# Research Statement

Marcel Böhme

Software controls everything. It executes your financial transactions, powers the breaks in your car, governs critical systems on your plane ride, and controls very intricate medical instruments. Imagine the threat to public safety if the software is compromised. Recently, the ransomware WannaCry exploited a software flaw to infect PCs on a massive scale; hospitals had to shut down.[1] My research concerns *automated, scalable, effective, and practical techniques to analyze, test, and repair industrial-scale software systems*. I developed several efficient vulnerability detection techniques. I formally showed how lightweight blackbox techniques can easily outperform smarter whitebox techniques. To move research closer to practice, I was among the first to conduct, support, and advertise user studies for the evaluation of automated debugging and repair techniques. I empirically showed how to reduce the time to find a vulnerability from several days to only a few hours. My tools exposed about 100 security-critical, previously unreported flaws and vulnerabilities in large and widely-used software systems. More than 40 of my security advisories are now listed in the US National Vulnerability Database.

In future, I plan to take an *interdisciplinary* approach and explore the intersection between Automated Software Engineering (ASE) and Machine Learning (ML) with applications to vulnerability detection and program analysis. ML provides a rich toolset for the *scalable analysis* of massive systems whereas *scalability* is one of the most critical problems of existing ASE techniques. While we develop research techniques that work well for selected toy examples, practitioners cannot adopt them if they do not work for normal-size software. I plan to leverage recent advances in ML to solve the most important research problems in ASE *in the large*; to develop fundamental push-button solutions that *can actually be used*. In order to achieve this scalability, we might trade strict guarantees for probabilistic ones, e.g., using statistical inference. We might trade specific quantities for approximate ones, e.g., using rare event modelling. We can use predictive analysis to determine trajectories of properties or techniques. At all times we shall keep the human in the loop.

## I. Scalable and Effective

**Dissertation work.** At first, I developed techniques that can expose errors which were introduced by software *changes*. My tools found 5 previously unreported errors in the most recent version of the Coreutils which is available on every Linux PC. Every day 60k lines of code are changed in the Linux kernel alone. Naturally, there is no time to test the complete kernel every time a change is committed. The techniques that I developed in my dissertation are *scalable* because they check the small commits instead of the large program and *effective* in showing that the changed program is at least as correct as the previous version. In fact, one technique is so effective that it can actually *prove* that no errors are introduced. However, I felt that my techniques were still too slow. They involved heavyweight program analysis and constraint solving.

**Post-doctoral work.** Recently, I have formally shown how *lightweight* blackbox techniques easily outperform the theoretically *most effective* whitebox testing technique in the realistic setting where developers have only limited time. To discuss my findings, I was invited by several colleagues to give a talk at UCL, SUTD, NTU, and TU Darmstadt. Inspired by the theoretical insights, I have been working on a practical forecasting method that given two testing techniques accurately *predicts* which one is likely to reveal more errors in a given time budget. I developed several very scalable testing techniques: Our tools i) exposed more than a dozen vulnerabilities in large programs, such as Adobe Reader and Windows Media Player, ii) exposed more than 40 previously unreported CVEs[2] in widely-used, security-critical software, iii) has been shared and evaluated by the community resulting in several posts on Hacker News, and iv) outperformed the state-of-the-art by *an order of magnitude*!

[1] https://www.forbes.com/sites/thomasbrewster/2017/05/17/wannacry-ransomware-hit-real-medical-devices

[2] CVEs are security-critical vulnerabilities that are listed in the US National Security Database and by MITRE.

**Impact on Practice.** I developed and contributed to the development of several tools that are publicly available. For instance, my boosted fuzzer AFLFast is available on GitHub[3] where it received 150+ stars and 80+ forks and 100+ tweets on Twitter. With the NUS Industry Liason Office (ILO) we are exploring commercialization avenues for our directed greybox fuzzer AFLGo, and our model-based whitebox fuzzer Hercules++. Moreover, I have contributed patches to several open-source projects, such as the GNU GCC compiler project. My open-source vulnerability detection tools have exposed 100+ bugs in widely-used open-source software and libraries, such as php, perl, valgrind, gdb, ErlangVM, coreutils, binutils, libiberty, libdwarf, libxml2, libming, and libav. For the most-critical vulnerabilities, I published several security advisories which are registered in the US National Vulnerability Database (42 CVEs).

**Going forward.** While my tools do outperform the state-of-the-art, they still only work for small, yet widely-distributed and security-critical libraries. My long-term plan is to develop automated techniques for industrial-size, heterogeneous software systems *that actually work*. Many existing techniques are indeed very effective but fail to work for large, real-world programs: They take too long, they cannot be interrupted to provide intermediate results, they report too many false positives, or they work only for a subset of the language. Machine Learning (ML) has been used to analyze systems of *arbitrary size* and structure. ML provides a rich toolset that draws from statistics, bayesian data analysis, and predictive analytics. I plan to utilize this toolset to build general, scalable, yet effective program analysis and vulnerability detection techniques.

For instance, in his thesis my student Björn Mathis developed a the *first obfuscation-resilient technique* to elicit critical information flows in Android programs (say, from the database storing your credit card number to a server in Croatia). Leveraging ML-based causal inferencing, he implemented a technique that outperforms the state-of-the-art static information flow detection tool on real-world apps using an order of magnitude less code. No false positives and probabilistic bounds on the false negatives. Even better. In principle, ML allows to predict at any time when Björn's technique is going to find the next information flow.

Predictive analytics is very useful also in the context of cybersecurity: "After 1h we found no vulnerabilities, but the chance of finding one in the next 2 days is 10E-9". ML can help to identify average or extremal values of numerical program properties, such as performance, energy consumption, or the "amount" of information that flows from one point to another. Such techniques can expose resource exhaustion vulnerabilities in devices connected through the Internet of Things causing quick battery drainage or in internet services causing excessive CPU and memory usage such that benign requests cannot be entertained. ML will help to build models of the behaviour of huge software systems that can actually be analyzed in our limited time. Our team at NUS has been developing several program repair techniques that leverage program analysis, symbolic execution, and constraint solving to patch, e.g., the Heartbleed vulnerability. I am excited to use this background to develop compiler techniques for generating provably safe and secure code.

Generally, I am focussing on research that has the potential of high impact both on the state-of-practice as well as on the research community. I believe that great research is fundamental and without expiry date. Each individual work should manifest a greater long-term vision. At the same time, I strongly believe that great research is informed by the challenges in industry and should solve real problems.

## II. Useful and Practical

**Dissertation work.** Many automated testing, debugging, and repair techniques have been evaluated based on benchmarks that contain artificial errors that are caused by simple injected faults. We found that real errors are very different from artificial ones, which puts at risk the validity of earlier studies. In my dissertation, I launched CoREBench, the first large, systematic collection of real bugs that were introduced and fixed by open source software developers and reported by users. ACM Software Engineering Notes listed this work in the *Top-10 most downloaded articles for three months.*[4] CoREBench is already being used by several research groups, including at Imperial College London, University of Luxembourg, CMU, Saarland University, and NUS.

[3] https://github.com/mboehme/aflfast
[4] http://portalparts.acm.org/2680000/2674632/fm/frontmatter.pdf

**Postdoctoral work.** Evidently, there is a chasm between software engineering *research and practice*. In a recent study, we found that after half a century of research of automated debugging, the practice is unchanged, manual and ad hoc. The recent spate in research of automated repair techniques seems to inspire among practitioners more suspicion than anticipation. In collaboration with Prof. Andreas Zeller at Saarland University, I have been conducting one of the longest running retrospective and observational studies on the subject of debugging and repair that involves hundreds of software engineering practitioners from all over the world. Given the errors in CoREBench, we ask the participants to localize, explain, and repair these errors. Before developing automated techniques, we learn about the manual processes and the expectations of practitioners on the output of these techniques. Our insights will be published in a sequence of conference and journal articles and help to close the prominent gap between the research and practice of debugging and repair.

In several ongoing works with Prof. Zeller and his PhD student Ezekiel Soremekun, we have been investigating the utility of the most prominent existing automated debugging techniques and been developing several very useful debugging techniques that explain in natural language the root cause of those errors that were introduced by program changes (relative to the correct behavior of the previous version; called delta narratives).

**Going forward.** My long-term plan is to develop automated techniques for industrial-size, heterogeneous software systems *that can actually be used*. We can construct more useful, less suspicious, and more effective automated testing, debugging, and repair techniques if we keep the human in the loop. Again, I can utilize the rich toolset of Machine Learning My automated repair techniques will be able to learn from human-generated bug fixes to generate repairs that developers can actually read and understand, that explain the reasoning for that commit. They will be able to harness large human-generated vulnerability databases that match the observed symptoms (e.g., resource exhaustion) to bug fixes and workarounds. I can leverage the recent spate in source code repositories and app stores containing millions of similar programs and human-generated test suites. Equipped with this information my techniques will be better able to classify program behavior into buggy and correct, which has been a long-standing problem. The generated test cases and repairs would be more readable / trustworthy. Then, developers may actually understand what is going wrong if a generated test fails.

## Selected Bibliography

- **[CCS'17] "**Directed Greybox Fuzzing", M. Böhme, V.T. Pham, M.D. Nguyen, A. Roychoudhury, Under Review at 24th ACM Conference on Computer and Communications Security (CCS) 2017
- **[ESEC/FSE'17] "**Where is the Bug and How is it Fixed? An Experiment with Practitioners", M. Böhme, E. O. Soremekun, S. Chattophadyay, E. Ugherughe, Andreas Zeller, 11th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE) 2017, pp. 1-11
- **[CCS'16]** "Coverage-based Greybox Fuzzing as Markov Chain", M. Böhme, V.T. Pham, A. Roychoudhury, *23rd ACM Conference on Computer and Communications Security (CCS) 2016.* pp. 1032-1043
- **[ASE'16]** "Model-based Whitebox Fuzzing", V.T. Pham, M. Böhme, A. Roychoudhury, 31st International Conference on Software Engineering (ASE) 2016, pp. 543-553
- **[TSE'15]** "A Probabilistic Analysis of the Efficiency of Automated Software Testing", M. Böhme and S. Paul, IEEE Transactions on Software Engineering, Vol. 42, Issue 4, pp. 345-360
- **[FSE'14]** "On the Efficiency of Automated Testing", M. Böhme and S. Paul, 22nd ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE) 2014, pp. 632-642
- **[ISSTA'14]** "CoREBench: Studying Complexity of Regression Errors", M. Böhme and A. Roychoudhury, *23rd ACM/SIGSOFT International Symposium on Software Testing and Analysis* (ISSTA) 2014, pp. 398-408
- **[ICSE'13]** "Partition-based Regression Verification", M. Böhme, B. C.d.S. Oliveira, and A. Roychoudhury, *ACM/IEEE International Conference on Software Engineering (ICSE)* 2013, pp.300-309
- **[FSE'13]** "Regression Tests to Expose Change Interaction Errors", M. Böhme, B. C.d.S. Oliveira, and A. Roychoudhury, *Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)* 2013, pp. 339-349