2 (assert
3 (let ((a (str.in.re S (re.++ re.allchar (re.++
4   (str.to.re "7;") (re.++ re.allchar
5     (str.to.re "aa"))))))
6 (let ((b (not (str.in.re S (re.union re.allchar
7   (str.to.re "X'jafa"))))))
8 (let ((c (and (not b) (not a))))
9 (not c))))))
10 (check-sat)
```

(b) Bug revealing instance.

Figure 2.2: Original seed instance from SMT-COMP 2019 on the top, and simplified instance revealing critical bug in Z3's Z3str3 string solver on the bottom.

bug spanned 194 lines. The minimization component of STORM reduced this instance to 15 lines. A simplified version of the instance is shown in Fig. 2.3b. (We discuss it later in this section.) A developer of the buggy tactic asked us which application generated this instance, thinking that it was a tool he developed during his PhD thesis. When we mentioned that it was STORM, he replied "*What? Your random generator could have done my PhD thesis?? &@#%, you should have told me sooner :)*". This demonstrates STORM's ability to generate realistic SMT instances that can be difficult to distinguish from instances produced by client applications of SMT solvers.

In Sect. 2.5, we describe in more detail our experience of using STORM to test both mature solver implementations as well as new features before their deployment.

Under the hood. We now give a high-level overview of our fuzzing technique, which operates in three phases. Fig. 2.1 depicts each of these phases.

The first phase, *seed fragmentation*, takes as input a seed SMT instance S. For instance, imagine an instance with multiple assertions. Each assertion contains a logical formula, such as f in the figure, potentially composed of Boolean sub-

```
1 (declare-fun A () Bool)
2 (declare-fun B () Bool)
3
4 (assert (not B))
5 (assert (not (and (not A) B)))
6 (assert A)
7
8 (check-sat-using dom-simplify)
```

(a) Original instance.

```
1 (declare-fun A () Bool)
2 (declare-fun B () Bool)
3
4 (assert (and (not B) A))
5
6 (check-sat-using dom-simplify)
```

(b) Bug revealing instance.

Figure 2.3: Simplified instance revealing critical bug in Z3's `dom-simplify` tactic on the top, and logically equivalent instance not revealing the bug on the bottom.

formulas (i.e., predicates), such as $f_2 \vee f_3$, f_1, f_2, and f_3 in the figure. Initially, STORM generates a random assignment of all free variables in the formulas in S. Then, STORM recursively fragments the formulas in S into all their possible sub-formulas. For example, f is broken down into f_1 and $f_2 \vee f_3$, each of these is in turn broken down into its Boolean sub-formulas, and so on. The valuation (i.e., truth value) of each (sub-)formula, T or F, is computed based on the random assignment. All formulas together with their valuations are inserted in an initial pool as shown in the figure.

The second phase, *formula generation*, uses the formulas in the initial pool to build new formulas. The valuation of each new formula is computed based on the valuations of its constituent initial formulas. All new formulas with their valuations are inserted in a construction pool as shown in the figure. For instance, initial formulas f_2 and f_3 are used to construct a new formula $f_2 \wedge f_3$.

The third phase, *instance generation*, uses formulas from both pools to generate new SMT instances. The reason for having the two pools is to be able to control the frequency with which initial and constructed formulas appear in the new instances. Instances generated during this phase have a different Boolean structure than the seeds. However, their basic building blocks, that is, the initial formulas that could not be fragmented further, remain unchanged. This is what allows STORM to generate realistic instances. In addition, all new instances are satisfiable by

construction.

Therefore, a critical bug is detected whenever an SMT solver returns unsat for a STORM-generated instance. In such a case, STORM uses *instance minimization* to minimize the size of the instance revealing the bug.

Examples. Fig. 2.2a shows a seed instance from the international SMT competition SMT-COMP 2019 [6]. Starting from this seed, STORM generated the (simplified) instance shown in Fig. 2.2b, which revealed the critical bug in Z3str3 described above. Z3str3 derives length constraints from regular-expression membership predicates. The bug that STORM exposed here is that such a length constraint, which is implied by membership in a regular expression, was not asserted by the string solver.

It is easy to see that the first asserted formula in Fig. 2.2a corresponds to variable a in Fig. 2.2b, while the second asserted formula in Fig. 2.2a corresponds to variable b in Fig. 2.2b. Therefore, the seed essentially checks for satisfiability of $a \wedge b$. In Fig. 2.2b, c is equivalent to $\neg a \wedge \neg b$, and the instance checks for satisfiability of $\neg c$, thus, of $a \vee b$. This shows that even small mutations to the Boolean structure of a formula can be effective in revealing critical issues in solvers. In fact, such mutations can result in triggering different parts of a solver's implementation, e.g., different simplifications, heuristics, or optimizations.

This is also evidenced by the example in Fig. 2.3. The instance in Fig. 2.3a reveals the critical bug in Z3's dom-simplify tactic described earlier. It essentially checks the satisfiability of $\neg B \wedge \neg(\neg A \wedge B) \wedge A$, which is logically equivalent to $\neg B \wedge A$. Observe, however, that the logically equivalent formula, shown in Fig. 2.3b, does not trigger the bug.

Consequently, the benefit of fuzzing the Boolean structure of seed instances is two-fold. First, it is effective in detecting critical issues in solvers. Such issues are by definition far more serious and complex than other types of bugs, such as crashes, since they can, for instance, result in verifying incorrect safety-critical code. Second, fuzzing only the Boolean structure of seeds helps generate realistic SMT instances. This is confirmed by the above comments on the tactic bug from the Z3 developer who thought that the STORM instance was generated by his own PhD tool. This was also confirmed by other solver developers with whom we interacted.

2.3. Our Approach

We now describe our fuzzing technique and how it solves two key challenges in detecting critical bugs in SMT solvers: (1) how to generate non-trivial SMT instances, and (2) how to determine if a critical bug is exposed. The latter demonstrates how STORM addresses the oracle problem [31] in the context of soundness testing for solvers. Finally, we describe how we minimize bug-revealing instances to reduce their size. This step significantly facilitates debugging for solver developers.

Algorithm 1: Core fuzzing procedure in STORM.

```
1  procedure PopulateInitialPool(S, D_max)
2    A ← GetAsserts(S)
3    M ← RandAssignment(A)
4    P ← EmptyPool()
5    for i = pred ∈ S to
6      if ¬ExceedsDepth(pred, D_max) then
7        v ← IsTrue(M, pred)
8        P ← Add(P, pred, v)
9    return P
10
11 procedure Fuzz(S, NC, NM, D_max, A_max)
12   // Phase 1: Seed fragmentation
13   P_init ← PopulateInitialPool(S, D_max)
14
15   // Phase 2: Formula generation
16   P_constr ← EmptyPool()
17   while Size(P_constr) < NC do
18     f_1, v_1 ← RandFormula(P_init, P_constr)
19     op ← RandOp()
20     if op = AND then
21       f_2, v_2 ← RandFormula(P_init, P_constr)
22       f ← AND(f_1, f_2)
23       v ← v_1 ∧ v_2
24     else
25       f ← NOT(f_1)
26       v ← ¬v_1
27     if ¬ExceedsDepth(f, D_max) then
28       P_constr ← Add(P_constr, f, v)
29
30   // Phase 3: Instance generation
31   B ← EmptyList()
32   m ← 0
33   while m < NM do
34     // Number of generated assertions
35     ac ← (RandInt() % A_max) + 1
36     A ← EmptyList()
37     while 0 < ac do
38       f, v ← RandFormula(P_init, P_constr)
39       if ¬v then
40         // Negation of f to guarantee assertion satisfiability
41         f ← NOT(f)
42       A ← Append(A, f)
43       ac ← ac − 1
44     // Invocation of SMT solver under test
45     r ← CheckSAT(A)
46     // Test oracle
47     if r = UNSAT then
48       B ← Append(B, A)
49     m ← m + 1
50   return B
```

2.3.1. Fuzzing Technique

Given an SMT instance as seed input, our fuzzing approach proceeds in three main phases: (1) seed fragmentation, (2) formula generation, and (3) instance generation. Seed fragmentation extracts sub-formulas from the seed. These will be used as building blocks for generating new formulas in the second phase. Lastly, instance generation creates new, satisfiable SMT instances based on the generated formulas, invokes the SMT solver under test on each of these instances, and uses the solver result as part of the test oracle to detect critical bugs.

Alg. 1 describes these three phases in detail. Function FUZZ takes the initial seed *S* and several additional parameters that bound the fuzzing process (explained below). As a first step, the function populates an initial pool P_{init} of formulas (line 13) with formula fragments of the seed *S*.

To this purpose, function POPULATEINITIALPOOL extracts all assertions in the seed and generates a random assignment *M*, i.e., an assignment of values to free variables. In our implementation, we use a separate SMT solver (i.e., different from the one under test) to generate a model for the assertions (or their negation if the assertions are unsatisfiable). Next, we iterate over all predicates (i.e., tree-shaped Boolean sub-formulas as in Fig. 2.1) in the seed. We use assignment *M* to evaluate those predicates for which the tree depth does not exceed a bound D_{max}. This valuation *v* is crucial for subsequent phases of the fuzzing process, and we add both the formula *pred* and *v* to the initial pool, which is essentially a map from formulas to valuations. Note that, by fragmenting the seed, the initial pool already contains a large number of non-trivial formulas that would be difficult to generate from scratch (e.g., with a grammar-based fuzzer).

In the second phase, we populate the construction pool P_{constr} by adding *NC* new formulas of maximum depth D_{max}. These formulas are generated randomly by selecting one of two Boolean operators, logical *AND* (lines 21–23) and *NOT* (lines 25–26). Note that this set of operators is functionally complete, thus allowing us to generate any Boolean formula. We construct a new formula *f* by conjoining two existing formulas (f_1 and f_2 with valuations v_1 and v_2) in the case of *AND* and negating an existing formula (f_1 with valuation v_1) in the case of *NOT*. Existing formulas are randomly selected from the pools. Before adding the resulting formula *f* to the construction pool, we derive its valuation *v* from the valuations of its sub-formulas (lines 23 and 26).

In essence, the second phase enriches the set of existing formulas by generating new ones without requiring a complete grammar for all syntactic constructs. Instead, we use a minimal, but functionally complete, grammar for *Boolean formulas*. This significantly simplifies formula generation without sacrificing expressiveness. Note that a separate pool for newly constructed formulas allows having control over how many of them are used in the instances generated in the third phase. In Fig. 2.2b, this step is responsible for generating the formulas on lines 8 and 9 that ultimately amount to checking the satisfiability of `a` $\vee$ `b`.

Once the two pools are populated, we use them to generate *NM* SMT instances

Algorithm 2: Depth-minimization procedure in STORM.

```
1 procedure MinimizeDepth(S, NC, NM, D_min, D_max, A_max)
2   if D_max ≤ D_min then
3     return S
4   D ← (D_min + D_max)/2
5   B ← Fuzz(S, NC, NM, D, A_max)
6   if 0 < Size(B) then
7     S_min ← SelectSeedWithSmallestDepth(B)
8     return MinimizeDepth(S_min, NC, NM, D_min, D, A_max)
9   return MinimizeDepth(S, NC, NM, D + 1, D_max, A_max)
```

that we feed to the solver under test. To assemble a new instance A, we create up to A_{max} assertions (*ac* on line 35) by randomly picking formulas from the pools. If the valuation of a selected formula is true, we directly assert it, otherwise we assert its negation. This ensures that all assertions are satisfiable. Of course, the same holds for instance A consisting of these assertions in addition to a satisfiability check. We now leverage this fact when feeding the SMT instance to the solver under test (line 45). The oracle reveals a critical bug if the solver returns *UNSAT*.

2.3.2. Instance Minimization

In practice, our fuzzing technique often generates bug-revealing instances that are very large, containing deeply nested formulas and several assertions. This can considerably complicate debugging for solver developers.

Adapting established minimization techniques based on delta debugging [270] might seem like a natural fit for this use case. However, the special nature of critical bugs complicates this task in comparison to other classes of bugs, such as crashes. For minimizing crashing instances, it is sufficient to minimize the original instance (e.g., by dropping assertions) *while preserving the crash*. In contrast, for instances that exhibit a critical bug, the behavior that should be preserved is more involved, that is, *the instance should be minimized such that the buggy solver still returns* `unsat` *while the ground truth remains* `sat`. This requires either satisfiability-preserving minimizations or a trusted second solver that can act as a ground-truth oracle by rejecting minimizations that do not preserve satisfiability. Unfortunately, the only state-of-the-art delta debugger for SMT instances, ddSMT [198], does not preserve satisfiability. (Note that ddSMT is the successor of deltaSMT [4], which was used to minimize instances generated by FuzzSMT [51].) Moreover, a second trusted solver is not always available (e.g., for new theories or solver-specific features and extensions).

To overcome these limitations, we developed a specialized minimization technique that directly leverages the bounds of our fuzzing procedure to obtain smaller instances (see Alg. 2 for depth minimization). By repeatedly running the fuzzing procedure on a *buggy* seed instance, this algorithm attempts to find the minimum

values for D_{max} and A_{max} that still reveal a critical bug. It uses binary search to first minimize the number of assertions (analogous to MINIMIZEDEPTH in Alg. 2) and subsequently the depth of asserted formulas. Note that the fuzzing procedure may report multiple bug-revealing instances, and we recursively minimize the smallest with respect to the bound being minimized (line 8). Our evaluation shows that this technique works more reliably than leveraging ddSMT and a second solver (see Sect. 2.5.5).

2.4. Implementation

Seeds. STORM uses the Python API in Z3 to manipulate SMT formulas for generating new instances. It can, therefore, only fuzz instances within the logics supported by Z3. In practice, this is not an important restriction since Z3 supports a very large number of logics. Moreover, STORM requires seeds to be expressed in an extension of the SMT-LIB v2 input format [9] supported by Z3. Note that SMT-LIB is the standard input format used across solvers.

Random assignments. STORM uses Z3 to generate a random model for a given seed (line 3 of Alg. 1). Note, however, that bugs in Z3 resulting in a wrong model do not affect our fuzzer. In fact, given any assignment, our technique just requires correct valuations for predicates in the initial pool. In theory, computing these valuations is relatively straightforward since the assignment provides concrete values for all free variables; simply substituting variables with values should be sufficient for quantifier-free predicates. In practice, we use Z3 to compute predicate valuations and have not encountered any bugs in this solver component.

Random choices. Our implementation provides concrete instantiations of functions RANDOP and RANDFORMULA from Alg. 1 as follows. RANDOP returns *AND* with probability 50% and *NOT* otherwise. Function RANDFORMULA selects a formula from one of the pools uniformly at random, but with probability 30% from the initial pool and from the construction pool otherwise.

Incremental mode. Many solvers support a feature called *incremental mode*. It allows client tools to push and pop constraints when performing a large number of similar satisfiability queries (e.g., checking feasibility of paths with a common prefix during symbolic execution). To efficiently support this mode, solvers typically use dedicated algorithms that reuse results from previous queries; in fact, SMT-COMP [6] features a separate track to evaluate these algorithms. To test incremental mode, STORM is able to generate SMT instances that contain push and pop instructions in addition to regular assertions.

2.5. Experimental Evaluation

In this section, we address the following research questions:

RQ1: How effective is STORM in detecting new critical bugs in SMT solvers?

RQ2: How effective is STORM in detecting known critical bugs in SMT solvers?

RQ3: How do the assertion and depth bounds of STORM impact its effectiveness?

RQ4: How effective is our instance minimization at reducing the size of bug-revealing instances?

RQ5: To what extent do STORM-generated instances increase code coverage of SMT solvers?

We make our implementation open source[5]. To support open science, we include all data, source code, and documentation necessary for reproducing our experimental results.

2.5.1. Solver Selection

We used STORM to test seven popular SMT solvers, which support the SMT-LIB input format [9] and regularly participate in the international SMT competition SMT-COMP [6]. Specifically, we selected Boolector [201], CVC4 [32], MathSAT5 [70], SMTInterpol [67], STP [102], Yices2 [91], and Z3 [84].

In addition to the above mature implementations, STORM was also used to test new features of solvers. In particular, the developers of Yices2 asked us to test the new bitvector theory in the MCSAT solver [138] of Yices2, which is based on the model-constructing satisfiability calculus [85]. MCSAT is an optional component of Yices2, which is dedicated to quantifier-free non-linear real arithmetic. STORM did not find bugs in this new theory of MCSAT, and the theory was integrated with the main version of Yices2 shortly after. In our experimental evaluation, it is therefore tested as part of Yices2.

Moreover, the developers of Z3 asked us to test a new arithmetic solver (we refer to it as Z3-AS), which they had been preparing for the last two years. It came with better non-linear theories and replaced the legacy arithmetic solvers in Z3. According to the Z3 developers, STORM could help expedite the integration of this new feature by finding bugs early, which it did. Since Z3-AS was later integrated in the main version of Z3, and we tested it independently, we include it separately in our evaluation.

Due to the success of STORM in detecting intricate critical bugs in Z3-AS, the Z3 developers described our fuzzer as being "*extremely useful*" and asked us to test Z3's debug branch (let us refer to it as Z3-DBG). Z3-DBG implemented a

[5] https://github.com/Practical-Formal-Methods/storm

variety of new solver features in which STORM also detected a critical bug (see Sect. 2.5.5).

Finally, the developers of the Z3str3 string solver [40] asked us to provide them with STORM-generated string instances. They became aware of STORM since it detected several critical issues in Z3str3, which we reported. Note that Z3str3 is developed by the same group of people as StringFuzz [46]. We, therefore, suspect that STORM found bugs in Z3str3 that StringFuzz could not find, especially since StringFuzz does not target critical bugs. The STORM-generated instances that we provided (in addition to the bug-revealing ones that we reported) were used as a regression test suite during the development of performance enhancements in Z3str3. According to a developer of Z3str3, our instances helped reveal critical bugs introduced by these enhancements. Most of these bugs were due to missing or incorrect axioms in Z3str3.

2.5.2. Logic Selection

In our experimental evaluation, for each solver, we identified well supported logics based on its participation in SMT-COMP 2019 [6]. In certain cases, we also added logics identified as error-prone by the solver developers, such as `QF_FP`. In general however, STORM can handle the intersection of all logics supported by the SMT-LIB v2 input format and all logics supported by Z3. The latter constraint emerges because our implementation relies on Z3's APIs for generating the mutated SMT instances (see Sect. 2.4).

Tab. 2.2 shows the tested logics for each solver. (The second column and second to last row of the table should be ignored for now.) The logic abbreviations are explained in the SMT-LIB standard [9], but generally speaking, the following rules hold. `QF` stands for quantifier-free formulas, `A` for arrays, `AX` for arrays with extensionality, `BV` for bitvectors, `FP` for floating-point arithmetic, `IA` for integer arithmetic, `RA` for real arithmetic, `IRA` for integer real arithmetic, `IDL` for integer difference logic, `RDL` for rational difference logic, `L` before `IA`, `RA`, or `IRA` for the linear fragment of these arithmetics, `N` before `IA`, `RA`, or `IRA` for the non-linear fragment, `UF` for the extension that allows free sort and function symbols, `S` for strings, and `DT` for datatypes.

2.5.3. Benchmark Selection

For our experiments, we used as seeds all non-incremental SMT-LIB instances in SMT-COMP 2019 [6]. We also used all SMT-LIB instances in the regression test suites of CVC4, Yices2, and Z3. The second column of Tab. 2.2 shows how many seeds correspond to each tested logic. The second to last row of the table ("Unsp.") refers to instances in which the logic is unspecified—the solver may use any.

In general, we only tested each solver with logics, and thus instances, it supports.

Table 2.2: The tested logics per solver and the number of seed instances per logic.

Logic	Seeds	SMT Solvers						
		Boolector	CVC4	MathSAT5	SMTInterpol	STP	Yices2	Z3
ALIA	42		✓		✓			✓
AUFNIA	3		✓					✓
LRA	2444		✓		✓			✓
QF_ALIA	42		✓		✓		✓	✓
QF_AUFNIA	3		✓	✓				✓
QF_DT	1602		✓					✓
QF_LRA	1049		✓		✓		✓	✓
QF_RDL	261		✓				✓	✓
QF_UFIDL	444		✓				✓	✓
QF_UFNRA	38		✓	✓			✓	✓
UFDTLIA	327		✓					✓
AUFDTLIA	728		✓					✓
AUFNIRA	1490		✓					✓
NIA	14		✓					✓
QF_ANIA	8		✓	✓				✓
QF_AX	555		✓	✓	✓		✓	✓
QF_FP	40418		✓	✓				✓
QF_NIA	23901		✓	✓			✓	✓
QF_S	24323		✓					✓
QF_UFLIA	580		✓		✓		✓	✓
UFLIA	9524		✓		✓			✓
AUFLIA	3273		✓		✓			✓
BV	5750	✓	✓					✓
NRA	3813		✓					✓
QF_AUFBV	49	✓	✓				✓	✓
QF_BV	3872	✓	✓			✓	✓	✓
QF_IDL	843		✓		✓		✓	✓
QF_NIRA	3		✓	✓			✓	✓
QF_UF	7481		✓		✓		✓	✓
QF_UFLRA	936		✓		✓		✓	✓
UF	7596		✓		✓			✓
UFLRA	17		✓		✓			✓
AUFLIRA	2268		✓		✓			✓
LIA	388		✓		✓			✓
QF_ABV	8310	✓	✓				✓	✓
QF_AUFLIA	1310		✓		✓			✓
QF_BVFP	17196		✓	✓				✓
QF_LIA	2104		✓		✓		✓	✓
QF_NRA	4067		✓	✓			✓	✓
QF_UFBV	1238	✓	✓				✓	✓
QF_UFNIA	478		✓	✓			✓	✓
UFDT	4527		✓					✓
UFNIA	4446		✓					✓
Unsp.	5825	–	–	–	–	–	–	–
Total	193586	5	43	10	17	1	19	43

For seeds without a specified logic, we only generated mutations of those that each solver could handle.

2.5.4. Experimental Setup

For our experiments, we used the following setting for STORM unless stated otherwise: $D_{max} = 64$, $A_{max} = 64$, NC between 200 and 1500, and NM between 300 and 1000 (see Alg. 1). Both NC and NM were adjusted dynamically within the above ranges based on the size of the initial pool. The goal was to use larger values for larger initial pools, and thus, larger seeds.

We performed all experiments on a 32-core Intel ® Xeon ® E5-2667 v2 CPU @ 3.30GHz machine with 256GB of memory, running Debian GNU/Linux 10 (buster).

Comparison with state of the art. Except for a single tool [55], all existing SMT solver testing tools at the time did not use oracles to detect critical bugs. They, therefore, required differential testing of multiple solvers to identify such bugs. In RQ2, we evaluate the effectiveness of STORM at detecting existing critical bugs, including the publicly reported bugs found by the most closely related tool at the time [55]. Recall that this tool supports only the theory of strings.

2.5.5. Experimental Results

We now discuss our experimental results for each of the above research questions.

RQ1: New critical bugs. Tab. 2.3 shows critical bugs found by STORM in the SMT solvers we tested between November 2019 and February 2020. The first column of the table shows the solvers. We list Z3str3 separately as it is not the default string solver in Z3. The second column denotes whether bugs were found in the incremental mode of a solver, which essentially corresponds to a different solver variant. The third column lists the logics in which bugs were found, and the last column shows the number of bugs. **During our three months of testing, STORM detected 29 critical bugs in three mature solvers (or nine solver variants) and 15 different logics.**

All of these bugs were previously unknown, unique, and confirmed by the solver developers. Out of the 29 critical bugs, 27 were fixed in the latest solver versions at the time of writing this dissertation. Note that the bugs were only detected by STORM-generated instances, i.e., none were detected by the seeds. In addition to the bugs in the table, STORM was also able to detect known bugs as well as other issues (i.e., of classes C and D) as a by-product, which we do not report here.

The feedback from solver developers is very positive, and we have been discussing it throughout the chapter. As another example, a Yices2 developer told

Table 2.3: Previously unknown, unique, and confirmed critical bugs found by STORM in the tested SMT solvers.

SMT Solver	Incremental Mode	Logics	Critical Bugs
MathSAT5		QF_FP QF_BVFP	2
Yices2		QF_UFIDL QF_UF	2
Yices2	✓	QF_UFIDL QF_UFLRA	2
Z3		QF_UFLIA QF_BV UF LIA QF_BVFP QF_LIA	8
Z3	✓	QF_FP QF_S	3
Z3str3		QF_S	6
Z3-AS		AUFNIRA QF_NIA AUFLIRA QF_NRA	4
Z3-AS	✓	AUFNIRA	1
Z3-DBG		QF_NIA	1

us that STORM found real bugs and that it is especially useful to have the ability to test the incremental mode of solvers. He also mentioned that they used to run FuzzSMT [51] on all theories, and that now this fuzzer runs continuously on new theories generating "*infinite*" instances. FuzzSMT, however, does not target critical bugs, and for this reason, they ran VoteSMT [12] to differentially test solvers and detect incorrect Yices2 results. Despite this, STORM detected four new critical bugs in Yices2.

Another Yices2 developer commented on the severity of two of the bugs that STORM found. He mentioned that one was in the pre-processing component and "*easy to fix (and an obvious mistake in retrospect) but it was in a part of Yices that had probably not been exercised much*". "*The other one was much more tricky to trace and fix, it was related to a combination of features and optimization in the E-graph, not localized to a single module*".

RQ2: Known critical bugs. In this research question, we evaluate the effectiveness of STORM in reproducing known critical bugs. We, therefore, collected

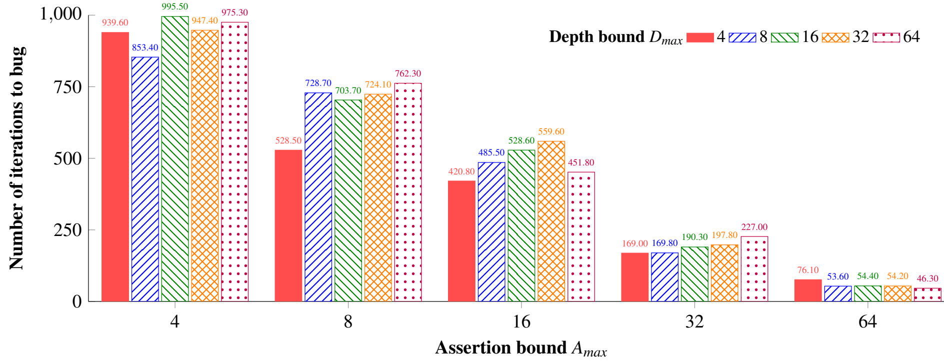


Figure 2.4: Median number of iterations to find bugs with different configurations of STORM. Each bar corresponds to a configuration with a certain depth and assertion bound.

all critical bugs that were reported for the solvers under test during the three-month period between Nov 15 and Feb 15, 2020. We focused only on bugs with a subsequent fix (i.e., closed issues on GitHub). Out of the seven solvers, we exclude MathSAT5 because it is closed source, and bugs may only be reported via email. We also exclude Boolector, SMTInterpol, and STP because no critical bugs were reported for these solvers during the above time period. For the remaining three solvers, CVC4, Yices2, and Z3, there were 6, 1, and 14 critical bugs with a fix, respectively, after excluding all the bugs that we reported.

We ran STORM on the solver version in which each bug was found. Since developers typically add fixed bugs to their regression tests, we removed all seeds that revealed any of these bugs (without being mutated). We collected all generated instances for which each solver incorrectly returned `unsat`. To ensure that STORM actually found the reported bug (and not a different one), we ran all bug-revealing instances against the first solver version with the corresponding fix. If the solver now returned `sat` for at least one of the instances, we counted the bug as reproduced.

For each of the three solvers, STORM was able to reproduce 1 (CVC4), 1 (Yices2), and 4 (Z3) critical bugs, so 6 out of a total of 21. Therefore, **if STORM had run on these solver versions, it would have prevented approximately 1/3 of the critical-bug reports in a three-month period.** Given that during this period we reported 10 additional bugs detected by STORM in these solvers, it is possible that our fuzzer would have been able to reproduce more bugs if it had run longer or if it was being run continuously.

We also ran STORM on the publicly reported critical bugs found by Bugariu and Müller [55] (regardless of when they were reported). STORM was able to reproduce them.

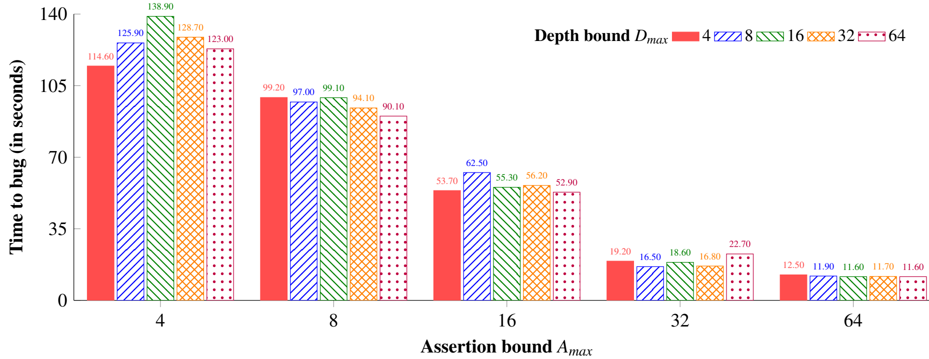


Figure 2.5: Median time (in seconds) to find bugs with different configurations of STORM. Each bar corresponds to a configuration with a certain depth and assertion bound.

RQ3: Fuzzing bounds. To evaluate the effect of the fuzzing bounds of STORM, we only considered closed bugs. We used all 19 closed bugs reported by us from RQ1 except for those in Z3-AS (the original commits could not be retrieved due to a rebase in the branch) for a remaining of 14 bugs. In addition, we used all reproduced bugs from RQ2 for a total of 21 bugs.

For each of these bugs, we randomly selected a seed file that had allowed STORM to detect the bug in RQ1 or RQ2. We performed eight independent runs of STORM (with random seeds different from the ones used in RQ1 and RQ2 to avoid bias) to evaluate the effect of the different fuzzing bounds. STORM was unable to reproduce one Yices2 bug from RQ1 with any of the eight random seeds; we therefore do not include it in the results shown in Fig. 2.4.

For the assertion and depth bounds A_{max} and D_{max}, we used five different settings: 4, 8, 16, 32, and 64. Fig. 2.4 shows the median number of iterations (i.e., generated instances) until the bug was found for different combinations of these settings. We can observe that **a large assertion bound reduces the number of iterations significantly (e.g., up to 12x for $D_{max} = 4$).** In contrast, the trend for the depth bound is less clear, which suggests that it has a less significant effect and is mostly useful for minimizing instances. We can observe very similar trends when comparing the median time to find the bug (see Fig. 2.5).

RQ4: Instance minimization. We now evaluate the effectiveness of our instance minimization. To this end, we collect all instances revealing the 20 bugs of RQ3 that are generated by STORM with its default configuration (Sect. 2.5.4).

The results of minimizing these instances using binary search (BS) and delta debugging (ddSMT [198]) are shown in Tab. 2.4. We perform eight independent minimization runs and report median results. Instance size is measured in terms of the number of bytes, the number of assertions, and the maximum formula depth in an assertion. A dash for ddSMT means either that the instance could not be

Table 2.4: Size of original and minimized bug-revealing instances. Instance size is shown in terms of the number of bytes / number of assertions / maximum formula depth.

Bug ID	Unminimized Instances			Minimized by BS			Minimized by ddSMT		
1	23430/	64/	5	20801/	61/	5	321/	4/	0
2	3756/	6/	12	3756/	6/	12	–/	–/	–
3	9641/	20/	8	5276/	19/	9	–/	–/	–
4	66209/	33/	56	13086/	8/	2	–/	–/	–
5	64071/	44/	27	24326/	24/	6	–/	–/	–
6	37943/	51/	2	4247/	5/	0	575/	4/	0
7	19408/	64/	3	1025/	5/	2	–/	–/	–
8	19235/	27/	2	4002/	5/	0	–/	–/	–
9	23659/	51/	4	2004/	5/	0	–/	–/	–
10	4275/	6/	1	1514/	5/	1	–/	–/	–
11	39585/	64/	16	5832/	16/	2	–/	–/	–
12	22017/	58/	5	1013/	5/	2	–/	–/	–
13	180082/	62/	8	7210/	5/	2	–/	–/	–
14	7934/	10/	8	4431/	10/	5	–/	–/	–
15	72490/	50/	0	5455/	5/	2	–/	–/	–
16	35725/	33/	3	2591/	5/	2	–/	–/	–
17	17180/	21/	57	1146/	5/	0	421/	1/	0
18	10176/	14/	0	2586/	14/	0	–/	–/	–
19	16812/	51/	4	13137/	33/	6	–/	–/	–
20	16826/	30/	1	5163/	5/	1	601/	7/	0

minimized or that ddSMT does not support a construct in the instance. As outlined in Sect. 2.3.2, we had to adapt ddSMT for this use case by invoking a second solver to reject minimizations that would not preserve satisfiability; we used the version of the solver that fixed the corresponding bug for this purpose.

Despite these adaptations, we observed that ddSMT could not minimize the instances for bugs 2, 3, 4, 5, 13, 14, and 18. We suspect that its search space of possible minimizations might not contain more complex transformations that would be required to both preserve satisfiability *and* the bug. We observed the same outcome when running ddSMT on instances that were first minimized using binary search.

For bugs 10, 11, and 19, ddSMT does not support `str.to.re` and `str.at`, which are supported by Z3str3. For bugs 7, 8, 9, 12, 15, and 16, ddSMT does not support **`check-sat-using`**, which is supported by Z3. Recall that STORM accepts seed instances expressed in the extension of the SMT-LIB format that is supported by Z3 (Sect. 2.4), whereas ddSMT only supports the standard.

Overall, this experiment shows that **our minimization procedure works more reliably and is able to significantly reduce buggy instances (median reduction of 82.7%).** However, for the cases where both procedures produced results, the

Table 2.5: **Code coverage increase as more instances are generated by STORM.**

Generated Instances	Line Coverage	Function Coverage
0	58219	26256
100	66945	30498
200	67063	30524
300	67119	30547
400	67208	30598
500	67759	30861

ddSMT-based minimization procedure was able to produce smaller instances. This is not entirely surprising given that BS uses the fuzzer, which treats predicates not containing other predicates (i.e., ground- or leaf-predicates) as atomic building blocks. For instance, for bug 17, the instance that was minimized with BS contains several complex ground-predicates that ddSMT is able to minimize further. We expect that more involved combinations of the two approaches could produce even better results.

RQ5: Code coverage. A Yices2 developer mentioned that they use fuzzer-generated instances to enrich their regression tests such that they achieve higher coverage. In this research question, we therefore evaluate whether STORM is able to increase coverage.

We selected one of the solvers (Z3) and four random logics in which we found bugs (`QF_UFLIA, AUFNIRA, UF, LIA`). We then computed the line and function coverage when running Z3 on all the instances from SMT-COMP 2019 [6] for these logics (10054 seeds). The result is shown in the first row of Tab. 2.5. At the same time, we randomly selected 5 instances from each logic and ran STORM with $NM = 500$ and a single new random seed to generate exactly 500 new instances for each of the 20 seed instances. Tab. 2.5 shows that, as more instances are generated, coverage increases noticeably (9540 more lines and 4605 more functions after only 500 generated instances). This demonstrates that **running STORM on only a small number of seed instances is able to result in a noticeable coverage increase over a large number of instances from a well known benchmark set.**

2.6. Threats to Validity

We identify the following threats to the validity of our experiments.

Selection of seeds. STORM requires seed instances as input, and our results do not necessarily generalize to other seeds [240]. However, we selected as seeds

instances from SMT-COMP 2019 [6] as well as regression test suites of solvers. We believe that our selection is sufficiently broad to mitigate this threat. In addition, we make our tool open source so it may be run with different seeds.

Selection of solvers. The bugs found by STORM depend on the solvers and logics that we tested. However, we selected a wide range of different, mature solvers and logics to mitigate this threat.

Randomness in fuzzing. A common threat when evaluating fuzzers is related to the internal validity [240] of their results. To mitigate systematic errors that may be introduced due to random choices of our fuzzer, we used random seeds to ensure deterministic results and performed experiments for eight different seeds.

2.7. Related Work

SMT solvers are core components in many program analyzers, and as a result, their reliability is of crucial importance. Although it is feasible to verify SAT and SMT *algorithms* [97, 157, 172], it is challenging and time consuming to verify even very basic SAT- or SMT-solver *implementations*. Verifying highly complex and high-performance solver implementations, such as CVC4 [32] and Z3 [84], is completely impractical. For these reasons, there is a growing interest in testing such solvers, alongside related efforts that focus on testing entire program analyzers.

Testing SAT and SMT solvers. FuzzSMT [51] focuses on finding crashes of SMT solvers for bitvector and array instances. It uses grammar-based blackbox fuzzing to generate crash-inducing instances and minimizes any such instances with delta debugging [4, 270]. Brummayer et al. [52] extend this line of work to SAT and QBF solvers. In contrast, STORM performs mutational fuzzing, and its minimization procedure leverages the fuzzer and its bounds regarding the number of assertions and the formula depth.

StringFuzz [46] targets testing of string solvers. In addition to randomly generating syntactically valid instances using a grammar, it is also able to mutate or transform formulas in existing instances. However, since not all of its transformations preserve satisfiability, it is not easily possible to leverage metamorphic testing [31] to detect critical bugs. In contrast to both FuzzSMT and StringFuzz, the satisfiability of all STORM-generated instances is known.

Previously to STORM, Bugariu and Müller [55] proposed an automated testing technique that synthesizes SMT instances for the string theory. The true satisfiability of the generated instances is derived by construction and used as a test oracle. In contrast, STORM performs mutational fuzzing and supports a wide range of theories.

In a subsequent work to STORM, Winterer *et al.* [89] introduced type aware mutation for SMT instances, a technique that replaces operators in existing SMT instances with operators of conforming types to generate well-typed mutant instances. These mutant instances are then used as test cases for differential testing to detect soundness bugs in SMT solvers. The approach is further generalized in [208]. In contrast to our work, the approach does not have a test oracle, i.e., to detect a bug, it relies on a disagreement between two SMT solvers. Another subsequent work by Winterer *et al.* [263] introduces semantic fusion, an approach that combines two equisatisfiable SMT instances (both instances are either satisfiable or unsatisfiable) into a new equisatisfiable one. Similar to our approach, semantic fusion uses seed SMT instances to generate new instances. However, in contrast to our approach, the satisfiability of the seed instances must be known in advance.

A subsequent work by Yao *et al.* [268] presents a metamorphic testing-based approach that takes a seed SMT instance ϕ and tests an SMT solver by identifying the inconsistency between the satisfiability result of ϕ and its equi-satisfiable mutants. These mutants are generated using predefined mutation rules. The key idea behind the approach is: an over-approximation of a satisfiable instance is satisfiable and an under-approximation of an unsatisfiable instance is unsatisfiable. STORM in contrast can generate a satisfiable formula independently of the satisfiability status of the seed formula. In another recent work, Yao *et al.* [267] present a feedback-driven grammar-based fuzzing technique that also considers the configuration space of the SMT solver during the fuzzing campaign. For a generated SMT instance, the fuzzer attempts to explore the configuration space of a solver by mutating the solver options. STORM does not need a grammar to generate new SMT instances and focuses on detecting critical soundness issues in default and the most widely used solver configurations. Furthermore, STORM-generated SMT instances can also be used with different options to detect soundness bugs in different solver configurations.

Recently Scott *et al.* [234] proposed a reinforcement learning based fuzzing system to detect performance issues in SMT solvers. The technique only focuses on the theory of strings and floating-point arithmetic. In contrast, STORM mainly targets critical soundness issues in SMT solvers and is not limited to a theory. Bringolf *et al.* [177] presented an approach to detect incompleteness bugs in SMT solvers by mutating an SMT instance using local satisfiability preserving transformations. An incompleteness bug is reported if the solver unexpectedly returns `unknown`. STORM targets critical soundness bugs in SMT solvers that are the hardest to detect and may cause unsoundness in program analyzers that rely on these solvers.

Unlike the above approaches that test solvers by generating input instances for their textual interface in either SMT-LIB or some solver-specific format, Artho et al. [18] and Niemetz et al. [199, 200] developed model-based API testing frameworks for SAT and SMT solvers to test a solver's application programming interface. They focus on testing various API parameters and solver options. These frameworks generate a random but valid sequence of solver API calls based on a

customizable API model.

Testing program analyzers. Kapus and Cadar [141] combine random program generation with differential testing [180] to find bugs in symbolic-execution engines. Their technique is inspired by existing compiler-testing techniques (e.g., Csmith [266]) and used to test KLEE [58], CREST [3], and FuzzBALL [174].

Cuoq et al. [82] use randomly generated programs to test the Frama-C static-analysis platform [74]. Bugariu et al. [56] present a fuzzing technique for detecting soundness and precision issues in implementations of abstract domains—the core components of abstract interpreters [76]. They use algebraic properties of abstract domains as test oracles and find bugs in widely used domains. Recently, Taneja et al. [248] proposed a testing technique for identifying soundness and precision issues in static dataflow analyses by comparing results with a sound and maximally precise SMT-based analysis; they rely on the SMT solver to provide correct results.

Zhang et al. [271] develop a practical and automated fuzzing technique to test software model checkers. They focus on testing control-flow reachability properties of programs. More specifically, they synthesize valid branch reachability properties using concrete program executions and then fuse individual properties of different branches into a single safety property.

Klinger et al. [148] propose an automated technique to test the soundness and precision of program analyzers in general. Their approach is based on differential testing. From seed programs, they generate program-analysis benchmarks on which they compare the results of different analyzers.

2.8. Summary and Remarks

In this chapter, we have presented a novel fuzzing technique for detecting critical bugs in SMT solvers—key components of many state-of-the-art program analyzers. Conceptually, STORM is a blackbox mutational fuzzer that uses fragments of existing SMT instances to generate new, realistic instances. Its formula-generation phase takes inspiration from grammar-based fuzzers; it leverages a minimal, but functionally complete, grammar for Boolean formulas to generate new formulas from fragments found in seeds. Finally, it solves the oracle problem by generating instances that are satisfiable by construction.

Chapter 3

Metamorphic Testing of Datalog Engines Using Conjunctive Queries

Datalog is a popular query language with applications in several domains. Like any complex piece of software, Datalog engines may contain bugs. The most critical ones manifest as incorrect results when evaluating queries—we refer to these as *query bugs*. Given the wide applicability of the language, query bugs may have detrimental consequences, for instance, by compromising the soundness of a program analysis that is implemented and formalized in Datalog. In this chapter, we present the first metamorphic-testing approach for detecting query bugs in Datalog engines. We ran our tool on three mature engines and found 13 previously unknown query bugs, some of which are deep and revealed critical semantic issues.

3.1. Introduction

Datalog [113] is a declarative, logic-based query language that is syntactically a subset of Prolog. Datalog is expressive, yet concise, and as a result, it is used as a domain-specific language in several application domains, such as natural-language processing [194], bio-informatics [145, 232], big-data analytics [118, 134], networking [164], program analysis [48, 83, 110, 195, 261], robotics [213], generic graph databases [239], and security [49, 50, 111, 251].

Query evaluation is performed by *Datalog engines*, prominent examples of which include Soufflé [136], bddbddb [260], DDlog [228], μZ [125], and LogicBlox [17]. However, as any complex piece of software, Datalog engines may contain bugs, resulting in incorrect query results. An incorrect result may manifest by including wrong entries or by missing entries that should have been included.

We refer to such bugs as *query bugs*.

Depending on the application domain, query bugs may have detrimental consequences. In particular, when a buggy Datalog engine is used in program analysis, it could compromise *soundness* of the verification process; in other words, it could cause an analyzer to verify incorrect software. As an example, imagine a static analyzer that uses Datalog to implement a may-alias (or must-alias) analysis. A query bug that results in computing fewer (or more) aliases could lead to missing critical bugs in the analyzed software.

In this chapter, we present the *first* automatic test-case generation approach for detecting query bugs in Datalog engines. A major challenge in finding such bugs is the lack of an *oracle* specifying *expected* query results. This problem may be overcome with a technique known as *differential testing* [180]. Differential testing would involve running multiple Datalog engines on a common set of programs and comparing their results for discrepancies. In our context, this would be extremely difficult as there exists no unified standard for Datalog syntax; as a result, many different dialects have emerged.

Our approach circumvents the lack of an oracle using an alternative technique, namely *metamorphic testing* [66]. It works by transforming a Datalog program such that the new result has an a-priori known relationship to the result of the original program. Examples of such a relationship are that the new result should be equivalent to the original, contained in the original, or containing the original. To ensure that these oracles are known in advance, we design metamorphic transformations based on database theory, and in particular, formal properties of *conjunctive queries*.

Despite their simplicity, conjunctive queries constitute an important class of database queries due to their theoretical properties. Specifically, while many fundamental problems in query optimization and minimization are computationally hard—or even undecidable—for general forms of queries, they are feasible for conjunctive queries. An example of such a problem is *query containment*, which we discuss in Sect. 3.3. The key insight behind our approach is to leverage properties of conjunctive queries to develop metamorphic transformations for full-blown Datalog programs.

We implement our approach in a tool called queryFuzz, which we use to test three mature Datalog engines. Not only did we find previously unknown query bugs in all engines, but we also detected 81% of all reported query bugs in the period between May 2020 till February 2021. Moreover, as we describe in Sect. 3.10, some of these bugs were hidden deep in the engine stack and revealed critical semantic issues.

Contributions. The contributions of this chapter are as follows:

1. We present the first metamorphic-testing approach for detecting query bugs in Datalog engines.

```
1  // declarations
2  edge(X:number, Y:number).
3  reachable(X:number, Y:number).
4  .output reachable
5
6  // facts
7  edge(1,2).
8  edge(2,3).
9  edge(4,2).
10 edge(2,5).
11
12 // rules
13 reachable(X,Y) :- edge(X,Y).
14 reachable(X,Z) :- edge(X,Y), reachable(Y,Z).
```

Figure 3.1: A simple Datalog program.

2. We implement our approach in an open-source tool[1], queryFuzz. We are already working closely with the developers of the mature Datalog engines in order to integrate queryFuzz in their development cycles.

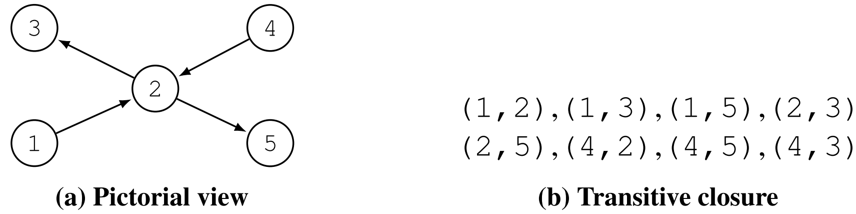
3. We evaluate the effectiveness of queryFuzz by testing three popular Datalog engines. Our tool detected 13 previously unknown query bugs in all three engines as well as many other bugs as a by-product.

Outline. The next section gives an overview of our approach. Sect. 3.3 provides background on properties of conjunctive queries, Sect. 3.4 explains the technical details of our approach for these queries, and Sect. 3.8 generalizes the approach to full-blown Datalog programs. In Sect. 3.9, we describe the implementation of queryFuzz. We present our experimental evaluation in Sect. 3.10, discuss related work in Sect. 3.12, and conclude in Sect. 3.13.

3.2. Overview

Datalog is a logic programming language where programs comprise a finite set of *rules* over *relations*. *Input relations* are given in the form of *facts*; they are also commonly referred to as *extensional database (EDB) relations*. *Intensional database (IDB) relations* are defined by logic rules, and one of them is specified as *output*. Fig. 3.1 shows an example of a simple Datalog program. The rules on

[1] https://github.com/Practical-Formal-Methods/queryFuzz



(a) Pictorial view **(b) Transitive closure**

Figure 3.2: Pictorial view and transitive closure of edge.

lines 13 and 14 define IDB relation `reachable`, which is specified as output on line 4 and computes the transitive closure of input relation `edge`.

Pictorially, `edge` represents the graph in Fig. 3.2a. There is an edge from node *x* to node *y* if `edge(`*x*`,`*y*`)` is a fact. Execution of this program is essentially a sequence of derivations, where each step adds an edge tuple to the output relation until a fixed point is reached. Fig. 3.2b shows the final tuples in `reachable`.

Approach. Using the above example as seed, we now give an overview of our metamorphic-testing approach for Datalog engines. Fig. 3.3 illustrates its main stages.

The first stage, *Program Generation*, generates a diverse set of programs to be transformed. It takes as input a (possibly empty) seed program, such as that of Fig. 3.1, and outputs a new program. In case the seed is empty, the new program is randomly generated based on a Datalog grammar. If the seed is not empty, this stage automatically extends it with randomly generated IDB relations using both existing and newly generated facts and rules (again based on a grammar). This is essentially a generalization of the above case where the seed is empty. One of the program relations is then specified as output.

The second stage, *Program Transformation*, applies metamorphic transformations to the newly generated program (or directly to the seed if the first stage is skipped). These transformations change rules of the program such that—when computing its output using a Datalog engine—the new result has an a-priori known relationship to the old result. In particular, the new result may contain the old one (as computed by program `exp.dl` in Fig. 3.3), it may be equivalent to the old one (as computed by `equ.dl`), or it may be contained in the old result (as computed by `con.dl`). For example, a transformation in which the new result should be equivalent to the old one is changing line 13 of Fig. 3.1 to the following:

```
reachable(X,Y) :- edge(X,Y), edge(W,Y).
```

As we will see in the next section, this change appears to be introducing a join, which however has no effect on the result. Another transformation could be applied to line 14 as follows:

```
reachable(X,Z) :- edge(X,X), reachable(X,Z).
```

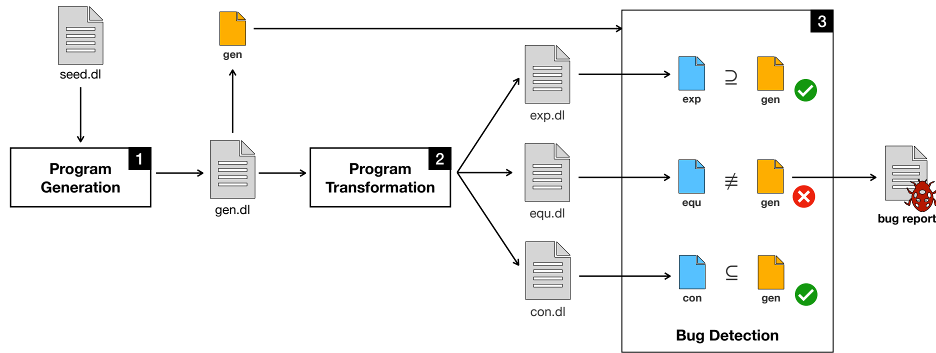


Figure 3.3: Overview of our approach.

In this case, the new result should be contained in the old one—in fact, the new result should be empty as there are no edges from a node to itself.

Finally, the third stage, *Bug Detection*, uses these relationships between new and old results (shown in blue and yellow, respectively, in Fig. 3.3) as *oracles* in order to detect query bugs in the underlying Datalog engine. For instance, imagine that, after transforming line 13 of Fig. 3.1 as described above, the Datalog engine returns all but one of the tuples shown in Fig. 3.2b. Since this transformation ensures that the new result is equivalent to the old one, a query bug has been detected. Note that a query bug is also detected if the old result is incorrect as long as the expected relationship to the new result does not hold.

Query bugs. In the rest of this section, we present two query bugs detected by queryFuzz in existing Datalog engines. We provide a complete list of detected bugs and more details in Sect. 3.10.

Fig. 3.4 shows a program snippet that was generated by queryFuzz in order to test μZ [125], the Datalog engine of the Z3 SMT solver [84] supporting the bddbddb [260] dialect. Relation `r` (line 4) is defined to compute all tuples in `in2` whose second element is in `in1`. Tuple `(25,10)` is the only one that satisfies this definition. Output relation `out` (line 5) obtains the first element of each tuple in `r`, that is, it computes `25`. This is also the result that is returned by μZ. Now, consider the following transformation applied by queryFuzz to line 5:

```
out(F) :- r(F,C), r(F,A), r(F,B).
```

The result of the new program should still be `25`, but μZ returns values `7–63`. We reported this bug on Z3's GitHub issue tracker[2], and it was immediately confirmed and fixed. In fact, a Z3 developer commented: "*These are good latent bugs. They exercise some edge cases that slipped through the cracks until now.*"

The code snippet in Fig. 3.5 was also generated by queryFuzz, this time when testing the Soufflé Datalog engine [136]. Relation `out` is the output relation of

[2] https://github.com/Z3Prover/z3/issues/4870

```
1 in1(49). in1(10).
2 in2(25,10). in2(16,13). in2(24,22).
3
4 r(V,M) :- in2(V,M), in1(M).
5 out(F) :- r(F,C).
```

Figure 3.4: Generated program snippet for testing μZ.

```
1 HqV(a) :- MZV(a,b), MZV(c,d).
2 gQk(jW) :- MZV(jW,jW).
3 QOq(aS,GF) :- MZV(GF,GF), gQk(M), HqV(aS), MZV(aS,M).
4 RwL(qr) :- QOq(u,qr), gQk(u), gQk(u).
5 out(jB,ym) :- gQk(h), RwL(ym), MZV(h,jB).
```

Figure 3.5: Generated program snippet for testing Soufflé.

the program. When line 1 is changed to

```
Hqv(a) :- MZV(a,b).
```

the program result should remain the same. However, we found that the result of the original program contained 240 entries, whereas that of the transformed program contained 306. We reported this query bug[3], which was immediately fixed.

These types of bugs, detected by queryFuzz, are extremely difficult for unsuspecting users to notice and might compromise upstream applications that rely on a Datalog engine.

3.3. Background

In this section, we review key concepts from database theory, and in particular query optimization, that form the basis of our metamorphic transformations.

A *database schema* $\mathbb{R}$ is a set of relations R. The *arity* of a relation is the number of attributes in the relation. For example, `edge` and `reachable` in Fig. 3.1 are relations of arity 2. An attribute in a relation can take values from a domain D. Let R be a relation of arity m. A *fact* over R is an expression of the form $R(a_1,...,a_m)$, where $a_i \in D_i$ for every $i = 1,\ldots,m$, e.g., `edge(1,2)` in Fig. 3.1. An *instance* of relation R is a finite set of facts over R. A *database instance* I over a database schema $\mathbb{R}$ is a collection of relational instances over the relations $R \in \mathbb{R}$.

A *conjunctive query* (CQ) is a single non-recursive function-free Horn rule,

[3] https://github.com/souffle-lang/souffle/issues/1453

e.g., every rule in Figs. 3.4 and 3.5 is a CQ. This is the simplest type of query that can be expressed over a database schema. Syntactically, a conjunctive query Q is an expression of the form

$$P(\vec{U}) \leftarrow R_1(\vec{U}_1), \ldots, R_n(\vec{U}_n)$$

where $\vec{U}$ and $\vec{U}_i$ $(1 \leq i \leq n)$ are vectors of variables and constants. Any variable appearing in $\vec{U}$ must also appear in some $\vec{U}_i$. The expression to the left of $\leftarrow$ is the *head* of the query, and the expression to the right is the *body*. Each $R_i(\vec{U}_i)$ in the body of the query is a *subgoal*, and $R_i \in \mathbb{R}$ is a *relation*. Note that subgoals can refer to the same relation. The set of answers for query Q w.r.t a database instance I is denoted by $Q(I)$.

Given two syntactically different CQs, we now define query *equivalence* and *containment*.

Definition 1 (Query Equivalence). *Two conjunctive queries Q_1 and Q_2 are equivalent, denoted by $Q_1 \equiv Q_2$, iff for every database instance I, we have $Q_1(I) = Q_2(I)$.*

Definition 2 (Query Containment). *Conjunctive query Q_1 is contained in conjunctive query Q_2, denoted by $Q_1 \subseteq Q_2$, iff for every database instance I, we have $Q_1(I) \subseteq Q_2(I)$.*

It is straightforward to see that if $Q_1 \subseteq Q_2$ and $Q_2 \subseteq Q_1$, then $Q_1 \equiv Q_2$. A decidable procedure for checking query containment [62] involves determining whether there exists a so-called *containment mapping* between two queries.

Definition 3 (Substitution). *A substitution θ is a mapping from a set of variables V to a set of variables V'.*

Definition 4 (Containment Mapping). *A substitution θ is a containment mapping from conjunctive query Q_2 to conjunctive query Q_1, if Q_2 can be transformed by means of θ to become Q_1.*

Formally, given two CQs

$$P(\vec{U}) \leftarrow R_1(\vec{U}_1), \ldots, R_n(\vec{U}_n) \qquad (Q_1)$$

$$P'(\vec{V}) \leftarrow S_1(\vec{V}_1), \ldots, S_m(\vec{V}_m) \qquad (Q_2)$$

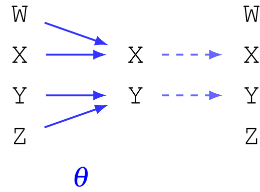
θ is a containment mapping from Q_2 to Q_1 if:

1. $\theta(P'(\vec{V})) = P(\vec{U})$, and
2. $\forall i \in \{1, \ldots, m\} \cdot \exists j \in \{1, \ldots, n\} \cdot \theta(S_i(\vec{V}_i)) = R_j(\vec{U}_j)$.

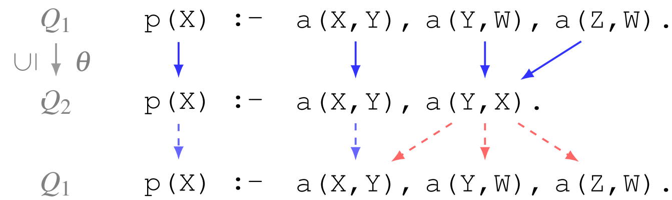
In words, a containment mapping maps variables of Q_2 to variables of Q_1 such that

1. the head of Q_2 becomes the head of Q_1, and
2. each subgoal of Q_2 becomes *some* subgoal of Q_1.

Theorem 1. *Let Q_1 and Q_2 be conjunctive queries. Q_2 is contained in Q_1 ($Q_2 \subseteq Q_1$) iff there exists a containment mapping from Q_1 to Q_2.*



(a) Containment mapping θ from Q_1 to Q_2.



(b) Mapping of head and subgoals induced by θ.

Figure 3.6: Containment mapping θ from Q_1 to Q_2 induces a mapping of subgoals. No mapping exists from Q_2 to Q_1.

As an example, consider the two CQs below (in Datalog syntax):

```
p(X) :- a(X,Y), a(Y,W), a(Z,W). // Q1
p(X) :- a(X,Y), a(Y,X).         // Q2
```

Q_2 is contained in Q_1 ($Q_2 \subseteq Q_1$) because there exists a containment mapping θ from Q_1 to Q_2 (shown using solid arrows in Fig. 3.6a; dotted arrows should be ignored for now). This is indeed a containment mapping because the head of Q_1 is the head of Q_2 and each subgoal of Q_1 becomes a subgoal of Q_2 (shown using solid arrows in Fig. 3.6b). On the other hand, Q_1 is not contained in Q_2 ($Q_1 \nsubseteq Q_2$) because there does not exist a containment mapping from Q_2 to Q_1, shown with dotted arrows in the figure. If `X` and `Y` are mapped to themselves (see Fig. 3.6a), then the head and first subgoal of Q_2 become the head and first subgoal of Q_1, but the second subgoal of Q_2 cannot become any subgoal of Q_1 (see Fig. 3.6b; red dotted arrows denote invalid subgoal mappings).

3.4. Metamorphic Transformations

Using the equivalence and containment properties of CQs, we now present their metamorphic transformations. Note that, in this section, we keep the presentation simple by describing a single transformation to a single conjunctive query. In practice however, our approach can perform sequences of transformations to multiple, more general queries (see Sects. 3.7.1 and 3.8 for more details).

Since any conjunctive query may be expressed as a Datalog rule, we refer to CQs as rules in the following. Given a Datalog rule Q, our metamorphic rule transformations are categorized into three *types*:

Addition (ADD): Q is transformed into $\text{ADD}(Q) = Q'$ by adding a subgoal.

Modification (MOD): Q is transformed into $\text{MOD}(Q) = Q'$ by the modifying a variable.

Removal (REM): Q is transformed into $\text{REM}(Q) = Q'$ by removing a subgoal.

Each of these transformation types may result in any of the following three *outcomes*:

Expansion (EXP): Original rule Q is contained in the transformed rule Q', i.e., $Q \subseteq Q'$.

Equivalence (EQU): Original rule Q is equivalent to the transformed rule Q', i.e., $Q \equiv Q'$.

Contraction (CON): Transformed rule Q' is contained in the original rule Q, i.e., $Q' \subseteq Q$.

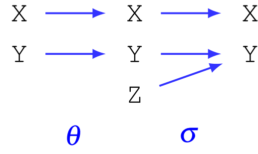
We refer to these outcomes as *oracles*.

Based on the above, a *rule transformation* combines a transformation type with an oracle. For instance, ADDCON refers to adding a subgoal to a rule Q such that the resulting rule Q' is contained in Q. Next, we describe these transformations in detail.

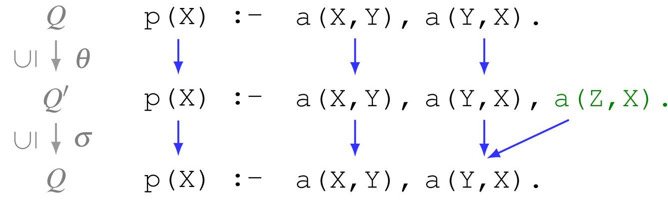
3.5. ADD Transformations

The ADD transformations add a subgoal $R(v_1, \ldots, v_n)$ to a rule Q, where $v_1, \ldots, v_n$ are variables—we ignore constants for simplicity.

ADDEXP. The ADDEXP transformation ensures that Q is contained in the resulting rule Q', i.e., $Q \subseteq Q'$. However, note that it is not possible to obtain a Q' such that $Q \subset Q'$ by adding a subgoal. The reason is that, when adding a subgoal to Q, there is *always* a containment mapping from Q to Q', i.e., $Q' \subseteq Q$. This is because the head of Q is the head of Q', and each subgoal of Q is in Q'. Consequently, even if there existed a containment mapping in the desirable



(a) Containment mapping θ from Q to Q' and mapping σ from Q' to Q.



(b) Mapping of head and subgoals induced by θ and σ.

Figure 3.7: Example of ADDEQU transformation.

direction, i.e., $Q \subseteq Q'$, then the two queries would be equivalent, a case that is already covered by ADDEQU.

ADDEQU. Given that a containment mapping from Q to Q' always exists, the ADDEQU transformation guarantees that $Q \equiv Q'$ by ensuring there also exists a containment mapping from Q' to Q. Intuitively, ADDEQU adds a new subgoal to Q while avoiding introducing new joins among the existing subgoals, thus preserving the original result. To ensure the existence of a containment mapping from Q' to Q when adding a subgoal $R(v_1, \ldots, v_n)$ to Q, relation R must already exist in the body of Q.

Example. Fig. 3.7 shows an example of an ADDEQU transformation. The new subgoal `a(Z,X)` (shown in green) maps to `a(Y,X)` when respecting the containment mapping σ from Q' to Q. Although it might appear that the new subgoal introduces a join, this join does not restrict the original result (as computed by the original subgoals) any further.

Algorithm. The algorithm performing this transformation is shown in procedure ADDEQU of Alg. 3. First, we extract the head and body of rule Q (line 2). Then, a random subgoal g and its arity n are retrieved from *body* (lines 3–4). On lines 5–8, we replace each of m variables in g with a fresh variable, where m is a random number from 1 to n. Each call to function FRESHVAR returns a new variable that is not already present in Q. This guarantees that no new joins are introduced.

Algorithm 3: ADD transformations

```
1  procedure ADDEQU(Q)
2    head, body ← Q
3    g ← RANDSUBGOAL(body)
4    n ← ARITY(g)
5    m ← RANDINTRANGE(1, n)
6    for (i ← 0, i < m, i++) do
7      j ← RANDINTRANGE(0, n − 1)
8      g.args[j] ← FRESHVAR(Q)
9    return head ← body, g.
10 procedure ADDCON(Q, relations)
11   g.rel ← RANDRELATION(relations)
12   n ← ARITY(g)
13   vars ← EXTRACTALLVARS(Q)
14   for (i ← 0, i < n, i++) do
15     g.args[i] ← RANDVAR(vars)
16   head, body ← Q
17   if g ∈ body then
18     return none
19   return head ← body, g.
```

Subgoal g is finally appended to *body*, and new rule Q' is returned (line 9). In the example of Fig. 3.7, we replace variable `Y` in subgoal `a(Y,X)` of Q with fresh variable `Z` and append this new subgoal to Q in order to generate Q'.

ADDCON. The ADDCON transformation ensures that rule Q' is contained in original rule Q, i.e., $Q' \subseteq Q$. Intuitively, ADDCON adds a new subgoal to Q introducing new joins, thus potentially contracting the original result. To differentiate this transformation from ADDEQU, we ensure that a containment mapping does not exist from Q' to Q, i.e., $Q \nsubseteq Q'$. Note, however, that the absence of such a mapping does not mean that Q' produces a *strictly* contracted result. In other words, $Q' \subset Q$ does not always hold; for example, for an empty database instance, the result of Q' is still equivalent to that of Q. To ensure the absence of a containment mapping from Q' to Q when adding a subgoal $R(v_1, \ldots, v_n)$ to Q, relation R must either not already exist in the body of Q, or if it does, its variables should prevent it from being mapped to any subgoal in Q.

Example. Fig. 3.8 shows an example of an ADDCON transformation. The new subgoal `a(Y,Y)` (shown in green) corresponds to relation `a`, which already appears in the body of Q. Despite this, the new subgoal does not map to any subgoal in Q since variable `Y` may not be mapped to both `X` and `Y`.

Algorithm. The algorithm is shown in procedure ADDCON of Alg. 3. As a first step, we create a subgoal g by randomly selecting a relation from the set of all relations in the program (line 11). On line 12, we retrieve its arity n. Then, all

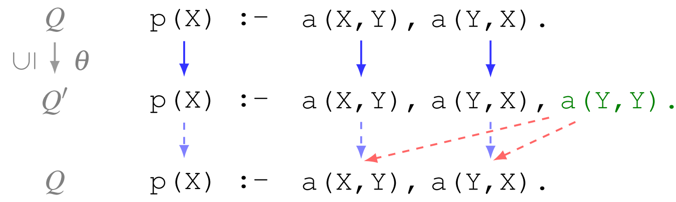
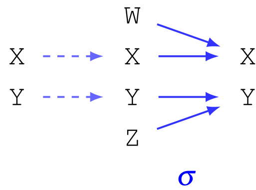
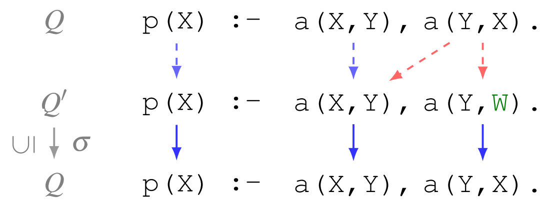


Figure 3.8: Example of ADDCON transformation.

variables of query Q are extracted in *vars* (line 13), and we initialize each argument of g with a random variable from *vars* (lines 14–15). Using variables in Q for this initialization guarantees that new joins are introduced unless g already appears in *body*. If so, we discard it (lines 17–18), otherwise, we append g to *body* and return new rule Q' (line 19). Note that, when `none` is returned, our implementation tries again. In the example of Fig. 3.8, we select relation `a`, initialize its arguments with variable `Y`, and append this new subgoal to Q.



(a) Containment mapping σ from Q' to Q.



(b) Mapping of head and subgoals induced by σ.

Figure 3.9: Example of MODEXP transformation.

Algorithm 4: MOD transformations

```
1  procedure ModExp(Q)
2     head, body ← Q
3     vars ← ExtractReusedVars(body)
4     v ← RandVar(vars)
5     body′ ← ReplaceRandOccurrence(body, v, FreshVar(Q))
6     return head ← body′
7  procedure ModCon(Q)
8     vars ← ExtractAllVars(Q)
9     if |vars| < 2 then
10        return none
11    v ← RandVar(vars)
12    w ← RandVar(vars \ {v})
13    Q′ ← ReplaceVar(Q, v, w)
14    return Q′
```

3.6. MOD Transformations

The MOD transformations modify a rule Q by renaming a variable appearing in its subgoals.

MODEXP. Intuitively, this transformation expands the result of Q by renaming a variable in a way that removes existing joins. This is achieved by creating a surjective containment mapping from Q' to Q, i.e., $Q \subseteq Q'$. Note that the mapping may not be bijective as this would make MODEXP equivalent to MODEQU.

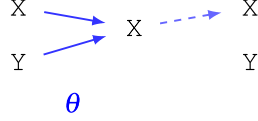
Example. Fig. 3.9 shows an example of a MODEXP transformation, where variable `X` of subgoal `a(Y,X)` is renamed to `W`.

Algorithm. The algorithm for this transformation is shown in procedure MODEXP of Alg. 4. We first extract variables *vars* that appear more than once in *body* of rule Q (line 3). A random variable v from *vars* is selected (line 4), and we replace a random occurrence of v in *body* with a fresh variable to get $body'$ (line 5). Replacing an occurrence of a reused variable with a fresh one guarantees that existing joins are removed. Finally, we return $head \leftarrow body'$ as transformed rule Q' (line 6). In the example of Fig. 3.9, we choose variable `X`, which appears twice in the body of Q, and replace its second occurrence with fresh variable `W`.

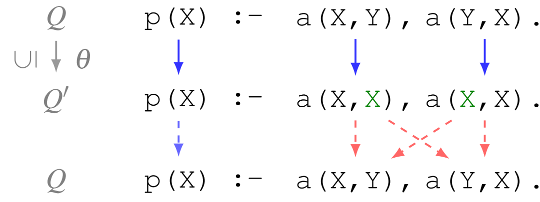
MODEQU. The MODEQU transformation ensures that the result of Q is equivalent to that of Q' by creating a bijective containment mapping between the two rules. A way to guarantee the existence of such a mapping is by replacing *all* occurrences of a variable in Q with those of a fresh variable. Note that this is a very simple transformation, which we include here mainly for completeness.

MODCON. Analogously to the MODEXP transformation, MODCON renames a variable in Q such that there exists a surjective (and not bijective) containment mapping from Q to Q'.

Example. Fig. 3.10 shows an example of a MODCON transformation, where all



(a) Containment mapping θ from Q to Q'.



(b) Mapping of head and subgoals induced by θ.

Figure 3.10: Example of MODCON transformation.

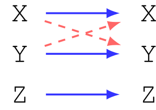
occurrences of variable `Y` are renamed to `X`.

Algorithm. The algorithm is shown in procedure MODCON of Alg. 4. As a first step, we extract all variables *vars* in Q (line 8). If there are fewer than two, we return `none` (lines 9–10). Otherwise, two (different) variables v and w are randomly selected from *vars* (lines 11–12), and we replace all occurrences of v in Q with w to get Q' (line 13). This ensures that new joins are introduced.

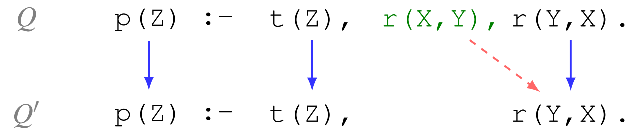
3.7. REM Transformations

The REM transformations remove a subgoal $R(v_1,\ldots,v_n)$ from a rule Q. Analogously to the ADD transformations, when removing the subgoal, there is *always* a containment mapping σ from Q' to Q, i.e., $Q \subseteq Q'$. This is because the head of Q' is the head of Q, and each subgoal of Q' is in Q.

REMEXP. This transformation checks the existence of a containment mapping from Q to Q'. If such a mapping does *not* exist, then $Q' \not\subseteq Q$ and $Q \subseteq Q'$ (due to σ), that is, the result of Q is expanded. Note that, in general, the problem of checking query containment is NP-complete. However, we can design a containment checker with linear-time complexity because Q' is derived from Q by removing one subgoal. Therefore, it is only necessary to check whether this subgoal of Q



(a) No containment mapping from Q to Q'.



(b) Dropped subgoal may not be mapped to any subgoal in Q'.

Figure 3.11: Example of REMEXP transformation.

may be mapped to any subgoal of Q'.

Example. Consider the example in Fig. 3.11. Removing the second subgoal of Q (shown in green) prevents the existence of a mapping from Q to Q' since it would require each of the variables `X` and `Y` of Q to be mapped to more than one variable of Q'. Consequently, this is a successful REMEXP transformation.

Algorithm. The algorithm for this transformation is shown in procedure REMEXP of Alg. 5. First, we randomly select a subgoal g from the body of Q (line 13) and remove it to get Q' (line 15). We then check the existence of a containment mapping from Q to Q' (line 16). This is done by simply checking if the removed subgoal g may be mapped to any subgoal in Q'. If no such mapping exists, then we return transformed rule Q', otherwise we return `none`.

The algorithm for checking the existence of a containment mapping from Q to Q', where Q' is derived from Q by removing a subgoal g is shown in procedure EXISTSCONTAINMENT of Alg. 5. As a first step, we extract all variables *vars* in Q and *vars*$'$ in Q' (lines 2–3). We then compute the set of removed variables *rmVars* (line 4). Function REPLACEWITHWILDCARD (line 5) creates a pattern expression p from g such that the first occurrence of each variable in g is replaced with a wildcard if the variable is also in *rmVars*. Any subsequent occurrences of the same variable are replaced with a back-reference to the first match; this ensures that equality constraints between variables are captured. In the example of Fig. 3.11, *rmVars* is empty, so p is g, that is, `r(X,Y)`. If any g' in the body of Q' matches this pattern, then g may be mapped to a subgoal in Q' and a mapping exists (lines 10–15). Otherwise, a mapping does not exist (line 10) as in Fig. 3.11.

Algorithm 5: REM transformations

```
1  procedure EXISTSCONTAINMENT(Q, Q′, g)
2     vars ← EXTRACTALLVARS(Q)
3     vars′ ← EXTRACTALLVARS(Q′)
4     rmVars ← vars \ vars′
5     p ← REPLACEWITHWILDCARD(g, rmVars)
6     head′, body′ ← Q′
7     for each g′ ∈ body′ do
8        if g′ matches p then
9           return true
10    return false
11 procedure REMEXP(Q)
12    head, body ← Q
13    g ← RANDSUBGOAL(body)
14    Q′.head ← head
15    Q′.body ← body \ {g}
16    if ¬ EXISTSCONTAINMENT(Q, Q′, g) then
17       return Q′
18    return none
```

REMEQU. If, after removing a subgoal of Q, a containment mapping θ from Q to Q' does exist, then we have a REMEQU transformation because both $Q' \subseteq Q$ and $Q \subseteq Q'$ hold. The former holds due to θ and the latter due to σ.

Example. Fig. 3.12 shows an example of a REMEQU transformation.

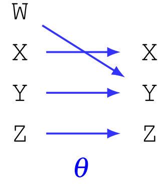
Algorithm. This algorithm is analogous to the one for REMEXP (see Alg. 5). For this transformation however, we return Q' when a containment mapping from Q to Q' *does* exist, i.e., EXISTSCONTAINMENT on line 16 is not negated. In the example of Fig. 3.12, *rmVars* is a singleton containing variable `W`, thus pattern p on line 5 of Alg. 5 is `r(*,X)`, where `*` is a wildcard. Subgoal `r(Y,X)` in Q' matches this pattern, and as a result, a mapping exists.

REMCON. Analogously to ADDEXP, this transformation can only be the same as REMEQU.

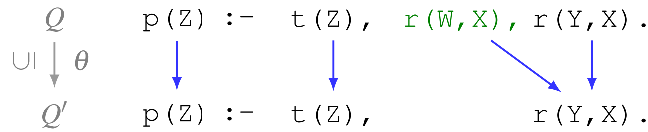
3.7.1. Transformation Sequences

Until now, we have focused on applying a single transformation to a single rule, which is a conjunctive query. However, our approach is also able to apply sequences of transformations to such a rule.

More specifically, a rule macro-transformation T may be composed of a sequence of micro-transformations $[t_1, \ldots, t_n]$ as the ones that we described so far. However, every micro-transformation $t_i \in T$ must preserve the intended oracle for the rule (i.e., EXP, EQU, CON). In particular, for an expanding macro-



(a) Containment mapping θ from Q to Q'.



(b) Mapping of head and subgoals induced by θ.

Figure 3.12: Example of REMEQU transformation.

transformation T_{EXP}, in which $Q \subseteq Q'$, the sequence of micro-transformations may have oracles EQU or EXP, but not CON. Analogously, for a contracting macro-transformation T_{CON}, in which $Q' \subseteq Q$, the micro-transformations may have oracles EQU or CON, but not EXP. For an equivalent macro-transformation T_{EQU}, in which $Q \equiv Q'$, all micro-transformations must also have EQU oracles.

In the following section, we generalize our approach from a single conjunctive query to a Datalog program, containing rules that are not necessarily CQs.

3.8. Beyond Conjunctive Queries

Let us first show how the oracle of a rule (macro-)transformation generalizes to any *positive*-Datalog program, i.e., any program without negation. To do this, we need to explain *monotonicity* of conjunctive queries. Intuitively, when adding more entries to a database instance, a monotonic query never produces a smaller result.

Definition 5 (Monotonicity)**.** *A conjunctive query Q over a database schema $\mathbb{R}$ is monotonic iff, for every two instances I and J of $\mathbb{R}$, it holds that $Q(I) \subseteq Q(J)$ when $I \subseteq J$.*

In a program P, the output relation is called a *Datalog query*, Q_P. Suppose our approach transforms a rule Q in P to get new rule Q', and therefore, new

program P' and Datalog query Q'_P. Now, the same oracle that should hold between Q and Q' should also hold between Q_P and Q'_P. This is because, in positive Datalog, all rules are monotonic. Therefore, due to the fixed-point computation, any change in the result of Q propagates monotonically to all rules that (directly or transitively) depend on Q. Ultimately, this includes the final Datalog query, and thus, the program result. Consequently, we may "lift" our oracles from individual conjunctive queries to full-blown positive-Datalog programs. Naturally, this also allows us to transform more than one rule in a positive-Datalog program as long as all transformations have the same intended oracle.

Let us now explain how our approach handles *any* Datalog program (not only positive ones). Of course, the EQU oracle trivially extends to any program. However, queryFuzz is able to accept any Datalog program for *all* oracles: it enforces that all rules depending on a transformed rule Q' are monotonic (e.g., they do not contain any negated subgoals). Intuitively, should the result of a rule Q' "flow" into a non-monotonic rule, the effect on the program result could be "flipped", for instance, it could be contracted instead of expanded. This is undesirable as it could lead to false positives. To handle negation, existing Datalog engines impose a computation order on relations. More specifically, relations are assigned to *strata* via a process known as *stratification* [25, 225]. Lower strata are computed before higher ones during the fixed-point computation. Therefore, queryFuzz works on any Datalog program by only transforming rules that are in a higher stratum than any rules containing negation. As a consequence, no results of transformed rules can "flow" into non-monotonic rules.

Note that many Datalog dialects support rules with more expressive language features, such as comparison operators, aggregate functions, disjunctions, and recursion. While our transformations target the restricted subset of pure conjunctive queries (see Sect. 3.3), they may also be applied to more expressive dialects as long as the monotonicity constraints described above are maintained. In fact, our implementation does handle such dialects.

Based on the above, in the rest of this section, we present another transformation in queryFuzz, which is specific to Datalog programs (unlike the transformations in Sect. 3.4, which target CQs in general).

3.8.1. NEG Transformation

A NEG transformation changes a program P into an equivalent but further stratified program P' by introducing a double negation in a rule Q. In particular, introducing a negation causes the Datalog engine to split a stratum in two. When this negation is double, we guarantee the EQU oracle (i.e., the transformation is NEGEQU).

We introduce so-called *safe* negations, i.e., every variable in a negated subgoal must also appear in a positive subgoal. (Unsafe rules are traditionally not allowed in Datalog as they do not restrict all variables to finite domains.) As an example, consider:

```
p(X,Y) :- a(X,Y), b(Y,Z), c(Z).     // Q
```

NEGEQU selects a subgoal g in Q, say `c(Z)`, and replaces it with a new negated subgoal, say `!neg(Z)`. Relation `neg` is defined to have the same body as Q but with a negated g, thus introducing a double negation:

```
neg(Z) :- a(X,Y), b(Y,Z), !c(Z).
p(X,Y) :- a(X,Y), b(Y,Z), !neg(Z). // Q'
```

One can easily see that queries Q and Q' are equivalent when thinking about the transformation logically: $\mathtt{a} \wedge \mathtt{b} \wedge \neg\, \mathtt{neg} \equiv \mathtt{a} \wedge \mathtt{b} \wedge (\neg\, \mathtt{a} \vee \neg\, \mathtt{b} \vee \mathtt{c}) \equiv \mathtt{a} \wedge \mathtt{b} \wedge \mathtt{c}$. Such a transformation partitions the original stratum of relation `p` into two, where the stratum of `p` is strictly greater than that of `c`. Note that Datalog traditionally disallows NEG transformations when g (in this case `c`) has a cyclic dependency on the head of Q (in this case `p`), which would require them to be defined in the same stratum.

3.9. Implementation

We implemented queryFuzz in a total of 5,300 lines of Python. It supports three Datalog dialects, namely Soufflé [136], bddbddb [260] (used by μZ), and DDlog [228]. In the rest of this section, we discuss how we implement the bug-detection stage of our approach.

Bug detection. During bug detection, queryFuzz compares the result of a program (`gen` in Fig. 3.3) with that of its transformation (`exp`, `equ`, or `con` in the figure). However, a program result could potentially contain millions of entries. This is especially true for randomly generated programs. To efficiently check an oracle, queryFuzz uses Datalog rules that decide result containment.

For instance, the rules that check EQU oracles are the following:

```
equ1(Z) :- gen(Z), !equ(Z).
equ2(Z) :- equ(Z), !gen(Z).
```

The first rule checks whether $\mathtt{gen} \subseteq \mathtt{equ}$ and the second whether $\mathtt{equ} \subseteq \mathtt{gen}$. A bug is detected if the result of either `equ1` or `equ2` is non-empty.

3.10. Experimental Evaluation

In this section, we address the following research questions:

RQ1: How effective is queryFuzz in detecting previously unknown query bugs in Datalog engines?

RQ2: Is the number of detected bugs significant?

RQ3: How deep are the detected bugs?

RQ4: What are characteristics of the detected bugs?

Table 3.1: **Query bugs detected by queryFuzz.**

Bug ID	Datalog Engine	Metamorphic Transformations	Bug Status
1	Soufflé	ADD	fixed
2	Soufflé	REM, REM, REM, MOD	fixed
3	Soufflé	MOD, ADD, ADD	confirmed
4	Soufflé	NEG	fixed
5	Soufflé	MOD, ADD, ADD	confirmed
6	Soufflé	MOD, ADD, MOD, REM	confirmed
7	Soufflé	REM, MOD, ADD	confirmed
8	Soufflé	ADD, ADD, MOD	confirmed
9	μZ	ADD, MOD, ADD	fixed
10	μZ	ADD, ADD, ADD, MOD	fixed
11	μZ	ADD, MOD	fixed
12	μZ	MOD, ADD	confirmed
13	DDlog	ADD, ADD, ADD	confirmed

RQ5: How efficient is queryFuzz?

3.10.1. Experimental Setup

We tested Soufflé, μZ, and DDlog, three popular and mature Datalog engines that are publicly available on GitHub. We completed the development of the first version of queryFuzz, with a subset of the metamorphic transformations and limited support for different language features, in May 2020, and initially focused on testing Soufflé. We only added support for the dialects of μZ and DDlog in late Dec 2020 to evaluate the generality of our transformations.

To avoid burdening developers and reporting duplicate issues, we only filed reports for bugs that were clearly different than the ones we had already reported until these were fixed. Of course, this hinders bug reporting, but it was greatly appreciated by the developers.

3.10.2. Experimental Results

We now discuss our experimental results for each of the above research questions.

RQ1: Query bugs. Tab. 3.1 shows the list of *unique* query bugs detected by queryFuzz in the Datalog engines we tested. Note that we confirmed bug uniqueness with the engine developers themselves. The first column of the table assigns an identifier to each bug; all identifiers link to the corresponding bug reports on the GitHub issue tracker of each engine. The second column of the

table shows the engine in which the bug was found, the third the sequence of metamorphic transformations that revealed the bug, and the last column shows the current status of the bug (i.e., open, confirmed, or fixed).

Overall, queryFuzz detected 13 previously unknown query bugs in all three engines. All bugs have been confirmed by the developers, and 6 have already been fixed. Bugs 3 and 5 are labeled as questions on the issue tracker even though developers have confirmed them. The reason is that they reveal a deep semantic issue in logic programming that cannot be easily addressed (see RQ4). As shown in the third column of the table, each of our metamorphic transformations (i.e., ADD, MOD, REM, and NEG) contributed to detecting at least one query bug. Moreover, the fact that each tested engine implements its own Datalog dialect speaks to the generality of these transformations. Note, however, that our public bug reports do not show all the applied transformations as we tried to localize issues as much as possible and aid developers in debugging; our tool repository[4] contains instructions on how to reproduce all bug-revealing transformations.

In addition to query bugs, queryFuzz also detected several crash bugs as a by-product; they are shown in Tab. 3.2. Even though such bugs are less critical, they expose robustness issues, and developers were still interested in them. In fact, a developer of Soufflé said: "*Bug reports like this are definitely welcome, especially because they might also point to other potential issues in our setup. [These issues] have already been super useful.*"

In general, we found many more bugs in Soufflé in comparison to the other engines. However, this does not necessarily mean that Soufflé is more buggy. A reason is that we tested it for a longer period of time (see Sect. 3.10.1). Another reason is that the Z3 developers generally have very limited bandwidth to devote to μZ as they are working on a new core SMT engine—we, therefore, decided against filing more bugs for the time being. In addition, DDlog is quite slow as it compiles every input program into a Rust project; this also slows down the testing process (see RQ5).

RQ2: Significance of bug numbers. To evaluate the significance of our bug-finding results, we compare the number of query bugs detected by queryFuzz to the total number of such bugs reported from May 1, 2020 to Feb 15, 2021. For this research question, we consider all three engines, and we collect the total number of reported bugs from their GitHub issue trackers. We inspect issues since May 1, 2020 because this is when we started testing Soufflé.

The results are shown in Fig. 3.13. In the considered time period, a total of 16 query bugs were reported in the three Datalog engines we tested, and queryFuzz detected 13 of them (81%). This ratio, though very high, is not surprising since query bugs are very hard to detect without an oracle. In the same time period, 41 crash bugs were reported, and queryFuzz detected 14 of them (34%) as a by-product.

RQ3: Bug depth. To understand the depth of the detected bugs, we analyzed

[4] https://github.com/Practical-Formal-Methods/queryFuzz

Table 3.2: **By-product bugs detected by queryFuzz.**

Bug ID	Datalog Engine	Bug Type	Bug Status
14	Soufflé	floating-point exception	confirmed
15	Soufflé	aborted evaluation	fixed
16	Soufflé	segmentation fault	fixed
17	Soufflé	segmentation fault	fixed
18	Soufflé	segmentation fault	fixed
19	Soufflé	segmentation fault	fixed
20	Soufflé	segmentation fault	fixed
21	Soufflé	assertion failure	fixed
22	Soufflé	assertion failure	fixed
23	Soufflé	assertion failure	fixed
24	Soufflé	assertion failure	fixed
25	Soufflé	assertion failure	confirmed
26	Soufflé	compiler error	fixed
27	μZ	performance bug	fixed

all Soufflé bugs together with the engine developers. In general, they revealed issues across the stack.

The Soufflé engine essentially consists of the following components, from front- to back-end: (1) ASTGEN for parsing and abstract-syntax-tree (AST) generation, (2) ASTOPT for AST analysis and optimization, (3) ASTRAM for translation from AST to relational-algebra machine (RAM), (4) RAMOPT for RAM optimization, and (5) INTSYN for interpretation or synthesis. The interpreter evaluates its RAM input, whereas the synthesizer translates RAM into C++ code, which is then compiled and executed.

Tab. 3.3 categorizes all Soufflé bugs into the engine component in which they were found—ignore bugs A, B, C, D, and E for now. Note that no bugs were found in ASTGEN and that we include a row INFRA, referring to infrastructure bugs, e.g., in utilities, that could affect the entire stack. As shown in the table, queryFuzz detected bugs in all components except ASTGEN, which is the most shallow.

We compare the depth of these bugs with that of bugs detected using of-the-shelf fuzzers and reported from May 1, 2020 to Feb 15, 2020. There were 6 such bugs, 3 of which were detected with Radamsa [8] and the other 3 with AFL [11]. One of the Radamsa bugs[5] was not confirmed by the developers, who labeled it as 'wontfix'. The other 5 bugs are shown in Tab. 3.3 as A, B, C, D, and E. They revealed issues in the ASTGEN and ASTOPT components of Soufflé, which are

[5]https://github.com/souffle-lang/souffle/issues/1757

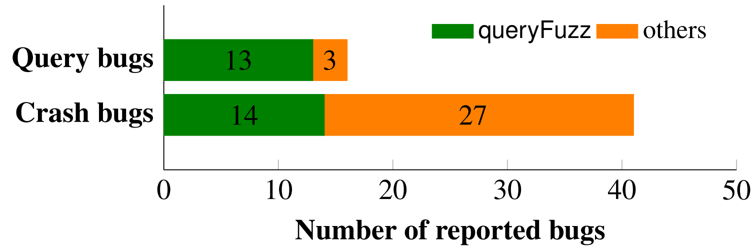


Figure 3.13: All bugs reported in the three Datalog engines from May 1, 2020 to Feb 15, 2021.

at the top of the stack—for instance, bugs A and E crash the engine during, or even before, parsing. The reason why queryFuzz bugs are much deeper is that it generates valid Datalog programs and its oracle-driven transformations are more likely to reveal semantic issues. Note that we do not further compare our approach with other off-the-shelf fuzzers as they are not able to detect query bugs due to lack of oracles.

RQ4: Bug characteristics. To demonstrate the nature of the detected bugs, we now provide a few interesting bug samples.

Bug 1 was found in Soufflé's ASTOPT component, and specifically, in the minimization pass that simplifies the program by removing equivalent rules and subgoals. This pass missed a corner case for singleton relations, i.e., with arity 1. The program and transformation that revealed this bug are discussed in Sect. 3.2 (see Fig. 3.5). Bug 4 was detected in the same component, but in its magic-set transformation [25, 28, 35, 225], which aims to derive only those facts that are relevant for the program's Datalog query. Our approach revealed this bug using a NEG transformation. The rule in which the negation was introduced depended on another rule containing a comparison operator, which in turn caused a mislabeling

Table 3.3: Categorization of Soufflé bugs into the components in which they were found.

Soufflé Component	Bug IDs
ASTGEN	A, E
ASTOPT	1, 2, 3, 4, 5, 15, 17, 20, 21, 25, B, C, D
ASTRAM	8, 22, 23, 24
RAMOPT	19
INTSYN	6, 7, 14, 26
INFRA	16, 18

```
1 PQRI(v) :- Z(v), Z(nbj).
2 PLEY(o) :- PQRI(x), Z(o), Z(x).
3 NFUV(q) :- Z(fym), PLEY(q).
4 OUT(t) :- NFUV(ssz), PLEY(arv), PLEY(t).
```

Figure 3.14: Generated program snippet for testing DDlog.

of relations as positive. Naturally, correct positive labeling is essential to the stratification process. Bug 2 revealed another issue in the magic-set transformation. In general, developers mentioned that implementing optimization passes on the AST is quite complex for a feature-rich Datalog dialect. They also expressed the need for verifying the correctness of such passes, as done by Bégay *et al.* [36].

According to developers, bugs 3 and 5 reveal an important semantic issue in logic programming. There is no clear execution order of instructions, which may result in numerical-stability issues in the presence of floating-point numbers. For these bugs to be fixed, the developers would have to build symbolic machinery that dictates the order of optimizations and instructions such that numerical stability is maximized. However, this is an open research problem, which is why these bugs were labeled as questions.

Bug 7 was detected in Soufflé's INTSYN component, at the very bottom of the stack. According to the developers, the problem lies in a data-structure representation for relations, namely brie [137], which does not properly implement element insertion and count. This bug existed at least since an old release of Soufflé (of 1.5 years ago at the time). A developer commented about this bug: "*I don't know how it could have been missed until now, but that's the first time I've seen anyone point this out.*" Bug 6 revealed a different issue with the same data structure; in this case, the computation of lower and upper bound values of its elements was incorrect.

Bug 8 was found in the ASTRAM component of Soufflé. Our transformation caused a silent internal failure in this component, which manifested itself with an incorrect result. A developer commented: "*Well spotted! Great work!*"

Bug 13, was detected in DDlog after randomly generating the program in Fig. 3.14 and then adding two subgoals to rule PLEY:

```
PLEY(o) :- PQRI(x), Z(o), Z(x), PQRI(z), PQRI(x).
```

The original program repeatedly computes Cartesian products of the different relations and generates a non-empty result. However, after the above transformation, which should preserve the original result, the new program generates an empty result. This is because DDlog stores all intermediate relations as multi-sets, where the multiplicity of each element is the number of times it was derived. Currently, multiplicities are stored as 32-bit integers to reduce the memory footprint of the program, and the above transformation caused an integer overflow, manifesting itself as an empty result. This bug was confirmed by the developers, who are considering several solutions to the problem, such as using 64-bit integers to

store multiplicities, internally converting multi-sets to sets using the `distinct` operator in Rust, or statically analyzing the program to estimate the number of derivations.

RQ5: Performance. Regarding the performance of queryFuzz, it expectedly varies significantly depending on the tested Datalog engine. On average, Soufflé requires 0.078 seconds to run a test (12.9 tests per second) in interpreter mode and 12 seconds in synthesizer mode, DDlog needs 1.2 minutes per test, and μZ 0.1 seconds (10 tests per second). On average, the first stage of queryFuzz generates 47.6 programs per second, and the second stage performs 303 transformations per second. As shown from these numbers, the performance bottleneck are the engines themselves.

3.11. Threats to Validity

We identified two threats to the validity of our experiments.

Selection of seeds. Our approach may use seeds as input, and its effectiveness in bug finding could depend on their selection. However, we used non-empty seeds only when testing Soufflé, and we selected all of its semantically valid regression tests[6]. Our seed selection is, therefore, sufficiently broad to mitigate this threat. Moreover, queryFuzz does not require non-empty seeds as, in their absence, it generates random Datalog programs (see Sect. 3.2). In fact, 7 of the detected bugs were found using non-empty seeds.

Selection of Datalog engines. The detected bugs also depend on our selection of Datalog engines. However, we chose three mature engines, which even support different dialects, to mitigate this threat and demonstrate the generality of our approach.

3.12. Related Work

In this chapter, we present the first testing approach for detecting query bugs in Datalog engines. It uses metamorphic testing to solve the common problem of finding a suitable oracle [31] taking inspiration from query optimization in database theory. Of course, query optimization has been studied in other domains as well, such as in Datalog or Prolog (e.g., [108, 215, 230, 257]). However, optimization targets a goal different than ours, that of finding an equivalent query that performs faster. In contrast, queryFuzz tests Datalog engines by exploring a state space of queries that are not necessarily equivalent, let alone more optimal. In the following, we focus on testing work from related areas, such as database systems, compilers, and program analyzers.

Metamorphic testing. Metamorphic testing [66] is an effective technique to test

[6] We selected all tests in the 'evaluation', 'example', and 'semantic' folders under https://github.com/souffle-lang/souffle/tree/master/tests.

software systems without user-provided oracles. It works by mutating test cases via metamorphic relations that allow inferring the expected output of the mutated test cases. Over the years, it has been used to test a variety of systems, from web services [61], over compilers [153], to machine-learning applications [265]. Segura *et al.* [235] conducted a comprehensive survey on metamorphic testing in different domains.

Testing database systems. Database-management systems lie at the heart of most large-scale software applications today. Ensuring their correctness and robustness is of critical importance and has been a focus of many researchers and practitioners for decades.

In 1998, Slutz [244] proposed a technique, based on differential testing, to detect bugs in database systems. Another approach—also based on differential testing—was used by Jinho *et al.* to detect performance bugs [139]. Jepsen [146], developed by Kingsbury, is a practical tool for detecting safety bugs in distributed database systems; these can occur due to asynchronous interactions between components, data loss due to networking issues, node failures, etc. Recently, Rigger and Su proposed a series of testing techniques [221–223], which they implemented in a tool called SQLancer. Their tool detected hundreds of bugs in various relational database systems.

Fuzzing is also applied to detect crashes and other robustness issues in database systems. For instance, SQLsmith [236] is a popular SQL-query generator that has detected hundreds of crashes in widely used database systems. Other query-generation approaches include ones relying on constraint solvers [53, 143, 144, 192, 212, 255], symbolic execution [42, 163], and reverse query processing [41].

Testing compilers. Compiler testing is another important and active research area [65, 159, 246, 266]. Le *et al.* proposed a metamorphic-testing technique [153], known as equivalence modulo inputs (EMI), which mutates a seed program to generate equivalent programs. The technique and its extensions [154, 247, 272] have detected hundreds of bugs in GCC and Clang. A related approach was also used to test graphics shader compilers [90, 159]. Livinskii *et al.* recently developed a technique for generating expressive programs without undefined behavior to test C and C++ compilers [162]. The programs are then compiled using different compilers, and their outputs are compared to detect bugs. Recently, such techniques have also been extended to compilers for specialized domains, such as deep learning [161, 264] and quantum computers [206].

Testing program analyzers. Work on detecting bugs—in particular soundness bugs—in implementations of program-analysis techniques [57] has received significant attention in recent years. Various different approaches have been proposed to test a wide range of analysis techniques, such as model checking [271], abstract interpretation [148], symbolic execution [141], or dataflow analysis [248], as well as their underlying components, such as abstract domains [56] or constraint solvers [55, 169, 252, 262, 263].

3.13. Summary and Remarks

We have presented the first approach for metamorphic testing of Datalog engines. Our tool, queryFuzz, detected 13 previously unknown query bugs in three different engines. Query bugs are critical since, unlike crashes, they typically remain undetected. Given that Datalog is frequently used to formalize and implement security analyses or verification tools, such bugs can be catastrophic. As a result, we received overwhelmingly positive reactions from engine developers about the bugs we reported, several of which revealed deep—sometimes even fundamental—issues.

Chapter 4

Dependency-Aware Metamorphic Testing of Datalog Engines

In this chapter, we present another powerful metamorphic testing technique to find query bugs in Datalog engines. The technique uses rich precedence information capturing dependencies among relations in a Datalog program. This enables much more general and effective metamorphic transformations than the ones presented in chapter 3. We implement our approach in a tool called DLSmith, which detected 16 previously unknown query bugs in four Datalog engines.

4.1. Introduction

As discussed in chapter 3, Datalog engines are complex, especially since they typically employ advanced query transformation, optimization, and compilation techniques to improve their performance and scalability. As a result of this complexity, Datalog engines are prone to query bugs. A query bug causes the engine to return incorrect results that, for example, contain more, fewer, or different entries than they should. These bugs are severe—they may compromise the soundness of an upstream program analyzer, leading to catastrophic consequences in safety-critical settings.

In chapter 3, we presented the first ever metamorphic testing approach to detect query bugs in Datalog engines and implemented it in a tool called queryFuzz. The technique, however, is limited in the metamorphic transformations it can perform. In particular, it selects an existing rule in a given Datalog program and carefully modifies it without considering the surrounding program. More specifically, its transformations only consist in adding an atom to a rule, removing an atom from a rule, or modifying a rule variable. queryFuzz requires that such changes do not introduce negation and may only be performed for a specific set of rules (i.e., those at the highest stratum) such that the resulting program is still valid. In short,

transformations in queryFuzz are limited to ones that can be performed locally without considering the entire program.

The technique introduced in this chapter overcomes these limitations by inferring an *annotated precedence graph* for a given Datalog program, capturing rich information about any dependencies among program relations. Hence, this graph provides a global view of the program, thereby allowing for more radical transformations, including adding entirely new rules, removing existing rules, and handling negation. At the same time, our approach incorporates all existing queryFuzz transformations. In other words, we significantly extend the range of possible transformations, and thus, increase the effectiveness of metamorphic testing in finding query bugs in Datalog engines. Moreover, by defining all our transformations on the annotated precedence graph, our approach can easily support many Datalog dialects.

We implemented this approach in our tool called DLSmith, which we used to test six Datalog engines, each supporting a different dialect. DLSmith detected 16 previously unknown query bugs in four of these engines. All bugs were confirmed and eleven were fixed; only two could have been found by queryFuzz. An engine developer commented: *"The bugs found are hidden deep inside the query plan generation and optimization pipeline. Due to the complexity of Datalog rules and data for triggering the error, the bugs are very hard to find with regular unit testing. Automatic tools are extremely helpful to identify the issue and improve the robustness of our Datalog engine."*

Contributions. This chapter makes the following contributions:

1. We present the most comprehensive and effective metamorphic testing approach for detecting query bugs in Datalog engines to date.
2. We implemented our approach in a publicly available tool called DLSmith.
3. We evaluated DLSmith on six Datalog engines supporting different dialects; our tool detected 16 previously unknown query bugs in four of these engines and received very positive feedback from their developers.

Outline. The next section provides necessary background. Sect. 4.3 gives an overview of our approach, while Sects. 4.4 and 4.5 explain the technical details of its key components. In Sect. 4.6, we describe the implementation of DLSmith. We present our experimental evaluation in Sect. 4.7, discuss related work in Sect. 4.8, and conclude in Sect. 4.9.

4.2. Background

In this section, we give an overview of Datalog programs and their associated precedence graphs.

4.2.1. Datalog Programs

Rules. A *term* is either a variable $x, y, z, \ldots$ or a constant $a, b, c, \ldots$. An *atom* is an expression of the form $R(\vec{U})$, where R is a *relation symbol* of arity m and $\vec{U}$ is an m-vector of terms, e.g., $M(x, y, a)$. A *ground atom* is an atom without variables, e.g., $M(a_1, \ldots, a_m)$, where a_i are constants. A *Datalog rule* is an expression of the form

$$R(\vec{U}) \leftarrow R_1(\vec{U}_1), \ldots, R_n(\vec{U}_n).$$

where $R_i(\vec{U}_i)$ for $1 \leq i \leq n$ are atoms. Note that atoms can refer to the same relation. The expression to the left of $\leftarrow$ is the *head* of the rule, and the expression to the right is the *body*. Any variable appearing in $\vec{U}$ must also appear in some $\vec{U}_i$. A relation R can have more than one rule, each of which is identified by a unique *rule number* k, where k ranges between 1 and the total number of rules for the relation.

Programs. As discussed in chapter 3, relation symbols are divided in two categories. First, there are *input relations* whose contents are given in the form of *facts* (ground atoms). These are commonly referred to as *extensional database (EDB) relations*. We use F to denote the set of facts. Second, there are *intensional database (IDB) relations* that are defined by Datalog rules, and one of them is specified as *output*. A Datalog program P is a finite set of facts and Datalog rules. P is *recursive* if a relation symbol appears in both the head and the body of a rule. For example, the following is a recursive Datalog program with three facts and two rules:

```
// facts (representing edges)
E(1,2). E(2,3). E(3,4).
// rules (computing the transitive closure of E)
C(x,z) :- E(x,z).
C(x,z) :- C(x,y), C(y,z).
```

The input to the program is `E` (EDB relation) and the output, `C` (IDB relation), represents the transitive closure of the edge relation `E`.

Stratified Datalog. A *stratification* of a Datalog program assigns a non-negative integer, called a *stratification number* or *stratum*, to every IDB relation in the program such that, for every rule, the following hold.

- For every positive (i.e., not negated) atom R_i in the rule body, the stratum of R_i is greater than or equal to the stratum of rule head R.
- For every negative (i.e., negated) atom R_i in the rule body, the stratum of R_i is strictly greater than the stratum of rule head R.

Stratification allows providing well defined semantics for evaluating Datalog programs [160], and consequently, most Datalog engines only support stratifiable programs. The evaluation of a program starts with the highest stratum, for which a fixpoint is computed. The computed results of any IDB relation in the highest

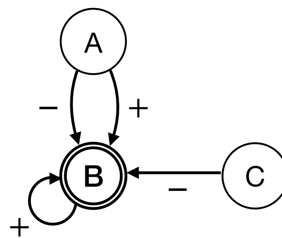
stratum are then used in the second highest stratum. The process is repeated until all strata are traversed.

P is a *stratified Datalog program* if it does not contain recursion involving negation. For example, the following program is not stratifiable because relation `P` negatively depends on relation `L`, which again depends on `P`:

```
L(a) :- M(a), P().
P(a) :- R(a), not L(a).
```

4.2.2. Precedence Graphs

A Datalog program *P* has an associated directed graph, called *precedence graph* and denoted by G_P. G_P has a node for each relation in the program and an edge from node *N* to *M* whenever a relation *N* is in the body of a certain rule and relation *M* is the head of the same rule. A precedence graph is, therefore, used



(a)

```
1 // declarations
2 edge(X:number, Y:number).
3 reachable(X:number, Y:number).
4 .output reachable
5
6 // facts
7 edge(1,2).
8 edge(2,3).
9 edge(4,2).
10 edge(2,5).
11
12 // rules
13 reachable(X,Y) :- edge(X,Y).
14 reachable(X,Z) :- edge(X,Y), reachable(Y,Z).
```

(b)

Figure 4.1: Precedence graph (a) for a simple Datalog program (b).

to capture dependencies between relations in the program. If a relation appears positively in the rule body, the corresponding edge is annotated with label $+$, otherwise with $-$. When P is non-recursive, its precedence graph is by definition acyclic. As an example, consider the program in Fig. 4.1 with its associated precedence graph.

Definition 6 (Precedence Graph)**.** *Given a Datalog program P, a precedence graph* $G_P = (V, E, \theta, \lambda)$ *is a directed, labeled hyper-graph, where V is a set of nodes. Each node in V represents a unique relation in P. Function* $\theta : Q \to V$ *assigns a relation in Q to a node in V, where Q is the set of all relations in P.* $E \subseteq (V x V)$ *is a set of directed edges. Function* $\lambda : E \to sign$*, where sign is* $\{+,-\}$*, assigns labels to edges.*

4.3. Overview

In this section, we give an overview of our approach (shown in Fig. 4.2), which is divided into four phases. On a high level, it uses a seed program to generate a random annotated precedence graph, applies metamorphic transformations on this graph, and compares the results of the programs corresponding to these two graphs.

The *first* phase takes as input a seed Datalog program S and produces a random precedence graph G_P. It does so by first extracting all relation symbols in S. For each relation symbol, it generates a graph node, called a *seed node*. Next, it randomly generates a number of new nodes, called *generated nodes*. G_P is then produced using these seed and generated nodes. Note, however, that no incoming edges are added to seed nodes, that is, the corresponding rules remain unchanged—this allows handling complex seed programs containing unique language features (e.g., SMT formulas in Formulog rules). Since program S in Fig. 4.2 consists of only one relation, G_P contains one seed node (`fib`); node `a` is randomly generated.

The *second* phase takes S and G_P as input and produces a corresponding program P as well as its result O_P. To achieve this, the graph annotator first uses G_P to generate an *annotated precedence graph* $\overline{G_P}$, which extends G_P by decorating its nodes and edges with properties. S and $\overline{G_P}$ are then used by the program generator to produce P, which is in turn executed to compute O_P. Note that the construction of $\overline{G_P}$ by the graph annotator ensures that P is stratifiable and passes all syntactic, semantic, and type checks of the target Datalog engine.

Under the hood, the program generator starts by creating a new relation for each node in $\overline{G_P}$. Relations created from seed nodes are called *seed relations*, and all others are *generated relations*. Rules for a seed relation are copied directly from S (since the corresponding seed node in $\overline{G_P}$ has no incoming edges). Rules for a generated relation are created based on the incoming edges of the corresponding node. Edge properties in $\overline{G_P}$ (*number*, *sign*, and *vars*) are directly reflected in program *syntax*. For example, when considering the edge from `fib` to `a`, its

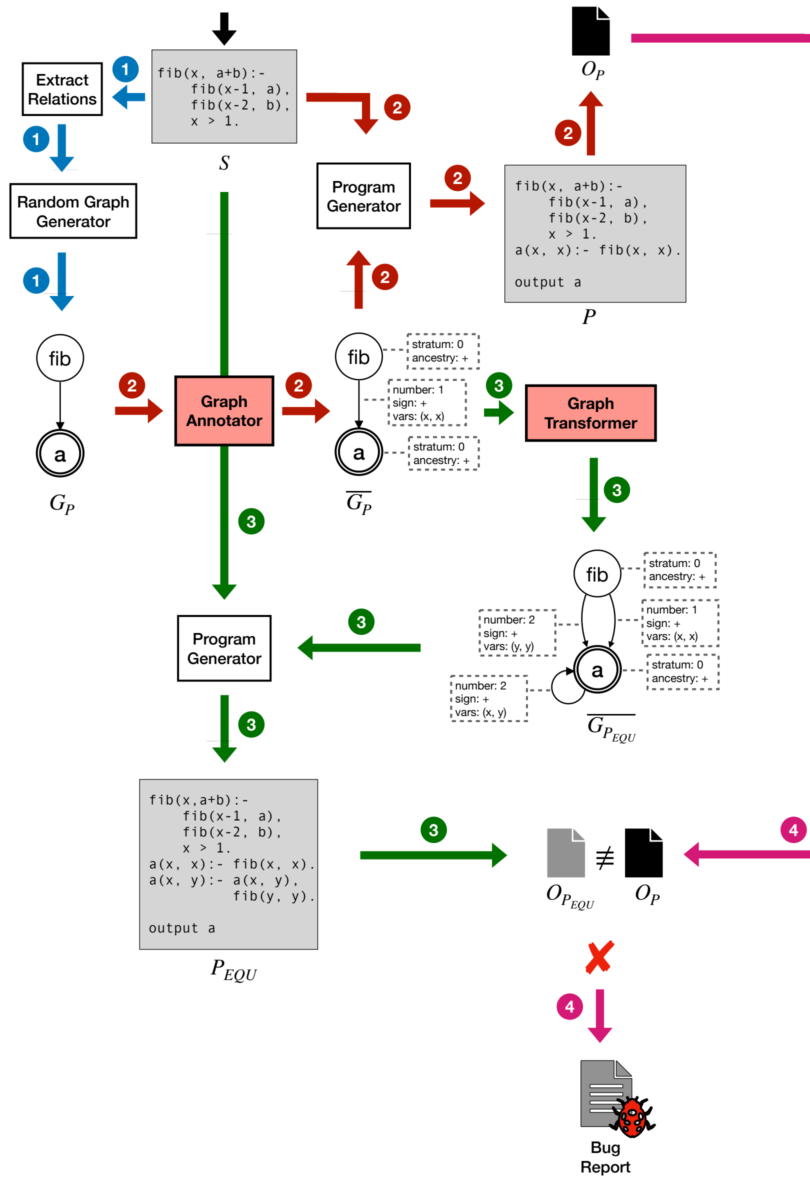


Figure 4.2: Overview of our approach.

properties denote that atom `fib(x,x)` (*vars*) appears positively (*sign*) in the first rule (*number*) for relation `a`. Node properties (*stratum* and *ancestry*) are *semantic*; they are computed using a lightweight static analysis on $\overline{G_P}$.

The *third* phase takes S and $\overline{G_P}$ as input and produces a new program P_{tr}, which constitutes a metamorphic transformation of P, as well as its result $O_{P_{tr}}$. In particular, we transform P to obtain P_{tr} such that $O_{P_{tr}}$ has a known relation with O_P. Examples of such relations are *equivalent*, *contracting*, and *expanding* transformations, i.e., $O_P \equiv O_{P_{EQU}}$, $O_P \supseteq O_{P_{CON}}$, and $O_P \subseteq O_{P_{EXP}}$. This is achieved

with the graph transformer, which applies graph rewrite rules on $\overline{G_P}$ to obtain $\overline{G_{P_{tr}}}$, while again ensuring that no incoming edges are added to seed nodes. Next, the program generator, which is the same as in the previous phase, converts $\overline{G_{P_{tr}}}$ into transformed program P_{tr}. In the end, P_{tr} is executed to compute $O_{P_{tr}}$. In Fig. 4.2, the graph transformer adds two edges from `fib` to `a` to generate a metamorphically equivalent annotated precedence graph $\overline{G_{P_{EQU}}}$.

The *fourth* phase compares results O_P and $O_{P_{tr}}$ according to oracle *tr*. A bug report is generated when the oracle does not hold.

The following two sections explain the key components of our approach in more detail, namely, the graph annotator and the graph transformer.

4.4. Graph Annotator

Recall that, in the second phase of our approach, the graph annotator takes as input a randomly generated precedence graph, G_P, and produces an annotated precedence graph, $\overline{G_P}$, by decorating the nodes and edges in G_P with *property-value pairs*. Such graphs are also known as *property graphs*.

Fig. 4.3 shows an annotated precedence graph for generating the program of Fig. 4.1. As in a standard precedence graph, we represent each relation symbol in the program by a node, called *relational node*. Each relational node maintains two properties: *stratum* and *ancestry*. The output relation in the program, which is a relational node in the graph, is additionally called an *output node*—the output node is shown with a double line in the figure. We call edges between relational nodes *relational edges*. Each relational edge maintains three properties: *number*, *sign*, and *vars*. We represent a ground atom (i.e., fact) by a distinct type of node, called *fact node*—fact nodes are shown with dotted lines in the figure. We associate each fact node with a relational node using a *fact edge*. We denote fact values as labels in fact nodes, e.g., label "`1,2`" for fact `A(1,2)`. Fact nodes and fact edges are property-less.

Definition 7 (Annotated Precedence Graph)**.** *Given a Datalog program P, an annotated precedence graph, denoted by $\overline{G_P} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, is a directed, attributed hyper-graph. V is the set of relational nodes, V_F the set of fact nodes, and $O \in V$ the output node. $E \subseteq (V x V)$ is the set of relational edges and $E_F \subseteq (V_F x V)$ the set of fact edges. There is a fact edge from* every *node $u \in V_F$ to some node $v \in V$. Function $\theta : Q \rightarrow V$ assigns a relation in Q to a relational node in V, where Q is the set of all relations in P. Similarly, $\theta_F : F \rightarrow V_F$ assigns a fact in F to a fact node in V_F, where F is the set of all facts in P. Relational nodes are assigned properties using function $\lambda : V x K^v \rightarrow vals^v$, where $K^v = \{stratum, ancestry\}$ is the set of node property keys and $vals^v$ the set of node property values such that $stratum \in \mathbb{N}$ and $ancestry \in \{+, -, ?, \texttt{none}\}$. For output node O, λ is defined to assign $stratum = 0$ and $ancestry = +$. Relational edges are assigned properties using function $\mu : E x K^e \rightarrow vals^e$, where $K^e = \{number, sign, vars\}$ is the set of edge property keys and $vals^e$ the set of edge*

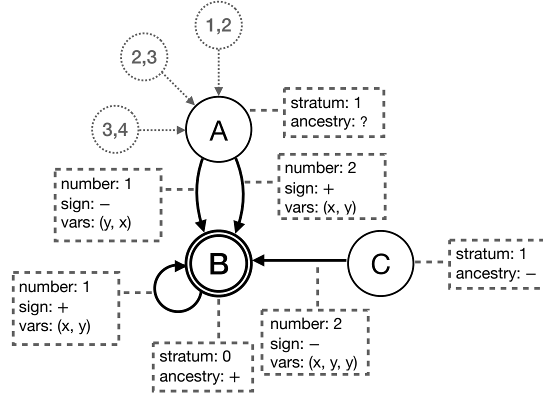


Figure 4.3: Annotated precedence graph for generating the program of Fig. 4.1.

property values such that number $\in \mathbb{N}$*, sign* $\in \{+,-\}$*, and vars is a tuple of variables and/or constants.*

On a high level, extending a standard precedence graph to an annotated one involves three steps: (1) randomly adding fact nodes and edges, (2) randomly generating properties for relational edges, and (3) inferring properties for relational nodes.

Fact nodes and edges. A fact node represents a ground atom of a relation. A fact edge e_f from a fact node F to a relational node N denotes that F represents a ground atom of relation N. In Fig. 4.3, we have three fact nodes connected to relational node A; these represent the three facts in the program of Fig. 4.1.

Relational-edge properties. As in standard precedence graphs, there is a relational edge e from a relational node N to a relational node M if N appears in the body of a rule r and M is the head of the same rule. Property *number* for e is k, where k is the rule number for r. *sign* is $+$ if N appears positively in r, otherwise it is $-$. *vars* is $\vec{U}_i$ if N is the ith atom in the body of r. Note that these properties are syntactic; they are later used by the program generator to produce a valid Datalog program.

In Fig. 4.3, we have an edge from B to B with property values $number = 1$, $sign = +$, and $vars = (x, y)$. In the program of Fig. 4.1, we therefore have a recursive relation B, where B appears positively in the first rule with variables (x, y). In Fig. 4.3, we also have an edge from C to B with property values $number = 2$, $sign = -$, and $vars = (x, y, y)$. As a result, in Fig. 4.1, relation C appears negatively in the second rule for B with variables (x, y, y).

We call an edge e *positive* if property $sign = +$, otherwise we call it *negative*. We call a path between two relational nodes in $\overline{G}$ a *dataflow path* (denoted by π). π from N to M is *positive* if the number of negative edges in π is even, otherwise π is *negative*. It is important to note that data flows monotonically along any dataflow path between N and M. In the case of a positive path, an increase or

decrease in data in N increases or decreases (although not necessarily strictly) the data in M, respectively. In the case of a negative path, an increase or decrease in data in N decreases or increases (although not necessarily strictly) the data in M, respectively.

Relational-node properties. Property *ancestry* for a node N is $+$ if all (possibly infinite) dataflow paths from N to output node O are positive; *ancestry* for N is $-$ if all (possibly infinite) paths from N to O are negative; *ancestry* is ? if there is at least one positive and one negative path from N to O; and *ancestry* is `none` if there is no dataflow path from N to O. We say that a node is in *positive ancestry* of O if *ancestry* is $+$ for that node, the node is in *negative ancestry* if *ancestry* is $-$, the node is in *unknown ancestry* if *ancestry* is ?, and the node is *not in the ancestry* if *ancestry* is `none`. We compute the value of *ancestry* for a node N by performing a backward depth-first traversal of $\overline{G}$ from O to N.

Property *stratum* for N is the stratification number of N. Essentially, it is the largest number of negative edges along any path from N to O in $\overline{G}$. Note that, in a stratified Datalog program, all relations have a finite stratum, that is, the precedence graph of a stratified program has no cycle that contains a negative edge. For example, in Fig. 4.3, C has $stratum = 1$ and $ancestry = -$ since there is a negative path from C to B with one negative edge. A has $ancestry = ?$ since there is at least one positive and one negative path from A to B.

Relational-node properties are semantic; they are computed by the graph annotator with a lightweight static analysis of $\overline{G}$ and are later used by the graph transformer to apply valid metamorphic transformations.

4.5. Graph Transformer

In this section, we define several primitive rewrite rules for annotated precedence graphs, introduce a methodology for specifying metamorphic transformations using these rules, and provide concrete example transformations.

4.5.1. Graph Rewrite Rules

Graph rewriting transforms a host graph, in our context the annotated precedence graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, by adding, removing, or altering the properties of graph elements (nodes or edges) using declaratively defined rules. A *graph rewrite rule* $\mathbb{R}(**g, **atr)$ is a variadic function, i.e., a function of indefinite arity, that takes as input a number of host graph elements (represented as $**g$) and rewrite attributes (represented as $**atr$); it returns a result graph $\overline{G_{tr}}$.

ADD rewrite rules

ADD rewrite rules transform $\overline{G}$ by adding a relational node, a fact node, or a relational edge between two existing relational nodes.

AddRelNode. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule adds a relational node n with property values *nvals* to $\overline{G}$; we denote it as $\mathbb{R}_{\text{AddRelNode}}(**g, **atr)$, where $**g = \{n\}$ and $**atr = \{nvals\}$. The result graph is $\overline{G_{tr}} = (V', V_F, O, E, E_F, \theta', \theta_F, \lambda', \mu)$, where $V' = V \cup \{n\}$ and θ', λ' only differ from θ, λ by including n.

AddFact. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule adds a fact node f and a fact edge e_f from f to relational node $v \in V$; we denote it as $\mathbb{R}_{\text{AddFact}}(**g, **atr)$, where $**g = \{f, v\}$ and $**atr = \emptyset$. The result graph is $\overline{G_{tr}} = (V, V'_F, O, E, E'_F, \theta, \theta'_F, \lambda, \mu)$, where $V'_F = V_F \cup f$, $E'_F = E_F \cup e_f$, and θ'_F only differs from θ_F by including f.

AddRelEdge. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule adds a relational edge e_r with property value *evals* from u to v, where $u, v \in V$; we denote it as $\mathbb{R}_{\text{AddRelEdge}}(**g, **atr)$, where $**g = \{u, v\}$ and $**atr = \{evals\}$. The result graph is $\overline{G_{tr}} = (V, V_F, O, E', E_F, \theta, \theta_F, \lambda, \mu')$, where $E' = E \cup e_r$ and μ' only differs from μ by including e_r.

Del rewrite rules

Del rewrite rules transform $\overline{G}$ by deleting a relational node, a fact node, or a relational edge between two existing relational nodes.

DelRelNode. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule deletes a node $v \in V$ along with its edges; we denote it as $\mathbb{R}_{\text{DelRelNode}}(**g, **atr)$, where $**g = \{v\}$ and $**atr = \emptyset$. The result graph is $\overline{G_{tr}} = (V', V_F, O, E', E_F, \theta', \theta_F, \lambda', \mu')$, where $V' = V \setminus v$, $E' = E \setminus E_v$ if E_v is the set of incoming and outgoing edges of v, and θ', λ' only differ from θ, λ by excluding v.

DelFact. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule deletes a fact node f associated with relational node $v \in V$ and the fact edge e_f from f to v; we denote it as $\mathbb{R}_{\text{DelFact}}(**g, **atr)$, where $**g = \{f, v\}$ and $**atr = \emptyset$. The result graph is $\overline{G_{tr}} = (V, V'_F, O, E, E'_F, \theta, \theta'_F, \lambda, \mu)$, where $V'_F = V_F \setminus f$, $E'_F = E_F \setminus e_f$, and θ'_F only differs from θ_F by excluding f.

DelRelEdge. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule deletes a relational edge $e \in E$; we denote the rule as $\mathbb{R}_{\text{DelRelEdge}}(**g, **atr)$, where $**g = \{e\}$ and $**atr = \emptyset$. The result graph is $\overline{G_{tr}} = (V, V_F, O, E', E_F, \theta, \theta_F, \lambda, \mu')$, where $E' = E \setminus e$ and μ' only differs from μ by excluding e.

Mod rewrite rules

Mod rewrite rules transform $\overline{G}$ by modifying property values of an existing relational node or edge.

ModRelNode. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule modifies the property values of a node $v \in V$; we denote it as

$\mathbb{R}_{\textsc{ModRelNode}}(**g, **atr)$, where $**g = \{v\}$ and $**atr = \{nvals\}$. The result graph is $\overline{G_{tr}} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda', \mu)$, where λ' only differs from λ by assigning property values *nvals* to v.

ModRelEdge. Given a graph $\overline{G} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu)$, this rule modifies the property values of an edge $e \in E$; we denote it as $\mathbb{R}_{\textsc{ModRelEdge}}(**g, **atr)$, where $**g = \{e\}$ and $**atr = \{evals\}$. The result graph is $\overline{G_{tr}} = (V, V_F, O, E, E_F, \theta, \theta_F, \lambda, \mu')$, where μ' only differs from μ by assigning property values *evals* to e.

4.5.2. Specifying Metamorphic Transformations

Recall that the graph transformer (see Fig. 4.2) applies graph rewrite rules on the annotated precedence graph $\overline{G_P}$ of program P to obtain $\overline{G_{P_{tr}}}$, which will then be converted into transformed program P_{tr}. Here, *tr* is the metamorphic relation that holds between the output of P (O_P) and that of P_{tr} ($O_{P_{tr}}$). Specifically, we have

- $O_P \equiv O_{P_{EQU}}$ for equivalent transformations,
- $O_P \supseteq O_{P_{CON}}$ for contracting transformations, and
- $O_P \subseteq O_{P_{EXP}}$ for expanding transformations.

We enforce relation *tr* by applying a graph rewrite rule $\mathbb{R}$ on a set of graph elements that satisfy a precondition ϕ. In general, we define a metamorphic transformation as a triple

```
assume(φ(**g))

**atr = G_P.generate_attributes(**arg)
G_P_tr = G_P.ℝ(**g, **atr)

assert(out(G_P) ≈_tr out(G_P_tr))
```

stating that if precondition ϕ holds for graph elements $**g$, then applying rewrite rule $\mathbb{R}$ on $**g$ establishes a relation *tr* between the output of P (`out(`$\overline{G_P}$`)`) and that of P_{tr} (`out(`$\overline{G_{P_{tr}}}$`)`). In this context, designing a metamorphic transformation essentially consists in defining precondition ϕ, one or more rewrite rules, and an attribute generation scheme implemented in method `generate_attributes`.

As an example, consider rewrite rule $\mathbb{R}_{\textsc{AddRelEdge}}(u, v, evals)$, which adds an edge e from relational node u to relational node v. In the transformed program P_{tr}, this means that a new atom is added in a rule for R, where $R = \theta^{-1}(v)$ is the relation corresponding to v. When $u \in V$, where V is the set of all relational nodes in $\overline{G_P}$, and $v \in V_{\texttt{none}}$, where $V_{\texttt{none}}$ is the set of all nodes that are not in the ancestry of the output node O, we have an equivalent (EQU) transformation. This

Table 4.1: Remaining metamorphic transformations implemented in DLSmith (grouped by oracles EQU, CON, and EXP).

Transformation	Description
EQU-ADDFACT	Adds a fact node to a relational node that is not in the ancestry of the output
EQU-DELFACT	Deletes a fact node from a relational node that is not in the ancestry of the output
EQU-DELRELNODE	Deletes a relational node that is not in the ancestry of the output
EQU-DELRELEDGE	Deletes an incoming relational edge from a node that is not in the ancestry of the output
CON-ADDFACT	Adds a fact node to a relational node that is in negative ancestry of the output
CON-DELFACT	Deletes a fact node from a relational node that is in positive ancestry of the output
CON-DELRELEDGES	Deletes all incoming relational edges from a node that is in positive ancestry of the output
EXP-ADDFACT	Adds a fact node to a relational node that is in positive ancestry of the output
EXP-DELFACT	Deletes a fact node from a relational node that is in negative ancestry of the output
EXP-DELRELEDGE	Deletes an incoming relational edge from a node that is in positive ancestry of the output

is because there is no dataflow path from v to O, and thus, when adding e from u to v, there is still no data flowing to O through e. We represent this transformation as follows.

$$\texttt{assume}(u \in \overline{G_P}.\texttt{get_all_nodes()} \bigwedge v \in \overline{G_P}.\texttt{get_nodes_with_ancestry(none)})$$

$$evals = \overline{G_P}.\texttt{generate_attributes}(u, v)$$
$$\overline{G_{P_{EQU}}} = \overline{G_P}.\mathbb{R}_{\text{ADDRELEDGE}}(u, v, evals)$$

$$\texttt{assert}(\texttt{out}(\overline{G_P}) \equiv \texttt{out}(\overline{G_{P_{EQU}}}))$$

Method `get_all_nodes` retrieves all nodes in the annotated precedence graph, whereas method `get_nodes_with_ancestry` retrieves all nodes with a particular ancestry, in this case `none`. Method `generate_attributes` returns values for properties *number*, *sign*, and *vars* of new edge e.

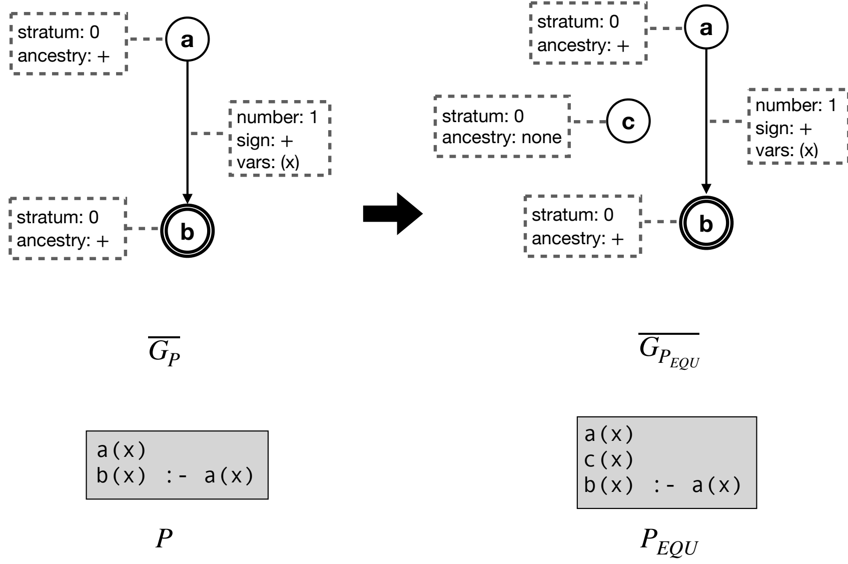


Figure 4.4: An example EQU-AddRelNode transformation.

4.5.3. Example Metamorphic Transformations

We now present a sample of the transformations implemented in DLSmith—these are the transformations that detected query bugs in the Datalog engines we tested. For the remaining transformations in DLSmith, see Tab. 4.1.

EQU-ADDRELNODE. This is an equivalent transformation adding a new relational node (see Fig. 4.4 for an example).

`assume(`$true$`)`

$$nvals = \{stratum : 0,\ ancestry : \texttt{none}\}$$
$$\overline{G_{P_{EQU}}} = \overline{G_P}.\mathbb{R}_{\text{ADDRELNODE}}(n, nvals)$$

`assert(out(`$\overline{G_P}$`)` $\equiv$ `out(`$\overline{G_{P_{EQU}}}$`))`

Here, $\{stratum : 0,\ ancestry : \texttt{none}\}$ is the implementation of `generate_attributes`. Recall that property *stratum* of a node is the largest number of negative edges along any path from the node to the output in the annotated precedence graph. Here, since *ancestry* is `none`, there is no path from the new node to the output, and thus, *stratum* must be 0.

EQU-ADDRELEDGES. This is an equivalent transformation adding a positive and a negative relational edge between two existing nodes (see Fig. 4.5 for an

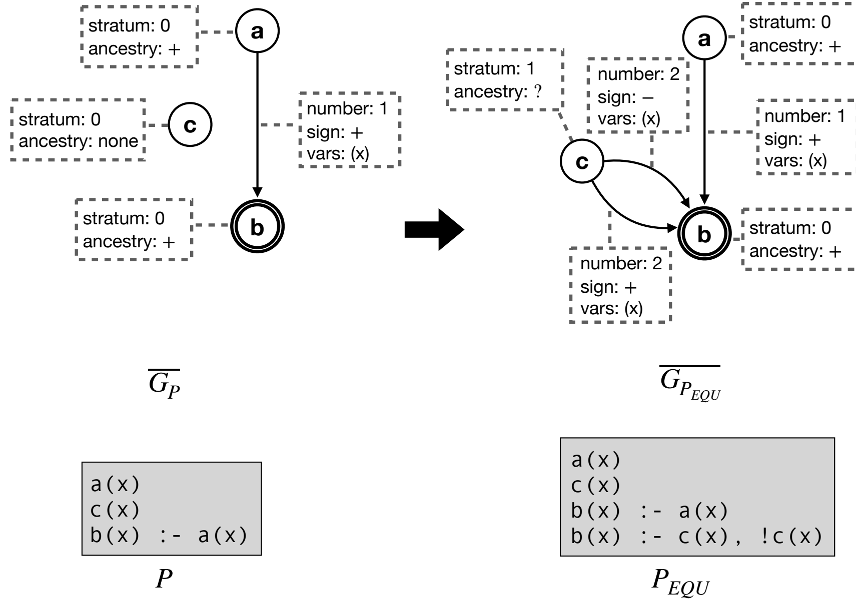


Figure 4.5: An example EQU-AddRelEdges transformation.

example). In particular, it adds two edges e and e' from relational node u to relational node v in $\overline{G_P}$, where $sign = +$ for e and $sign = -$ for e'. Node v may be any relational node in $\overline{G_P}$, but to ensure that the resulting program P_{EQU} is stratifiable, node u may not be a descendant of v.

---

assume($v \in \overline{G_P}$.get_all_nodes() $\bigwedge$
    $u \in \overline{G_P}$.get_all_nodes() $\setminus \overline{G_P}$.get_descendants(v))

$k = \overline{G_P}$.get_max_rule_number(v) $+1$
$args = \overline{G_P}$.generate_vars(v,k)
$evals = \{number: k,\ sign: +,\ vars: args\}$
$evals' = \{number: k,\ sign: -,\ vars: args\}$
$s = \overline{G_P}$.get_stratum(v) $+1$
$a =$ **if** $\overline{G_P}$.get_ancestry(v) $==$ none **then** none **else** ?
$nvals = \{stratum: s,\ ancestry: a\}$
$\overline{G_{P_{EQU}}} = \overline{G_P}.\mathbb{R}_{\text{ADDRELEDGE}}(u, v, evals)$
$\overline{G_{P_{EQU}}} = \overline{G_{P_{EQU}}}.\mathbb{R}_{\text{ADDRELEDGE}}(u, v, evals')$
$\overline{G_{P_{EQU}}} = \overline{G_{P_{EQU}}}.\mathbb{R}_{\text{MODRELNODE}}(u, nvals)$

---

assert(out($\overline{G_P}$) $\equiv$ out($\overline{G_{P_{EQU}}}$))

Note that adding two such edges to any node makes the corresponding rule

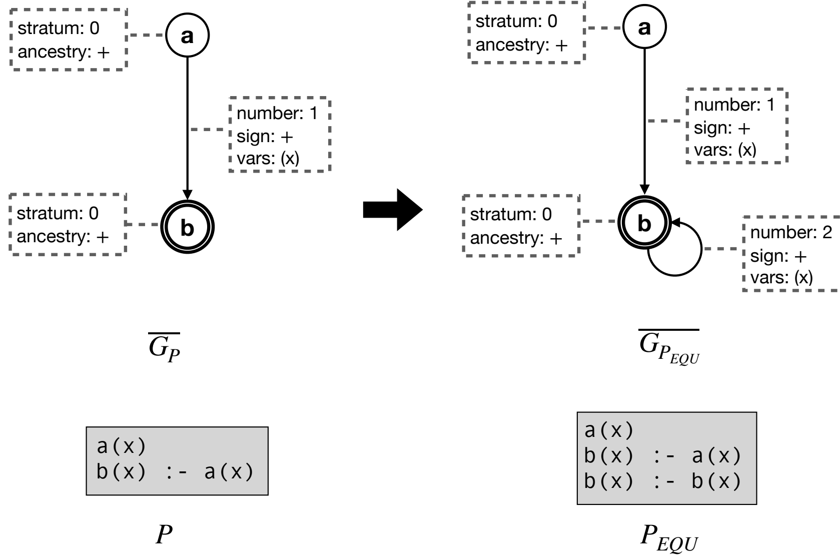


Figure 4.6: An example EQU-AddSelfEdge transformation.

compute an empty result. Therefore, for the transformation to be equivalent, we apply it on a new rule of an existing relation R, where $R = \theta^{-1}(v)$. Method `get_max_rule_number` retrieves the total number of existing rules for R, and thus, k must be `get_max_rule_number(`v`)` $+1$. Method `generate_vars` generates random variables, which however satisfy the type constraints of R. Finally, we update the properties of node u to reflect the added edges—*stratum* is incremented by 1 due to the negative edge, and if there is a path from v to the output, then *ancestry* becomes ? due to the edges having different *sign* values.

EQU-ADDSELFEDGE. This transformation adds a positive self edge e to an existing relational node v, that is, the edge connects v to itself (see Fig. 4.6 for an example). Similarly to EQU-ADDRELEDGES, for the transformation to be equivalent, we apply it on a new rule of an existing relation.

`assume(`$v \in \overline{G_P}$`.get_all_nodes())`

$k = \overline{G_P}$`.get_max_rule_number(`v`)` $+1$
$args = \overline{G_P}$`.generate_vars(`v, k`)`
$evals = \{number : k,\ sign : +,\ vars : args\}$
$\overline{G_{P_{EQU}}} = \overline{G_P}.\mathbb{R}_{\text{ADDRELEDGE}}(v, v, evals)$

`assert(out(`$\overline{G_P}$`)` $\equiv$ `out(`$\overline{G_{P_{EQU}}}$`))`

EQU-FACTINLINE. This transformation removes all incoming edges to a relational node v. Removing these edges effectively removes all rules for relation

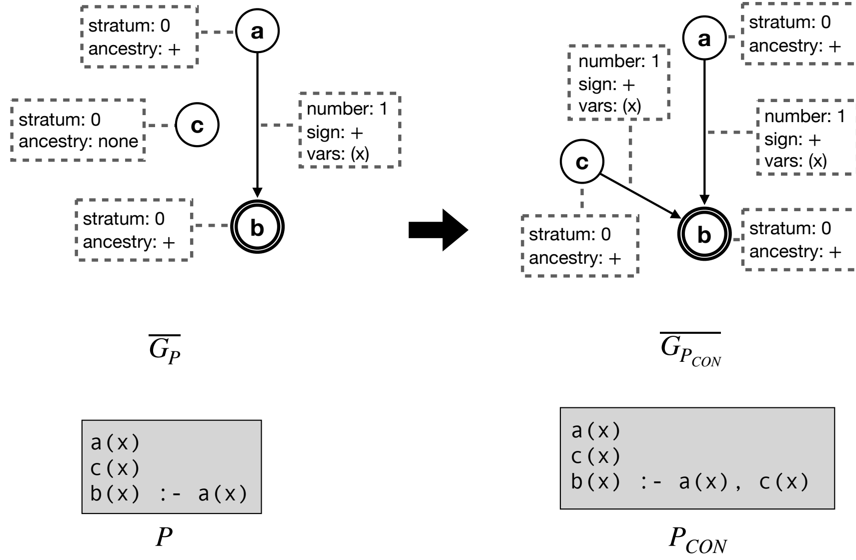


Figure 4.7: An example CON-AddRelEdge transformation.

$R = \theta^{-1}(v)$. For the output of P_{EQU} to remain equivalent to the output of P, the removed rules are replaced with the corresponding facts that these rules would compute—we retrieve these facts by executing the rules for R. In particular, the transformation creates a fact node for each retrieved fact and associates it with v using the ADDFACT rewrite rule.

assume($v \in \overline{G_P}$.get_all_nodes())

$E = \overline{G_P}$.get_incoming_edges(v)
$\overline{G_{P_{EQU}}} = \overline{G_P}$
foreach $e \in E : \overline{G_{P_{EQU}}} = \overline{G_{P_{EQU}}}.\mathbb{R}_{\text{DELRELEDGE}}(e)$
$F = \overline{G_P}$.get_facts(v)
foreach $f \in F : \overline{G_{P_{EQU}}} = \overline{G_{P_{EQU}}}.\mathbb{R}_{\text{ADDFACT}}(f, v)$
$\overline{G_{P_{EQU}}} = \overline{G_{P_{EQU}}}$.annotate()

assert(out($\overline{G_P}$) $\equiv$ out($\overline{G_{P_{EQU}}}$))

At the end of this radical transformation, we re-annotate the resulting graph since all ancestors of v might now have different *stratum* and *ancestry* values.

CON-ADDRELEDGE. This is a contracting transformation that adds a positive relational edge e from u to v, where v is in positive ancestry of output node O and u either has the same stratification number as v or is not an ancestor of v at

all (see Fig. 4.7 for an example). This is to ensure that e does not introduce a cycle with negation between u and v, thus rendering the transformed program unstratifiable. Adding an incoming edge to v corresponds to adding an atom in any rule for relation $R = \theta^{-1}(v)$, which contracts the result of R—adding an atom is essentially a conjunction in Datalog. Since data flows monotonically from R to the output relation, this transformation will also contract the result of the program.

```
assume(v ∈ G_P.get_nodes_with_ancestry(+) ⋀
       u ∈ G_P.get_nodes_with_stratum(v) ⋃
           (G_P.get_all_nodes() \ G_P.get_ancestors(v)))
```

$k = \texttt{generate_number}(1, \overline{G_P}.\texttt{get_max_rule_number}(v))$
$args = \overline{G_P}.\texttt{generate_vars}(v, k)$
$evals = \{number : k,\ sign : +,\ vars : args\}$
$s = \overline{G_P}.\texttt{get_stratum}(v)$
$a = \overline{G_P}.\texttt{get_ancestry}(u)$
$nvals = \{stratum : s,\ ancestry : a\}$
$\overline{G_{P_{CON}}} = \overline{G_P}.\mathbb{R}_{\text{ADDRELEDGE}}(u, v, evals)$
$\overline{G_{P_{CON}}} = \overline{G_{P_{CON}}}.\mathbb{R}_{\text{MODRELNODE}}(u, nvals)$

```
assert(out(G_P) ⊇ out(G_P_CON))
```

EXP-ADDRELEDGE. This is an expanding transformation that adds a positive relational edge e from u to v (see Fig. 4.8 for an example). Similar to the previous transformation, v is in positive ancestry of output node O, and u either has the same stratification number as v or is not an ancestor of v at all. Contrary to the previous transformation, adding e creates a new rule for relation $R = \theta^{-1}(v)$, which expands the result of R—adding a rule is essentially a disjunction in Datalog. Consequently, the program result is also expanded.

```
assume(v ∈ G_P.get_nodes_with_ancestry(+) ⋀
       u ∈ G_P.get_nodes_with_stratum(v) ⋃
           (G_P.get_all_nodes() \ G_P.get_ancestors(v)))
```

$k = \overline{G_P}.\texttt{get_max_rule_number}(v) + 1$
$args = \overline{G_P}.\texttt{generate_vars}(v, k)$
$evals = \{number : k,\ sign : +,\ vars : args\}$
$s = \overline{G_P}.\texttt{get_stratum}(v)$
$a = \overline{G_P}.\texttt{get_ancestry}(u)$
$nvals = \{stratum : s,\ ancestry : a\}$
$\overline{G_{P_{EXP}}} = \overline{G_P}.\mathbb{R}_{\text{ADDRELEDGE}}(u, v, evals)$
$\overline{G_{P_{EXP}}} = \overline{G_{P_{EXP}}}.\mathbb{R}_{\text{MODRELNODE}}(u, nvals)$

```
assert(out(G_P) ⊆ out(G_P_EXT))
```

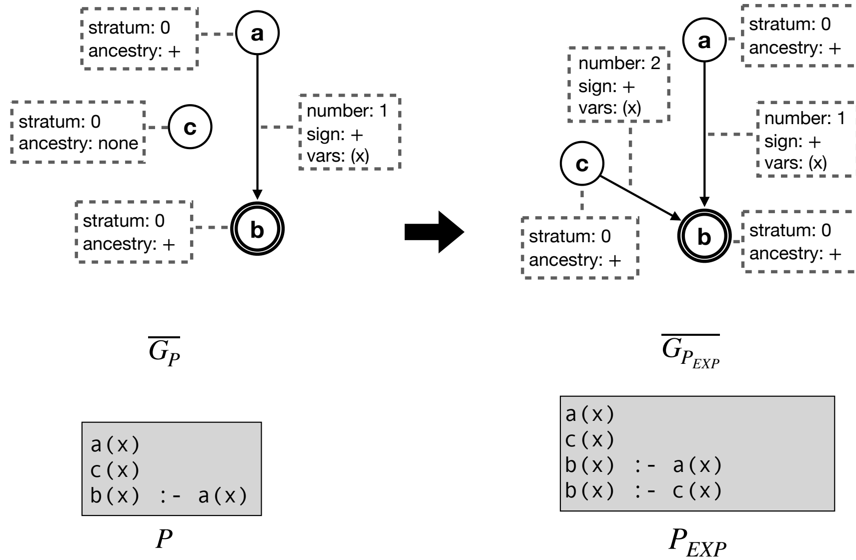


Figure 4.8: An example EXP-AddRelEdge transformation.

4.6. Implementation

We implemented DLSmith in a total of 6,300 lines of Python code. It currently supports six Datalog dialects, namely, Ascent [231], DDlog [228], Flix [167], Formulog [37], Scallop [128], and Soufflé [136]. In the rest of this section, we discuss how to implement new metamorphic transformations as well as the existing queryFuzz transformations [168] in DLSmith.

Implementing new transformations. The transformations described in the previous section require (on average) 40 lines of Python code to implement. Implementing a transformation for an already supported Datalog engine involves the following steps: (1) expressing a precondition, (2) retrieving the graph elements satisfying the precondition, (3) generating attributes for the graph rewrite rules(s), (4) calling the graph rewrite rule(s), and (5) expressing a postcondition. Implementing a transformation for a new engine additionally requires extracting relations from seed programs and generating programs from annotated precedence graphs.

Implementing queryFuzz transformations. queryFuzz implements metamorphic transformations based on formal properties of conjunctive queries, namely, query containment and equivalence. As described earlier however, these transformations are limited since the approach does not have a global view of the program being transformed. On the other hand, DLSmith subsumes queryFuzz—not only can all queryFuzz transformations be expressed using the specifications in Sect. 4.5, but its transformations can now also be applied in *any* stratum of

the Datalog program. In particular, this is how queryFuzz transformations can be expressed in DLSmith:

- ADD transformations add an atom $R(v_1,\ldots,v_n)$ to a rule of relation Q. These can be expressed in DLSmith by adding a relational edge e from a relational node u to a relational node v, where $u = \theta(R)$ and $v = \theta(Q)$, using rewrite rule $\mathbb{R}_{\text{ADDRELEDGE}}$.
- MOD transformations modify a rule of relation Q by renaming a variable appearing in its atoms. These can be expressed in DLSmith by modifying property *vars* of the incoming edges to a node v, where $v = \theta(R)$, using rewrite rule $\mathbb{R}_{\text{MODRELEDGE}}$.
- REM transformations remove an atom $R(v_1,\ldots,v_n)$ from a rule of relation Q. These can be expressed in DLSmith by removing a relational edge e from a relational node u to a relational node v, where $u = \theta(R)$ and $v = \theta(Q)$, using rewrite rule $\mathbb{R}_{\text{DELRELEDGE}}$.
- NEG transformations replace an atom $R(v_1,\ldots,v_n)$ in a rule of relation Q with the negated atom of a new relation, say *neg*. For instance, the following rule

  ```
  p(X, Y) :- a(X, Y), b(Y, Z), c(Z).
  ```

  is transformed into

  ```
  neg(Z) :- a(X, Y), b(Y, Z), not c(Z).
  p(X, Y) :- a(X, Y), b(Y, Z), not neg(Z).
  ```

  Relation *neg* is defined to have the same body as the rule for Q but with a negated R, thereby introducing double negation. These transformations are always equivalent and may be expressed in DLSmith by generating a new relational node for *neg*, adding edges for the atoms of *neg*, and adjusting the edges for the atoms of Q.

4.7. Experimental Evaluation

In this section, we address the following research questions:

RQ1: How effective is DLSmith in detecting previously unknown query bugs in diverse Datalog engines?

RQ2: What are characteristics of the detected bugs?

RQ3: How effective is DLSmith in terms of code coverage?

RQ4: How efficient is DLSmith?

4.7.1. Setup

We tested six mature Datalog engines, namely, Ascent, DDlog, Flix, Formulog, Scallop, and Soufflé. All engines are publicly available on GitHub. We completed the implementation of the first version of DLSmith in January 2022 and started our testing campaign with Soufflé. Until September 2022, we added support for the remaining five engines as well as for more transformations. On average, we spent about 1.5 months testing each engine. As seeds, we used semantically valid test cases from the engine repositories.

We performed all experiments on a 32-core Intel ® Xeon ® E5-2667 v2 CPU @ 3.30GHz machine with 256GB of memory, running Debian GNU/Linux 10 (buster).

4.7.2. Results

We now discuss our findings for each research question.

RQ1: Query bugs in diverse engines. We tested six active Datalog implementations, each supporting a different dialect. We give a brief overview of these engines next.

Ascent can integrate with arbitrary application logic written in the Rust programming language. In particular, it allows Datalog rules to call into Rust code and vice versa.

DDlog is used for incremental computation. Specifically, developers declaratively specify a desired input-output mapping, and DDlog uses it to synthesize an efficient incremental implementation.

Flix is a functional, imperative, and logic programming language, which looks like Scala and provides support for algebraic data types, pattern matching, higher order functions, etc. In Flix, Datalog programs are first class values, and Datalog constraints have more expressive power.

Formulog is a domain-specific dialect with support for constructing and reasoning about SMT formulas.

Scallop is a Datalog-based neuro-symbolic programming language, supporting discrete, probabilistic, and differential reasoning modes. Rules may be integrated with machine-learning models, facts may have associated probabilities, results may be computed with a success probability, etc.

Soufflé is a fast and scalable dialect, whose syntax was inspired by bddbddb [260] and μZ in Z3 [125]. Its primary goal is speed, thereby tailoring program execution to multi-core servers with large memory.

Table 4.2: Query bugs detected by DLSmith.

Bug ID	Datalog Engine	Metamorphic Transformation	Bug Status
1	Soufflé	EQU-ADDRELEDGES	Fixed
2	Soufflé	EQU-ADDRELEDGES	Fixed
3	Soufflé	EQU-ADDRELEDGES	Fixed
4	Soufflé	ADDEQU	Fixed
5	Soufflé	ADDEQU	Fixed
6	Soufflé	EQU-ADDRELEDGES	Confirmed
7	Formulog	EQU-ADDSELFEDGE	Fixed
8	Ascent	ADDEQU	Fixed
9	Scallop	CON-ADDRELEDGE	Fixed
10	Scallop	CON-ADDRELEDGE	Fixed
11	Scallop	EXP-ADDRELEDGE	Fixed
12	Scallop	CON-ADDRELEDGE	Fixed
13	Scallop	EQU-ADDRELEDGES	Confirmed
14	Soufflé	EQU-ADDRELEDGES	Confirmed
15	Soufflé	EQU-FACTINLINE	Confirmed
16	Soufflé	EQU-ADDRELNODE, EQU-ADDRELEDGES	Confirmed

Tab. 4.2 shows the list of unique and previously unknown query bugs detected by DLSmith. The first column of the table provides an identifier for each bug and links to the (anonymized) bug report on GitHub. The second column shows the engine where the bug was found, the third the metamorphic transformation that was applied, and the last column the status of the bug. In total, DLSmith detected 16 query bugs in four engines, all of which are confirmed by the developers and eleven are fixed.

Note that ADDEQU (bugs 4, 5, 8) is a queryFuzz transformation implemented in DLSmith. Out of all detected bugs, only bugs 4 and 8 could have been detected by queryFuzz. Bug 5 is detected with a queryFuzz transformation, which however is not applied at the highest stratum—this is only possible in DLSmith. Also note that our tool applies sequences of transformations, and bug 16 required a sequence of two transformations to be detected.

RQ2: Characteristics of detected bugs. To better understand the characteristics of the detected bugs, we now discuss them in detail.

Soufflé. Bugs 1, 2, 5, and 15 were found in the implementation of the `eqrel` (equivalence relation) data structure. According to the developers, they recently applied performance-related changes to this code, which is perhaps when these bugs were introduced. Bug 3 was due to the incorrect implementation of utility function `range` in the presence of unsigned bounds. Bug 4 was caused by a mistake in the implementation of subsumption for the `btree_delete` data

structure. The developers called it a "nasty bug to find and fix". Bug 6 was detected in the code generation mechanism for the interpreter. Currently, the interpreter checks floating point number equivalence via bitwise comparison rather than floating point comparison. The developers confirmed the issue but need time to resolve it. Bug 14 was again caused by floating point equivalence checking in the `brie` data structure. This bug, however, only manifested in compiler mode. Bug 16 was also found in `brie`; it manifested when using "auto-scheduling" for automatic performance tuning.

Formulog. Bug 7 was a non-deterministic query bug in the procedure for computing strata, which incorrectly depended on non-deterministic hash values. As a result, relations ended up being computed in the wrong order.

Ascent. Bug 8 occurred because of an atom having a repeated variable in a rule body. This case was not taken into account when reordering atoms at runtime to increase performance.

Scallop. Bugs 9 and 11 were due to incorrect optimization conditions in the query plan optimizer. Bug 10 was related to incorrect variable deduplication in the query plan generator. Bug 12 revealed an issue with incorrect optimization of negative atoms, and bug 13 was caused by incorrectly detecting a negative cycle between two relations.

RQ3: Code coverage. In this research question, we evaluate the code coverage achieved by DLSmith. We first compare it with the coverage achieved by hand-crafted tests in the engine repositories, which we use as seeds for DLSmith. We also compare with queryFuzz when using the same seeds and with DLSmith when using empty seeds. The results are shown in Tab. 4.3 for Soufflé (written in C++) and Scallop (written in Rust), where we detected the most bugs. Note that, for this experiment, the total number of seeds for Soufflé is 240 and for Scallop 39. When running DLSmith and queryFuzz, for each seed program (empty or not), we generate 500 transformed programs.

As shown in the table, DLSmith is the most effective, while DLSmith-Empty (with empty seeds) is the least effective. When comparing DLSmith to running the seeds alone for Scallop, we observe a 4.8% and 7.5% increase in line and function coverage, respectively. For Soufflé, we observe a 4.1% and 2.8% increase in line and function coverage, respectively. When comparing DLSmith to queryFuzz for Scallop, we observe a 3.9% and 6.7% increase in line and function coverage, respectively. For Soufflé, we observe a 2.7% and 2.3% increase in line and function coverage, respectively.

RQ4: Performance. The performance of DLSmith depends on the Datalog engine under test. At the end of the second phase in Fig. 4.2, DLSmith on average generates a program per 0.006 seconds (or 163 programs per second). However, as expected, it is slowed down by the engine running each of these programs, and Tab. 4.4 shows by how much. The second column computes the average

Table 4.3: Code coverage achieved by seeds alone, queryFuzz, DLSmith with empty seeds, and DLSmith. L represents line coverage, and F function coverage.

Datalog Engine	Seeds		queryFuzz	
	L	F	L	F
Scallop	18,056	2,709	18,201	2,729
Soufflé	63,452	40,350	64,298	40,544
	DLSmith-Empty		DLSmith	
	L	F	L	F
Scallop	11,388	1,826	18,915	2,912
Soufflé	50,180	30,205	66,027	41,484

Table 4.4: Average running time (in seconds) of DLSmith when executing its first two phases.

Datalog Engine	Running Time
Ascent	17.221
DDlog	612.121
Flix	240.031
Formulog	1.512
Scallop	0.146
Soufflé	0.734

time of generating *and* running a single program (i.e., executing the first two phases of Fig. 4.2). The graph transformer in the third phase of Fig. 4.2 on average generates a transformed annotated precedence graph per 0.00035 seconds (or 2857 transformed annotated precedence graphs per second), where the maximum transformation sequence length is 100. Note that these averages are computed over three runs of DLSmith, each seeded with 500 programs.

4.7.3. Threats to Validity

Our experimental results, and especially the detected query bugs, depend on the Datalog engines we tested as well as the seed programs that DLSmith takes as input. Regarding the former, we selected six diverse and active Datalog implementations. Regarding the latter, we used all (syntactically and semantically) valid test cases from the engine repositories as seeds for DLSmith. We also perform an experiment using only empty seeds to show their effect on our code coverage results.

4.8. Related Work

In this chapter, we presented the most comprehensive approach to detecting query bugs in Datalog engines. Our approach uses metamorphic testing to address the oracle problem [31] by first generating a Datalog program from its corresponding annotated precedence graph and transforming the program by transforming its annotated precedence graph using graph rewrite rules.

Graphs are a powerful and general notation that is used to express and model complex systems in a variety of areas in computer science, including software engineering [126], software security [54], program slicing [64], computer networks [181], and bioinformatics [209], to name a few. Graph rewriting [211] is used to formalize how a complex structure represented by a graph evolves over time. Together with graph transformation [26, 27, 210], it has been extensively studied in the graph theory community.

In chapter 3, we presented the first metamorphic testing approach to detect bugs in Datalog engines [168]. We have already discussed the most closely related areas of work in section 3.12.

4.9. Summary and Remarks

In this chapter, we have presented DLSmith, another powerful approach for detecting query bugs in Datalog engines using dependency-aware metamorphic test oracles. The test oracles presented in this chapter use rich precedence information capturing dependencies among relations in the program. DLSmith detected 16 previously unknown query bugs in four Datalog engines, and our evaluation showed that only two of these bugs could have been detected using queryFuzz.

Part II

Balancing Soundness, Precision, and Performance

Chapter 5

Automatically Tailoring Abstract Interpretation to Custom Usage Scenarios

In recent years, there has been significant progress in the development and industrial adoption of static analyzers, specifically of abstract interpreters. Such analyzers typically provide a large, if not huge, number of configurable options controlling the analysis precision and performance. A major hurdle in integrating them in the software development life cycle is tuning their options to custom usage scenarios, such as a particular code base or certain resource constraints.

In this chapter, we propose a technique that automatically tailors an abstract interpreter to the code under analysis and any given resource constraints. We implement this technique in a framework, TAILOR, which we use to perform an extensive evaluation on real-world benchmarks. Our experiments show that the configurations generated by TAILOR are vastly better than the default analysis options, vary significantly depending on the code under analysis, and most remain tailored to several subsequent code versions.

5.1. Introduction

Static analysis inspects code, without running it, in order to prove properties or detect bugs. Typically, static analysis approximates code behavior, for instance, because checking the correctness of most properties is undecidable. *Performance* is another important reason for this approximation. Typically, the closer the approximation is to the actual code behavior, the less efficient and the more *precise* the analysis is, that is, the fewer false positives it reports. For less tight approximations, the analysis tends to become more efficient but less precise.

Recent years have seen tremendous progress in the development and industrial

adoption of static analyzers. Notable successes include Facebook's Infer [59, 60] and AbsInt's Astrée [45]. Many popular analyzers, such as these, are based on *abstract interpretation* [76], a technique that abstracts the concrete program semantics and reasons about its abstraction. In particular, program states are abstracted as elements of *abstract domains*. Most abstract interpreters offer a wide range of abstract domains that impact the precision and performance of the analysis. For instance, the Intervals domain [75] is typically faster but less precise than Polyhedra [81], which captures linear inequalities among variables.

In addition to the domains, abstract interpreters usually provide a large number of other options, for instance, whether backward analysis should be enabled or how quickly a fixpoint should be reached. In fact, the sheer number of option combinations (over 6 million in our experiments) is bound to overwhelm users, especially non-expert ones. To make matters worse, the best option combinations may vary significantly depending on the code under analysis and the resources, such as time or memory, that users are willing to spend.

In light of this, we suspect that most users resort to using the default options that the analysis designer pre-selected for them. However, these are definitely not suitable for all code. Moreover, they do not adjust to different stages of software development, e.g., running the analysis in the editor should be much faster than running it in a continuous integration (CI) pipeline, which in turn should be much faster than running it prior to a major release. The alternative of enabling the (in theory) most precise analysis can be even worse, since in practice it often runs out of time or memory as we show in our experiments. As a result, the widespread adoption of abstract interpreters is severely hindered, which is unfortunate since they constitute an important class of practical analyzers.

Our approach. To address this issue, we present the first technique that automatically tailors a generic abstract interpreter to a custom usage scenario. With the term *custom usage scenario*, we refer to a particular piece of code and specific resource constraints. The key idea behind our technique is to phrase the problem of customizing the abstract-interpretation configuration to a given usage scenario as an optimization problem. Specifically, different configurations are compared using a cost function that penalizes those that prove fewer properties or require more resources. The cost function can guide the configuration search of a wide range of existing optimization algorithms. This problem of tuning abstract interpreters can be seen as an instance of the more general problem of *algorithm configuration* [129]. In the past, algorithm configuration has been used to tune algorithms for solving various hard problems, such as SAT solving [130, 131], and more recently, training of machine-learning models [38, 95, 249].

We implement our technique in an open-source framework called TAILOR[1], which configures a given abstract interpreter for a given usage scenario using a

[1]The framework's implementation can be found on Github at `https://github.com/Practical-Formal-Methods/tailor` and an installation at `https://doi.org/10.5281/zenodo.4719604`.

given optimization algorithm. As a result, TAILOR enables the abstract interpreter to prove as many properties as possible within the resource limit without requiring any domain expertise on behalf of the user.

Using TAILOR, we find that tailored configurations vastly outperform the default options pre-selected by the analysis designers. In fact, we show that this is possible even with very simple optimization algorithms. Our experiments also demonstrate that tailored configurations vary significantly depending on the usage scenario—in other words, there cannot be a single configuration that fits all scenarios. Finally, most of the generated configurations remain tailored to several subsequent code versions, suggesting that re-tuning is only necessary after major code changes.

Contributions. In this chapter, we make the following contributions:

1. We present the first technique for automatically tailoring abstract interpreters to custom usage scenarios.
2. We implement our technique in an open-source framework called TAILOR.
3. Using a state-of-the-art abstract interpreter, CRAB [116], with millions of configurations, we show the effectiveness of TAILOR on real-world benchmarks.

5.2. Overview

We now illustrate the workflow and tool architecture of TAILOR and provide examples of its effectiveness.

Terminology. In the following, we refer to an abstract domain with all its options (e.g., enabling backward analysis or more precise treatment of arrays etc.) as an *ingredient*.

As discussed earlier, abstract interpreters typically provide a large number of such ingredients. To make matters worse, it is also possible to combine different ingredients into a sequence (which we call a *recipe*) such that more properties are verified than with individual ingredients. For example, a user could configure the abstract interpreter to first use Intervals to verify as many properties as possible and then use Polyhedra to attempt verification of any remaining properties. Of course, the number of possible configurations grows exponentially in the length of the recipe (over 6 million in our experiments for recipes up to length 3).

Workflow. The high-level architecture of TAILOR is shown in Fig. 5.1. It takes as input the code to be analyzed (i.e., any program, file, function, or fragment), a user-provided resource limit, and optionally an optimization algorithm. We focus on time as the constrained resource in this chapter, but our technique could be easily extended to other resources, such as memory.

The optimization engine relies on a recipe generator to generate a fresh recipe. To assess its quality in terms of precision and performance, the recipe evaluator computes a cost for the recipe. The cost is computed by evaluating how precise and efficient the abstract interpreter is for the given recipe. This cost is used by the

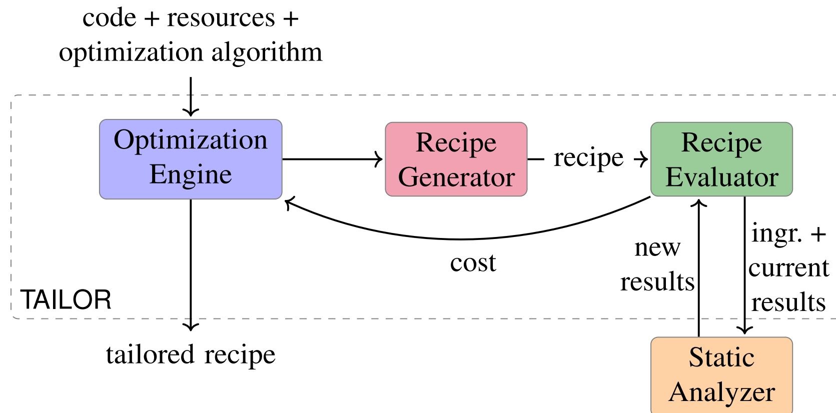


Figure 5.1: Overview of our framework.

optimization engine to keep track of the best recipe so far, i.e., the one that proves the most properties in the least amount of time. TAILOR repeats this process for a given number of iterations to sample multiple recipes and returns the recipe with the lowest cost.

Zooming in on the evaluator, a recipe is processed by invoking the abstract interpreter for each ingredient. After each analysis (i.e., one ingredient), the evaluator collects the new verification results, that is, the verified assertions. All verification results that have been achieved so far are subsequently shared with the analyzer when it is invoked for the next ingredient. Verification results are shared by converting all verified assertions into assumptions. After processing the entire recipe, the evaluator computes a cost for the recipe, which depends on the number of unverified assertions and the total analysis time.

In general, there might be more than one recipe tailored to a particular usage scenario. Naïvely, finding one requires searching the space of all recipes. Sect. 5.4.3 discusses several optimization algorithms for performing this search, which TAILOR already incorporates in its optimization engine.

Examples. As an example, let us consider the usage scenario where a user runs the CRAB abstract interpreter [116] in their editor for instant feedback during code development. This means that the allowed time limit for the analysis is very short, say, 1 sec. Now assume that the code under analysis is a program file[2] of the multimedia processing tool FFMPEG, which is used to evaluate the effectiveness of TAILOR in our experiments. In this file, CRAB checks 45 assertions for common bugs, i.e., division by zero, integer overflow, buffer overflow, and use after free.

Analysis of this file with the default CRAB configuration takes 0.35 sec to complete. In this time, CRAB proves 17 assertions and emits 28 warnings about the properties that remain unverified. For this usage scenario, TAILOR is able to tune the abstract-interpreter configuration such that the analysis time is 0.57 sec and the number of verified properties increases by 29% (i.e., 22 assertions are

[2] https://github.com/FFmpeg/FFmpeg/blob/master/libavformat/idcin.c

proved). Note that the tailored configuration uses a completely different abstract domain than the one in the default configuration. As a result, the verification results are significantly better, but the analysis takes slightly longer to complete (although remaining within the specified time limit). In contrast, enabling the most precise analysis in CRAB verifies 26 assertions but takes over 6 min to complete, which by far exceeds the time limit imposed by the usage scenario.

While it takes TAILOR 4.5 sec to find the above configuration, this is time well invested; the configuration can be re-used for several subsequent code versions. In fact, in our experiments, we show that generated configurations can remain tailored for at least up to 50 subsequent commits to a file under version control. Given that changes in the editor are typically much more incremental, we expect that no re-tuning would be necessary at all during an editor session. Re-tuning may be beneficial after major changes to the code under analysis and can happen offline, e.g., between editor sessions, or in the worst case overnight.

As another example, consider the usage scenario where CRAB is integrated in a CI pipeline. In this scenario, users should be able to spare more time for analysis, say, 5 min. Here, let us assume that the analyzed code is a program file[3] of the CURL tool for transferring data by URL, which is also used in our evaluation. The default CRAB configuration takes 0.23 sec to run and only verifies 2 out of 33 checked assertions. TAILOR is able to find a configuration that takes 7.6 sec and proves 8 assertions. In contrast, the most precise configuration does not terminate even after 15 min.

Both scenarios demonstrate that, even when users have more time to spare, the default configuration cannot take advantage of it to improve the verification results. At the same time, the most precise configuration is completely impractical since it does not respect the resource constraints imposed by these scenarios.

5.3. A Generic Abstract Interpreter

Many successful abstract interpreters (e.g., Astrée [45], C Global Surveyor [256], Clousot [94], CRAB [116], IKOS [47], Sparrow [203], and Infer [60]) follow the generic architecture in Fig. 5.2. In this section, we describe its main components to show that our approach should generalize to such analyzers.

Memory domain. Analysis of low-level languages such as C and LLVM-bitcode requires reasoning about pointers. It is, therefore, common to design a *memory domain* [189] that can simultaneously reason about pointer aliasing, memory contents, and numerical relations between them.

Pointer domains resolve aliasing between pointers, and *array domains* reason about memory contents. More specifically, array domains can reason about individual memory locations (cells), infer universal properties over multiple cells, or both. Typically, reasoning about individual cells trades performance for precision unless there are very few array elements (e.g., [106, 189]). In contrast, reasoning about

[3] https://github.com/curl/curl/blob/master/lib/cookie.c

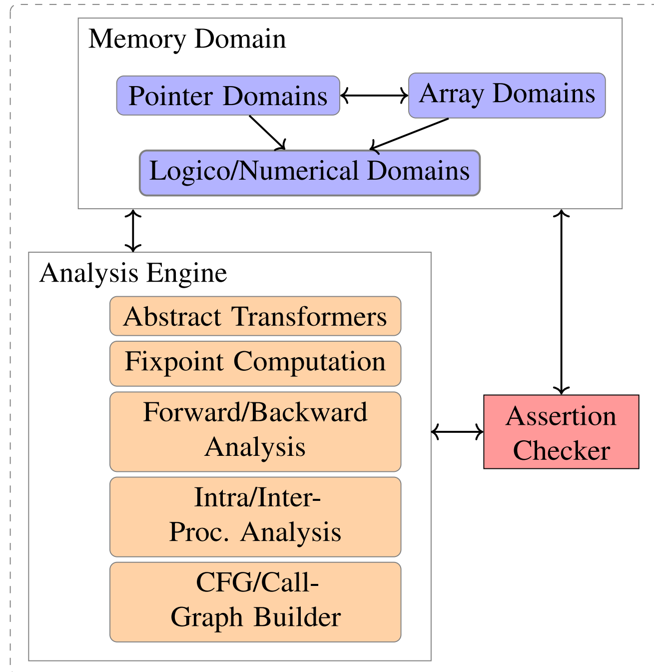


Figure 5.2: Generic architecture of an abstract interpreter.

multiple memory locations (*summarized cells*) trades precision for performance. In our evaluation, we use *Array smashing* domains [45] that abstract different array elements into a single summarized cell. *Logico-numerical domains* infer relationships between program and *synthetic* variables, introduced by the pointer and array domains, e.g., summarized cells.

Next, we introduce domains typically used for proving the absence of runtime errors in low-level languages. *Boolean domains* (e.g., flat Boolean, BDDApron [1]) reason about Boolean variables and expressions. *Non-relational domains* (e.g., Intervals [75], Congruence [109]) do not track relations among different variables, in contrast to *relational domains* (e.g., Equality [142], Zones [188], Octagons [190], Polyhedra [81]). Due to their increased precision, relational domains are typically less efficient than non-relational ones. *Symbolic domains* (e.g., Congruence closure [63], Symbolic constant [191], Term [103]) abstract complex expressions (e.g., non-linear) and external library calls by uninterpreted functions. *Non-convex domains* express disjunctive invariants. For instance, the DisInt domain [94] extends Intervals to a finite disjunction; it retains the scalability of the Intervals domain by keeping only non-overlapping intervals. On the other hand, the Boxes domain [115] captures arbitrary Boolean combinations of intervals, which can often be expensive.

Fixpoint computation. To ensure termination of the fixpoint computation, Cousot and Cousot introduce *widening* [76, 78], which usually incurs a loss of precision. There are three common strategies to reduce this precision loss, which

however sacrifice efficiency. First, *delayed widening* [45] performs a number of initial fixpoint-computation iterations in the hope of reaching a fixpoint before resorting to widening. Second, *widening with thresholds* [152, 186] limits the number of program expressions (thresholds) that are used when widening. The third strategy consists in applying *narrowing* [76, 78] a certain number of times.

Forward and backward analysis. Classically, abstract interpreters analyze code by propagating abstract states in a *forward* manner. However, abstract interpreters can also perform *backward* analysis to compute the execution states that lead to an assertion violation. Cousot and Cousot [77, 79] define a *forward-backward refinement* algorithm in which a forward analysis is followed by a backward analysis until no more refinement is possible. The backward analysis uses invariants computed by the forward analysis, while the forward analysis does not explore states that cannot reach an assertion violation based on the backward analysis. This refinement is more precise than forward analysis alone, but it may also become very expensive.

Intra- and inter-procedural analysis. An *intra-procedural* analysis analyzes a function ignoring the information (i.e., call stack) that flows into it, while an *inter-procedural* analysis considers all flows among functions. The former is much more efficient and easy to parallelize, but the latter is usually more precise.

5.4. Our Technique

This section describes the components of TAILOR in detail; Sects. 5.4.1, 5.4.2, 5.4.3 explain the optimization engine, recipe evaluator, and recipe generator (Fig. 5.1).

5.4.1. Recipe Optimization

Alg. 6 implements the optimization engine. In addition to the code P and the resource limit r_{max}, it also takes as input the maximum length of the generated recipes l_{max} (i.e., the maximum number of ingredients), a function to generate new recipes GENERATERECIPE (i.e., the recipe generator from Fig. 5.1), and four other parameters, which we explain later.

A tailored recipe is found in two phases. The first phase aims to find the best abstract domain for each ingredient, while the second tunes the remaining analysis settings for each ingredient (e.g., whether backward analysis should be enabled). Parameters i_{dom} and i_{set} control the number of iterations of each phase. Note that we start with a search for the best domains since they have the largest impact on the precision and performance of the analysis.

During the first phase, the algorithm initializes the best recipe rec_{best} with an initial recipe rec_{init} (line 3). The cost of this recipe is evaluated with function EVALUATE, which implements the recipe evaluator from Fig. 5.1. The subsequent nested loop (line 5) samples a number of recipes, starting with the shortest recipes

Algorithm 6: Optimization engine.

```
1  procedure OPTIMIZE(P, r_max, i_dom, i_set, rec_init, GENERATERECIPE, ACCEPT)
2    // Phase 1 (optimize domains)
3    rec_best := rec_curr := rec_init
4    cost_best := cost_curr := EVALUATE(P, r_max, rec_best)
5    for (l ← 0, l <= l_max, l++) do
6      for (i ← 1, i <= i_dom · l, i++) do
7        rec_next := GENERATERECIPE(rec_curr, l)
8        cost_next := EVALUATE(P, r_max, rec_next)
9        if cost_next < cost_best then
10         rec_best, cost_best := rec_next, cost_next
11       if ACCEPT(cost_curr, cost_next) then
12         rec_best, cost_best := rec_next, cost_next
13
14   // Phase 2 (optimize settings)
15   for (i ← 0, i <= l_max, i++) do
16     rec_mut := MUTATESETTINGS(rec_best)
17     cost_mut := EVALUATE(P, r_max, rec_mut)
18     if cost_mut < cost_best then
19         rec_best, cost_best := rec_mut, cost_mut
20   return rec_best
```

(l := 1) and ending with the longest recipes ($l := l_{max}$). The inner loop generates i_{dom} ingredients for each ingredient in the recipe (i.e., $i_{dom} \cdot l$ total iterations) by invoking function GENERATERECIPE, and in case a recipe with lower cost is found, it updates the best recipe (lines 9–10). Several optimization algorithms, such as hill climbing and simulated annealing, search for an optimal result by mutating some of the intermediate results. Variable rec_{curr} stores intermediate recipes to be mutated, and function ACCEPT decides when to update it (lines 11–12).

As explained earlier, the purpose of the first phase is to identify the best sequence of abstract domains. The second phase (lines 14–19) focuses on tuning the other settings of the best recipe so far. This is done by randomly mutating the best recipe via MUTATESETTINGS (line 16), and updating the best recipe if better settings are found (lines 18–19). After exploring i_{set} random settings, the best recipe is returned to the user (line 20).

5.4.2. Recipe Evaluation

The recipe evaluator from Fig. 5.1 uses a cost function to determine the quality of a fresh recipe with respect to the precision and performance of the abstract interpreter. This design is motivated by the fact that analysis imprecision and

inefficiency are among the top pain points for users [68].

Therefore, the cost function depends on the number of generated warnings w (that is, the number of unverified assertions), the total number of assertions in the code w_{total}, the resource consumption r of the analyzer, and the resource limit r_{max} imposed on the analyzer:

$$cost(w, w_{total}, r, r_{max}) = \begin{cases} \dfrac{w + \frac{r}{r_{max}}}{w_{total}}, & \text{if } r \leq r_{max} \\ \infty, & \text{otherwise} \end{cases}$$

Note that w and r are measured by invoking the abstract interpreter with the recipe under evaluation. The cost function evaluates to a lower cost for recipes that improve the precision of the abstract interpreter (due to the term w/w_{total}). In case of ties, the term r/r_{max} causes the function to evaluate to a lower cost for recipes that result in a more efficient analysis. In other words, for two recipes resulting in equal precision, the one with the smaller resource consumption is assigned a lower cost. When a recipe causes the analyzer to exceed the resource limit, it is assigned infinite cost.

5.4.3. Recipe Generation

In the literature, there is a broad range of optimization algorithms for different application domains. To demonstrate the generality and effectiveness of TAILOR, we instantiate it with four adaptations of three well-known optimization algorithms, namely random sampling [175], hill climbing (with regular restarts) [227], and simulated annealing [147, 184]. Here, we describe these algorithms in detail, and in Sect. 5.5, we evaluate their effectiveness.

Before diving into the details, let us discuss the suitability of different kinds of optimization algorithms for our domain. There are algorithms that leverage mathematical properties of the function to be optimized, e.g., by computing derivatives as in Newton's iterative method. Our cost function, however, is evaluated by running an abstract interpreter, and thus, it is not differentiable or continuous. This constraint makes such analytical algorithms unsuitable. Moreover, evaluating our cost function is expensive, especially for precise abstract domains such as Polyhedra. This makes algorithms that require a large number of samples, such as genetic algorithms, less practical.

Now recall that Alg. 6 is parametric in how new recipes are generated (with GENERATERECIPE) and accepted for further mutations (with ACCEPT). Instantiations of these functions essentially constitute our search strategy for a tailored recipe. In the following, we discuss four such instantiations. Note that, in theory, the order of recipe ingredients matters. This is because any properties verified by one ingredient are converted into assumptions for the next, and different assumptions may lead to different verification results. Therefore, all our instantiations are able to explore different ingredient orderings.

Algorithm 7: A recipe-generator instantiation.

```
1  procedure GenerateRecipe(rec, l_max)
2    act := RandomAction({ADD: 0.2, MOD: 0.8})
3    if act = ADD ∧ Len(rec) < l_max then
4      ingr_new := RandomPosetLeastIncomparable(rec)
5      rec_mut := AddIngredient(rec, ingr_new)
6    else
7      ingr := RandomIngredient(rec)
8      act_m := RandomAction({GT: 0.5, LT: 0.3, INC: 0.2})
9      if act_m = GT then
10       ingr_new := PosetGreaterThan(ingr)
11     else if act_m = LT then
12       ingr_new := PosetLessThan(ingr)
13     else
14       rec_rem := RemoveIngredient(rec, ingr)
15       ingr_new := RandomPosetLeastIncomparable(rec_rem)
16     rec_mut := ReplaceIngredient(rec, ingr, ingr_new)
17
18   if ¬PosetCompatible(rec_mut) then
19     rec_mut := GenerateRecipe(rec, l_max)
20   return rec_mut
```

Random sampling. Random sampling (RS) just generates random recipes of a certain length. Function ACCEPT always returns *false* as each recipe is generated from scratch, and not as a result of any mutations.

Domain-aware random sampling. RS might generate recipes containing abstract domains of comparable precision. For instance, the Octagons domain is typically strictly more precise than Intervals. Thus, a recipe consisting of these domains is essentially equivalent to one containing only Octagons.

Now, assume that we have a partially ordered set (poset) of domains that defines their ordering in terms of precision. An example of such a poset for a particular abstract interpreter is shown in Fig. 5.3. An optimization algorithm can then leverage this information to reduce the search space of possible recipes.

Given such a poset, we therefore define domain-aware random sampling (DARS), which randomly samples recipes that do not contain abstract domains of comparable precision. Again, ACCEPT always returns *false*.

Simulated annealing. Simulated annealing (SA) searches for the best recipe by mutating the current recipe rec_{curr} in Alg. 6. The resulting recipe (rec_{next}), if accepted on line 12, becomes the new recipe to be mutated. Alg. 7 shows an instantiation of GENERATERECIPE, which mutates a given recipe such that the poset precision constraints are satisfied (i.e., there are no domains of comparable precision). A recipe is mutated either by adding new ingredients with 20% probability or by modifying existing ones with 80% probability (line 2). The probability

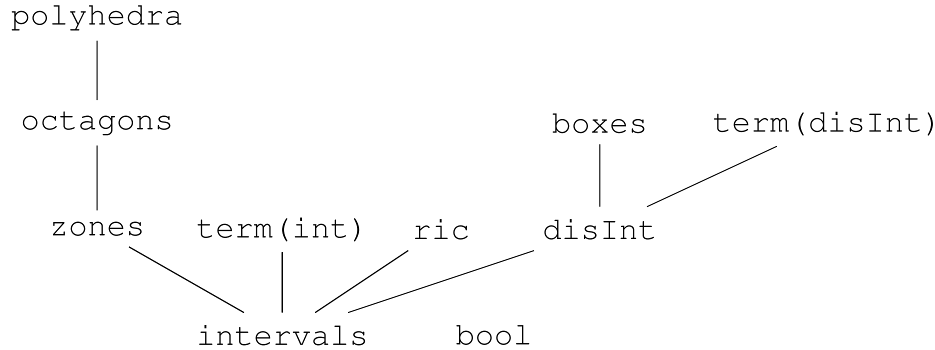


Figure 5.3: Comparing logico-numerical domains in CRAB. A domain d_1 is less precise than d_2 if there is a path from d_1 to d_2 going upward, otherwise d_1 and d_2 are incomparable.

of adding ingredients is lower to keep recipes short.

When adding a new ingredient (lines 4–5), Alg. 7 calls RANDOMPOSETLEAST-INCOMPARABLE, which considers all domains that are incomparable with the domains in the recipe. Given this set, it randomly selects from the domains with the least precision to avoid adding overly expensive domains. When modifying a random ingredient in the recipe (lines 7–16), the algorithm can replace its domain with one of three possibilities: a domain that is immediately more precise (i.e., not transitively) in the poset (via POSETGREATERTHAN), a domain that is immediately less precise (via POSETLESSTHAN), or an incomparable domain with the least precision (via RANDOMPOSETLEASTINCOMPARABLE). If the resulting recipe does not satisfy the poset precision constraints, our algorithm retries to mutate the original recipe (lines 18–19).

For simulated annealing, ACCEPT returns *true* if the new cost (for the mutated recipe) is less than the current cost. It also accepts recipes whose cost is higher with a certain probability, which is inversely proportional to the cost increase and the number of explored recipes. That is, recipes with a small cost increase are likely to be accepted, especially at the beginning of the exploration.

Hill climbing. Our instantiation of hill climbing (HC) performs regular restarts. In particular, it starts with a randomly generated recipe that satisfies the poset precision constraints, generates 10 new valid recipes, and restarts with a random recipe. ACCEPT returns *true* only if the new cost is lower than the best cost, which is equivalent to the current cost.

5.5. Experimental Evaluation

To evaluate our technique, we aim to answer the following research questions:

Table 5.1: CRAB settings and their possible values as used in our experiments. Default settings are shown in bold.

Setting	Possible Values
NUM_DELAY_WIDEN	$\{\mathbf{1}, 2, 4, 8, 16\}$
NUM_NARROW_ITERATIONS	$\{1, \mathbf{2}, 3, 4\}$
NUM_WIDEN_THRESHOLDS	$\{\mathbf{0}, 10, 20, 30, 40\}$
BACKWARD ANALYSIS	$\{\mathbf{OFF}, \mathit{ON}\}$
ARRAY SMASHING	$\{\mathit{OFF}, \mathbf{ON}\}$
ABSTRACT DOMAINS	all domains in Fig. 5.3

RQ1: Is our technique effective in tailoring recipes to different usage scenarios?

RQ2: Are the tailored recipes optimal?

RQ3: How diverse are the tailored recipes?

RQ4: How resilient are the tailored recipes to code changes?

5.5.1. Implementation

We implemented TAILOR by extending CRAB [116], a parametric framework for modular construction of abstract interpreters[4]. We extended CRAB with the ability to pass verification results between recipe ingredients as well as with the four optimization algorithms discussed in Sect. 5.4.3.

Tab. 5.1 shows all settings and values used in our evaluation. The first three settings refer to the strategies discussed in Sect. 5.3 for mitigating the precision loss incurred by widening. For the initial recipe, TAILOR uses Intervals and the CRAB default values for all other settings (in bold in the table). To make the search more efficient, we selected a representative subset of all possible setting values.

CRAB uses a DSA-based [117] pointer analysis and can, optionally, reason about array contents using array smashing. It offers a wide range of logico-numerical domains, shown in Fig. 5.3. The `bool` domain is the flat Boolean domain, `ric` is a reduced product of Intervals and Congruence, and `term(int)` and `term(disInt)` are instantiations of the Term domain with `intervals` and `disInt`, respectively. Although CRAB provides a bottom-up inter-procedural analysis, we use the default intra-procedural analysis; in fact, most analyses deployed in real usage scenarios are intra-procedural due to time constraints [68].

5.5.2. Benchmark Selection

For our evaluation, we systematically selected popular and (at some point) active C projects on GitHub. In particular, we chose the six most starred C repositories with

[4] CRAB is available at https://github.com/seahorn/crab.

Table 5.2: Overview of projects.

Project	Description
CURL	Tool for transferring data by URL
DARKNET	Convolutional neural-network framework
FFMPEG	Multimedia processing tool
GIT	Distributed version-control tool
PHP-SRC	PHP interpreter
REDIS	Persistent in-memory database

over 300 commits that we could successfully build with the Clang-5.0 compiler. We give a short description of each project in Tab. 5.2.

For analyzing these projects, we needed to introduce properties to be verified. We, thus, automatically instrumented these projects with four types of assertions that check for common bugs; namely, division by zero, integer overflow, buffer overflow, and use after free. Introducing assertions to check for runtime errors such as these is common practice in program analysis and verification.

As projects consist of different numbers of files, to avoid skewing the results in favor of a particular project, we randomly and uniformly sampled 20 LLVM-bitcode files from each project, for a total of 120. To ensure that each file was neither too trivial nor too difficult for the abstract interpreter, we used the number of assertions as a complexity indicator and only sampled files with at least 20 assertions and at most 100. Additionally, to guarantee all four assertion types were included and avoid skewing the results in favor of a particular assertion type, we required that the sum of assertions for each type was at least 70 across all files—this exact number was largely determined by the benchmarks.

Overall, our benchmark suite of 120 files totals 1346 functions, 5557 assertions (on average 4 assertions per function), and 667,927 LLVM instructions (Tab. 5.3).

5.5.3. Results

We now present our experimental results for each research question. We performed all experiments on a 32-core Intel ® Xeon ® E5-2667 v2 CPU @ 3.30GHz machine with 264GB of memory, running Ubuntu 16.04.1 LTS.

RQ1: Is our technique effective in tailoring recipes to different usage scenarios? We instantiated TAILOR with the four optimization algorithms described in Sect. 5.4.3: RS, DARS, SA, and HC. We constrained the analysis time to simulate two usage scenarios: 1 sec for instant feedback in the editor, and 5 min for feedback in a CI pipeline. We compare TAILOR with the default recipe (DEF), i.e., the default settings in CRAB as defined by its designer after careful tuning on a large set of benchmarks over the years. DEF uses a combination of two domains,

Table 5.3: Benchmark characteristics (20 files per project). The last three columns show the number of functions, assertions, and LLVM instructions in the analyzed files.

Project	Functions	Assertions	LLVM Instructions
CURL	306	787	50541
DARKNET	130	958	55847
FFMPEG	103	888	27653
GIT	218	768	102304
PHP-SRC	268	1031	305943
REDIS	321	1125	125639
Total	1346	5557	667927

namely, the reduced product of Boolean and Zones. The other default settings are in Tab. 5.1.

For this experiment, we ran TAILOR with each optimization algorithm on the 120 benchmark files, enabling optimization at the granularity of files. Each algorithm was seeded with the same random seed. In Alg. 6, we restrict recipes to contain at most 3 domains ($l_{max} = 3$) and set the number of iterations for each phase to be 5 and 10 ($i_{dom} = 5$ and $i_{set} = 10$).

The results are presented in Fig. 5.4, which shows the number of assertions that are verified with the best recipe found by each algorithm as well as by the default recipe. All algorithms outperform the default recipe for both usage scenarios, verifying almost twice as many assertions on average. The random-sampling algorithms are shown to find better recipes than the others, with DARS being the most effective. Hill climbing is less effective since it gets stuck in local cost minima despite restarts. Simulated annealing is the least effective because it slowly climbs up the poset toward more precise domains (see Alg. 7). However, as we explain later, we expect the algorithms to converge on the number of verified assertions for more iterations.

Fig. 5.5 gives a more detailed comparison with the default recipe for the time limit of 5 min. In particular, each horizontal bar shows the total number of assertions verified by each algorithm. The orange portion represents the assertions verified by both the default recipe and the optimization algorithm, while the green and red portions represent the assertions only verified by the algorithm and default recipe, respectively. These results show that, in addition to verifying hundreds of new assertions, TAILOR is able to verify the vast majority of assertions proved by the default recipe, regardless of optimization algorithm.

In Fig. 5.6, we show the total time each algorithm takes for all iterations. DARS takes the longest. This is due to generating more precise recipes thanks to its domain knowledge. Such recipes typically take longer to run but verify more assertions (as in Fig. 5.4). On average, for all algorithms, TAILOR requires only

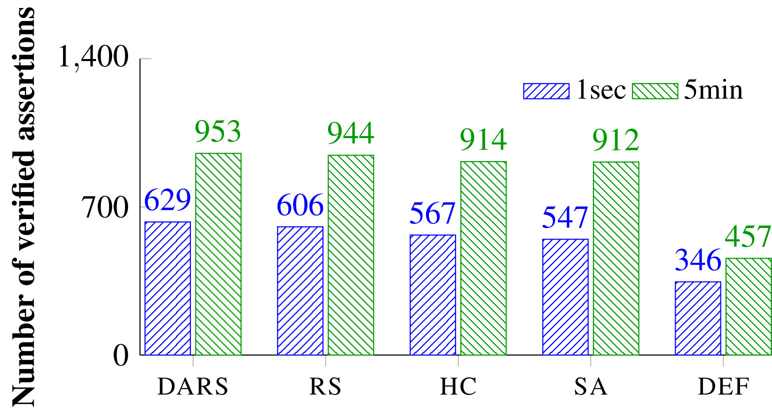


Figure 5.4: Comparison of the number of assertions verified with the best recipe generated by each optimization algorithm and with the default recipe, for varying timeouts.

30 sec to complete all iterations for the 1-sec timeout and 16 min for the 5-min timeout. As discussed in Sect. 5.2, this tuning time can be spent offline.

Fig. 5.7 compares the total number of assertions verified by each algorithm when TAILOR runs for 40 ($i_{dom} = 5$ and $i_{set} = 10$) and 80 ($i_{dom} = 10$ and $i_{set} = 20$) iterations. The results show that only a relatively small number of additional assertions are verified with 80 iterations. In fact, we expect the algorithms to eventually converge on the number of verified assertions, given the time limit and precision of the available domains.

As DARS performs best in this comparison, we only evaluate DARS in the remaining research questions. We use a 5-min timeout.

> **RQ1 takeaway:** TAILOR verifies between $1.6 - 2.1\times$ the assertions of the default recipe, regardless of optimization algorithm, timeout, or number of iterations. In fact, even very simple algorithms (such as RS) significantly outperform the default recipe.

RQ2: Are the tailored recipes optimal? To check the optimality of the tailored recipes, we compared them with the most precise (and least efficient) CRAB configuration. It uses the most precise domains from Fig. 5.3 (i.e., `bool`, `polyhedra`, `term(int)`, `ric`, `boxes`, and `term(disInt)`) in a recipe of 6 ingredients and assigns the most precise values to all other settings from Tab. 5.1. We generously gave a 30-min timeout to this recipe.

For 21 out of 120 files, the most precise recipe ran out of memory (264GB). For 86 files, it terminated within 5 min, and for 13, it took longer (within 30 min)—in many cases, this was even longer than TAILOR's tuning time in Fig. 5.6. We compared the number of assertions verified by our tailored recipes (which do not exceed 5 min) and by the most precise recipe. For the 86 files that terminated

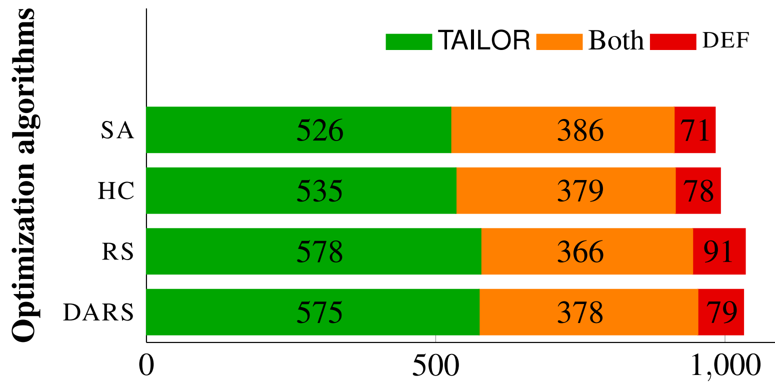


Figure 5.5: Comparison of the number of assertions verified by a tailored vs. the default recipe.

within 5 min, our recipes prove 618 assertions, whereas the most precise recipe proves 534. For the other 13 files, our recipes prove 119 assertions, whereas the most precise recipe proves 98.

Consequently, our (in theory) less precise and more efficient recipes prove more assertions in files where the most precise recipe terminates. Possible explanations for this non-intuitive result are: (1) Polyhedra coefficients may overflow, in which case the constraints are typically ignored by abstract interpreters, and (2) more precise domains with different widening operations may result in less precise results [16, 193].

We also evaluated the optimality of tailored recipes by mutating individual parts of the recipe and comparing to the original. In particular, for each setting in Tab. 5.1, we tried all possible values and replaced each domain with all other comparable domains in the poset of Fig. 5.3. For example, for a recipe including `zones`, we tried `octagons`, `polyhedra`, and `intervals`. In addition, we

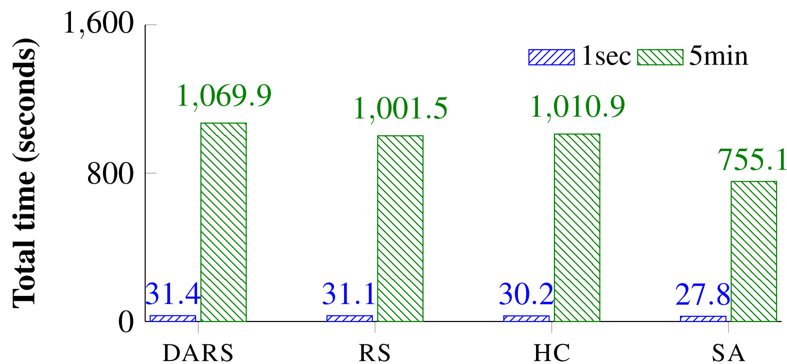


Figure 5.6: Comparison of the total time (in sec) that each algorithm requires for all iterations, for varying timeouts.

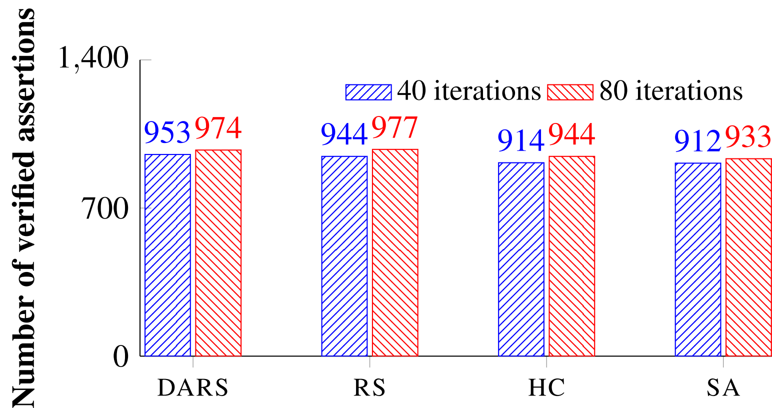


Figure 5.7: Comparison of the number of assertions verified with the best recipe generated by the different optimization algorithms, for different numbers of iterations.

tried all possible orderings of the recipe ingredients, which in theory could produce different results. We observed whether these changes resulted in a difference in the precision and performance of the analyzer.

Fig. 5.8 shows the results of this experiment, broken down by setting. Equal (in orange) indicates that the mutated recipe proves the same number of assertions within ± 5 seconds of the original. Positive (in green) indicates that it either proves more assertions or the same number of assertions at least 5 seconds faster. Negative (in red) indicates that the mutated recipe either proves fewer assertions or the same number of assertions at least 5 seconds slower.

The results show that, for our benchmarks, mutating the recipe found by TAILOR rarely led to an improvement. In particular, at least 93% of all mutated recipes were either equal to or worse than the original recipe. In the majority of these cases, mutated recipes are equally good. This indicates that there are many optimal or close-to-optimal solutions and that TAILOR is able to find one.

> **RQ2 takeaway:** As compared to the most precise recipe, TAILOR verified more assertions across benchmarks where the most precise recipe terminated. Furthermore, mutating recipes found by TAILOR resulted in improvement only for less than 7% of recipes.

RQ3: How diverse are the tailored recipes? To motivate the need for optimization, we must show that tailored recipes are sufficiently diverse such that they could not be replaced by a well-crafted default recipe. To better understand the characteristics of tailored recipes, we manually inspected all of them.

TAILOR generated recipes of length greater than 1 for 61 files. Out of these, 37 are of length 2 and 24 of length 3. For 77% of generated recipes, `NUM_DELAY_WIDEN` is not set to the default value of 1. Additionally, 55% of the ingredients

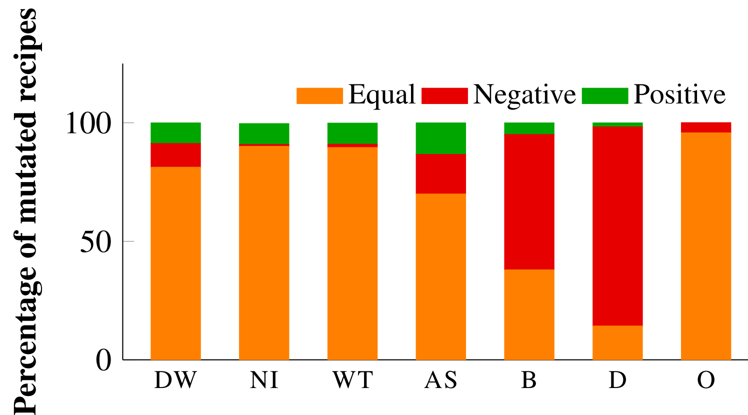


Figure 5.8: Effect of different settings on the precision and performance of the abstract interpreter. (DW: `NUM_DELAY_WIDEN`, NI: `NUM_NARROW_ITERATIONS`, WT: `NUM_WIDEN_THRESHOLDS`, AS: array smashing, B: backward analysis, D: abstract domain, O: ingredient ordering).

enable array smashing, and 32% enable backward analysis.

Fig. 5.9 shows how often (in percentage) each abstract domain occurs in a best recipe found by TAILOR. We observe that all domains occur almost equally often, with 6 of the 10 domains occurring in between 9% and 13% of recipes. The most common domain was `bool` at 18%, and the least common was `intervals` at 5%. We observed a similar distribution of domains even when instrumenting the benchmarks with only one assertion type, e.g., checking for integer overflow.

We also inspected which domain combinations are frequently used in the tailored recipes. One common pattern is combinations between `bool` and numerical domains (18 occurrences). Similarly, we observed 2 occurrences of `term(disInt)` together with `zones`. Interestingly, the less powerful variants of combining `disInt` with `zones` (3 occurrences) and `term(int)` with `zones` (6 occurrences) seem to be sufficient in many cases. Finally, we observed 8 occurrences of `polyhedra` or `octagons` with `boxes`, which are the most precise convex and non-convex domains. Our approach is, thus, not only useful for users, but also for designers of abstract interpreters by potentially inspiring new domain combinations.

RQ3 takeaway: The diversity of tailored recipes prevents replacing them with a single default recipe. Over half of the tailored recipes contain more than one ingredient, and ingredients use a variety of domains and their settings.

RQ4: How resilient are the tailored recipes to code changes? We expect

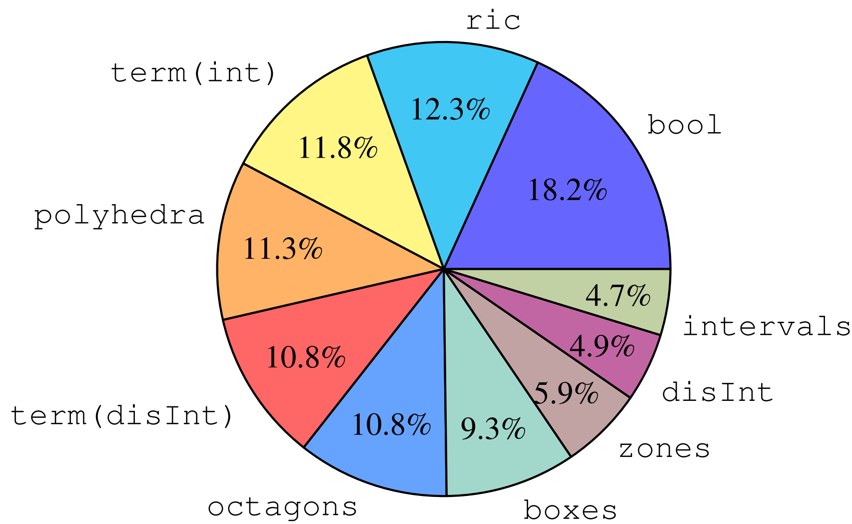


Figure 5.9: Occurrence of domains (in %) in the best recipes for all assertion types.

tailored recipes to be resilient to code changes, i.e., to retain their optimality across several changes without requiring re-tuning. We now evaluate if a recipe tailored for one code version is also tailored for another, even when the two versions are 50 commits apart.

For this experiment, we took a random sample of 60 files from our benchmarks and retrieved the 50 most recent commits per file. We only sampled 60 out of 120 files as building these files for each commit is quite time consuming—it can take up to a couple of days. We instrumented each file version with the four assertion types described in Sect. 5.5.2. It should be noted that, for some files, we retrieved fewer than 50 versions either because there were fewer than 50 total commits or our build procedure for the project failed on older commits. This is also why we did not run this experiment for over 50 commits.

We analyzed each file version with the best recipe, R_o, found by TAILOR for the oldest file version. We compared this recipe with new best recipes, R_n, that were generated by TAILOR when run on each subsequent file version. For this experiment, we used a 5-min timeout and 40 iterations.

Note that, when running TAILOR with the same optimization algorithm and random seed, it explores the same recipes. It is, therefore, very likely that recipe R_o for the oldest commit is also the best for other file versions since we only explore 40 different recipes. To avoid any such bias, we performed this experiment by seeding TAILOR with a different random seed for each commit. The results are shown in Fig. 5.10.

In Fig. 5.10, we give a bar chart comparing the number of files per commit that have a positive, equal, and negative difference in the number of verified assertions,

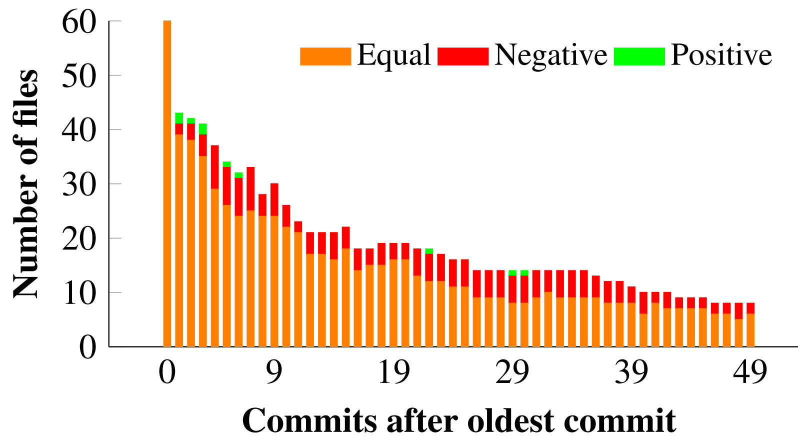


Figure 5.10: Difference in the safe assertions across commits.

where commit 0 is the oldest commit and 49 the newest. An equal difference (in orange) means that recipe R_o for the oldest commit proves the same number of assertions in the current file version, f_n, as recipe R_n found by running TAILOR on f_n. To be more precise, we consider the two recipes to be equal if they differ by at most 1 verified assertion or 1% of verified assertions since such a small change in the number of safe assertions seems acceptable in practice (especially given that the total number of assertions may change across commits). A positive difference (in green) means that R_o achieves better verification results than R_n, that is, R_o proves more assertions safe (over 1 assertion or 1% of the assertions that R_n proves). Analogously, a negative difference (in red) means that R_o proves fewer assertions. We do not consider time here because none of the recipes timed out when applied on any file version.

Note that the number of files decreases for newer commits. This is because not all files go forward by 50 commits, and even if they do, not all file versions build. However, in a few instances, the number of files increases going forward in time. This happens for files that change names, and later, change back, which we do not account for.

For the vast majority of files, using recipe R_o (found for the oldest commit) is as effective as using R_n (found for the current commit). The difference in safe assertions is negative for less than a quarter of the files tested, with the average negative difference among these files being around 22% (i.e., R_o proved 22% fewer assertions than R_n in these files). On the remaining three quarters of the files tested however, R_o proves at least as many assertions as R_n, and thus, R_o tends to be tailored across code versions.

Commits can result in both small and large changes to the code. We therefore also measured the average difference in the number of verified assertions per changed line of code with respect to the oldest commit. For most files,

regardless of the number of changed lines, we found that R_o and R_n are equally effective, with changes to 1000 LOC or more resulting in little to no loss in precision. In particular, the median difference in safe assertions across all changes between R_o and R_n was 0 (i.e., R_o proved the same number of assertions safe as R_n), with a standard deviation of 15 assertions. We manually inspected a handful of outliers where R_o proved significantly fewer assertions than R_n (difference of over 50 assertions). These were due to one file from GIT where R_o is not as effective because the widening and narrowing settings have very low values.

RQ4 takeaway: For over 75% of files, TAILOR's recipe for a previous commit (from up to 50 commits previous) remains tailored for future versions of the file, indicating the resilience of tailored recipes across code changes.

5.5.4. Threats to Validity

We have identified the following threats to the validity of our experiments.

Benchmark selection. Our results may not generalize to other benchmarks. However, we selected popular GitHub projects from different application domains (see Tab. 5.2). Hence, we believe that our benchmark selection mitigates this threat and increases generalizability of our findings.

Abstract interpreter and recipe settings. For our experiments, we only used a single abstract interpreter, CRAB, which however is a mature and actively supported tool. The selection of recipe settings was, of course, influenced by the available settings in CRAB. Nevertheless, CRAB implements the generic architecture of Fig. 5.2, used by most abstract interpreters, such as those mentioned at the beginning of Sect. 5.3. We, therefore, expect our approach to generalize to such analyzers.

Optimization algorithms. We considered four optimization algorithms, but in Sect. 5.4.3, we explain why these are suitable for our application domain. Moreover, TAILOR is configurable with respect to the optimization algorithm.

Assertion types. Our results are based on four types of assertions. However, these cover a wide range of runtime errors that are commonly checked by static analyzers.

5.6. Related Work

The impact of different abstract-interpretation configurations has been previously evaluated [259] for Java programs and partially inspired this work. To the best of our knowledge, we are the first to propose tailoring abstract interpreters to custom usage scenarios using optimization.

However, optimization is a widely used technique in many engineering disciplines. In fact, it is also used to solve the general problem of algorithm configu-

ration [129], of which there exist numerous instantiations, for instance, to tune hyper-parameters of learning algorithms [38, 95, 249] and options of constraint solvers [130, 131]. Existing frameworks for algorithm configuration differ from ours in that they are not geared toward problems that are solved by sequences of algorithms, such as analyses with different abstract domains. Even if they were, our experience with TAILOR shows that there seem to be many optimal or close-to-optimal configurations, and even very simple optimization algorithms such as RS are surprisingly effective (see RQ2); similar observations were made about the effectiveness of random search in hyper-parameter tuning [39].

In the rest of this section, we focus on the use of optimization in program analysis. It has been successfully applied to a number of program-analysis problems, such as automated testing [99, 100], invariant inference [238], and compiler optimizations [233].

Recently, researchers have started to explore the direction of enriching program analyses with machine-learning techniques, for example, to automatically learn analysis heuristics [121, 132, 216, 242]. A particularly relevant body of work is on adaptive program analysis [122–124], where existing code is analyzed to learn heuristics that trade soundness for precision or that coarsen the analysis abstractions to improve memory consumption. More specifically, adaptive program analysis poses different static-analysis problems as machine-learning problems and relies on Bayesian optimization to solve them, e.g., the problem of selectively applying unsoundness to different program components (e.g., different loops in the program) [124]. The main insight is that program components (e.g., loops) that produce false positives are alike, predictable, and share common properties. After learning to identify such components for existing code, this technique suggests components in unseen code that should be analyzed unsoundly.

In contrast, TAILOR currently does not adjust soundness of the analysis. However, this would also be possible if the analyzer provided the corresponding configurations. More importantly, adaptive analysis focuses on learning analysis heuristics based on existing code in order to generalize to arbitrary, unseen code. TAILOR, on the other hand, aims to tune the analyzer configuration to a custom usage scenario, including a particular program under analysis. In addition, the custom usage scenario imposes user-specific resource constraints, for instance by limiting the time according to a phase of the software-engineering life cycle. As we show in our experiments, the tuned configuration remains tailored to several versions of the analyzed program. In fact, it outperforms configurations that are meant to generalize to arbitrary programs, such as the default recipe.

5.7. Summary and Remarks

In this chapter, we have proposed a technique and framework that tailors a generic abstract interpreter to custom usage scenarios. We instantiated our framework with a mature abstract interpreter to perform an extensive evaluation on real-world

benchmarks. Our experiments show that the configurations generated by TAILOR are vastly better than the default options, vary significantly depending on the code under analysis, and typically remain tailored to several subsequent code versions.

Chapter 6

Input Splitting for Cloud-Based Static Application Security Testing Platforms

As software development teams adopt DevSecOps practices, application security is increasingly the responsibility of development teams, who are required to set up their own Static Application Security Testing (SAST) infrastructure. Since development teams often do not have the necessary infrastructure and expertise to set up a custom SAST solution, there is an increased need for cloud-based SAST *platforms* that operate as a service and run a variety of static analyzers. Adding a new static analyzer to a cloud-based SAST platform can be challenging because static analyzers greatly vary in complexity, from linters that scale efficiently to interprocedural dataflow engines that use cubic or even more complex algorithms. Careful manual evaluation is needed to decide whether a new analyzer would slow down the overall response time of the platform or may timeout too often.

We explore the question of whether this can be simplified by splitting the input to the analyzer into partitions and analyzing the partitions independently. Depending on the complexity of the static analyzer, the partition size can be adjusted to curtail the overall response time. We report on an experiment where we run different analysis tools with and without splitting the inputs. The experimental results show that simple splitting strategies can effectively reduce the running time and memory usage per partition without significantly affecting the findings produced by the tool.

6.1. Introduction

With the increasing popularity of DevSecOps development practices, Static Application Security Testing (SAST) shifts further to the left in the software de-

velopment life-cycle and becomes the responsibility of developers rather than security experts. This creates a growing demand for easy-to-use solutions. Many development teams do not have the capacity or expertise to configure and maintain their own static analysis infrastructure and prefer SAST *platforms* that offer a variety of static analyses on demand. Open-source platforms, such as the Software Assurance Marketplace (SWAMP) [151] or ShipShape [229], and their commercial alternatives offer a convenient abstraction. They provide a simple interface through which developers submit code and build artifacts (in their languages of choice) and receive recommendations on how to improve the code. Internally, such cloud-based SAST platforms may employ a variety of static analysis tools, such as [5, 10, 13, 71, 214, 220].

SAST platforms typically are run as a cloud-based service, and the individual analysis tools are containerized and instantiated on-demand on cloud-based machines. Developers expect such a SAST platform to handle inputs (codebases) of arbitrary complexity, and still deliver results within a certain time window. This is especially true for customers that integrate SAST platforms in their continuous integration and deployment (CI/CD) pipelines.

To maintain a predictable response time, SAST platforms face the challenge that they need to be able to scale to different sizes of inputs, and that, every time they add a new analysis tool, they have to ensure that the new tool does not slow down the response time for existing customers.

Vertical scaling by adding more memory or faster machines is not a cost-effective solution to the risk of running out of time or space when analyzing complex inputs. Provisioning machines large enough to handle the most complex analysis inputs would make the service unnecessarily expensive for customers that analyze smaller and simpler codebases. In the cloud, a large number of small machines is significantly less expensive than a small number of high-performance machines [237]. Moreover, since many SAST tools have superlinear time complexity [5, 217], even the most powerful machine will eventually not suffice. Much research has been conducted on adding various optimizations to improve the scalability of specific analysis engines, such as summarization of method calls [20, 217, 226], caching and reuse of partial results from prior analyses [5, 19], and incremental analysis [72, 253]. However, when operating a SAST platform, modifying the individual tools may not be an option because the tools might be proprietary or maintaining forks with custom modifications may be too costly.

Thus, a horizontal scaling strategy to distribute and balance the analysis load is still needed. Horizontal scaling needs to split up inputs into pieces such that each analysis tool employed by the platform can handle its input within the expected response time. The different pieces can then be analyzed on parallel instances of a given analysis tool. Such a horizontal scaling can be configured per analysis tool, but without modifying the tool itself. More complex tools can be configured to handle smaller pieces of code than light-weight tools to ensure that the overall latency of the platform does not change when a new complex tool gets added.

In this chapter, we present an approach to horizontally scale analysis tools in a

static analysis platform. Our approach takes as input a program and a bound for the size of code that should be analyzed by each single machine. It then employs a configurable splitting strategy to split the input program into partitions such that the amount of code in each partition is below the provided bound. We evaluate how this splitting process affects the accuracy of different static analysis tools and how the computational cost of analyzing partitions in parallel relates to the cost of analyzing the entire input program.

Splitting code into partitions comes with several challenges. The first challenge is that information may be lost because dependent code fragments are placed in separate partitions. This may impact the precision and recall of static analysis tools. For example, a real defect arising from the interaction between two classes may become a false negative if those classes end up in different partitions. Similarly, the evidence that a vulnerability has been correctly mitigated may become invisible when defect and mitigation split across partitions, yielding a false positive. This leads to our first research question; **RQ1:** What is the impact of splitting a program and analyzing the partitions in isolation on a tool's accuracy?

The second challenge when splitting code into partitions is that the complexity of static analysis may not be tied just to the size of the code. For example, data-flow analysis is cubic in the size of data-flow facts that are tracked [218]. That is, if data-flow facts are not evenly distributed across the program, splitting may not reduce the overall time or memory consumption of data-flow analysis if all facts end up in the same partition. Other analysis techniques, such as bi-abduction used by INFER [60], may require a different type of partitioning since their complexity is not tied to data-flow facts. Hence, we cannot guarantee that analyzing a partition uses less time or memory than analyzing the original program. So our second research question is **RQ2:** How do static analyzers perform on the partitions compared to the original program in terms of time and memory usage?

A third challenge is to find a splitting strategy that works for different kinds of static analysis tools. Splitting strategies may have different complexities for static analysis tools targeting different languages. E.g., identifying the direct dependencies of a Java class file is roughly constant since it is sufficient to look at the constant pool [43]. In Python, however, one has to iterate over the entire syntax tree of a file to determine its dependencies. A splitting strategy that takes dependencies into consideration is computationally more expensive for Python than for Java. Thus, our third research question is **RQ3:** What kinds of static analysis tools would benefit from splitting strategies discussed in the chapter?

To answer these three research questions, we implement a splitting approach that works with two different strategies to create partitions. The first strategy, SIZELIMITING, naïvely splits the input program into partitions based on an upper bound S on the number of files (or classes) per partition. Sorted files in lexicographical order are added to a partition until this bound S is reached and then, a new partition is started. The second strategy, SPLITMERGE, uses dependency information between the files of the input program to create partitions that include the necessary dependencies of a file. In SIZELIMITING, all partitions are disjoint,

while in SPLITMERGE, partitions can overlap.

We apply these two splitting strategies to a set of benchmark programs and analyze the resulting partitions with the static analysis tools RAPID [92] and INFER [60]. We evaluate the impact of both splitting strategies over non-splitting on these analysis tools in terms of reported findings and computational performance.

Contributions. The contributions of this chapter are as follows:

1. We motivate why input splitting is a relevant problem for SAST platforms and why additional research in this area is required.
2. We present experimental results that input splitting can work in practice with different SAST tools.
3. We show that with a proper selection of splitting strategy, all evaluated SAST tools can benefit from splitting. Yet finding the right splitting strategy depends on the complexity of the used SAST tool. While tools like RAPID and INFER which perform complex analyses benefit most from dependency-guided-splitting strategy like SPLITMERGE in terms of reduction in latency, memory consumption and minimizing the loss of findings, for inexpensive linter-like or intra-procedural analyses such as Bandit, a naive strategy like SIZELIMITING may be more beneficial.

We do not claim that any of the proposed strategies are optimal, nor that splitting is the only way to increase the maximum tractable problem size. Instead, this evaluation demonstrates how a lightweight splitting strategy can already significantly improve latency, scalability, and cost-effectiveness of cloud-based SAST platforms. For the future, we envision that such strategies can be used to reduce the cost of integrating new static analysis tools into a SAST platform. Moreover, instead of developing and benchmarking explicit splitting strategies for every new tool, a splitting algorithm could be generalized to adjust the splitting strategy based on the number of observed timeouts.

6.2. Motivating Example

We motivate the need for splitting with an example from the OWASP Benchmark[1]. This standard benchmark for Java SAST tools consists of 2,740 test cases for different types of security vulnerabilities. Each test case is a single Java file. The benchmark also has an additional 162 Java files that contain common helper classes which are used by multiple test cases.

The benchmark's public repository also includes score cards that show the performance of different SAST tools, an excerpt of which is displayed in Table 6.1. For example, the open-source tool FindSecBugs [205] (v1.4.6) analyzes the

[1] https://github.com/OWASP-Benchmark/BenchmarkJava/tree/53878cc8751e348b63de951b91a6d47cf29121d8/

Table 6.1: Score (based on precision and recall) and analysis time for several SAST tools on the OWASP Benchmark v1.1. Data taken from the OWASP Benchmark public repository.

Tool Name	OWASP Score	Total Time
FBwFindSecBugs v1.4.6	39.10%	0:02:02
SonarQube Java Plugin v3.14	33.34%	0:05:30
Commercial SAST-01	16.74%	2:55:20
Commercial SAST-02	30.60%	135:23:38
Commercial SAST-03	24.89%	1:52:00
Commercial SAST-04	32.64%	13:54:20

benchmark in just over two minutes and obtains a score of 39.1% – the OWASP score is based on the precision and recall of a tool's findings, with 100% for finding all and only the (known) vulnerabilities and 0% for only false positives and false negatives. FindSecBugs performs a lightweight analysis based on type propagation and thus scales linearly with the size of the program. For other tools in the benchmark's score cards, we can see that scalability may be an issue. The tools denoted as SAST-01 to SAST-04 in Table 6.1 have running times ranging from hours to days. Delays of such magnitude might be unacceptable for CI/CD customers.

If we provide a static analysis platform that runs multiple tools as a portfolio, customers would have to wait for the slowest tool to terminate before getting the final results (there are usually postprocessing steps, like de-duplication, before the unified results are returned). This makes it harder to add new tools, like SAST-02, to the portfolio. Hence, we would like a mechanism to split the program under analysis into smaller partitions, assuming that the analysis tool that we want to integrate terminates faster on (most) partitions so we can analyze these partitions in parallel and return results without increasing the latency of our analysis platform.

That is, for OWASP, we would like to split the 2,740 test cases into a set of partitions, each of them bounded by some size S that ensures our analysis terminates within an acceptable amount of time. We would also like each partition to contain the subset of the shared 162 classes that are used by any of the tests in that partition. Finally, we would like to minimize the number of partitions, since a very large number of very small partitions would amplify the impact of per-partition overhead, thus decreasing efficiency.

We illustrate the idea of splitting and the different splitting strategies using the listing in Figure 6.1, a simplified version of the test `BenchmarkTest01025`. The test contains a CWE22 (Path Traversal) vulnerability: the value received from `request.getHeader` is used in a relative pathname without input validation. An attacker could provide an input like `../../etc/passwd` to try to access sensitive data. This test calls helper method `doSomething` in class `Test1`, which is one of the 162 classes that are used by multiple tests. This

```java
public class Thing1 implements ThingInterface {
  public String doSomething(String i) {
    String r = i;
            return r;
    }
}

// Simplified version of the OWASP BenchmarkTest 01025
public class BenchmarkTest01025 extends HttpServlet {
    public void doPost(HttpServletRequest req,
                      HttpServletResponse response)
                      throws Exception {
        String p = req.getHeader("foo");
        String bar = new Thing1().doSomething(p);
        File fileTarget = new File("./tmp", bar);
        response.getWriter().println("...");
    }
}
```

Figure 6.1: Simplified version of an OWASP test that uses a shared class. The method *doSomething* is referenced 347 times in different OWASP tests.

method is called by a total of 347 tests in the OWASP benchmark, such as `BenchmarkTest01026` and `BenchmarkTest01029`.

Suppose the available compute instances (virtual machines) allow a certain tool to analyze up to 100 files before it risks exceeding the SLA (Service Level Agreement) time limit. This means we must split the OWASP benchmark into a set of partitions, each of them of size at most $S \leq 100$.

Naïve splitting (SIZELIMITING). First, we discuss a naïve strategy called SIZELIMITING which splits the codebase into non-overlapping subsets of up to S files each. To ensure determinism, the files are sorted in lexicographical order with respect to their names. Splitting is then performed on the sorted files. For the OWASP benchmark, which has 2,740 test classes and 162 shared classes (for a total of 2,902 files), this may produce, for example, 29 partitions of size 100 and one partition of size 2.

Since method `doSomething` in class `Test1` is called by 347 tests, we know these tests will be distributed over at least 4 partitions. That is, all but one of these partitions will not have access to the implementation of `doSomething` when running the static analysis. Depending on the analysis tool and its assumption on missing methods, this may result in a loss of findings, if the analysis under-approximates; or it may lead to false positives, if the analysis over-approximates; or it may crash the tool.

For this example, we need a splitting strategy that is able to create overlapping partitions to reduce the number of unavailable code dependencies in each partition.

In the following sections we outline such a strategy, called SPLITMERGE, and then evaluate its effect on the number of findings compared to the naïve strategy and to not splitting at all. We also evaluate the overhead of computing partitions and possibly reanalyzing code that is shared between partitions.

6.3. The SPLITMERGE Strategy

We aim to distribute the analysis of a program P, consisting of n files[2] $F = \{f_1, ..., f_n\}$, by splitting the program into partitions $R = \{r_1, \ldots, r_m\}$ (with $m \leq n$) such that each partition r_i contains no more than S files and can be analyzed independently with the target analysis tools. We ensure that the union of all partitions contains all files ($\cup_i r_i = F$). In general, partitions are not required to be disjoint, i.e., the same file may be replicated across multiple ones.

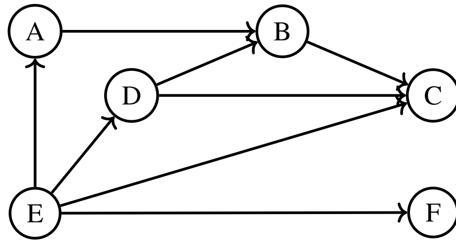
Algorithm overview. SPLITMERGE consists of three steps. Initially, a partition is created for each file in the codebase, which includes the file itself and its transitive dependencies up to a distance k. The distance k is a parameter of SPLITMERGE that allows to trade-off the size vs the degree of self-containment of the initial partitions. For the example in Figure 6.1, for $k = 2$, the initial partitions are {`BenchmarkTest01025`, `Thing1`, `ThingInterface`} and {`Thing1`, `ThingInterface`}.

The second step – *Split* – ensures none of the initial partitions exceeds the maximum size S by splitting any partition exceeding the size limit, while doing its best effort to preserve the dependency relations it contains. This step replicates the nodes with high degree of connectivity in all the split subsets, with the intuition that units with high connectivity are likely to carry semantic information shared by multiple subproblems.

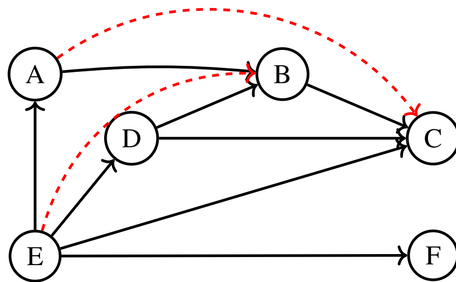
Finally, the third step – *merge* – takes as input a set of partitions of size less or equal than S and performs two tasks: 1) eliminate redundant partitions subsumed by others and 2) merge small partitions into larger ones to balance the load and further increase self-containment. A partition is redundant if it is entirely contained into another. In our example, the partition {`Thing1`, `ThingInterface`} can be dropped since the remaining partitions entirely cover its files and local dependencies. Merging small partitions to maximize the size of their union, constrained by this size being smaller than S, can be framed as a restricted instance of a bin-packing problem [135, 173]. The optimal solution to this problem converges to the smallest number of partitions with approximately uniform size S that cover the input codebase and is expected to balance the analysis load by assigning one partition to each executor.

In the remainder of this section we will detail each step of SPLITMERGE, with the help of a simplified example.

[2]In this chapter we focus on files as the elementary units to partition for analysis, which is a suitable setting for Java and Python. Our splitting strategy can in principle be applied to other language-specific units.



(a) Initial dependency graph of program P.



(b) Dependency graph of P augmented with transitive relations up to radius $k = 2$ (red dashed edges).

Figure 6.2: Dependency graphs of P.

Running example. Consider an example program P containing six files: A, B, C, D, E, F. The dependencies among these files are described in Figure 6.2a, where a directed edge (x, y) from x to y denotes that x depends on y (symmetrically, that y is a dependency of x). Such dependencies can typically be computed statically in linear time with the size of P, using tools such as JDeps [204] for Java or Snakefood [44] for Python. In the following, we will refer to files and vertices, and dependencies and edges interchangeably via the dependency graph.
Step 1: Initial partitions. This step produces an initial set of partitions of the program P aiming at preserving local dependencies. Given a program P composed of a finite set of files $F = \{f_0, f_1, \dots\}$ and a neighborhood radius $k > 0$, Alg. 8 constructs for each file a partition including the file itself and its neighbors up to distance k. A large value for k makes the algorithm more conservative in preserving dependency information. However, it also increases redundancy and the likelihood to produce partitions larger than the size limit S. In Alg. 8, after computing the dependency graph, the first loop augments the dependency relation to include edges linking a vertex to its neighbors up to distance k, while the second loop builds one partition per vertex including it transitive dependencies up to distance k. For a sparse enough dependency graph with n vertices and $k << n$, which is a common situation in practical systems where coupling should be minimized, the algorithm runs in nearly $\Theta(n)$; the worst case complexity would be $O(n^3)$ for $k \approx n$ and a fully connected graph (by reduction to computing the graph transitive closure), although it is unlikely for any realistic program to resemble this situation. The function `computeDependencyGraph` returns the vertices and edges of

Algorithm 8: Generate initial partitions

```
1  procedure GENERATEINITIALPARTITIONS(F, k)
2    (V,E) ← COMPUTEDEPENDENCYGRAPH(F)
3    // augment dependency relation
4    for each v ∈ V do
5      neighbors ← VERTICESWITHINDISTANCE(v,k)
6      for each n ∈ neighbors do
7        E ← E ∪ (v,n)
8    // build intial partitions
9    E ← ∅
10   for each v ∈ V do
11     r ← {v}
12     for each v ∈ V do
13       E ← r ∪ {u}
14     R ← R ∪ r
15   return R
```

the dependency graph. Each vertex of the graph corresponds to one file of the program under analysis.

Example. The dependency graph of our example program P is shown in Figure 6.2a. After the execution of the first loop in Alg. 8 with $k = 2$, the dependency relation is augmented with the transitive dependencies shown in red in Figure 6.2b – (A, C) and (E, B). The resulting initial partitions are thus:

$$\{\text{A, B, C}\}, \{\text{B, C}\}, \{\text{C}\}, \{\text{D, B, C}\}, \{\text{E, A, B, C, D, F}\}, \{\text{F}\}$$

Step 2: Split. Some initial partitions may have size larger than the maximum S. This is especially likely for larger values of the neighborhood radius k. This step aims at splitting an oversized partition r_i into smaller sets that fit within the size limit. However, uniformly splitting r_i into the minimum number or necessary disjoint subsets is likely to delete relevant dependency information. Instead, we deliberately produce a non-minimal number of subsets allowing redundancy to preserve dependency information. In particular, for a partition r_i that exceeds the maximum size ($|r_i| > S$), we sort the vertices in descending degree of connectivity (number of incoming and outgoing edges) and identify two sets of vertices: *high-connectivity*, which includes the p (a percentage) of vertices with the largest degrees of connectivity, and *low-connectivity* ones, which includes the rest of the vertices. The underlying intuition is that files involved with many dependency chains are likely to be relevant for the analysis of most subsets of r_i. Therefore, Alg. 9 first identifies these two sets and then partitions the low-connectivity vertices uniformly into small enough subsets to allow adding to each such subset the high-connectivity vertices. This operation is formalized in the `split` function, which is applied on each partition whose size exceeds S (line 12

Algorithm 9: Split

```
1  procedure SplitPartitions(G, R, p, S)
2    for each r ∈ R do
3      if |r| > S then
4        R ← (R \ {r}) ∪ Split(r, p, G, S)
5    return R

6  procedure Split(r, p, G, S)
7    (V_T, E_T) ← ExtractSubGraph(G, r)
8    V_{T_S} ← SortByDegreeDesc(V_T)
9    // the p highest degree nodes are replicated in each subset
10   p_r ← ⌊p · |r|⌋
11   if |r| > S then
12     p_r ← ⌊p · S⌋
13   // high-connectivity
14   hdn ← [v_{T_s 0}, ..., v_{T_s p_r}]
15   // low-connectivity
16   ldn ← [v_{T_s p_r + 1}, ...]
17   nSubsets ← ⌊(|r| − p_r) / (S − p_r)⌋ + 1
18   // Divide ldn uniformly into nSubsets parts
19   ldn ← {ldn_0, ldb_1, ..., ldn_{nSubsets−1}}
20   return {hdn ∪ ldn_0, hdn ∪ ldb_1, ..., hdn ∪ ldn_{nSubsets−1}}
```

handles the corner case of the chosen p not small enough to ensure splitting all oversized partitions.)

Example. Consider $S = 4$. The partition $\{E,A,B,C,D,F\}$ exceeds this size. In Figure 6.2b, vertex E has a degree of connectivity 5, B and C have degree 4, A and D have degree 3, F has degree 1. Let $p = 1/3$, E and B are selected as the high-connectivity vertices, leading to new partitions $\{E,B,A,C\}, \{E,B,D,F\}$ as replacement of $\{E,A,B,C,D,F\}$ (where vertices with the same degree have been sorted alphabetically).

Step 3: Merge. The last step of SplitMerge reduces the redundancy introduced by the previous steps and computes the final partition (Alg. 10). Some partitions computed by the first two steps may be subsumed by others. For example, $\{B,C\} \subseteq \{A,B,C\}$ in the partitions for our program P. In these situations, the information contained in the larger set subsumes the information in any of its subsets. The subsets can therefore be discarded, without loss of information (first loop in Alg. 10).

The second part of this step aims at grouping together partitions for the sake of balancing the analysis load distribution across multiple executors. This can be framed as an instance of the bin packing problem [173], where a set of items – the partitions – have to fit within the minimum number of bins of size S. While

Algorithm 10: Merge partitions

```
1  procedure MERGE(G, R, p, S)
2    for each r_i ∈ R do
3      if ∃r_j ∈ R s.t. r_i ⊆ r_j and i ≠ j then
4        R ← R \ {r_i}
5    return NEXTFIT(R, S)

6  procedure NEXTFIT(R, S)
7    T_s ← SORTBYSIZEASC(R)
8    R' ← ∅
9    r ← ∅
10   for each t ∈ T_s do
11     if |r| + |t| ≤ S then
12       r ← r ∪ t
13     else
14       R' ← R' ∪ r
15       r ← {t}
16   R' ← R' ∪ r
17   return R'
```

finding the optimal solution is NP-hard, many heuristics have been proposed to efficiently compute near-optimal solutions [135]. Among these, we adopted *next fit* [23], which has a time complexity of $O(n \log n)$ in the number of partitions n (due to sorting). Although, it may result in up to twice the optimal number of partitions, its fast execution time is preferred for the sake of minimizing the maximum analysis latency. Different algorithms can replace `nextFit` to trade off latency for a smaller number of parallel executors.

Example. In our small-size example, the merge phase would result in the final partitioning already after the redundancy reduction phase, since any further merging by `nextFit` would result in an oversized partition. The final partitions are: $\{D,B,C\},\{E,B,A,C\},\{E,B,D,F\}$.

After the three steps of SPLITMERGE, the resulting partitions satisfy the desired properties: (1) each partition is smaller than the prescribed size S, i.e., $|r_i| \leq S$; (2) the union of the partitions contains all files of the input program, i.e., $\cup_i r_i = F$. In the next section, we introduce our empirical evaluation on the impact of splitting strategies in comparison to non-splitting strategies.

6.4. Experimental Evaluation

In this section we report on our experiments using SPLITMERGE with three analysis tools on a portfolio of Java and Python benchmarks. Our evaluation will revolve around the following three research questions:

RQ1: What is the impact of splitting a program and analyzing the partitions in isolation on a tool's accuracy?

RQ2: How do static analyzers perform on the partitions compared to the original program in terms of time and memory usage?

RQ3: What kinds of static analysis tools would benefit from splitting strategies discussed in the chapter?

6.4.1. Experimental Settings

Static analysis tools. We used two industrial static analysis tools with interprocedural analysis capabilities –RAPID [92] and INFER [5]. RAPID is a tool developed at AWS that performs IFDS/IDE-based [217] type-state analysis to detect incorrect usage of cloud-service APIs. INFER is a static analysis tool developed at Facebook that uses separation logic to detect memory-related issues such as null pointer exceptions, resource leaks, and concurrency race conditions. A third set of experiments will instead use Bandit [13], a static analysis tool to find common security issues in Python. Unlike the other tools in our experiments, Bandit processes each source code file individually.

Benchmark programs. We use three different benchmark suites in our experiment:

- The OWASP Benchmark (v1.2) [14] (`OWASP`), a well-known Java-based web application designed to evaluate the accuracy, coverage, and speed of automated software vulnerability detection tools. It contains 2,740 labeled test cases that demonstrate common web app vulnerabilities, including, e.g., command injection, weak cryptography, path traversal.

- The Juliet Test Suite For Java [202] (`Juliet`), created by the NSA's Center for Assured Software (CAS) specifically for testing static analysis tools. It comprises 28,881 test cases that contain vulnerabilities for 112 different CWEs.

- Open-source packages from Maven Central [178] (`Maven`): Starting from a set of 26,142 open-source Java. packages randomly sampled from Maven, we ran RAPID on all packages and, for each package, recorded the number of "seeds" (elements in the codebase that may lead to potential findings). This can be done in linear time. We filtered out packages with no seeds, since their analysis with RAPID is very inexpensive. We also removed any packages that crashed the tool. To make the splitting problem more challenging, out of the 4,611 remaining packages we selected those with at least 1,500 classes. That left us with 138 Maven packages.

Baseline and experiments. We evaluate the SPLITMERGE splitting strategy in comparison with the naïve SIZELIMITING splitting strategy described in Sect. 6.2, and also against two baseline configurations that do not perform any splitting:

- No Splitting, Unlimited Time (`NoSplit-UT`): no splitting, 16Gb memory, 24h timeout. This strategy approximates the absence of latency constraints. We use the findings reported with `NoSplit-UT` as reference to assess accuracy drops due to splitting.

- No Splitting, Unlimited Memory (`NoSplit-UM`): no splitting, 144Gb memory, 10 minutes timeout. This strategy imposes the same timeout we will use for splitting, but allows the analyzers to use virtually unlimited memory (no tool saturated the available memory in our experiments).

SPLITMERGE and SIZELIMITING are allowed 16Gb of memory and 10 minutes timeout. We run all the experiments on Amazon EC2 C5.18xlarge instances (72 vCPUs, 144Gb RAM). We do not limit the number of cores a tool can use. We use Amazon Linux as operating system and `ulimit` to enforce memory limits.
Performance metrics. To evaluate our research questions, we collect the following metrics throughout the experimental campaign:

- **Total findings**: The number of unique findings reported by each tool, used as a proxy to detect accuracy losses. When different splitting strategies are applied, we compare the number of findings against the baselines to estimate the impact of splitting. Notably, a tool may also report false positive findings in either the baseline or after splitting. In general, we do not have a reliable means to discriminate between true and false positives and for the sake of this work we pragmatically assume that, ideally, a splitting strategy should result in exactly the same set of findings as `NoSplit-UT`; differences would suggest an impact on the accuracy of the tool.

- **Best possible latency**: The longest analysis time for any of the partitions of the input program. This is the minimum waiting time for the user, excluding other network and service invocation latency.

- **Total time**: Cumulative analysis time for all partitions. An index of the cumulative cost in computation time. Its value is related to the computational overhead induced by the redundancy allowed when splitting.

- **Peak heap usage**: The maximum Java heap memory used by an analysis tool written in Java. In our experiment, this metric is only measured for RAPID, which is written in Java.

- **Peak memory/max resident set size (RSS)**: The maximum amount of memory held by the process running an analysis tool at any time.

- **Number of partitions**: The number of partitions produced by a strategy for a given benchmark.

- **Sum of partition sizes**: The sum of storage size for all partitions produced in an experiment.

Configuration. SPLITMERGE is executed with: maximum partition size $S = 500$, neighborhood radius $k = 2$, and the percentage of high-connectivity vertices $p = 0.1$. These configuration values control the trade-off between latency, total CPU time, maximum memory, and impact on precision. For the experiments reported in this chapter, we prescribed a maximum allowed latency of 10 minutes and a maximum of 16Gb of memory per tool process and systematically swept the configuration space to make sure the selected configuration comfortably allows analyzing a partition within our prescribed latency and memory limits. We acknowledge that different latency and resource constraints, benchmarks, and analysis tools may require different tuning of the parameters.

6.4.2. Experimental Results

We now present our experimental results for each research question.

RQ1: What is the impact of splitting a program and analyzing the partitions in isolation on a tool's accuracy?

We answer this question by looking at the total findings detected by the analyzers shown in Tab. 6.2 reporting on the `OWASP`, `Juliet`, and `Maven` benchmarks. In the table, **X** means the analyzer did not terminate within the given timeout or crashed, thus no results were reported. On all three benchmarks, SPLITMERGE allows both RAPID and INFER to detect more findings in comparison to SIZELIMITING.

Regarding accuracy, we take the results of `NoSplit-UT` as the baseline for comparison (except for RAPID on `Juliet`, where this strategy did not produce a result). On `OWASP`, SPLITMERGE allowed RAPID to detect exactly the same number of findings (3,326) as with `NoSplit-UT`, without losing accuracy. We also compared the output of the tool with both strategies: the set of findings is exactly the same. In contrast, we lost 509 (3,326 − 2,817) findings with the naïve splitting strategy SIZELIMITING, corresponding to 15% (509/3,326) of the total findings that can be detected by RAPID without splitting. For INFER, splitting the original code using SIZELIMITING impacts its recall negatively, as INFER detected much fewer findings compared with non-splitting strategies (230 vs. 401). In contrast, SPLITMERGE allowed INFER to detect exactly the same findings as with non-splitting strategies.

On `Juliet`, RAPID did not finish the analysis using non-splitting strategies, which gave us no baseline to assess the impact of splitting on its accuracy, besides observing that SPLITMERGE returned more findings than SIZELIMITING. Similarly to `OWASP`, splitting also resulted in loss of findings on Juliet for INFER. It also turns out that INFER crashed (exited with non-zero return code) when analyzing the partitions. As shown in Table 6.3, the crash rate is 2% with SIZELIMITING and 6% with SPLITMERGE. We conjecture INFER is less tolerant to absences of dependent classes in comparison to RAPID, but further investigation is needed.

Table 6.2: Comparing both SizeLimiting and SplitMerge to baselines on `OWASP`, `Juliet` and `Maven`. The percentages in the rows for SizeLimiting and SplitMerge strategies correspond to the reduction (or gain) in the number of findings, total time and memory usage when compared with `NoSplit-UT`. Best Possible Latency column shows the speedup achieved with SizeLimiting and SplitMerge strategies. In the cases where `NoSplit-UT` failed to give a result within 24 hours, we report the speedup as ∞x and the number of findings as N/A.

Strategy	Total Findings		Best Possible Latency (min)		Total Time (min)		Peak Heap Usage (MB)	Peak Memory Max RSS (MB)		No. of Part.	Sum of Part. Sizes (MB)
	Rapid	Infer	Rapid	Infer	Rapid	Infer	Rapid	Rapid	Infer		
OWASP											
NoSplit-UT	3,326	401	55.5	1.3	55.5	1.3	2,215	5,940.8	212.2	1	24.8
NoSplit-UM	**X**	401	**X**	1.3	**X**	1.3	**X**	**X**	213.6	1	24.8
SizeLimiting	2,817 (-15%)	230 (-42%)	1 (55x)	0.01 (130x)	6.8 (-87.7%)	0.7 (-46.1%)	271 (-87.7%)	5,406.8 (-8.9%)	81.8 (-61.4%)	10	24.8
SplitMerge	3,326 (0%)	401 (0%)	1.4 (39x)	0.01 (130x)	8.6 (-84.5%)	0.9 (-30.7%)	304 (-86.2%)	5,997.2 (+0.9%)	79.9 (-62.3%)	14	25.6
Juliet											
NoSplit-UT	**X**	14,183	**X**	2.4	**X**	24.2	**X**	**X**	2,844.9	1	249.7
NoSplit-UM	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	**X**	1	249.7
SizeLimiting	7,758 (N/A)	12,456 (-12%)	1.6 (∞x)	0.09 (26x)	101.1 (N/A)	37.1 (+34.7%)	1,855.7 (N/A)	7,098.5 (N/A)	131.5 (-95.3%)	95	249.7
SplitMerge	8,803 (N/A)	13,866 (-2%)	6 (∞x)	0.08 (30x)	185.4 (N/A)	41.1 (+45.5%)	2,623.6 (N/A)	7,933.4 (N/A)	135.8 (-95.2%)	151	298.7
Maven											
NoSplit-UT	1,193	31,246	32.3	2.5	767.4	307.4	14,730.9	17,501	1,776	138	4,205
NoSplit-UM	1,186	32,172	10	1.0	521.8	270.3	14,012.0	48,802	1,790	138	4,213
SizeLimiting	991 (-17%)	18,087 (-42%)	9.1 (3.5x)	1.0 (2.5x)	137 (-82.1%)	190.2 (-38.1%)	11,483.4 (-22%)	16,861 (-3.6%)	964 (-45.7%)	778	4,381
SplitMerge	1,113 (-6.7%)	31,793 (+1.7%)	8.5 (3.8x)	1.0 (2.5x)	543.2 (-30.3%)	715.1 (+132%)	12,181.7 (-17.03%)	17,182 (-1.8%)	1,025 (-42.2%)	2,641	15,756

Table 6.3: Timeout, crash and success rates of analysis runs.

Strategy	RAPID Timeout	INFER Timeout	RAPID Crash	INFER Crash	RAPID Success	INFER Success
OWASP						
NoSplit-UT	0%	0%	0%	0%	100%	100%
NoSplit-UM	100%	0%	0%	0%	0%	100%
SIZELIMITING	0%	0%	0%	20%	100%	80%
SPLITMERGE	0%	0%	0%	0%	100%	100%
Juliet						
NoSplit-UT	100%	0%	0%	0%	0%	100%
NoSplit-UM	100%	100%	0%	0%	0%	0%
SIZELIMITING	0%	0%	0%	2%	100%	98%
SPLITMERGE	0%	0%	0%	6%	100%	94%
Maven						
NoSplit-UT	0%	4%	0%	24%	100%	72%
NoSplit-UM	0%	4%	0%	22%	100%	74%
SIZELIMITING	0%	0.4%	0%	9.6%	100%	90%
SPLITMERGE	0%	1%	0%	16%	100%	83%

For the Maven benchmark, we conducted an experiment for each of the 138 Maven packages separately and aggregated the results (which show 138 partitions for the no-split strategies corresponding to the 138 packages analyzed). From the results on Maven, we can see that SPLITMERGE has less negative impact on both INFER and RAPID's findings compared to SIZELIMITING, i.e., RAPID only lost 6.7% ((1,193-1,113)/1,193) of the findings using SPLITMERGE, while it is 17% ((1,193-991)/1,193) using SIZELIMITING. On the other hand, INFER detected more findings with SPLITMERGE than NoSplit-UT, as it crashed less frequently (Table 6.3).

We remark once again that the number of findings is a coarse proxy to evaluate the differential accuracy, while assessing precision and recall of the different tools would require scoring each tool's output against a ground truth to establish which findings are true and false, which is beyond the scope of this study.

RQ1 takeaway: Splitting the original program can sometimes negatively impact a tool's accuracy. However our experiments show that smarter splitting strategies like SPLITMERGE can controllably reduce accuracy loss. For very large codebases (e.g., Juliet) and strict resource constraints, splitting may be the only option if we want to retrieve any findings, since the tools do not scale and thus end up returning no findings at all.

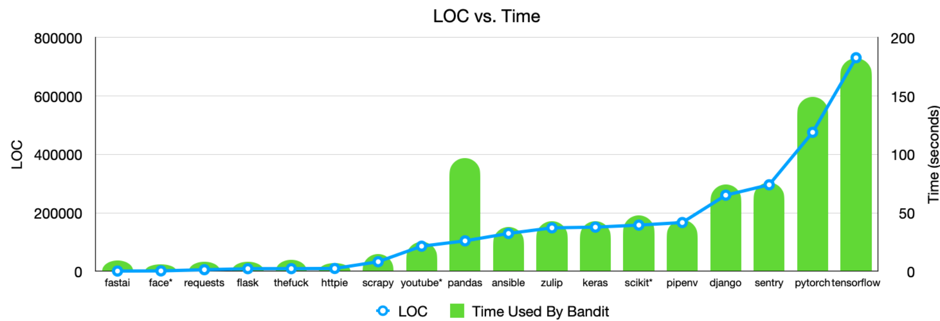


Figure 6.3: LOC of the 17 open source Python projects with dependency analysis and analysis time used by Bandit.

RQ2: How do static analyzers perform on the partitions compared to the original program in terms of time and memory usage?

After evaluating the accuracy loss of splitting, with this research question, we evaluate analysis time and resource demand. A benefit of splitting a program and analyzing each partition in isolation is the possibility of running an instance of the analysis tool on each partition in parallel, thus reducing the user's waiting time. The best possible latency of analyzing partitions is the maximum analysis time required to analyze any such partitions. On all three benchmark suites, both RAPID and INFER achieved much better latency with splitting strategies on both SAST benchmarks (`OWASP` and `Juliet`) and real-world applications (`Maven`). Using SPLITMERGE, the analyzers achieved more than 2x speedup (RAPID: 32.3/8.5, INFER: 2.5/1.0) on the `Maven` packages in comparison to `NoSplit-UT`. Since RAPID is written in Java, we also measured the peak heap usage from the JVM and observed that with SPLITMERGE, RAPID used less than 13.7% of the peak heap consumption of `NoSplit-UT` on `OWASP` and 82% on `Maven`. We also measured the maximum resident set memory size, which indicates a significant reduction of peak memory consumption also for INFER.

RQ2 takeaway: Splitting the codebase allows us to analyze the partitions in parallel. On all three benchmark suites, our experiments show that splitting significantly reduced the latency for both analysis tools. Splitting also significantly reduced the memory required to analyze the benchmark suites compared to non-splitting.

RQ3: What kinds of static analysis tools would benefit from splitting strategies discussed in the chapter?

Comparing the results of RAPID and INFER, the first observation is that splitting allowed RAPID to analyze `Juliet`, which was intractably large for the non-

splitting strategies. We also observed that RAPID is more tolerant than INFER with partitions that miss dependencies. On OWASP and Juliet, splitting increased the number of crashes (Table 6.3), resulting in a reduction of the number of findings. On Maven we observed the opposite effect, where analyzing the 138 packages individually led to more crashes than grouping all their sources and splitting them with SIZELIMITING or SPLITMERGE. However, for most benchmark applications, the difference in number of findings with and without splitting was limited for both RAPID and INFER, while RAPID was much faster in analyzing all subjects, significantly reducing both the best possible latency (by 73-97% with SPLITMERGE) and the total computation time (29-84% with SPLITMERGE). Also for INFER the best possible latency dropped by 60-99%, while the total computation time remained around the same order of magnitude, taking into account that the different number of crashes between NoSplit-UT and SPLITMERGE make it difficult to discern how much computation time ultimately led to results rather than crashing. INFER also showed a significant reduction in memory demand, up to 95% on Juliet.

A worst-case scenario for SPLITMERGE. To better define the target application scope of a dependency-aware splitting strategy like SPLITMERGE, we report on an additional experiment using Bandit [13], a static analysis tool for Python. Bandit processes each source file independently by building an abstract syntax tree and inspecting it for error patterns. While the previous experiments analyzed Java applications, SPLITMERGE is not restricted to a specific language, provided that the user can specify the elementary code units to partition (e.g., files, modules, or classes) and can identify their dependencies. In the case of Python, dependencies between its classes can be extracted using Snakefood [44] and Importlab [93].

We ran the three tools of the analysis pipeline (Snakefood, Importlab, and Bandit) on 17 popular Python open-source projects [3] and recorded the analysis time. Figure 6.3 shows the lines of code (LOC, bars) vs analysis time for each program (line). The analysis pipeline including dependency analysis and Bandit shows, as expected, very high scalability, taking only about 3 minutes to analyze tensorflow, the largest program in the set. The breakdown of the time required for each step of this analysis pipeline is shown in Figure 6.4, from which it is evident that dependency analysis takes a significant proportion of the pipeline time, up to 73% for youtube, while Bandit only takes a smaller fraction of the pipeline time.

This experiment represents a worst-case scenario for SPLITMERGE—the analysis does not benefit from preserving dependency information, thus dropping the benefits of SPLITMERGE's heuristics. To parallelize Bandit's analysis it would be more effective to use SIZELIMITING, avoiding the overhead of dependency analysis, which, contrary to the case of costlier interprocedural analyses with RAPID and INFER, does not pay off with Bandit's per-file analysis.

[3] https://dev.to/biplov/17-popular-python-opensource-projects-on-github-21ae

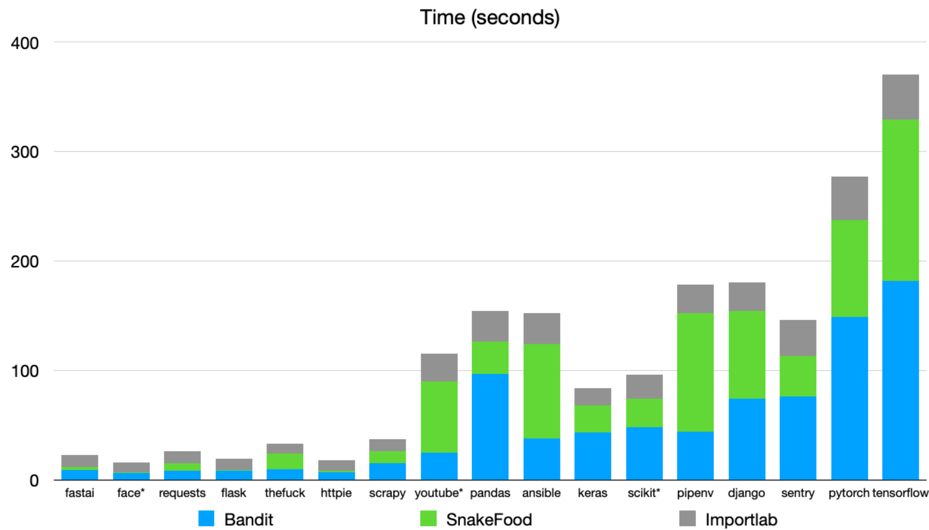


Figure 6.4: Time used by dependency analysis compared to time used by Bandit.

RQ3 takeaway: Our experiments show that splitting is useful to limit the per-machine resource consumption of static analysis tools and different tools require different splitting strategies. Inter-procedural analyses benefit most from splitting even with a naive strategy like SIZELIMITING. For example, splitting the input codebase allowed RAPID to analyze code much faster than without splitting. Whereas for INFER, the most prominent benefit of splitting was the reduction in memory usage. Our experiments also show that for inexpensive linter-like or intra-procedural analyses such as Bandit, SIZELIMITING may be more beneficial than dependency-guided-splitting strategy like SPLITMERGE.

6.5. Related Work

SAST platforms. Several SAST platforms in the literature provide static analysis as a service and present a simple interface to developers that allows them to run a variety of static analysis tools. One of the earliest platforms is Review Bot [24], which integrates FindBugs [96], PMD, and CheckStyle into the code review process. Review Bot runs in the code review process on changes small enough to be reviewed by humans, and the tools are fast linting tools, so it did not have an immediate need for splitting.

The Khasiana1 web portal [196] integrates three static analyzers (FindBugs [96], Safe [105], and Xylem [197]) under a unified service interface. It focuses on discussing the usefulness of the reported findings. The tools are evaluated on hand-picked projects in which scalability is not an issue.

Two more recent platforms are Software Assurance Marketplace

(SWAMP) [151] and Google's Tricorder [229] (and its open-source version, ShipShape). These platforms focus on making it easy to plug in additional analyzers. They provide orchestration as well as aggregation and management of recommendations. They do not focus on how to deal with large inputs. Our splitting approach is agnostic to the specific SAST platform and could be applied to Khasiana1, SWAMP, or Tricorder/ShipShape as well.

An approach that shares a similar motivation to ours is implemented by Cheetah [88] which integrates several static analysis tools into an IDE. Cheetah achieves fast response times by gradually increasing the scope of the analysis: it starts by analyzing only the method currently being edited in the IDE, then increases the scope to class, package, and project level. This allows them to deliver some results early on while gradually increasing the precision if the user is willing to wait for it. In this chapter, rather than gradually increasing the scope until reaching a time limit, we reduce the scope by splitting—simplifying the input, to get results within certain resource constraints.

A wide range of static program analysis problems can be viewed as instances of the Context-Free Language (CFL) Reachability problem [218], e.g., inter-procedural dataflow analysis [217], control flow analysis [254], set-constraints [149], specification-inference [34], shape analysis [218], object-flow analysis [269], pointer and alias analysis [166, 273], and program slicing [127] to name a few. Unfortunately, the worst case time complexity for solving CFL-reachability problems is $O(n^3)$, which is known as the "cubic bottleneck" [119]. As a result, highly precise analysis of large-scale software is challenging. In the literature, most work only focuses on very specific optimizations to a particular analysis that are not applicable in general.

Splitting input representation. Singh *et al.* proposed a technique to speed analysis with Polyhedra domain in abstract interpretation by partitioning variables into subsets such that the constraints only exist between variables in the same subset [241]. To mitigate the path explosion problem in symbolic execution, Trabish *et al.* [250] introduced *chopped symbolic execution*, an approach in which users can identify *unimportant* parts in the code, and the symbolic analysis tries to avoid those parts. Barnat *et al.* [29] propose a distributed algorithm for model checking LTL-formulas. The algorithm works by first partitioning and then exploring the state space in parallel. In model checking based on symbolic state representation, the technique in [114] partitions BDDs into smaller BDDs (each representing a subset of states) which are subsequently given to different processes. Kumar *et al.* [150] present a technique for distributed explicit state model checking to improve run time performance.

Parallel/distributed static analysis. The problem of scaling static analysis to potentially very large inputs is also discussed in [104] where the authors present a distributed call-graph construction algorithm designed to run in the cloud. We share the motivation that static analysis needs to be elastic to scale to very large inputs but our approach is designed to be agnostic to specific static analysis tools and techniques. Mendez-Lojo *et al.* [183] proposed a technique to parallelize

inclusion-based points-to analysis by formulating it in terms of constraint graph rewrite rules. Su *et al.* [245] introduced a parallel solution to CFL-reachability based pointer analysis by avoiding redundant graph traversals using data sharing and query scheduling. Rodriguez *et al.* [224] presented an actor-model-based parallel algorithm for solving IFDS data-flow problems. Albarghouthi *et al.* [15] proposed a framework to parallelize top down inter-procedural analysis using the MapReduce paradigm. Facebook uses INFER [60] based on bi-abduction which is modular—generates summaries for methods in the program and compositional—composes summaries at call sites. Nevertheless, as we show in our experiments, even a modular analysis like INFER does not scale to large or complex inputs under practical resource constraints. BigSpa [274] provides a data-parallel algorithm for CFL-reachability based static analysis. Graspan [258] and Grapple [275] are single-machine, disk-based tools that model specific static analysis problems, taint analysis and type-state analysis respectively, as transitive closure computation on graphs.

The above approaches parallelize or distribute the analysis workload in a way that is specific to the tool or technique at hand. Most of them modify the existing tool. In our case, we do not introduce any changes to the SAST tools; an important design goal is to be able to add new off-the-shelf tools and update existing tools (as new versions are released) without having to maintain our own modified versions of them.

Automated refactoring. The goal of splitting is to break a large program into smaller units that are sufficiently self-contained to be analyzed in isolation. This is similar to the refactoring problem of breaking up a monolithic system into smaller components. An overview of such refactoring techniques is given in [98]. Recently, we see approaches based on machine learning (e.g., [86]), dynamic analysis (e.g., [140]), and static analysis (e.g., [179]). The splitting problem discussed in this chapter is simpler than the problem of automatically refactoring in the sense that our partitions do not need to be functioning programs; they just need to to be sufficiently self-contained for analysis purposes.

Graph partitioning and communities. There is a rich body of work [133] on graph clustering for community detection in networks studied as graphs, e.g., social networks, academic citation networks, and collaboration networks—which might provide a theoretical grounding for future research. However, to the best of our knowledge this line of work currently focuses on preserving maximal connectivity, as opposed to our goal of attaining an effective trade-off between preserving connectivity and load-balancing partitions.

6.6. Summary and Remarks

We discussed how splitting of static analysis inputs can be used to effectively limit the maximum resource consumption of an analysis tool. We motivated that this is an important problem when operating a SAST platform, as adding new

analysis tools to the platform must not increase the maximum latency for existing customers.

Our evaluation shows that the splitting strategy has a significant impact on the outcome of the static analysis tool and that not all strategies are suitable for all tools. We showed that, for more complex (super-linear) static analysis tools, more advanced splitting strategies are needed to minimize the effect on the reported number of findings. For inexpensive linter-style tools like Bandit, we see that the overhead of a complex splitting strategy outweighs the benefits and that a simple splitting strategy is sufficient.

Chapter 7

Conclusion and Future Work

The aim of the work presented in this dissertation is to improve software correctness and reliability making it easier and more practical for developers of all skill levels to incorporate state of the art static program analyzers in their software development workflow. While the potential benefits of program analyzers are clear, their usability and effectiveness in mainstream software development workflows often comes into question and can prevent software developers from using these tools to their full potential. Practical program analyzers, therefore, have to make tradeoffs (e.g., in soundness, precision, or performance) to maintain a delicate balance between returning the minimum number of false negatives/positives, scaling to very large industrial codebases, being highly automatic, and having minimum overhead for the developers.

Designing, implementing, and deploying program analyzers, is an extremely challenging task. This makes them extremely complicated pieces of software with a high likelihood of having correctness bugs themselves (challenge 1) or tradeoffs not suitable for every piece of code under every usage scenario (challenge 2). In this dissertation, we focus our attention on improving the state of the art in these two challenge areas that can prevent typical software development teams from using these tools to their full potential.

Detecting unintentional unsoundness. In the first part of this dissertation, we present algorithms to detect correctness bugs in fundamental program analysis components such as SMT solvers and Datalog engines. Correctness bugs in these components can compromise the results computed by an upstream program analyzer. This can have disastrous consequences e.g., when analyzing software used for electronic voting, financial systems, transportation, or secure communication.

In chapter 2, we presented a general blackbox fuzzing technique for detecting critical bugs in any SMT solver. In contrast to existing work at the time, our technique does not require a grammar to synthesize SMT instances from scratch. Instead, it takes inspiration from state-of-the-art mutational fuzzers and generates new instances by mutating existing ones. We implemented the technique in an

open-source tool called STORM, which has the additional ability to effectively minimize the size of bug-revealing instances to facilitate debugging. In only three months of testing, with STORM, we were able to detect 29 previously unknown critical bugs in three mature SMT solvers and 15 different logics. In this chapter, we focused our attention on detecting refutational soundness bugs (the solver returns `unsat` for a satisfiable instance) because such bugs are the most difficult to detect. However, in the future, STORM can be extended to detect solution soundness bugs (the solver returns `sat` for an unsatisfiable instance). We also plan to extend STORM by integrating complementary techniques proposed subsequently to our work. For example, the fuzzing technique presented in STORM can be combined with OpFuzz that fuzzes operators in an SMT instance.

In chapter 3, we presented the first ever metamorphic testing based approach to detect query bugs in Datalog engines. The approach is based on the formal properties of conjunctive queries. Despite their simplicity, conjunctive queries constitute an important class of database queries due to their theoretical properties. We implemented our approach in an open source tool called queryFuzz, which we used to test three mature Datalog engines. queryFuzz detected 13 previously unknown query bugs in all three Datalog engines. In future work, we also plan to use the transformations presented in this chapter to test other data query systems since many first-order queries can be written as conjunctive queries.

In chapter 4, we presented a metamorphic testing approach that allows us to define much more general and effective transformations than the ones presented in chapter 3. This is because queryFuzz can only apply transformations locally without considering the entire program. The approach presented in chapter 4 uses an annotated precedence graph, that captures rich semantic and syntactic information about a given Datalog program. This graph allows us to define more radical program-wide transformations that also incorporates all existing queryFuzz transformations. In the future, we plan to explore ways of extending the annotated precedence graph with dialect-specific features e.g., aggregates.

Balancing soundness, precision, and performance. In the second part of this dissertation, we introduce techniques to reduce friction in the integration of program analyzers in everyday software development workflows. Smoothly integrating and tuning an advanced program analyzer for a particular codebase under certain resource constraints and different usage scenarios can be a challenging task for software developers, most of whom lack an advanced understanding of these analyzers.

In chapter 5, we presented a technique that can automatically tailor a generic abstract interpreter to the code under analysis and any given resource constraints. The key idea behind our approach is to phrase the problem of customizing the abstract-interpretation configuration to a given usage scenario as an optimization problem. Our experiments show that configurations generated by our technique vastly outperform the default options pre-selected by the analysis designers. We implemented our approach in an open-source framework called TAILOR and

demonstrate its effectiveness using a state-of-the-art abstract interpreter CRAB, with millions of configurations, on real-world benchmarks. In the future, we plan to explore the challenges that an inter-procedural analysis would pose, for instance, by using a different recipe for computing a summary of each function or each calling context.

In chapter 6, we presented an approach to horizontally scale static analysis tools in cloud-based static analysis platforms. Our approach takes as input a program to be analyzed and a bound for the size of code that should be analyzed by each single cloud instance. It then employs a configurable splitting strategy to split the input program into partitions such that the amount of code in each partition is below a provided bound. Our experiments show that splitting the input code base into partitions and analyzing each partition in a separate cloud instance can be an effective strategy to scale static analysis tools in static analysis platforms. We show that with a proper selection of a splitting strategy, all evaluated static analyzers can benefit from such an approach. The approach was developed in collaboration with Amazon Web Services and is currently being used in production in their CodeGuru service. We believe that, in the future, the process of picking a splitting strategy and tuning the parameters when adding a tool to a cloud-based static analysis platform can be fully automated. A tool can be run with different configurations on regression data to identify the best combination of strategy and configuration. Also, the configuration parameters for each tool can be re-adjusted on the fly as the available hardware improves or the static analysis tools get updated.

References

1. The BDDAPRON logico-numerical abstract domains library. http://www.inrialpes.fr/pop-art/people/bjeannet/bjeannet-forge/bddapron.
2. Coverity scan. https://scan.coverity.com/.
3. CREST: Concolic test generation tool for C. http://www.burn.im/crest/.
4. DeltaSMT. http://fmv.jku.at/deltasmt.
5. Infer. http://fbinfer.com.
6. The international satisfiability modulo theories competition. https://smt-comp.github.io.
7. Klocwork. http://www.klocwork.com.
8. Radamsa. https://gitlab.com/akihe/radamsa.
9. The satisfiability modulo theories library. http://smtlib.cs.uiowa.edu.
10. SonarQube. http://www.sonarqube.org.
11. Technical “whitepaper” for AFL. http://lcamtuf.coredump.cx/afl/technical_details.txt.
12. VoteSMT. http://fmv.jku.at/votesmt.
13. Bandit. https://bandit.readthedocs.io/en/latest/, 2008.
14. OWASP. https://owasp.org/www-project-benchmark/, 2022.
15. Aws Albarghouthi, Rahul Kumar, Aditya V. Nori, and Sriram K. Rajamani. Parallelizing top-down interprocedural analyses. In *PLDI*, pages 217–228. ACM, 2012.
16. Gianluca Amato and Marco Rubino. Experimental evaluation of numerical domains for inferring ranges. *ENTCS*, 334:3–16, 2018.
17. Molham Aref, Balder ten Cate, Todd J. Green, Benny Kimelfeld, Dan Olteanu, Emir Pasalic, Todd L. Veldhuizen, and Geoffrey Washburn. Design and implementation of the LogicBlox system. In *SIGMOD*, pages 1371–1382. ACM, 2015.
18. Cyrille Artho, Armin Biere, and Martina Seidl. Model-based testing for verification back-ends. In *TAP*, volume 7942 of *LNCS*, pages 39–55. Springer, 2013.
19. Steven Arzt and Eric Bodden. Reviser: Efficiently updating ide-/ifds-based data-flow analyses in response to incremental program changes. In *ICSE*, page 288–298. ACM, 2014.
20. Steven Arzt and Eric Bodden. Stubdroid: Automatic inference of precise data-flow summaries for the android framework. In *ICSE*, pages 725–735, 2016.
21. Nathaniel Ayewah, William W. Pugh, J. David Morgenthaler, John Penix,

and YuQian Zhou. Evaluating static analysis defect warnings on production software. In *PASTE*, pages 1–8. ACM, 2007.

22. John Backes, Pauline Bolignano, Byron Cook, Catherine Dodge, Andrew Gacek, Kasper Søe Luckow, Neha Rungta, Oksana Tkachuk, and Carsten Varming. Semantic-based automated reasoning for AWS access policies using SMT. In *FMCAD*, pages 1–9. IEEE, 2018.
23. Brenda S Baker and Edward G Coffman, Jr. A tight asymptotic bound for next-fit-decreasing bin-packing. *JADM*, 2(2):147–152, 1981.
24. Vipin Balachandran. Reducing human effort and improving quality in peer code reviews using automatic static analysis and reviewer recommendation. In *ICSE*, page 931–940. IEEE, 2013.
25. Isaac Balbin, Graeme S. Port, Kotagiri Ramamohanarao, and Krishnamurthy Meenakshi. Efficient bottom-up computation of queries on stratified databases. *JLP*, 11:295–344, 1991.
26. Paolo Baldan, Andrea Corradini, and Barbara König. A static analysis technique for graph transformation systems. In *CONCUR*, volume 2154 of *LNCS*, pages 381–395. Springer, 2001.
27. Paolo Baldan and Barbara König. Approximating the behaviour of graph transformation systems. In *ICGT*, volume 2505 of *LNCS*, pages 14–29. Springer, 2002.
28. François Bancilhon, David Maier, Yehoshua Sagiv, and Jeffrey D. Ullman. Magic sets and other strange ways to implement logic programs. In *PODS*, pages 1–15. ACM, 1986.
29. Jiri Barnat, Lubos Brim, and Jitka Stríbrná. Distributed LTL model-checking in SPIN. In *SPIN*, volume 2057 of *LNCS*, pages 200–216. Springer, 2001.
30. Michael Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *FMCO*, volume 4111 of *LNCS*, pages 364–387. Springer, 2005.
31. Earl T. Barr, Mark Harman, Phil McMinn, Muzammil Shahbaz, and Shin Yoo. The oracle problem in software testing: A survey. *TSE*, 41:507–525, 2015.
32. Clark W. Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. CVC4. In *CAV*, volume 6806 of *LNCS*, pages 171–177. Springer, 2011.
33. Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*, pages 305–343. Springer, 2018.
34. Osbert Bastani, Saswat Anand, and Alex Aiken. Specification inference using context-free language reachability. In *POPL*, pages 553–566. ACM, 2015.
35. Catriel Beeri and Raghu Ramakrishnan. On the power of magic. *JLP*, 10:255–299, 1991.

36. Pierre-Léo Bégay, Pierre Crégut, and Jean-François Monin. Developing and certifying Datalog optimizations in Coq/MathComp. In *CPP*, pages 163–177. ACM, 2021.
37. Aaron Bembenek, Michael Greenberg, and Stephen Chong. Formulog: Datalog for SMT-based static analysis. In *OOPSLA*, pages 141:1–141:31. ACM, 2020.
38. James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. Algorithms for hyper-parameter optimization. In *NIPS*, pages 2546–2554, 2011.
39. James Bergstra and Yoshua Bengio. Random search for hyper-parameter optimization. *JMLR*, 13:281–305, 2012.
40. Murphy Berzish, Vijay Ganesh, and Yunhui Zheng. Z3str3: A string solver with theory-aware heuristics. In *FMCAD*, pages 55–59. IEEE Computer Society, 2017.
41. Carsten Binnig, Donald Kossmann, and Eric Lo. Reverse query processing. In *ICDE*, pages 506–515. IEEE Computer Society, 2007.
42. Carsten Binnig, Donald Kossmann, Eric Lo, and M. Tamer Özsu. QAGen: Generating query-aware test databases. In *SIGMOD*, pages 341–352. ACM, 2007.
43. Andrew Binstock. Gitleaks: a SAST tool for detecting and preventing hardcoded secrets like passwords, api keys, and tokens in git repositories. https://blogs.oracle.com/javamagazine/post/java-class-file-constant-pool, 2022.
44. Martin Blais. Snakefood. https://furius.ca/snakefood/doc/snakefood-doc.html, 2007.
45. Bruno Blanchet, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. A static analyzer for large safety-critical software. In *PLDI*, pages 196–207. ACM, 2003.
46. Dmitry Blotsky, Federico Mora, Murphy Berzish, Yunhui Zheng, Ifaz Kabir, and Vijay Ganesh. StringFuzz: A fuzzer for string solvers. In *CAV*, volume 10982 of *LNCS*, pages 45–51. Springer, 2018.
47. Guillaume Brat, Jorge A. Navas, Nija Shi, and Arnaud Venet. IKOS: A framework for static analysis based on abstract interpretation. In *SEFM*, volume 8702 of *LNCS*, pages 271–277. Springer, 2014.
48. Martin Bravenboer and Yannis Smaragdakis. Strictly declarative specification of sophisticated points-to analyses. In *OOPSLA*, pages 243–262. ACM, 2009.
49. Lexi Brent, Anton Jurisevic, Michael Kong, Eric Liu, François Gauthier, Vincent Gramoli, Ralph Holz, and Bernhard Scholz. Vandal: A scalable security analysis framework for smart contracts. *CoRR*, abs/1809.03981, 2018.
50. Neville Brent, Lexiand Grech, Sifis Lagouvardos, Bernhard Scholz, and

Yannis Smaragdakis. Ethainter: A smart contract security analyzer for composite vulnerabilities. In *PLDI*, pages 454–469. ACM, 2020.
51. Robert Brummayer and Armin Biere. Fuzzing and delta-debugging SMT solvers. In *SMT*, pages 1–5. ACM, 2009.
52. Robert Brummayer, Florian Lonsing, and Armin Biere. Automated testing and debugging of SAT and QBF solvers. In *SAT*, volume 6175 of *LNCS*, pages 44–57. Springer, 2010.
53. Nicolas Bruno, Surajit Chaudhuri, and Dilys Thomas. Generating queries with cardinality constraints for DBMS testing. *TKDE*, 18:1721–1725, 2006.
54. Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. Detecting self-mutating malware using control-flow graph matching. In *DIMVA*, volume 4064 of *LNCS*, pages 129–143. Springer, 2006.
55. Alexandra Bugariu and Peter Müller. Automatically testing string solvers. In *ICSE*, pages 1459–1470. ACM, 2020.
56. Alexandra Bugariu, Valentin Wüstholz, Maria Christakis, and Peter Müller. Automatically testing implementations of numerical abstract domains. In *ASE*, pages 768–778. ACM, 2018.
57. Cristian Cadar and Alastair F. Donaldson. Analysing the program analyser. In *ICSE*, pages 765–768. ACM, 2016.
58. Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, pages 209–224. USENIX, 2008.
59. Cristiano Calcagno and Dino Distefano. Infer: An automatic program verifier for memory safety of C programs. In *NFM*, volume 6617 of *LNCS*, pages 459–465. Springer, 2011.
60. Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NFM*, volume 9058 of *LNCS*, pages 3–11. Springer, 2015.
61. W. K. Chan, S. C. Cheung, and Karl R. P. H. Leung. Towards a metamorphic testing methodology for service-oriented software applications. In *QSIC*, pages 470–476. IEEE Computer Society, 2005.
62. Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90. ACM, 1977.
63. Bor-Yuh Evan Chang and K. Rustan M. Leino. Abstract interpretation with alien expressions and heap structures. In *VMCAI*, volume 3385 of *LNCS*, pages 147–163. Springer, 2005.
64. J.-L. Chen, F.-J. Wang, and Y.-L. Chen. An object-oriented dependency graph for program slicing. In *TOOLS*, pages 121–130. IEEE Computer Society, 1997.
65. Junjie Chen, Jibesh Patra, Michael Pradel, Yingfei Xiong, Hongyu Zhang, Dan Hao, and Lu Zhang. A survey of compiler testing. *Comput. Surv.*,

53:4:1–4:36, 2020.
66. Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. Metamorphic testing: A new approach for generating next test cases. Technical Report HKUST–CS98–01, HKUST, 1998.
67. Jürgen Christ, Jochen Hoenicke, and Alexander Nutz. SMTInterpol: An interpolating SMT solver. In *SPIN*, volume 7385 of *LNCS*, pages 248–254. Springer, 2012.
68. Maria Christakis and Christian Bird. What developers want and need from program analysis: An empirical study. In *ASE*, pages 332–343. ACM, 2016.
69. Maria Christakis, Thomas Cottenier, Antonio Filieri, Linghui Luo, Lee Pike, Nico Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. Input splitting for cloud-based static application security testing platforms. In *ESEC/FSE*, pages 1367–1378. ACM, 2022.
70. Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The MathSAT5 SMT solver. In *TACAS*, volume 7795 of *LNCS*, pages 93–107. Springer, 2013.
71. Justin Collins. Brakeman: a static vulnerability scanner specifically designed for Ruby on Rails applications. https://brakemanscanner.org/.
72. Christopher L. Conway, Kedar S. Namjoshi, Dennis Dams, and Stephen A. Edwards. Incremental algorithms for inter-procedural analysis of safety properties. In *CAV*, pages 449–461. Springer, 2005.
73. David Coppit and Jiexin Lian. yagg: an easy-to-use generator for structured test inputs. In David F. Redmiles, Thomas Ellman, and Andrea Zisman, editors, *ASE*, pages 356–359. ACM, 2005.
74. Loïc Correnson, Pascal Cuoq, Florent Kirchner, Virgile Prevosto, Armand Puccetti, Julien Signoles, and Boris Yakobowski. *Frama-C User Manual*, 2011. http://frama-c.com//support.html.
75. Patrick Cousot and Radhia Cousot. Static determination of dynamic properties of programs. In *ISOP*, pages 106–130. Dunod, 1976.
76. Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252. ACM, 1977.
77. Patrick Cousot and Radhia Cousot. Abstract interpretation and application to logic programs. *JLP*, 13:103–179, 1992.
78. Patrick Cousot and Radhia Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *PLILP*, volume 631 of *LNCS*, pages 269–295. Springer, 1992.
79. Patrick Cousot and Radhia Cousot. Refining model checking by abstract interpretation. *Autom. Softw. Eng.*, 6:69–95, 1999.
80. Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The astreé analyzer. In *ESOP*, LNCS, pages 21–30. Springer, 2005.

81. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96. ACM, 1978.
82. Pascal Cuoq, Benjamin Monate, Anne Pacalet, Virgile Prevosto, John Regehr, Boris Yakobowski, and Xuejun Yang. Testing static analyzers with randomly generated programs. In *NFM*, volume 7226 of *LNCS*, pages 120–125. Springer, 2012.
83. Oege de Moor, Damien Sereni, Mathieu Verbaere, Elnar Hajiyev, Pavel Avgustinov, Torbjörn Ekman, Neil Ongkingco, and Julian Tibble. .QL: Object-oriented queries made easy. In *GTTSE*, volume 5235 of *LNCS*, pages 78–133. Springer, 2007.
84. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.
85. Leonardo de Moura and Dejan Jovanovic. A model-constructing satisfiability calculus. In *VMCAI*, volume 7737 of *LNCS*, pages 1–12. Springer, 2013.
86. Utkarsh Desai, Sambaran Bandyopadhyay, and Srikanth Tamilselvam. Graph neural network to dilute outliers for refactoring monolith application. In *AAAI*, pages 72–80. AAAI, 2021.
87. Dino Distefano, Manuel Fähndrich, Francesco Logozzo, and Peter W. O'Hearn. Scaling static analyses at Facebook. *CACM*, 62:62–70, 2019.
88. Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson Murphy-Hill. Just-in-time static analysis. In *ISSTA*, page 307–317. ACM, 2017.
89. Winterer Dominik, Zhang Chengyu, and Su Zhendong. On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. In *OOPSLA*, pages 1–25. ACM, 2020.
90. Alastair F. Donaldson, Hugues Evrard, Andrei Lascu, and Paul Thomson. Automated testing of graphics shader compilers. *PACMPL*, 1:93:1–93:29, 2017.
91. Bruno Dutertre. Yices 2.2. In *CAV*, volume 8559 of *LNCS*, pages 737–744. Springer, 2014.
92. Michael Emmi, Liana Hadarean, Ranjit Jhala, Lee Pike, Nicolás Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. Rapid: Checking API usage for the cloud in the cloud. In *ESEC/FSE*, pages 1416–1426. ACM, 2021.
93. Martin DeMello et al. Importlab. `https://github.com/google/importlab`, 2017.
94. Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS*, volume 6528 of *LNCS*, pages 10–30. Springer, 2010.
95. Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. In *ICML*, volume 80 of *PMLR*, pages

1436–1445. PMLR, 2018.
96. Cormac Flanagan, K Rustan M Leino, Mark Lillibridge, Greg Nelson, James B Saxe, and Raymie Stata. Extended static checking for java. In *PLDI*, pages 234–245, 2002.
97. Jonathan Ford and Natarajan Shankar. Formal verification of a combination decision procedure. In *CADE*, volume 2392 of *LNCS*, pages 347–362. Springer, 2002.
98. Jonas Fritzsch, Justus Bogner, Alfred Zimmermann, and Stefan Wagner. From monolith to microservices: A classification of refactoring approaches. In *DEVOPS*, volume 11350 of *LNCS*, pages 128–141. Springer, 2018.
99. Zhoulai Fu and Zhendong Su. Mathematical execution: A unified approach for testing numerical code. *CoRR*, abs/1610.01133, 2016.
100. Zhoulai Fu and Zhendong Su. Achieving high coverage for floating-point code via unconstrained programming. In *PLDI*, pages 306–319. ACM, 2017.
101. Vijay Ganesh. *Decision Procedures for Bit-Vectors, Arrays and Integers*. PhD thesis, Stanford University, USA, 2007.
102. Vijay Ganesh and David L. Dill. A decision procedure for bit-vectors and arrays. In *CAV*, volume 4590 of *LNCS*, pages 519–531. Springer, 2007.
103. Graeme Gange, Jorge A. Navas, Peter Schachte, Harald Søndergaard, and Peter J. Stuckey. An abstract domain of uninterpreted functions. In *VMCAI*, volume 9583 of *LNCS*, pages 85–103. Springer, 2016.
104. Diego Garbervetsky, Edgardo Zoppi, and Benjamin Livshits. Toward full elasticity in distributed static analysis: The case of callgraph analysis. In *ESEC/FSE*, page 442–453. ACM, 2017.
105. Emmanuel Geay, Eran Yahav, and Stephen Fink. Continuous code-quality assurance with safe. In *PEPM*, pages 145–149. ACM, 2006.
106. Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted Linux kernel extensions. In *PLDI*, pages 1069–1084. ACM, 2019.
107. Patrice Godefroid, Michael Y. Levin, and David A. Molnar. Automated whitebox fuzz testing. In *NDSS*, pages 151–166. The Internet Society, 2008.
108. Markian M. Gooley and Benjamin W. Wah. Efficient reordering of Prolog programs. In *ICDE*, pages 110–117. IEEE Computer Society, 1988.
109. Philippe Granger. Static analysis of arithmetical congruences. *International Journal of Computer Mathematics*, 30:165–190, 1989.
110. Sergey Grebenshchikov, Nuno P. Lopes, Corneliu Popeea, and Andrey Rybalchenko. Synthesizing software verifiers from proof rules. In *PLDI*, pages 405–416. ACM, 2012.
111. Neville Grech, Michael Kong, Anton Jurisevic, Lexi Brent, Bernhard Scholz, and Yannis Smaragdakis. MadMax: Surviving out-of-gas conditions in Ethereum smart contracts. *PACMPL*, 2:116:1–116:27, 2018.

112. Sergio Greco and Cristian Molinaro. *Datalog and Logic Databases*. Morgan & Claypool, 2015.
113. Sergio Greco and Cristian Molinaro. *Datalog and Logic Databases*. Morgan & Claypool, 2015.
114. Orna Grumberg, Tamir Heyman, Nili Ifergan, and Assaf Schuster. Achieving speedups in distributed symbolic reachability analysis through asynchronous computation. In *IFIP*, volume 3725 of *LNCS*, pages 129–145. Springer, 2005.
115. Arie Gurfinkel and Sagar Chaki. Boxes: A symbolic abstract domain of boxes. In *SAS*, volume 6337 of *LNCS*, pages 287–303. Springer, 2010.
116. Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A. Navas. The SeaHorn verification framework. In *CAV*, volume 9206 of *LNCS*, pages 343–361. Springer, 2015.
117. Arie Gurfinkel and Jorge A. Navas. A context-sensitive memory model for verification of C/C++ programs. In *SAS*, volume 10422 of *LNCS*, pages 148–168. Springer, 2017.
118. Daniel Halperin, Victor Teixeira de Almeida, Lee Lee Choo, Shumo Chu, Paraschos Koutris, Dominik Moritz, Jennifer Ortiz, Vaspol Ruamviboonsuk, Jingjing Wang, Andrew Whitaker, Shengliang Xu, Magdalena Balazinska, Bill Howe, and Dan Suciu. Demonstration of the Myria big data management service. In *SIGMOD*, pages 881–884. ACM, 2014.
119. Nevin Heintze and David A. McAllester. On the cubic bottleneck in subtyping and flow analysis. In *LICS*, pages 342–351. IEEE, 1997.
120. Matthias Heizmann, Jochen Hoenicke, and Andreas Podelski. Software model checking for people who love automata. In *CAV*, volume 8044 of *LNCS*, pages 36–52. Springer, 2013.
121. Kihong Heo, Hakjoo Oh, and Hongseok Yang. Learning a variable-clustering strategy for octagon from labeled data generated by a static analysis. In *SAS*, volume 9837 of *LNCS*, pages 237–256. Springer, 2016.
122. Kihong Heo, Hakjoo Oh, and Hongseok Yang. Resource-aware program analysis via online abstraction coarsening. In *ICSE*, pages 94–104. IEEE Computer Society/ACM, 2019.
123. Kihong Heo, Hakjoo Oh, Hongseok Yang, and Kwangkeun Yi. Adaptive static analysis via learning with Bayesian optimization. *TOPLAS*, 40:14:1–14:37, 2018.
124. Kihong Heo, Hakjoo Oh, and Kwangkeun Yi. Machine-learning-guided selectively unsound static analysis. In *ICSE*, pages 519–529. IEEE Computer Society/ACM, 2017.
125. Krystof Hoder, Nikolaj Bjørner, and Leonardo de Moura. μZ—An efficient engine for fixed points with constraints. In *CAV*, volume 6806 of *LNCS*, pages 457–462. Springer, 2011.
126. Susan Horwitz and Thomas W. Reps. The use of program dependence graphs in software engineering. In *ICSE*, pages 392–411. ACM, 1992.

127. Susan Horwitz, Thomas W. Reps, and David W. Binkley. Interprocedural slicing using dependence graphs. In *PLDI*, pages 35–46. ACM, 1988.
128. Jiani Huang, Ziyang Li, Binghong Chen, Karan Samel, Mayur Naik, Le Song, and Xujie Si. Scallop: From probabilistic deductive databases to scalable differentiable reasoning. In *NeurIPS*, pages 25134–25145, 2021.
129. Frank Hutter. *Automated Configuration of Algorithms for Solving Hard Computational Problems*. PhD thesis, The University of British Columbia, Canada, 2009.
130. Frank Hutter, Domagoj Babic, Holger H. Hoos, and Alan J. Hu. Boosting verification by automatic tuning of decision procedures. In *FMCAD*, pages 27–34. IEEE Computer Society, 2007.
131. Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *AAAI*, pages 1152–1157. AAAI, 2007.
132. Sehun Jeong, Minseok Jeon, Sung Deok Cha, and Hakjoo Oh. Data-driven context-sensitivity for points-to analysis. *PACMPL*, 1:100:1–100:28, 2017.
133. Di Jin, Zhizhi Yu, Pengfei Jiao, Shirui Pan, Dongxiao He, Jia Wu, Philip Yu, and Weixiong Zhang. A survey of community detection approaches: From statistical modeling to deep learning. *TKDE*, 2021.
134. Seo Jiwon, Guo Stephen, and Lam Monica S. SociaLite: Datalog extensions for efficient social network analysis. In *ICDE*, pages 278–289. IEEE Computer Society, 2013.
135. David S Johnson. *Near-optimal bin packing algorithms*. PhD thesis, Massachusetts Institute of Technology, 1973.
136. Herbert Jordan, Bernhard Scholz, and Pavle Subotic. Soufflé: On synthesis of program analyzers. In *CAV*, volume 9780 of *LNCS*, pages 422–430. Springer, 2016.
137. Herbert Jordan, Pavle Subotic, David Zhao, and Bernhard Scholz. Brie: A specialized trie for concurrent Datalog. In *PPoPP*, pages 31–40. ACM, 2019.
138. Dejan Jovanovic, Clark Barrett, and Leonardo de Moura. The design and implementation of the model-constructing satisfiability calculus. In *FMCAD*, pages 173–180. IEEE Computer Society, 2013.
139. Jinho Jung, Hong Hu, Joy Arulraj, Taesoo Kim, and Woon-Hak Kang. APOLLO: Automatic detection and diagnosis of performance regressions in database systems. *VLDB*, 13:57–70, 2019.
140. Anup K. Kalia, Jin Xiao, Rahul Krishna, Saurabh Sinha, Maja Vukovic, and Debasish Banerjee. Mono2micro: A practical and effective tool for decomposing monolithic java applications to microservices. In *ESEC/FSE*, page 1214–1224. ACM, 2021.
141. Timotej Kapus and Cristian Cadar. Automatic testing of symbolic execution engines via program generation and differential testing. In *ASE*, pages 590–600. IEEE Computer Society, 2017.

142. Michael Karr. Affine relationships among variables of a program. *Acta Inf.*, 6:133–151, 1976.
143. Shadi Abdul Khalek, Bassem Elkarablieh, Yai O. Laleye, and Sarfraz Khurshid. Query-aware test generation using a relational constraint solver. In *ASE*, pages 238–247. IEEE Computer Society, 2008.
144. Shadi Abdul Khalek and Sarfraz Khurshid. Automated SQL query generation for systematic testing of database engines. In *ASE*, pages 329–332. ACM, 2010.
145. Ross D. King. Applying inductive logic programming to predicting gene function. *AI Mag.*, 25:57–68, 2004.
146. Kyle Kingsbury. Jepsen. https://jepsen.io.
147. Scott Kirkpatrick, C. Daniel Gelatt Jr., and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.
148. Christian Klinger, Maria Christakis, and Valentin Wüstholz. Differentially testing soundness and precision of program analyzers. In *ISSTA*, pages 239–250. ACM, 2019.
149. John Kodumal and Alexander Aiken. The set constraint/cfl reachability connection in practice. In *PLDI*, pages 207–218. ACM, 2004.
150. Rahul Kumar and Eric G. Mercer. Load balancing parallel explicit state model checking. *ENTCS*, 128:19–34, 2005.
151. James A. Kupsch, Barton P. Miller, Vamshi Basupalli, and Josef Burger. From continuous integration to continuous assurance. In *STC*, pages 1–8, 2017.
152. Lies Lakhdar-Chaouch, Bertrand Jeannet, and Alain Girault. Widening with thresholds for programs with complex control graphs. In *ATVA*, volume 6996 of *LNCS*, pages 492–502. Springer, 2011.
153. Vu Le, Mehrdad Afshari, and Zhendong Su. Compiler validation via equivalence modulo inputs. In *PLDI*, pages 216–226. ACM, 2014.
154. Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *OOPSLA*, pages 386–399. ACM, 2015.
155. K. Rustan M. Leino. Dafny: An automatic program verifier for functional correctness. In *LPAR*, volume 6355 of *LNCS*, pages 348–370. Springer, 2010.
156. Xavier Leroy. Formal verification of a realistic compiler. *CACM*, 52:107–115, 2009.
157. Stéphane Lescuyer and Sylvain Conchon. A reflexive formalization of a SAT solver in Coq. In *TPHOLs*, 2008.
158. Chris Lewis, Zhongpeng Lin, Caitlin Sadowski, Xiaoyan Zhu, Rong Ou, and E. James Whitehead Jr. Does bug prediction support human developers? findings from a Google case study. In *ICSE*, pages 372–381. ACM, 2013.
159. Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. Many-core compiler fuzzing. In *PLDI*, pages 65–76. ACM, 2015.

160. Vladimir Lifschitz. On the declarative semantics of logic programs with negation. In *Foundations of Deductive Databases and Logic Programming*, pages 177–192. Morgan Kaufmann, 1988.
161. Jiawei Liu, Jinkun Lin, Fabian Ruffy, Cheng Tan, Jinyang Li, Aurojit Panda, and Lingming Zhang. Finding deep-learning compilation bugs with NNSmith. *CoRR*, abs/2207.13066, 2022.
162. Vsevolod Livinskii, Dmitry Babokin, and John Regehr. Random testing for C and C++ compilers with YARPGen. *PACMPL*, 4:196:1–196:25, 2020.
163. Eric Lo, Carsten Binnig, Donald Kossmann, M. Tamer Özsu, and Wing-Kai Hon. A framework for testing DBMS features. *VLDB*, 19:203–230, 2010.
164. Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *CACM*, 52:87–95, 2009.
165. Nuno P. Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with Alive. In *PLDI*, pages 22–32. ACM, 2015.
166. Yi Lu, Lei Shang, Xinwei Xie, and Jingling Xue. An incremental points-to analysis with cfl-reachability. In *CC*, volume 7791 of *LNCS*, pages 61–81. Springer, 2013.
167. Magnus Madsen and Ondrej Lhoták. Safe and sound program analysis with FLIX. In *ISSTA*, pages 38–48. ACM, 2018.
168. Muhammad Numair Mansur, Maria Christakis, and Valentin Wüstholz. Metamorphic testing of datalog engines. In *ESEC/FSE*, pages 639–650. ACM, 2021.
169. Muhammad Numair Mansur, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *ESEC/FSE*, pages 701–712. ACM, 2020.
170. Muhammad Numair Mansur, Benjamin Mariano, Maria Christakis, Jorge A. Navas, and Valentin Wüstholz. Automatically tailoring abstract interpretation to custom usage scenarios. In *CAV*, volume 12760 of *LNCS*, pages 777–800. Springer, 2021.
171. Muhammad Numair Mansur, Valentin Wüstholz, and Maria Christakis. Dependency-aware metamorphic testing of datalog engines. In *ISSTA*. ACM, 2023.
172. Filip Maric. Formal verification of a modern SAT solver by shallow embedding into Isabelle/HOL. *TCS*, 411:4333–4356, 2010.
173. Silvano Martello and Paolo Toth. *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.
174. Lorenzo Martignoni, Stephen McCamant, Pongsin Poosankam, Dawn Song, and Petros Maniatis. Path-exploration lifting: Hi-Fi tests for Lo-Fi emulators. In *ASPLOS*, pages 337–348. ACM, 2012.
175. I. Mátyáš. Random optimization. *Avtomat. i Telemekh.*, 26:246–253, 1965.
176. Peter M. Maurer. Generating test data with enhanced context-free grammars.

IEEE Software, 7(4):50–55, 1990.

177. Bringolf Mauro, Winterer Dominik, and Su Zhendong. Finding and understanding incompleteness bugs in smt solvers. In *ASE*. ACM, 2022.
178. Maven. List of Maven Packages. https://gist.github.com/linghuiluo/1b82866051e4c4ebb0fd065549f60100, 2022.
179. Genc Mazlami, Jürgen Cito, and Philipp Leitner. Extraction of microservices from monolithic software architectures. In Ilkay Altintas and Shiping Chen, editors, *ICWS*, pages 524–531. IEEE, 2017.
180. William M. McKeeman. Differential testing for software. *Digital Technical Journal*, 10:100–107, 1998.
181. John M. McQuillan. Graph theory applied to optimal connectivity in computer networks. *Comput. Commun. Rev.*, 7:13–41, 1977.
182. Muhammad Izhar Mehar, Charles Louis Shier, Alana Giambattista, Elgar Gong, Gabrielle Fletcher, Ryan Sanayhie, Henry M. Kim, and Marek Laskowski. Understanding a revolutionary and flawed grand experiment in blockchain: The DAO attack. *JCIT*, 21(1):19–32, 2019.
183. Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. Parallel inclusion-based points-to analysis. In *OOPSLA*, pages 428–443. ACM, 2010.
184. Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, 21:1087–1092, 1953.
185. Jan Midtgaard and Anders Møller. QuickChecking static analysis properties. *Softw. Test., Verif. Reliab.*, 27, 2017.
186. Bogdan Mihaila, Alexander Sepp, and Axel Simon. Widening as abstract domain. In *NFM*, volume 7871 of *LNCS*, pages 170–184. Springer, 2013.
187. Barton P. Miller, Lars Fredriksen, and Bryan So. An empirical study of the reliability of UNIX utilities. *Commun. ACM*, 33(12):32–44, 1990.
188. Antoine Miné. A few graph-based relational numerical abstract domains. In *SAS*, volume 2477 of *LNCS*, pages 117–132. Springer, 2002.
189. Antoine Miné. Field-sensitive value analysis of embedded C programs with union types and pointer arithmetics. In *LCTES*, pages 54–63. ACM, 2006.
190. Antoine Miné. The Octagon abstract domain. *HOSC*, 19:31–100, 2006.
191. Antoine Miné. Symbolic methods to enhance the precision of numerical abstract domains. In *VMCAI*, volume 3855 of *LNCS*, pages 348–363. Springer, 2006.
192. Chaitanya Mishra, Nick Koudas, and Calisto Zuzarte. Generating targeted queries for database testing. In *SIGMOD*, pages 499–510. ACM, 2008.
193. David Monniaux and Julien Le Guen. Stratified static analysis based on variable dependencies. *ENTCS*, 288:61–74, 2012.
194. Raymond J. Mooney. Inductive logic programming for natural language processing. In *ILP*, volume 1314 of *LNCS*, pages 3–22. Springer, 1996.

195. Mayur Naik, Alex Aiken, and John Whaley. Effective static race detection for Java. In *PLDI*, pages 308–319. ACM, 2006.
196. Mangala Gowri Nanda, Monika Gupta, Saurabh Sinha, Satish Chandra, David Schmidt, and Pradeep Balachandran. Making defect-finding tools work for you. In *ICSE*, page 99–108. ACM, 2010.
197. Mangala Gowri Nanda and Saurabh Sinha. Accurate interprocedural null-dereference analysis for java. In *ICSE*, pages 133–143. IEEE, 2009.
198. Aina Niemetz and Armin Biere. ddSMT: A delta debugger for the SMT-LIB v2 format. In *SMT*, pages 36–45, 2013.
199. Aina Niemetz, Mathias Preiner, and Clark W. Barrett. Murxla: A modular and highly extensible API fuzzer for SMT solvers. In *CAV*, volume 13372 of *LNCS*, pages 92–106. Springer, 2022.
200. Aina Niemetz, Mathias Preiner, and Armin Biere. Model-based API testing for SMT solvers. In *SMT*, page 10 pages, 2017.
201. Aina Niemetz, Mathias Preiner, Clifford Wolf, and Armin Biere. Btor2, BtorMC and Boolector 3.0. In *CAV*, volume 10981 of *LNCS*, pages 587–595. Springer, 2018.
202. NIST. Juliet Test Suite for Java. https://samate.nist.gov/SRD/testsuite.php, 2022.
203. Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for C-like languages. In *PLDI*, pages 229–238. ACM, 2012.
204. Oracle. JDeps - Java Platform, Standard Edition Tools Reference. https://docs.oracle.com/javase/9/tools/jdeps.htm, 2022.
205. OWASP. FindSecBugs: the SpotBugs plugin for security audits of Java web applications. https://find-sec-bugs.github.io/, 2022.
206. Matteo Paltenghi and Michael Pradel. Bugs in quantum computing platforms: An empirical study. *PACMPL*, 6:1–27, 2022.
207. Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. Generative type-aware mutation for testing SMT solvers. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–19, 2021.
208. Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. Generative type-aware mutation for testing SMT solvers. *Proc. ACM Program. Lang.*, 5(OOPSLA):1–19, 2021.
209. Georgios A. Pavlopoulos, Maria Secrier, Charalampos N. Moschopoulos, Theodoros G. Soldatos, Sophia Kossida, Jan Aerts, Reinhard Schneider, and Pantelis G. Bagos. Using graph theory to analyze biological networks. *BioData Min.*, 4, 2011.
210. Karl-Heinz Pennemann. Development of correct graph transformation systems. In *ICGT*, volume 5214 of *LNCS*, pages 508–510. Springer, 2008.
211. Detlef Plump. On termination of graph rewriting. In *WG*, volume 1017 of *LNCS*, pages 88–100. Springer, 1995.
212. Meikel Poess and John M. Stephens. Generating thousand benchmark

queries in seconds. In *VLDB*, pages 1045–1053. Morgan Kaufmann, 2004.
213. David Poole. Logic programming for robot control. In *IJCAI*, pages 150–157. Morgan Kaufmann, 1995.
214. Praetorian, Inc. Gokart: a security-oriented static analysis for Golang with a focus on minimizing false positives. https://github.com/praetorian-inc/gokart/.
215. Raghu Ramakrishnan, Catriel Beeri, and Ravi Krishnamurthy. Optimizing existential Datalog queries. In *PODS*, pages 89–102. ACM, 1988.
216. Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties from 'big code'. *CACM*, 62:99–107, 2019.
217. Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *POPL*, page 49–61. ACM, 1995.
218. Thomas W. Reps. Program analysis via graph reachability. In *ISLP*, pages 5–19. MIT, 1997.
219. H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74:358–366, 1953.
220. Zachary Rice. Understanding the constant pool inside a Java class file. https://github.com/zricethezav/gitleaks/.
221. Manuel Rigger and Zhendong Su. Detecting optimization bugs in database engines via non-optimizing reference engine construction. In *ESEC/FSE*, pages 1140–1152. ACM, 2020.
222. Manuel Rigger and Zhendong Su. Finding bugs in database systems via query partitioning. *PACMPL*, 4:211:1–211:30, 2020.
223. Manuel Rigger and Zhendong Su. Testing database engines via pivoted query synthesis. In *OSDI*, pages 667–682. USENIX, 2020.
224. Jonathan Rodriguez and Ondrej Lhoták. Actor-based parallel dataflow analysis. In *CC*, volume 6601 of *LNCS*, pages 179–197. Springer, 2011.
225. Kenneth A. Ross. Modular stratification and magic sets for Datalog programs with negation. In *PODS*, pages 161–171. ACM, 1990.
226. Atanas Rountev, Mariana Sharp, and Guoqing Xu. Ide dataflow analysis in the presence of large object-oriented libraries. In *CC*, pages 53–68. Springer, 2008.
227. Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2010.
228. Leonid Ryzhyk and Mihai Budiu. Differential Datalog. In *Datalog*, volume 2368 of *CEUR*, pages 56–67. CEUR-WS.org, 2019.
229. Caitlin Sadowski, Jeffrey van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. Tricorder: Building a program analysis ecosystem. In *ICSE*, page 598–608. IEEE, 2015.
230. Yehoshua Sagiv. Optimizing Datalog programs. In *Foundations of Deductive Databases and Logic Programming*, pages 659–698. Morgan Kaufmann, 1988.

231. Arash Sahebolamri, Thomas Gilray, and Kristopher K. Micinski. Seamless deductive inference via macros. In *CC*, pages 77–88. ACM, 2022.
232. José Carlos Almeida Santos, Houssam Nassif, David Page, Stephen H. Muggleton, and Michael J. E. Sternberg. Automated identification of protein-ligand interaction features using inductive logic programming: A hexose binding case study. *BMC Bioinform.*, 13:162, 2012.
233. Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ASPLOS*, pages 305–316. ACM, 2013.
234. Joseph Scott, Federico Mora, and Vijay Ganesh. Banditfuzz: A reinforcement-learning based performance fuzzer for SMT solvers. In *VSTTE*, volume 12549 of *LNCS*, pages 68–86. Springer, 2020.
235. Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz Cortés. A survey on metamorphic testing. *TSE*, 42:805–824, 2016.
236. Andreas Seltenreich. SQLsmith. `https://github.com/anse1/sqlsmith`.
237. Amazon Web Services. Elastic Compute Cloud (EC2). `https://aws.amazon.com/ec2`.
238. Rahul Sharma and Alex Aiken. From invariant checking to invariant inference using randomized search. In *CAV*, volume 8559 of *LNCS*, pages 88–105. Springer, 2014.
239. Alexander Shkapsky, Kai Zeng, and Carlo Zaniolo. Graph queries in a next-generation Datalog system. *VLDB*, 6:1258–1261, 2013.
240. Janet Siegmund, Norbert Siegmund, and Sven Apel. Views on internal and external validity in empirical software engineering. In *ICSE*, pages 9–19. IEEE Computer Society, 2015.
241. Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Making numerical program analysis fast. In *PLDI*, pages 303–313. ACM, 2015.
242. Gagandeep Singh, Markus Püschel, and Martin T. Vechev. Fast numerical program analysis with reinforcement learning. In *CAV*, volume 10981 of *LNCS*, pages 211–229. Springer, 2018.
243. Emin Gün Sirer and Brian N. Bershad. Using production grammars in software testing. In *DSL*, pages 1–13. ACM, 1999.
244. Donald R. Slutz. Massive stochastic testing of SQL. In *VLDB*, pages 618–622. Morgan Kaufmann, 1998.
245. Yu Su, Ding Ye, and Jingling Xue. Parallel pointer analysis with cfl-reachability. In *ICPP*, pages 451–460. IEEE Computer Society, 2014.
246. Chengnian Sun, Vu Le, and Zhendong Su. Finding and analyzing compiler warning defects. In *ICSE*, pages 203–213. ACM, 2016.
247. Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *OOPSLA*, pages 849–863. ACM, 2016.
248. Jubi Taneja, Zhengyang Liu, and John Regehr. Testing static analyses for precision and soundness. In *CGO*, pages 81–93. ACM, 2020.
249. Chris Thornton, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown.

Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *KDD*, pages 847–855. ACM, 2013.

250. David Trabish, Andrea Mattavelli, Noam Rinetzky, and Cristian Cadar. Chopped symbolic execution. In *ICSE*, pages 350–360. ACM, 2018.
251. Petar Tsankov, Andrei Marian Dan, Dana Drachsler-Cohen, Arthur Gervais, Florian Bünzli, and Martin T. Vechev. Securify: Practical security analysis of smart contracts. In *CCS*, pages 67–82. ACM, 2018.
252. Muhammad Usman, Wenxi Wang, and Sarfraz Khurshid. TestMC: Testing model counters using differential and metamorphic testing. In *ASE*, pages 709–721. IEEE Computer Society, 2020.
253. Jens Van der Plas, Quentin Stiévenart, Noah Van Es, and Coen De Roover. Incremental flow analysis through computational dependency reification. In *SCAM*, pages 25–36, 2020.
254. Dimitrios Vardoulakis and Olin Shivers. CFA2: A context-free approach to control-flow analysis. In *ESOP*, volume 6012 of *LNCS*, pages 570–589. Springer, 2010.
255. Manasi Vartak, Venkatesh Raghavan, and Elke A. Rundensteiner. QRelX: Generating meaningful queries that provide cardinality assurance. In *SIGMOD*, pages 1215–1218. ACM, 2010.
256. Arnaud Venet and Guillaume P. Brat. Precise and efficient static array bound checking for large embedded C programs. In *PLDI*, pages 231–242. ACM, 2004.
257. Jian Wang, Jungsoon P. Yoo, and Thomas J. Cheatham. Efficient reordering of C-PROLOG. In *Conference on Computer Science*, pages 151–155. ACM, 1993.
258. Kai Wang, Aftab Hussain, Zhiqiang Zuo, Guoqing Xu, and Ardalan Amiri Sani. Graspan: A single-machine disk-based graph system for interprocedural static analyses of large-scale systems code. In *ASPLOS*, page 389–404. ACM, 2017.
259. Shiyi Wei, Piotr Mardziel, Andrew Ruef, Jeffrey S. Foster, and Michael Hicks. Evaluating design tradeoffs in numeric static analysis for Java. In *ESOP*, volume 10801 of *LNCS*, pages 653–682. Springer, 2018.
260. John Whaley, Dzintars Avots, Michael Carbin, and Monica S. Lam. Using Datalog with binary decision diagrams for program analysis. In *APLAS*, volume 3780 of *LNCS*, pages 97–118. Springer, 2005.
261. John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *PLDI*, pages 131–144. ACM, 2004.
262. Dominik Winterer, Chengyu Zhang, and Zhendong Su. On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. *PACMPL*, 4:193:1–193:25, 2020.
263. Dominik Winterer, Chengyu Zhang, and Zhendong Su. Validating SMT solvers via semantic fusion. In *PLDI*, pages 718–730. ACM, 2020.

264. Dongwei Xiao, Zhibo Liu, Yuanyuan Yuan, Qi Pang, and Shuai Wang. Metamorphic testing of deep learning compilers. *Proc. ACM Meas. Anal. Comput. Syst.*, 6:15:1–15:28, 2022.
265. Xiaoyuan Xie, Joshua Wing Kei Ho, Christian Murphy, Gail E. Kaiser, Baowen Xu, and Tsong Yueh Chen. Application of metamorphic testing to supervised classifiers. In *QSIC*, pages 135–144. IEEE Computer Society, 2009.
266. Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. Finding and understanding bugs in C compilers. In *PLDI*, pages 283–294. ACM, 2011.
267. Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. Fuzzing SMT solvers via two-dimensional input space exploration. In *ISSTA*, pages 322–335. ACM, 2021.
268. Peisen Yao, Heqing Huang, Wensheng Tang, Qingkai Shi, Rongxin Wu, and Charles Zhang. Skeletal approximation enumeration for SMT solver testing. In *ESEC/FSE*, pages 1141–1153. ACM, 2021.
269. Hao Yuan and Patrick Th. Eugster. An efficient algorithm for solving the dyck-cfl reachability problem on trees. In *ESOP*, volume 5502 of *LNCS*, pages 175–189. Springer, 2009.
270. Andreas Zeller. *Why Programs Fail, Second Edition: A Guide to Systematic Debugging*. Morgan Kaufmann, 2009.
271. Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, and Zhendong Su. Finding and understanding bugs in software model checkers. In *ESEC/FSE*, pages 763–773. ACM, 2019.
272. Qirun Zhang, Chengnian Sun, and Zhendong Su. Skeletal program enumeration for rigorous compiler testing. In *PLDI*, pages 347–361. ACM, 2017.
273. Xin Zheng and Radu Rugina. Demand-driven alias analysis for C. In *POPL*, pages 197–208. ACM, 2008.
274. Zhiqiang Zuo, Rong Gu, Xi Jiang, Zhaokang Wang, Yihua Huang, Linzhang Wang, and Xuandong Li. Bigspa: An efficient interprocedural static analysis engine in the cloud. In *IPDPS*, pages 771–780, 2019.
275. Zhiqiang Zuo, John Thorpe, Yifei Wang, Qiuhong Pan, Shenming Lu, Kai Wang, Guoqing Harry Xu, Linzhang Wang, and Xuandong Li. Grapple: A graph system for static finite-state property checking of large-scale systems code. In *EuroSys*. ACM, 2019.

Curriculum Vitae

Research Interests

I am broadly interested in automatic testing, debugging, security, formal analysis, and verification of complex software systems. The overarching goal of my research is to improve software correctness and reliability by making it easier and practical for developers of all skill levels to incorporate state of the art formal methods tools in their software development workflow.

Education

06/2018 - 11/2022 **Max Planck Institute for Software Systems, Kaiserslautern, Germany**
Doctoral Researcher in Computer Science, advised by Prof. Maria Christakis

04/2014 - 04/2018 **University of Freiburg, Freiburg, Germany**
Masters of Science in Computer Science

09/2009 - 08/2013 **National University of Sciences and Technology, Islamabad, Pakistan**
Bachelors of Engineering in Computer Engineering

Employment

11/2022 - present **Amazon Web Services, Berlin, Germany**
Applied Scientist in Automated Reasoning Group

06/2018 - 11/2022 **Max Planck Institute for Software Systems, Kaiserslautern, Germany**
Doctoral Researcher in Computer Science

06/2021 - 10/2021 **Amazon Web Services, Berlin, Germany**
Applied Scientist Intern in Automated Reasoning Group

Publications

1. Maria Christakis, Matthias Heizmann, **Muhammad Numair Mansur**, Christian Schilling, and Valentin Wüstholz. Semantic fault localization and suspiciousness ranking. In *TACAS*, volume 11427 of LNCS, pages 226–243. Springer, 2019.

2. **Muhammad Numair Mansur**, Maria Christakis, Valentin Wüstholz, and Fuyuan Zhang. Detecting critical bugs in SMT solvers using blackbox mutational fuzzing. In *ESEC/FSE*, pages 701–712. ACM, 2020.
3. **Muhammad Numair Mansur**, Benjamin Mariano, Maria Christakis, Jorge A. Navas, and Valentin Wüstholz. Automatically tailoring abstract interpretation to custom usage scenarios. In *CAV*, volume 12760 of LNCS, pages 777–800. Springer, 2021.
4. **Muhammad Numair Mansur**, Maria Christakis, and Valentin Wüstholz. Metamorphic testing of datalog engines. In *ESEC/FSE*, pages 639–650. ACM, 2021.
5. Maria Christakis, Thomas Cottenier, Antonio Filieri, Linghui Luo, **Muhammad Numair Mansur**, Lee Pike, Nico Rosner, Martin Schäf, Aritra Sengupta, and Willem Visser. Input splitting for cloud-based static application security testing platforms. In *ESEC/FSE*, pages 1367–1378. ACM, 2022.
6. **Muhammad Numair Mansur**, Valentin Wüstholz, Maria Christakis. Dependency-Aware Metamorphic Testing of Datalog Engines. In *ISSTA*, ACM, 2023.