Report from Dagstuhl Seminar 23131

Software Bug Detection: Challenges and Synergies

Marcel Böhme*[1], Maria Christakis*[2], Rohan Padhye*[3], Kostya Serebryany*[4], Andreas Zeller*[5], and Hasan Ferit Eniser†[6]

1 **MPI-SP – Bochum, DE & Monash University – Melbourne, AU.** `marcel.boehme@mpi-sp.org`
2 **TU Wien, AT.** `maria.christakis@tuwien.ac.at`
3 **Carnegie Mellon University – Pittsburgh, US & Amazon Web Services, US.** `rohanpadhye@cmu.edu`
4 **Google – Mountain View, US.** `kcc@google.com`
5 **CISPA – Saarbrücken, DE.** `zeller@cispa.saarland`
6 **MPI-SWS – Kaiserslautern, DE.** `hfeniser@mpi-sws.org`

Abstract

This report documents the program and the outcomes of Dagstuhl Seminar 23131 "Software Bug Detection: Challenges and Synergies". This seminar brought together researchers from academia and industry working on various aspects of software bug detection, with two broad goals: identifying challenges in practical deployment of bug-finding tools and discovering new synergies among bug-finding techniques and research methods. The seminar focused discussion on bug-finding tools and their relevance and adoption in industry.

Seminar March 26–31, 2023 – https://www.dagstuhl.de/23131
2012 ACM Subject Classification Software and its engineering; Software and its engineering → Software testing and debugging; Software and its engineering → Software verification
Keywords and phrases Bug Finding, Coverage, ML4SE, Oracles, Software Testing, Software Verification
Digital Object Identifier 10.4230/DagRep.13.3.92

1 Executive Summary

Rohan Padhye (Carnegie Mellon University – Pittsburgh, US & Amazon Web Services, US)
Marcel Böhme (MPI-SP – Bochum, DE & Monash University – Melbourne, AU)
Maria Christakis (TU Wien, AT)
Kostya Serebyany (Google – Mountain View, US)
Andreas Zeller (CISPA – Saarbrücken, DE)

Software bugs are inevitable when engineering complex systems, and the cost of their consequences can be enormous. Over the past several decades, there has been tremendous progress in advancing the state-of-the-art in automatic bug finding. Popular techniques include static analysis, dynamic analysis, formal methods and specification, verification, symbolic execution, fuzzing, and search-based test generation. However, with the rapid growth of new application domains and the ever-increasing complexity of software, practitioners are rarely faced with a one-size-fits-all solution for finding bugs in their software. Domain-specific

* Editor / Organizer
† Editorial Assistant / Collector

trade-offs must be made in choosing the right technique, in configuring a tool to work for a particular context, or in combining multiple approaches to provide better assurances. Currently, this is largely a manual activity and the burden is mainly on practitioners.

This Dagstuhl Seminar brought together researchers from academia and industry working on various aspects of software bug detection, with two broad goals: identifying challenges in practical deployment of bug-finding tools and discovering new synergies among bug-finding techniques and research methods.

The seminar focused discussion on bug-finding tools and their relevance and adoption in industry. Other questions that came up included: What are effective approaches to discover software bugs as fast as possible? How can we formally verify the absence of bugs? Which guarantees do our approaches provide about the correctness, reliability, and security of the software when no bugs are discovered? Which concerns do practitioners have when bug finding tools are integrated into their development process? What are effective approaches to automatically mitigate, diagnose, or repair certain kinds of bugs?

The seminar was organized to maximize time for open discussion. Seven attendees were invited to give short keynote talks of a topic of their choice, which occurred on mornings of the seminar. The afternoons were reserved for working groups and panel discussions. The topics for these discussions were crowdsourced using an ad-hoc voting system in the main seminar room. Working groups then broke out for discussion in smaller rooms and reconvened with summaries.

Overall, in the opinion of the organizers, the seminar was a huge success. The strong participation from researchers in industry and the diverse set of expertise among researchers in academia enabled open-minded discussion on topics of key importance that are not easily exchanged via traditional conference proceedings. This document summarizes the talks given and working groups conducted in the seminar.

2 Table of Contents

Executive Summary
Rohan Padhye, Marcel Böhme, Maria Christakis, Kostya Serebyany, and Andreas Zeller . . . 92

Overview of Talks

Reflections on Software Testing Research
Cristian Cadar . . . 96

Beyond the Crash Oracle: Challenges in Deploying Fuzzing to Find Functional Errors at Scale
Alastair F. Donaldson . . . 96

Lessons Learned from Designing Software Engineering Methods for Enabling AI
Miryung Kim . . . 97

Dependencies everywhere!
Anders Møller . . . 97

Hits and Misses From a Decade of Program Analysis in Industry
Peter O'Hearn . . . 97

Daunting and Exciting Reality of Industrial Bug Hunting
Dmitrii Viukov . . . 98

From Bug Detection to Bug Mitigation and Elimination: the Role of Tools in Memory Safety
Anna Zaks . . . 98

Working groups

Oracles 3
Eric Bodden . . . 98

Oracles 1
Hasan Ferit Eniser . . . 99

Severity Analysis
Caroline Lemieux . . . 99

Competitions and Evaluations
Rohan Padhye . . . 100

Correctness and Verification
Rohan Padhye . . . 100

Reproducibility and Artifacts
Rohan Padhye . . . 101

Oracles 2
Mathias Payer . . . 101

User Experience in Fuzzing
Van-Thuan Pham . . . 102

Large Language Models for Bug Detection
Michael Pradel . . . 103

Learning-based and Analysis-based Bug Detection
Michael Pradel 103

New Coverage Signals
Kostya Serebryany and Hasan Ferit Eniser 103

ML and Static Analysis
Dominic Steinhöfel 104

Dependencies
Andreas Zeller 104

Participants 105

3 Overview of Talks

3.1 Reflections on Software Testing Research

Cristian Cadar (Imperial College London, GB)

The talk covered a number of challenges and opportunities for software testing research, including understanding developer communities, benchmarking, incremental progress and research directions, the need for patch testing techniques, the problems on which academia should focus on, the challenges of maintaining tools in academia, and the many misaligned incentives that researchers in the area are facing.

3.2 Beyond the Crash Oracle: Challenges in Deploying Fuzzing to Find Functional Errors at Scale

Alastair F. Donaldson (Imperial College London, GB)

Main reference Alastair F. Donaldson, Hugues Evrard, Paul Thomson: "Putting Randomized Compiler Testing into Production (Experience Report)", in Proc. of the 34th European Conference on Object-Oriented Programming, ECOOP 2020, November 15-17, 2020, Berlin, Germany (Virtual Conference), LIPIcs, Vol. 166, pp. 22:1–22:29, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020.
URL https://doi.org//10.4230/LIPIcs.ECOOP.2020.22

Over the last decade, fuzzing has been shown to be extremely effective at finding security critical defects in software and is now deployed at scale by many large companies and open-source projects. However, most of the success of fuzzing at scale is restricted to finding inputs that cause the system under test to crash. While testing with respect to the "crash oracle" is important for finding vulnerabilities (especially when the crash oracle is boosted with compile-time instrumentation to detect undefined behaviour), its effectiveness for finding deep functional errors is limited – errors that lead to the system under test doing the wrong thing – but doing so without actually crashing.

The talk covered why designing and scaling up fuzzing techniques for finding functional errors is so much harder than in the context of crashes. Issues include the manual effort required to build smart input generators and mutators, the need to respect input validity constraints not only during generation/mutation, but also during test-case reduction, and the difficulty associated with de-duplicating bug-triggering test cases. Throughout the talk, the speaker drew on his experience designing GraphicsFuzz, a metamorphic fuzzing technique for GPU compilers on which he based a start-up company that was acquired by Google and subsequently used to fuzz Android graphics drivers to find functional bugs.

3.3 Lessons Learned from Designing Software Engineering Methods for Enabling AI

Miryung Kim (UCLA, US)

Software developers are rapidly adopting AI to power their applications. Current software engineering techniques do not provide the same benefits to this new class of compute and data-intensive applications. To provide productivity gains that developers desire, our research group has designed a new wave of software engineering methods.

First, the speaker discussed technical challenges of making custom hardware accelerators accessible to software developers. She showcased HeteroGen, an automated program repair and test input generation method for making heterogeneous application development with FPGA accessible to software developers. Second, she discussed technical challenge of designing automated testing and debugging methods for big data analytics. She also showcased BigTest, symbolic-execution based test generation for Apache Spark.

At the end, the speaker shared the lessons learned from designing SE methods that target big data and HW heterogeneity and discuss open problems in this data and compute-intensive domain.

3.4 Dependencies everywhere!

Anders Møller (Aarhus University, DK)

Modern software critically relies on reusable, open source software packages. They enable fast development of advanced applications, but also introduce major challenges with incompatibilities, security vulnerabilities, and breaking changes. In this talk, the speaker described ongoing work at coana.tech on building new program analysis tools that can assist library and application developers by providing accurate information about how dependencies are being used.

3.5 Hits and Misses From a Decade of Program Analysis in Industry

Peter O'Hearn (University College London, GB)

The speaker talked about hits and misses from a decade of program analysis in industry.

3.6 Daunting and Exciting Reality of Industrial Bug Hunting

Dmitrii Viukov (Google – München, DE)

In this talk, the speaker shared his experience deploying various bug detection tools and fuzzing at Google over the last decade. The talk touched on the goals, context, constraints and everyday life of the team. Then the talk proceeded to the common properties of the tools that found (and not found) adoption and concludes with a look into the future and what types of tools we are looking for.

3.7 From Bug Detection to Bug Mitigation and Elimination: the Role of Tools in Memory Safety

Anna Zaks (Apple Computer Inc. – Sunnyvale, US)

A key ingredient to securing a platform is mitigation and elimination of memory safety errors in software. Dynamic and static bug-finding tools can be used to find many memory-safety bugs such as buffer overflows, use-after-frees, and data races. However, most such tools assume no change in the environment – the compiler, the language, libraries, operating system and the hardware – which limits their impact. In this talk, the speaker described how to co-design language security features with the rest of the software stack and how that helps design tools and techniques for mitigation and elimination of whole classes of memory-safety bugs. She also talked about the new role bug-finding techniques like program analysis can play in a world where safer languages, libraries, and security mitigations are available.

4 Working groups

4.1 Oracles 3

Eric Bodden (Universität Paderborn, DE)

In this working group, attendees discussed oracles in different domains and ways to generalize them. For instance, one widely considered oracle type is memory corruption because it is very generic, easy to specify and detect (particularly if they lead to a crash). However, we also need better oracles in other domains such as in API testing, big data analytics and so on. One can utilize error handling code in APIs as oracles and test them using directed fuzzing techniques. In big data analytics, the available options to tackle the oracle problem are metamorphic and differential testing which come with their own limitations. A high-level idea was to define simpler proxy properties, whose violations indicate a problem whereas conformances are not very interesting, in domains where specifying complete oracles is hard. As a result, discussions converged towards the conclusion of benefiting the most suitable oracle option per domain such as heuristics, proxy properties, error handling code, metamorphic or differential testing.

4.2 Oracles 1

Hasan Ferit Eniser (MPI-SWS – Kaiserslautern, DE)

Metamorphic and differential testing techniques are examples of the most widely applied techniques to overcome the well-known oracle problem. However, manual effort for crafting metamorphic operations, little or no (e.g. coverage) feedback, and the non-existence of multiple subjects under test limit applicability of these techniques in practice. For this reason, “assertion” based oracles still play an important role in bug detection. Nonetheless, they also come with their own limitations. For example, hyperproperties are hard to express with assertions. Using ML to determine successful and failing executions or human-in-the-loop based solutions can also be considered for this problem. One example usage of ML can be to rank rules that serve as oracles for a particular program. Other open problems in this area include oracles for patch generation and dealing with bugs in specifications.

4.3 Severity Analysis

Caroline Lemieux (University of British Columbia – Vancouver, CA)

In a world of finite developer resources, we cannot prioritize the investigation and fixing of all potential bugs. The goal of severity analysis is to rank bugs by severity, so as to direct developer time to those bugs whose fix is most likely to improve the software system.

We considered two perspectives on severity. The first is severity from a security perspective. We discussed a few factors that should go into judging the severity of a bug from a security perspective. The second one is exploitability (currently, this is manually done by human analysts when a bug is e.g. submitted to MITRE for ranking as a CVE) Higher exploitability implies higher severity. And the last one is ease of discoverability where we assume easier to discover usually means higher severity, as bad actors may be more likely to discover it.

The perspective of functional severity is slightly different. When considering the severity of a functional (i.e., not likely to have security impacts, but which may crash or give a wrong result) bug, we may care more about user experience for example frequency of bug-revealing inputs in the usual input distribution. In this case, the more commonly seen in practice implies more severe bugs.

Additionally in both cases, how many different paths there are to reveal bug may relate to severity: higher number of potentially reaching paths means more severe.

4.4 Competitions and Evaluations

Rohan Padhye (Carnegie Mellon University – Pittsburgh, US & Amazon Web Services, US)

Joint work of Rohan Padhye, Dirk Beyer

This working group described the importance of competitions and standardized evaluation methodologies in supporting research on automated bug-finding. This approach has been very successful in accelerating research in domains such as SMT solvers. The discussion first addressed the goals such an evaluation, which can be to find as many bugs as possible in as little time as possible, as well as the use of other metrics such as code coverage. Care should also be taken to distinguish between general-purpose bug finding tools and customized tools that search for special classes of bugs. The group then identified several efforts in this space, such as the DARPA Cyber Grand Challenge (CGC) [1] dataset which contains 50 programs each having one bug, the LAVA-M [2] dataset of artificial bugs, the MAGMA [3] data-set containing historical bugs, and the FuzzBench [4] data-set containing known bugs in real programs that fuzzers can find. Some concerns with using such data sets include representativeness of bugs, whether the bugs in artificial datasets are realistic and likely to occur in practice, whether tools might overfit to these benchmarks, the computational effort required to find small numbers of bugs in large programs, and various constrained imposed by competitions such as a strict time limit. The group agreed that finding new bugs in real target is that are previously well-tested is a good way of demonstrating a technique's effectiveness, but it is not always possible to find such results. Formats such as registered reports, which allow papers to be peer-reviewed before evaluation results are available, appear to mitigate some of the risks with expecting authors to find such previously unknown bugs with new tools even if the technique is novel and sound.

References

1 Lee, N. (2015). Darp's cyber grand challenge (2014–2016). Counterterrorism and Cybersecurity: Total Information Awareness, 429-456.
2 Dolan-Gavitt, B., Hulin, P., Kirda, E., Leek, T., Mambretti, A., Robertson, W., ... & Whelan, R. (2016, May). Lava: Large-scale automated vulnerability addition. In 2016 IEEE symposium on security and privacy (SP) (pp. 110-121). IEEE.
3 Hazimeh, A., Herrera, A., & Payer, M. (2020). Magma: A ground-truth fuzzing benchmark. Proceedings of the ACM on Measurement and Analysis of Computing Systems, 4(3), 1-29.
4 Metzman, J., Szekeres, L., Simon, L., Sprabery, R., & Arya, A. (2021, August). Fuzzbench: an open fuzzer benchmarking platform and service. In Proceedings of the 29th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering (pp. 1393-1403).

4.5 Correctness and Verification

Rohan Padhye (Carnegie Mellon University – Pittsburgh, US & Amazon Web Services, US)

Joint work of Dirk Beyer

This working group discussed the importance of correctness guarantees in automated program analysis tools used for improving software quality. The properties to check could include functional correctness or things like reachability. Many static techniques need to use some form of underapproximation or overapproximation for scalability. Dynamic techniques like testing or fuzzing do not cover all paths and so provide no guarantees, but continuous integration is important. There is a scope to include some form of verification in CI. Deductive verification techniques can scale using composition, but require annotations from developers. Proof-of-concept projects from the operating systems domain provide some hope. Incremental verification is also a promising direction. The group discussed directions that researchers can investigate to increase the adoption and applicability of verification techniques. A theme that emerged is that verification must be thought not only as correctness, but as something to integrate into processes, so we need ways to integrate techniques and run them in dev processes. The verification process is also where you find out what you want to verify, during the process!

4.6 Reproducibility and Artifacts

Rohan Padhye (Carnegie Mellon University – Pittsburgh, US & Amazon Web Services, US)

Joint work of Dirk Beyer

The working group discussed the current state of artifact evaluation in ACM-sponsored conferences along and the relevance of this process. The group first identified what the various badges mean. The "Available" badge indicates that the artifact must have a DOI, must be identifiable, immutable, and long-term available. If an artifact is evaluated by a committee, it can be given a badge of "Functional" or "Reusable" based on criteria set out by the ACM and the artifact evaluation chairs. The artifact can also be considered "Validated" if the results of the accompanying research paper are independently "reproduced" (using author-provided artifacts) or "replicated" (by re-implementing the artifact using the information available in the paper). There seems to be considerable discrepancy between conferences and ACM SIGs on how to interpret these badges. For example, SIGMOD provides artifact "reproduced" badge even when the results are validated by an artifact evaluation committee, which in SIGSOFT conferences is just considered "functional" (reserving the "reproduced" badge for independent studies by future authors). The group agreed that more consistency is needed in issuing these badges. The group also discussed whether the effort of artifact evaluation is justifiable and whether there is any incentive for authors to produce high quality artifacts. Another question that came up was whether future authors who reuse the artifact should cite the artifact DOI or the original paper. The group identified some benefits for the artifact process, such as the fact that it allows students to serve on review committees and experience working with artifacts authored by their peers. Reproducibility and Artifacts

4.7 Oracles 2

Mathias Payer (EPFL – Lausanne, CH)

In the bug detection oracles work group we focused on ways to signal that a bug was triggered. Oracles implement a given policy and enforce this policy by instrumenting the target program with certain checks. These checks then continuously monitor if the policy has been violated.

We started by discussing sanitizers in general and focusing in particular on AddressSanitizer, UndefinedBehaviorSanitizer and ThreadSanitizer. These are the most well-known sanitizers and thoroughly used. In the discussion, they served as the basis to define Oracle policies and we continued towards program specifications.

If you want to go beyond these general sanitizer policies, you will require some specification that defines program behavior. Generalizing this is somewhat challenging and requires additional work from the developer. This can quickly become domain specific, e.g., protecting SQLlite against SQL-injection, scripts against command injection, or websites against cross-site scripting. At a higher level, data-flow may serve as a general policy but would require massive amounts of performance improvements.

Among others we also discussed possible hardware extensions and how to generalize anomaly detection for these different niches.

4.8 User Experience in Fuzzing

Van-Thuan Pham (The University of Melbourne, AU)

In this group, we discussed user experience in three stages of fuzzing: (i) set up fuzzing environments, (ii) run & monitor fuzzing progress, and (iii) analyze the results.

In the first stage, writing test harnesses/fuzz drivers seem to be the most time-consuming task and it could be challenging for developers. It is one of the reasons why the number of fuzz drivers in the open-source software packages is quite limited. Some solutions have been discussed including transforming unit tests to fuzz drivers (e.g., FuzzTest) or generating fuzz drivers in a fully automated manner (e.g., FUDGE, FuzzGen). Some companies like Google have bug bounty programs to reward researchers who contribute their fuzz drivers.

Writing fuzz drivers manually is time-consuming and error prone in the sense that developers might have difficulties in understanding if their fuzz drivers are working properly or not. So, in the second stage, companies like Meta/Facebook have built "health-check" mechanism to measure run-time metrics (e.g., code coverage improvement trajectory) and then flag potential malfunctional fuzz drivers. Moreover, Indeterminism is a challenge in some fuzz targets.

In the third stage, users expect better tools to support triaging the crashes and more detailed report. For instance, they would like to have better fault localization and root cause analysis utilities. They also expect to know about the severity of the identified bugs. They would also like to get suggested patches to fix the bugs. Another question developers would ask is when they should stop fuzzing. For ClusterFuzzLite setup at Google, they use a threshold of 10 mins while in fuzzing research papers, results are normally reported for 24-hour experiments.

4.9 Large Language Models for Bug Detection

Michael Pradel (Universität Stuttgart, DE)

The impressive power of large language models (LLMs) has lead to the question how to use these models for bug detection. We discussed two main directions. First, LLMs could generate inputs for testing programs, in a way similar to fuzzers or automated test generator. A strength of LLMs for this task could be to generate realistic inputs, as the model learns typicaly distributions of inputs from data. A challenge for this idea is how to obtain uncommon inputs from a model trained to predict likely token sequences. Second, LLMs could act in a way similar to static analysis and predict for given piece of code whether the code contains a bug, and ideally, more details on this bug. Initial experiments reported by the participants suggest that both directions are promising and worth exploring.

4.10 Learning-based and Analysis-based Bug Detection

Michael Pradel (Universität Stuttgart, DE)

Machine learning-based techniques for finding bugs and vulnerabilities are showing promising results. At the same time, traditional static and dynamic analysis approaches come with their own benefits. We discussed how to combine the strengths of learning-based and analysis-based bug detection techniques. One interesting direction is to use a trained model as a filter or ranking mechanism applied to warnings reported by a static analysis. For example, such a model could be trained on past warnings and records of how developers reacted to them. Another interesting direction is to feed information an analysis extracts from programs, e.g., aliasing relationships, as an input into a machine learning model. Finally, we discussed ways for more tightly integrating both kinds of approaches, e.g., by having a model query an analysis, or vice versa.

4.11 New Coverage Signals

Kostya Serebryany (Google – Mountain View, US) and Hasan Ferit Eniser (MPI-SWS – Kaiserslautern, DE)

In this working group, we discussed new ways of guiding feedback in the form coverage signals in fuzzers. Custom feedback signals for domain specific problems are almost always helpful to increase bug finding performance. Neuron coverage, which is a coverage metric devised for neural networks is a typical example. However, the challenge here is how to generalize the better coverage signal. For this, we need to think about reasonably abstract state that generalize for a good coverage signal (e.g. depth of stack to guide fuzzer to stack over flows).

4.12 ML and Static Analysis

Dominic Steinhöfel (CISPA – Saarbrücken, DE)

In this working group, we discussed how AI techniques could help in automated software testing. We believe AI-based techniques are too slow to be used as standalone input generators. Yet, they might be useful as an alternative to symbolic execution to solve "fuzzing blockers," i.e., to complement the usual graybox techniques. Similarly, they could help to reduce the overhead in fuzzer harness generation (e.g., for libfuzzer). Orthogonally to these considerations, we think that current advances in AI promise solutions to automatic bug explaining/root causing and other debugging problems such as coming up with an input reaching a chosen line of code. Concerning root causing, we discovered the "ChatDBG" project, which uses GPT to explain errors in interactive debuggers. We investigated this project and were astonished by how trivial the prompt used for GPT was. In conclusion, we think there is much potential in using AI to augment traditional automated testing techniques and address open issues in automated testing and debugging.

4.13 Dependencies

Andreas Zeller (CISPA – Saarbrücken, DE)

Generally speaking, Modularity is a big success of Software Engineering, as it enables the widespread reuse of components we see today. Unfortunately, being dependent on third-party modules also creates new problems, as we need to trust them not to introduce new bugs and vulnerabilities. Solutions discussed included (1) quarantining less trusted modules into sandboxes with least privileges to reduce the risk of vulnerabilities; and (2) being explicit about dependencies for both clients and providers of modules to reduce the risk of breaking compatibility.

Participants

- Cornelius Aschermann
Meta – Seattle, US
- Sébastien Bardin
CEA LIST – L'Hay les Roses, FR
- Lukas Bernhard
CISPA – Saarbrücken, DE
- Dirk Beyer
LMU München, DE
- Eric Bodden
Universität Paderborn, DE
- Marcel Böhme
MPI-SP – Bochum, DE & Monash University – Melbourne, AU
- Herbert Bos
VU University Amsterdam, NL
- Cristian Cadar
Imperial College London, GB
- Sang Kil Cha
KAIST – Daejeon, KR
- Maria Christakis
TU Wien, AT
- Jürgen Cito
TU Wien, AT
- Alastair F. Donaldson
Imperial College London, GB
- Hasan Ferit Eniser
MPI-SWS – Kaiserslautern, DE
- Rahul Gopinath
The University of Sydney, AU
- Alessandra Gorla
IMDEA Software Institute – Madrid, ES
- Reiner Hähnle
TU Darmstadt, DE
- Marc Heuse
marc heuse it security – Berlin, DE
- Christian Holler
Mozilla – Berlin, DE
- Miryung Kim
UC – Los Angeles, US
- Caroline Lemieux
University of British Columbia – Vancouver, CA
- Jonathan Metzman
Google – New York, US
- Peter Müller
ETH Zürich, CH
- Anders Møller
Aarhus University, DK
- Yannic Noller
National University of Singapore, SG
- Peter O'Hearn
University College London, GB
- Hakjoo Oh
Korea University – Seoul, KR
- Alessandro Orso
Georgia Institute of Technology – Atlanta, US
- Rohan Padhye
Carnegie Mellon University – Pittsburgh, US & Amazon Web Services, US
- Mathias Payer
EPFL – Lausanne, CH
- Van-Thuan Pham
The University of Melbourne, AU
- Michael Pradel
Universität Stuttgart, DE
- Manuel Rigger
National University of Singapore, SG
- Kostya Serebryany
Google – Mountain View, US
- Dominic Steinhöfel
CISPA – Saarbrücken, DE
- Dmitrii Viukov
Google – München, DE
- Valentin Wüstholz
ConsenSys – Wien, AT
- Anna Zaks
Apple Computer Inc. – Sunnyvale, US
- Andreas Zeller
CISPA – Saarbrücken, DE
- Lingming Zhang
University of Illinois – Urbana-Champaign, US