

# Testing Program Analyzers and Verifiers

Maria Christakis<sup>\*1</sup>, Alastair F. Donaldson<sup>\*2</sup>, John Regehr<sup>\*3</sup>, and Thodoris Sotiropoulos<sup>\*4</sup>

1 TU Wien, AT. [maria.christakis@tuwien.ac.at](mailto:maria.christakis@tuwien.ac.at)

2 Imperial College London, GB. [alastair.donaldson@imperial.ac.uk](mailto:alastair.donaldson@imperial.ac.uk)

3 University of Utah – Salt Lake City, US. [regehr@cs.utah.edu](mailto:regehr@cs.utah.edu)

4 ETH Zürich, CH. [theodoros.sotiropoulos@inf.ethz.ch](mailto:theodoros.sotiropoulos@inf.ethz.ch)

---

## Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 25242 “Testing Program Analyzers and Verifiers”. Program analyzers and verifiers are routinely employed during software development to prevent and detect faults. In this seminar, we examine the impact of faults within these tools, distinguishing between those that are critical and those that are less severe. We also explore and discuss state-of-the-art techniques for uncovering faults in program analyzers and verifiers, their connections to related domains such as compiler testing, and potential future directions for improving their reliability.

**Seminar** June 9–12, 2025 – <https://www.dagstuhl.de/25242>


**2012 ACM Subject Classification** Software and its engineering → Software testing and debugging; Software and its engineering → Software verification

**Keywords and phrases** formal methods, program analysis, static analysis, testing, verification

**Digital Object Identifier** 10.4230/DagRep.15.6.69

## 1 Executive Summary

*Thodoris Sotiropoulos (ETH Zürich, CH)*

**License**  Creative Commons BY 4.0 International license  
© Thodoris Sotiropoulos

In an era where software pervades every facet of modern life, ensuring the correctness of software systems is of paramount importance. Program analyzers and verifiers are integral ingredients of this endeavor. They provide developers with essential tools to identify and prevent potential software faults before they impact production systems and end users. However, just like all software, analyzers and verifiers are not free from faults. Faults in analysis and verification tools can potentially undermine the entire software ecosystem, leading to missed vulnerabilities, wasted development efforts, and more.

The goal of this Dagstuhl Seminar was to (1) identify the challenges associated with the problem of ensuring the reliability of program analyzers and verifiers, (2) discuss potential ways to address these challenges, and (3) connect both practitioners as well as researchers working in this domain. In particular, the seminar aimed to bring together experts in program analysis, verification, automated testing, and formal methods. The discussion focused on three themes:

- **Severity of faults within program analysis and verification tools:** This involved a detailed examination of how different types of faults can impact various user groups (e.g., end users, software developers) by taking into consideration the context and the

---

\* Editor / Organizer



Except where otherwise noted, content of this report is licensed under a Creative Commons BY 4.0 International license

Testing Program Analyzers and Verifiers, *Dagstuhl Reports*, Vol. 15, Issue 6, pp. 69–83

Editors: Maria Christakis, Alastair F. Donaldson, John Regehr, and Thodoris Sotiropoulos



DAGSTUHL  
REPORTS

Dagstuhl Reports  
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

intended users of the analysis. For example, we elucidated essential properties required in analyses (e.g., soundness for safety-critical software), while discussing which properties may be less critical.

- **Automated generation of test inputs for analyzers and verifiers:** We aimed to thoroughly discuss the challenge of test input generation for finding faults in program analysis and verification, and explored whether existing program generators used in other contexts (e.g., compiler testing) could be potentially applied to validate analyzers/verifiers. For example, program analysis and verification tools are routinely used to detect a huge variety of semantic errors (e.g., buffer overflows, type errors, integer overflows, and many more). A key technical challenge with this is the generation of programs that exhibit interesting semantic errors that are supposedly to be caught by the analyzer/verifier under test.
- **Test oracles to validate program analyzers and verifiers:** Automatically testing an analyzer or verifier requires a test oracle to determine whether it functions as expected. Given that the majority of program analyzers/verifiers lack a specification and are not standardized, this introduces a significant challenge in determining whether their behavior is correct. The goal was to discuss potential test oracles for different types of faults within program analyzers and verifiers.

The aforementioned discussions were realized through a combination of talks and panel discussions. The seminar began with a brief introduction of the attendees, followed by a keynote by John Regehr outlining the seminar's goals. Afterwards, 14 participants from both academia and industry presented work related to the seminar's scope, including novel test oracles for detecting faults in program analyzers and verifiers, testing in emerging domains (e.g., quantum platforms), and new program generation techniques tailored to analyzers.

The seminar schedule was intentionally open and flexible, encouraging attendees to propose discussion topics. Two dedicated discussion panels were held. The first panel focused on whether proactive testing of program analyzers and verifiers is worthwhile. The debate was structured around opposing viewpoints: John Regehr argued in favor of testing analyzers and verifiers, while Alastair Donaldson presented arguments against it.

The second panel covered a broader range of themes, including the significance of faults in analyzers and verifiers, the role of AI in testing, the problem of program generation saturation, and methods for assessing the performance of analyzers and verifiers.

## 2 Table of Contents

### Executive Summary

<i>Thodoris Sotiropoulos</i> . . . . .	69
--	----

### Overview of Talks

Better Fuzzing via Grammar Mutation and Repair <i>Cristian Cadar</i> . . . . .	73
Testing Equivalence Checkers <i>Alastair F. Donaldson</i> . . . . .	73
Search+LLM-based Testing for ARM Simulators <i>Karine Even-Mendoza</i> . . . . .	74
Constraint-Based Test Oracles for Program Analyzers <i>Markus Fleischmann, Maria Christakis, Anastasia Isychev, David Kaïndlstorfer, and Valentin Wüstholtz</i> . . . . .	74
Checkification: Testing Your (Static Analysis) Truths <i>Manuel Hermenegildo</i> . . . . .	75
Interrogation Testing of Program Analyzers for Soundness and Precision Issues <i>David Kaïndlstorfer, Maria Christakis, Anastasia Isychev, and Valentin Wüstholtz</i> .	76
UBGen: Generating UB Programs for testing Sanitizers <i>Shaohua Li</i> . . . . .	76
The Mopsa static analysis platform, and our quest to ease implementation & maintenance <i>Raphaël Monat</i> . . . . .	77
Semantic Metamorphic Testing for Finding Bugs in SMT Solvers. <i>Hakjoo Oh</i> . . . . .	77
Testing Quantum Computing Platforms <i>Michael Pradel</i> . . . . .	78
hevm, a flexible symbolic execution framework to verify EVM bytecode <i>Mate Soos</i> . . . . .	78
Synthesizing Test Cases for Testing Type Checkers <i>Thodoris Sotiropoulos</i> . . . . .	79
Termination (Resilience) Analysis, and Bugs in Its Implementation <i>Caterina Urban</i> . . . . .	79
Fuzzing Zero-Knowledge Infrastructure <i>Valentin Wüstholtz, Maria Christakis, and Anastasia Isychev</i> . . . . .	80
On Test Oracles for Program Analyzers and Verifiers <i>Chengyu Zhang</i> . . . . .	80

### Panel discussions

Proactively Testing Program Analyzers and Verifiers: Is It Worth It? <i>Thodoris Sotiropoulos</i> . . . . .	81
--	----

**72      25242 – Testing Program Analyzers and Verifiers**

Fault De-duplication, Prioritization, And Other Considerations <i>Thodoris Sotiropoulos</i> . . . . .	82
<b>Participants</b> . . . . .	<b>83</b>

## 3 Overview of Talks

### 3.1 Better Fuzzing via Grammar Mutation and Repair

*Cristian Cadar (Imperial College London, GB)*

**License** © Creative Commons BY 4.0 International license  
© Cristian Cadar

**Joint work of** Bachir Bendrissou, Alastair Donaldson, Cristian Cadar

**Main reference** Bachir Bendrissou, Cristian Cadar, Alastair F. Donaldson: “Grammar Mutation for Testing Input Parsers”, *ACM Trans. Softw. Eng. Methodol.*, Vol. 34(4), pp. 116:1–116:21, 2025.

**URL** <https://doi.org/10.1145/3708517>

In this talk, I presented our recent and ongoing work on enhancing grammar-based fuzzing via grammar mutation and repair.

Our project GMutator aims to generate edge inputs, particularly those incorrectly accepted by a program, via the novel concept of grammar mutation – where the grammar is first mutated before being used for fuzzing.

Our project AFLRepair combines grammar-based fuzzing with greybox fuzzing. To avoid input mutations leading to mostly invalid inputs, we combine standard byte-level mutations with a repair stage.

Both our projects found bugs that are out of reach of prior approaches.

### 3.2 Testing Equivalence Checkers

*Alastair F. Donaldson (Imperial College London, GB)*

**License** © Creative Commons BY 4.0 International license  
© Alastair F. Donaldson

**Joint work of** Michalis Pardalos, Alastair F. Donaldson, Emiliano Morini, Laura Pozzi, John Wickerson

**Main reference** Michalis Pardalos, Alastair F. Donaldson, Emiliano Morini, Laura Pozzi, John Wickerson: “Who checks the checkers? Automatically finding bugs in C-to-RTL formal equivalence checkers”, in *Proc. of the DVCCon Europe 2024; Design and Verification Conference and Exhibition Europe*, pp. 39–44, 2024.

**URL** <https://doi.org/10.30420/566438006>

C-to-RTL (register-transfer level) formal equivalence checkers (ECs) allow hardware implementations to be compared against software specifications. Thanks to their complete state-space coverage, ECs are trusted to authorise design sign-off. Therefore, ridding ECs of bugs is a top priority. In pursuit of this goal, we have developed Equifuzz, a technique and tool for randomized testing (fuzzing) of SystemC-to-RTL ECs. Equifuzz uses knowledge of SystemC semantics to generate rich designs that are known to be equivalent to trivial RTL designs. It has uncovered 7 unsoundness bugs in major commercial ECs (where the EC claimed equivalence incorrectly), and 5 incompleteness bugs (where the EC failed to prove equivalence between equivalent designs), all of which have been confirmed by the tool vendors. The fact that Equifuzz has been able to find serious bugs in extensively tested, major commercial ECs demonstrates that fuzzing is a valuable complement to the handcrafted tests that EC developers use as standard.

### 3.3 Search+LLM-based Testing for ARM Simulators

*Karine Even-Mendoza (King’s College London, GB)*

**License** © Creative Commons BY 4.0 International license  
© Karine Even-Mendoza

**Main reference** Karine Even-Mendoza, Héctor D. Menéndez, William B. Langdon, Aidan Dakhama, Justyna Petke, Bobby R. Bruce: “Search+LLM-Based Testing for ARM Simulators”, in Proc. of the 47th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice, SEIP@ICSE 2025, Ottawa, ON, Canada, April 27 – May 3, 2025, pp. 469–480, IEEE, 2025.

**URL** <https://doi.org/10.1109/ICSE-SEIP66354.2025.00047>

In order to aid quality assurance of large complex hardware architectures, system simulators have been developed. However, such system simulators do not always accurately mirror what would have happened on a real device. A significant challenge in testing these simulators comes from the complexity of having to model both the simulation and the infinite number of software that could be run on such a device.

Our previous work introduced SearchSYS, a testing framework for software simulators. SearchSYS leverages a large language model for initial seed C code generation which is then compiled, and the resultant binary is fed to a fuzzer. We then use differential testing by running the outputs of fuzzing on real hardware and a system simulator to identify mismatches.

In this talk, we present and discuss our solution to the problem of testing software simulators, using SearchSYS to test the gem5 VLSI digital circuit simulator, employed by ARM to test their systems. In particular, we focus on the simulation of the ARM silicon chip Instruction Set Architecture (ISA). SearchSYS can create test cases that activate bugs by combining LLMs, fuzzing, and differential testing. Using only LLM, SearchSYS identified 74 test cases that activated bugs. By incorporating fuzzing, this number increased by 93 additional bug-activating cases within 24 hours. Through differential testing, we identified 624 bugs with LLM-generated test cases and 126 with fuzzed test inputs. Out of the total number of bug-activating test cases, 4 unique bugs have been reported and acknowledged by developers. Additionally, we provided developers with a test case suite and fuzzing statistics, and open-sourced SearchSYS.

### 3.4 Constraint-Based Test Oracles for Program Analyzers

*Markus Fleischmann (TU Wien, AT), Maria Christakis (TU Wien, AT), Anastasia Isychev (TU Wien, AT), David Kaindlstorfer (TU Wien, AT), and Valentin Wüstholtz (Consensus – Wien, AT)*

**License** © Creative Commons BY 4.0 International license  
© Markus Fleischmann, Maria Christakis, Anastasia Isychev, David Kaindlstorfer, and Valentin Wüstholtz

**Main reference** Markus Fleischmann, David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholtz, Maria Christakis: “Constraint-Based Test Oracles for Program Analyzers”, in Proc. of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024, pp. 344–355, ACM, 2024.

**URL** <https://doi.org/10.1145/3691620.3695035>

Program analyzers implement complex algorithms and, as any software, can contain bugs. Bugs in their implementation may lead to analyzers being imprecise and failing to verify safe programs, i.e., programs with no reachable error locations; or worse, analyzer bugs may lead to reporting unsound results by verifying unsafe programs, i.e., programs with reachable error locations.

In this paper, we propose a method to detect such bugs by generating constraint-based test oracles for analyzers. We re-purpose and extend Fuzzle, a tool for benchmarking fuzzers, in a tool called Minotaur. Minotaur generates C programs from SMT constraints, and based on the satisfiability of the constraints, derives whether the generated programs are safe or unsafe. For instance, for an unsafe program, an analyzer under test contains a soundness issue if it proves it safe. Using Minotaur, we found 30 unique soundness and precision issues in 11 well-known analyzers that reason about reachability properties.

### 3.5 Checkification: Testing Your (Static Analysis) Truths

*Manuel Hermenegildo (IMDEA Software Institute – Pozuelo de Alarcón, ES & UPM – Madrid, ES)*

**License** © Creative Commons BY 4.0 International license  
© Manuel Hermenegildo

**Joint work of** Manuel Hermenegildo, Ignacio de Casso, Daniela Ferreiro, Pedro Lopez-Garcia, and Jose F. Morales

**Main reference** Daniela Ferreiro, Ignacio Casso, José F. Morales, Pedro López-García, Manuel V. Hermenegildo: “Checkification: A Practical Approach for Testing Static Analysis Truths”, CoRR, Vol. abs/2501.12093, 2025.

**URL** <https://doi.org/10.48550/ARXIV.2501.12093>

In this talk we will present and demo our “checkification” approach: a simple, automatic method for testing static analyzers. Broadly, it consists in checking that the properties inferred statically are satisfied dynamically. The main advantage of checkification lies in its simplicity, specially when framed within the Ciao assertion-based validation framework, which implements a blend of static and dynamic assertion checking.

We will demonstrate how in this setting analysis results can be tested with little effort by combining, via a simple program transformation, the basic components that comprise the framework itself: 1) the static analyzer, which outputs its results as the original program source with assertions interspersed; 2) the assertion run-time checking mechanism, which instruments a program to ensure that no assertion is violated at run time; 3) the random test case generator, which generates random test cases satisfying the properties present in assertion preconditions; and 4) the unit-testing framework, which executes those test cases. We will show the interaction of these components while checking the results of the CiaoPP abstract interpretation-based static analyzer, for several abstract domains for different properties, analysis (fixpoint) algorithms, etc.

### 3.6 Interrogation Testing of Program Analyzers for Soundness and Precision Issues

*David Kaindlstorfer (TU Wien, AT), Maria Christakis (TU Wien, AT), Anastasia Isychev (TU Wien, AT), and Valentin Wüstholtz (Consensys – Wien, AT)*

**License** © Creative Commons BY 4.0 International license

© David Kaindlstorfer, Maria Christakis, Anastasia Isychev, and Valentin Wüstholtz

**Main reference** David Kaindlstorfer, Anastasia Isychev, Valentin Wüstholtz, Maria Christakis: “Interrogation Testing of Program Analyzers for Soundness and Precision Issues”, in Proc. of the 39th IEEE/ACM International Conference on Automated Software Engineering, ASE 2024, Sacramento, CA, USA, October 27 - November 1, 2024, pp. 319–330, ACM, 2024.

**URL** <https://doi.org/10.1145/3691620.3695034>

Program analyzers are critical in safeguarding software reliability. However, due to their inherent complexity, they are likely to contain bugs themselves, and the question of how to detect them arises. Existing approaches, primarily based on specification-based, differential, or metamorphic testing, have been successful in finding analyzer bugs, but also come with certain limitations. In this paper, we present interrogation testing, a novel testing methodology for program analyzers, to address limitations in existing metamorphic-testing techniques. Specifically, interrogation testing introduces two key innovations by (1) incorporating more information from analyzer queries to construct more powerful oracles, and (2) introducing a knowledge base that maintains a history of diverse queries. We implemented interrogation testing in Sherlock and tested 8 mature analyzers—including model checkers, abstract interpreters, and symbolic-execution engines—that can prove the safety of assertions in C/C++ programs. We found 24 unique issues in these analyzers, 16 of which are soundness related, i.e., an analyzer does not report an assertion that can be violated. Our experimental evaluation demonstrates Sherlock’s effectiveness by finding issues between 7x and 906x faster than our baseline, which is inspired by the state of the art.

### 3.7 UBGen: Generating UB Programs for testing Sanitizers

*Shaohua Li (The Chinese University of Hong Kong, HK)*

**License** © Creative Commons BY 4.0 International license

© Shaohua Li

**Joint work of** Shaohua Li, Zhendong Su

**Main reference** Shaohua Li, Zhendong Su: “UBFuzz: Finding Bugs in Sanitizer Implementations”, in Proc. of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1, ASPLOS 2024, La Jolla, CA, USA, 27 April 2024- 1 May 2024, pp. 435–449, ACM, 2024.

**URL** <https://doi.org/10.1145/3617232.3624874>

In this talk, I will introduce our new program generator UBGen, which can generate C programs with various undefined behaviors, such as buffer overflows and null pointer dereferences. We have used UBGen to fuzz sanitizers (ASan, UBSan, and MSan) and have successfully detected more than 10 false negative bugs, where sanitizers failed to report the UB in programs.

### 3.8 The Mopsa static analysis platform, and our quest to ease implementation & maintenance

*Raphaël Monat (INRIA Lille, FR)*

**License** © Creative Commons BY 4.0 International license  
© Raphaël Monat

**Joint work of** Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné

**Main reference** Raphaël Monat, Abdelraouf Ouadjaout, Antoine Miné: “Easing Maintenance of Academic Static Analyzers”, International Journal on Software Tools for Technology Transfer, Vol. CSV 2024 Special Issue, Springer Verlag, 2025.

**URL** <https://doi.org/10.1007/s10009-024-00770-1>

Mopsa is a Modular Open Platform for Static Analysis, whose goal is to encourage research and education in static analysis, by providing a fully-featured and extensible open-source platform and usable analyses built with it. In particular, analyses in Mopsa can reach a high expressivity, thanks to a framework allowing an extensive use of relational domains, which are able to infer linear constraints between variables. In this talk, we will see a brief overview of Mopsa, our main design decisions and the results we have able to obtain so far.

Implementations of static analyzers are time-consuming to develop and to maintain, but necessary to enable building further research upon the implementation. This talk will present the tools and techniques we have come up with to simplify the maintenance of Mopsa. First, we describe an automated way to measure precision that does not require any manual inspection of the results, improves transparency of the analysis, and helps discovering regressions during continuous integration. Second, we have taken inspiration from standard tools observing the concrete execution of a program to design custom tools observing the abstract execution of the analyzed program itself, such as abstract debuggers and profilers. Finally, we report on some cases of automated testcase reduction.

### 3.9 Semantic Metamorphic Testing for Finding Bugs in SMT Solvers.

*Hakjoo Oh (Korea University – Seoul, KR)*

**License** © Creative Commons BY 4.0 International license  
© Hakjoo Oh

Ensuring the correctness of SMT solvers is thus critical, as they serve as the cornerstone of a wide range of software engineering applications, from symbolic execution and program verification to program synthesis and repair. However, existing testing techniques, such as differential and metamorphic testing, have limitations: the former requires multiple solvers and is confined to shared functionality, while the latter is often restricted to simple, syntactic-preserving transformations. In this talk, I present DIVER, a technique that overcomes these limitations through semantic metamorphic testing. Unlike prior approaches, DIVER performs oracle-guided, unrestricted random mutations based on the semantic model of a formula, enabling it to uncover deep, solver-specific soundness and model-generation bugs that are out of reach for existing tools. Using DIVER, we have discovered 25 new bugs in Z3, CVC5, and dReal, including subtle logical errors that had persisted in production releases for years.

### 3.10 Testing Quantum Computing Platforms

*Michael Pradel (Universität Stuttgart, DE)*

**License**  Creative Commons BY 4.0 International license  
© Michael Pradel

**Joint work of** Michael Pradel, Matteo Paltenghi


**Main reference** Matteo Paltenghi, Michael Pradel: “MorphQ: Metamorphic Testing of the Qiskit Quantum Computing Platform”, in Proc. of the 45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023, pp. 2413–2424, IEEE, 2023.

**URL** <https://doi.org/10.1109/ICSE48619.2023.00202>

Quantum computing is a rapidly evolving field with the potential to revolutionize a wide range of industries. At the core of this revolution are quantum computing platforms, which – similar to traditional analyzers and compilers – analyze, optimize, and translate quantum programs. Unfortunately, like any complex software, these platforms are susceptible to bugs that can undermine the correctness and reliability of quantum applications. This talk presents two automated testing techniques designed to address this challenge. The first, MorphQ, is a metamorphic testing approach tailored to quantum computing platforms. It combines a generator of quantum input programs with a suite of program transformations that exploit quantum-specific metamorphic relationships to uncover inconsistencies. The second technique, QITE, introduces a cross-platform testing framework for quantum computing. It is based on a novel ITE process that generates equivalent quantum programs by iteratively (I) Importing assembly code into platform-specific representations, (T) Transforming the programs via platform-specific optimizations and gate conversions, and (E) Exporting them back to assembly. Both approaches have successfully identified a range of previously unknown bugs in widely used quantum computing platforms, including Qiskit, PennyLane, and Pytket, thereby contributing to the robustness and trustworthiness of this emerging field.

### 3.11 hevm, a flexible symbolic execution framework to verify EVM bytecode

*Mate Soos (Ethereum – Berlin, DE)*

**License**  Creative Commons BY 4.0 International license  
© Mate Soos

**Joint work of** Mate Soos, dxo, Zoe Paraskevopoulou

**Main reference** Dxo, Mate Soos, Zoe Paraskevopoulou, Martin Lundfall, Mikael Brockman: “Hevm, a Fast Symbolic Execution Framework for EVM Bytecode”, in Proc. of the Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I, Lecture Notes in Computer Science, Vol. 14681, pp. 453–465, Springer, 2024.

**URL** [https://doi.org/10.1007/978-3-031-65627-9\\_22](https://doi.org/10.1007/978-3-031-65627-9_22)

We present hevm, a symbolic execution engine for the EVM. hevm can prove safety properties for EVM bytecode or verify semantic equivalence between two bytecode objects. It exposes a user-friendly API in Solidity that allows end-users to define symbolic tests using almost the same syntax as they would for their usual unit tests. We evaluate our framework against state-of-the-art tools, using a comprehensive set of benchmarks. Our empirical findings demonstrate that hevm outperforms its counterparts, effectively solving a greater number of problems within competitive time frames.

### 3.12 Synthesizing Test Cases for Testing Type Checkers

*Thodoris Sotiropoulos (ETH Zürich, CH)*

**License** © Creative Commons BY 4.0 International license  
© Thodoris Sotiropoulos

**Main reference** Thodoris Sotiropoulos, Stefanos Chaliasos, Zhendong Su: “API-Driven Program Synthesis for Testing Static Typing Implementations”, Proc. ACM Program. Lang., Vol. 8(POPL), pp. 1850–1881, 2024.

**URL** <https://doi.org/10.1145/3632904>

Type checkers are the most widely used form of static analysis, helping us identify bugs in programs during development. However, bugs in type checkers themselves can harm the programmer experience and, more critically, pose security risks, especially when they compromise the soundness of the checker.

In this talk, we address one of the central challenges in program generation for testing type checkers: saturation. We introduce THALIA, a framework that leverages APIs from real-world software libraries to synthesize small client programs designed to stress-test type checker implementations. The strength of THALIA comes from the inherent complexity of modern APIs, which often depend on advanced typing features such as parametric polymorphism and overloading. By exploiting these APIs, THALIA produces programs that exercise sophisticated typing behaviors without the need to explicitly generate those features from scratch.

When applied to popular type checkers, THALIA uncovered dozens of previously unknown bugs, many of which had been missed by existing program generation techniques.

#### References

- 1 Thodoris Sotiropoulos, Stefanos Chaliasos, Zhendong Su: API-Driven Program Synthesis for Testing Static Typing Implementations. Proc. ACM Program. Lang. 8(POPL): 1850-1881 (2024)
- 2 Stefanos Chaliasos, Thodoris Sotiropoulos, Diomidis Spinellis, Arthur Gervais, Benjamin Livshits, Dimitris Mitropoulos: Finding typing compiler bugs. PLDI 2022: 183-198
- 3 Stefanos Chaliasos, Thodoris Sotiropoulos, Georgios-Petros Drosos, Charalambos Mitropoulos, Dimitris Mitropoulos, Diomidis Spinellis: Well-typed programs can go wrong: a study of typing-related bugs in JVM compilers. Proc. ACM Program. Lang. 5(OOPSLA): 1-30 (2021)

### 3.13 Termination (Resilience) Analysis, and Bugs in Its Implementation

*Caterina Urban (INRIA & ENS Paris, FR)*

**License** © Creative Commons BY 4.0 International license  
© Caterina Urban

**Joint work of** Caterina Urban, Naïm Moussaoui Remil

We present a novel abstract interpretation-based static analysis for proving Termination Resilience, the absence of Robust Non-Termination vulnerabilities in software programs. Robust Non-Termination characterizes programs where an externally-controlled input can force infinite execution, independently of other uncontrolled variables. The approach is implemented in the open-source tool FuncTION. We conclude with an overview of the bugs that we accidentally found during its development, longing for a more principled way to uncover such issues.

### 3.14 Fuzzing Zero-Knowledge Infrastructure

*Valentin Wüstholtz (Consensys – Wien, AT), Maria Christakis (TU Wien, AT), Anastasia Isychev (TU Wien, AT)*

**License** © Creative Commons BY 4.0 International license

© Valentin Wüstholtz, Maria Christakis, and Anastasia Isychev

**Joint work of** Christoph Hochrainer, Anastasia Isychev, Valentin Wüstholtz, Maria Christakis

**Main reference** Christoph Hochrainer, Anastasia Isychev, Valentin Wüstholtz, Maria Christakis: “Fuzzing Processing Pipelines for Zero-Knowledge Circuits”, CoRR, Vol. abs/2411.02077, 2024.

**URL** <https://doi.org/10.48550/ARXIV.2411.02077>

Zero-knowledge (ZK) infrastructure is highly complex and highly critical for the correct operation of several privacy-focused applications, such as online voting and blockchains; that is, a single bug can result in massive financial and reputational damage. To find such potential million-dollar bugs before they are exploited, we have developed a novel fuzzing technique that can find logic flaws that impact soundness or completeness of ZK infrastructure. Our fuzzer has already found 20 such issues in four ZK systems, namely Circom, Corset, Gnark, and Noir.

### 3.15 On Test Oracles for Program Analyzers and Verifiers

*Chengyu Zhang (Loughborough University, GB)*

**License** © Creative Commons BY 4.0 International license

© Chengyu Zhang

**Joint work of** Chengyu Zhang, Zhendong Su, Dominik Winterer, Ting Su, Geguang Pu, Fuyuan Zhang, Yichen Yan, Weigang He, Peng Di, Mengli Ming, Shijie Li, Yulei Sui

**Main reference** Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, Zhendong Su: “Finding and understanding bugs in software model checkers”, in Proc. of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019, pp. 763–773, ACM, 2019.

**URL** <https://doi.org/10.1145/3338906.3338932>

Program analyzers and verifiers are fundamental to building reliable software systems. Consequently, developing effective methodologies and practical tools to solidify these foundational components is critical. However, constructing test oracles for program analyzers and verifiers poses significant challenges due to the inherent complexity of their tasks.

In this talk, I will summarize three effective methodologies for building test oracles for program analyzers, verifiers, and their underlying tools. These methodologies have proven successful in uncovering thousands of bugs across various software, including SMT solvers, software and hardware model checkers, program verifiers, and static analyzers.

The talk will focus on both the theoretical and practical challenges of constructing test oracles for program analyzers and verifiers, and introduce the latest advances.

#### References

- 1 Chengyu Zhang, Ting Su, Yichen Yan, Fuyuan Zhang, Geguang Pu, Zhendong Su: Finding and understanding bugs in software model checkers. ESEC/SIGSOFT FSE 2019: 763-773
- 2 Dominik Winterer, Chengyu Zhang, Zhendong Su: Validating SMT solvers via semantic fusion. PLDI 2020: 718-730
- 3 Dominik Winterer, Chengyu Zhang, Zhendong Su: On the unusual effectiveness of type-aware operator mutations for testing SMT solvers. Proc. ACM Program. Lang. 4(OOPSLA): 193:1-193:25 (2020)

- 4 Weigang He, Peng Di, Mengli Ming, Chengyu Zhang, Ting Su, Shijie Li, Yulei Sui: Finding and Understanding Defects in Static Analyzers by Constructing Automated Oracles. *Proc. ACM Softw. Eng.* 1(FSE): 1656-1678 (2024)
- 5 Chengyu Zhang, Zhendong Su: SMT2Test: From SMT Formulas to Effective Test Cases. *Proc. ACM Program. Lang.* 8(OOPSLA2): 222-245 (2024)

## 4 Panel discussions

### 4.1 Proactively Testing Program Analyzers and Verifiers: Is It Worth It?

*Thodoris Sotiropoulos (ETH Zürich, CH)*

License  Creative Commons BY 4.0 International license  
© Thodoris Sotiropoulos

The first discussion panel of the seminar explored whether it is worthwhile to proactively test program analyzers and verifiers, particularly through techniques such as fuzzing. Arguments were presented on both sides. We clarify that these arguments do not suggest that program analyzers and verifiers do not matter. Instead, we question whether it is important to *proactively* test them using fuzzing to find edge case bugs.

#### Arguments against proactive testing.

- Software systems rely on many components including user code (the code that should be verified or analyzed), specifications (describing what should be checked), libraries, compilers, operating systems, testing infrastructure and test suites, and program analyzers and verifiers. Why should our attention specifically go to program analyzers and verifiers? Isn't it more likely there will be serious programs with specifications, or issues in library code, or compiler bugs?
- For a verifier bug to cause a severe failure, several conditions must align:
  - A bug exists in the analyzer.
  - A bug exists in the user code.
  - The analyzer bug masks the user bug.
  - The masked bug is not caught by other means (testing, reviews, etc.).
  - The masked bug leads to a serious real-world failure.

The probability of all these factors aligning was considered small. As a concrete example, a bug in Dafny was mentioned, which was deemed too much of an edge case to realistically occur in practice.

- Proactive fuzzing requires ongoing investment in maintaining target systems across versions, which may not always be feasible or cost-effective.
- Even when bugs are found, developers may not respond effectively, whether because they are overwhelmed or because they prioritize other issues. This raises concerns about the return on investment in proactive testing program analyzers and verifiers.


#### Arguments in favor of proactive testing.

- A single fault in a verifier can affect many users and programs at the same time, amplifying its potential consequences compared to faults in user-specific code.
- If analyzers and verifiers are perceived as untrustworthy, the entire software ecosystem suffers. For example, large-scale systems like Amazon's infrastructure, which processes billions of queries daily, depend critically on trustworthy analyzers.
- As software development increasingly moves toward specification-driven approaches (programs verified rather than tested), the correctness of analyzers and verifiers becomes central to reliability.

- Bugs that affect users cannot always be predicted in advance. Proactive testing provides a way to uncover such issues before they manifest in practice.
- Past experience with unreliable compilers demonstrates the importance of testing infrastructure tools early, rather than assuming they are inherently reliable.
- Proactive testing adds an important safeguard. Even if most testing is carried out by users, regression testing and fuzzing provide an early line of defense.

## 4.2 Fault De-duplication, Prioritization, And Other Considerations

*Thodoris Sotiropoulos (ETH Zürich, CH)*

License  Creative Commons BY 4.0 International license  
© Thodoris Sotiropoulos

This session discussed several topics proposed by the attendees during the seminar. The discussions were around the importance of faults in program analyzers/verifiers, bug de-duplication and prioritization, saturation of program generation, AI for testing, and performance testing of program analyzers/verifiers.

**Fault importance and prioritization.** A key theme was the prioritization of bugs. Participants distinguished between bug de-duplication (eliminating duplicate reports) and bug prioritization (deciding which unique bugs are most important to address). Many attendees emphasized that de-duplication based on crash signatures can be misleading, as many similar crashes may stem from distinct underlying issues. Ultimately, only the tool developers are in a strong position to correctly identify duplicates.

The type of discovered issues may also influence prioritization. For example, some attendees shared their experience with their interaction with the Kotlin development team. In particular, Kotlin developers recently focused on regressions between Kotlin v1.0 and v2.0, rather than on bugs present in both versions, even when those bugs affected soundness of type checkers.

**AI for testing.** The panel discussed AI-assisted testing, including the Fuz4all project [1]. While details were not extensively covered, the discussion reflected growing interest in leveraging machine learning and AI to improve bug-finding effectiveness.

**Saturation of program generation.** The notion of saturation, that is, the point at which a program generator appears to stop finding new bugs, was revisited. Attendees argued that saturation is not necessarily negative: once saturation is reached, any additional bugs discovered are highly likely to be novel and meaningful. However, other attendees advocated that “there is no end”: even slight modifications to a generator or support for new features often lead to the discovery of new classes of bugs.

**Performance testing.** Performance was briefly mentioned, with particular focus on performance-critical software such as SMT solvers. Since analyzers often depend on solvers, small regressions in solver speed or behavior can have side-effects in program analysis tools. Sometimes, this can cause analyses to fail unexpectedly. While this was only lightly discussed, the brittleness of solver performance was recognized as a practical concern.

### References

- 1 Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, Lingming Zhang: Fuzz4All: Universal Fuzzing with Large Language Models. ICSE 2024: 126:1-126:13

**Participants**

- Cristian Cadar  
Imperial College London, GB
- Maria Christakis  
TU Wien, AT
- Pascal Cuoq  
TurstInSoft – Paris, FR
- Alastair F. Donaldson  
Imperial College London, GB
- Karine Even-Mendoza  
King’s College London, GB
- Markus Fleischmann  
TU Wien, AT
- Amber Gorzynski  
Imperial College London, GB
- Manuel Hermenegildo  
IMDEA Software Institute –  
Pozuelo de Alarcón, ES &  
UPM – Madrid, ES
- Anastasia Isychev  
TU Wien, AT
- David Kaindlstorfer  
TU Wien, AT
- Shaohua Li  
The Chinese University of  
Hong Kong, HK
- Muhammad Numair Mansur  
Amazon Web Services –  
Berlin, DE
- Raphaël Monat  
INRIA Lille, FR
- Hakjoo Oh  
Korea University – Seoul, KR
- Michael Pradel  
Universität Stuttgart, DE
- John Regehr  
University of Utah –  
Salt Lake City, US
- Mate Soos  
Ethereum – Berlin, DE
- Thodoris Sotiropoulos  
ETH Zürich, CH
- Hao Sun  
ETH Zürich, CH
- Caterina Urban  
INRIA & ENS Paris, FR
- Valentin Wüstholtz  
Consensys – Wien, AT
- Chengyu Zhang  
Loughborough University, GB

