Dominik Winterer
ETH Zurich, Switzerland

Research Statement

April 2025

Software is pervasive in our everyday lives. Still, software defects (called *bugs*) frequently result in major security and safety threats.[1] **I work towards a future in which critical software bugs no longer exist, thereby preventing dangerous threats to both security and safety.** Of key importance in bringing such a future to fruition are Formal Methods, a dedicated field in computer science featuring mathematically rigorous techniques to prevent bugs. Formal Methods have been studied for decades with substantial breakthroughs: Numerous techniques invented, powerful tools engineered, and several Turing Awards won for advances in the field. Yet, despite all this, Formal Methods have not been widely adopted.

One major obstacle preventing adoption is that Formal Methods tools are not reliable and user-friendly enough [2]. A second major obstacle is that existing tools, techniques, and results are not consolidated, hence largely inaccessible to the mainstream developer [2]. A third major reason is the *specification crisis*, i.e., the fact that most software today lacks precise descriptions of what it is supposed to do.

My career goal is to establish and develop Formal Methods Engineering (FME)—a new dedicated field in computer science for driving Formal Methods' mainstream adoption overcoming all these obstacles. FME is characterized by three major research challenges: *(C1) Making Formal Methods more mature, (C2) Best utilzing Formal Methods, and (C3) Overcoming the specification crisis*. Similar to Software Engineering for software, FME's aim is to transform the use of Formal Methods from an ad-hoc activity to a structured approach and so fundamentally change computer science. *I am a researcher in software engineering and programming languages*. My research yields widely applicable, practical, and ambitious methodologies. I focused on solidifying Satisfiability Modulo Theory (SMT) solvers—one of the most powerful and foundational Formal Methods. My research contribution realizes the first step towards FME by ensuring the reliability and correctness of SMT solvers and hence all software that builds on them. *My research and tools found 1,800+ bugs in SMT solvers, leading to 1,300+ bug fixes among them 350+ highly critical realizing one of the largest bug-hunting campaigns in academia.*

Research Contribution: Solidifying Modern SMT Solvers

I first introduce SMT solvers and clarify their key for software security and safety. Then, I describe my research contribution guided by three consecutive research questions: (RQ1) Do SMT solvers *even have* critical bugs? (RQ2) How to more effectively test SMT solvers? (RQ3) Have we tested enough?

SMT Solvers and their crucial role in ensuring software security and safety A Satisfiability Modulo Theories (SMT) solver is a tool that, given a formula ϕ in first-order logic, determines whether ϕ is satisfiable with respect to a background theory. It returns `sat` if there exists an assignment that satisfies ϕ, and `unsat` otherwise. As a simple example, the formula $\phi = x > 0 \wedge x < 2$, where x is an integer variable, is satisfiable (e.g., $x = 1$), whereas $\phi' = x > 0 \wedge x < 1$ is unsatisfiable. SMT solvers support theories important for reasoning about programs, such as booleans, bitvectors, integers, reals, floating-point numbers, and strings, as well as combinations of these.

SMT solvers are among the most powerful formal methods tools, and are foundational for program verification, model checking, solver-aided programming, symbolic execution, and program synthesis—important techniques to ensure the security and safety of software. SMT solvers are also core components in large-scale industrial applications. Examples include AWS's access policy analyzer Zelkova (used in a third of all AWS services worldwide), Microsoft's SecGuru (for verifying Azure network configurations), and AdaCore's SPARK toolchain (used in safety-critical domains such as aerospace, automotive, and railways). Since SMT solvers are usually deployed in security- and safety-critical contexts, correctness is paramount. For example, if an SMT solver incorrectly returns `unsat` for a satisfiable formula, or vice versa, this is called a soundness bug, which can have serious consequences. Such bugs may propagate to

[1] The most recent example is the CrowdStrike IT outage in July 2024, causing financial damage of 10+ billion USD.

downstream tools, leading to false correctness proofs, undetected bugs, invalid counterexamples, cascading failures in toolchains, and ultimately, a loss of trust in Formal Methods. Therefore, ensuring the correctness of SMT solvers is crucial for ensuring the software security and safety.

RQ1: Do SMT solvers *even have* critical bugs? As of early 2019, there was little reason to doubt the correctness of SMT solvers. GitHub issues of leading SMT solvers contained no soundness bugs. In the SMT solver competition, they were also rare. Moreover, research papers only found very few bugs in unstable solvers. This was indicating that besides a few known issues, SMT solvers should be trusted. *How to put this trust in SMT solvers to the test?* There are two key obstacles: (O1) fabricating complex formulas to thoroughly stress-test the SMT solvers and (O2) validating the SMT solver's result. O1 is challenging because SMT solvers have large codebases and an expressive input format, while O2 is challenging because of the trade-off between formula complexity and control over its satisfiability. I invented *Semantic Fusion*, an approach to solve both challenges [8]. There is a large collection of benchmarks (400k), complex, real-world SMT formulas on which SMT solvers are well-exercised. Semantic Fusion fabricates new formulas from these benchmarks on which SMT solvers are *not* well-exercised, hence potentially returning incorrect results. To create these new formulas, we fuse formula pairs of known satisfiability in many different ways, resulting in a large space of formulas closely resembling the original benchmarks but not quite. The key here is that we fabricate the new formulas so that we also know their satisfiability by construction. We first combine the formulas, resulting in a conjunction/disjunction. Then, we slice off a piece of each formula and combine the two formulas in an interesting manner, resulting in a fused formula that we can use to stress-test SMT solvers. *Do SMT solvers* even have *critical bugs? Yes! They do, and quite a few.* We found 76 bugs in the Z3 and CVC4, the most powerful and popular SMT solvers with this idea. Strikingly, we found soundness bugs, the most critical bug category, that can invalidate the results of verification tools. Many of these bugs were undetected for years and some were shockingly simple. *The work had an eye-opening effect on the community and won a Distinguished Paper Award at PLDI '20.*

RQ2: How to more effectively test SMT solvers? Semantic Fusion is a highly targeted attack, particularly effective on nonlinear and string logic. As fusing two ancestor formulas produces a test formula with a size equal to the combined size of its ancestors, it is crucial to carefully select the ancestor formulas to avoid excessive timeouts during testing. This raises the question of whether we can design a more lightweight technique that lets us leverage all available formulas. Operators are central to the semantics of SMT formulas. If we change them, it affects the solution space of formulas, exercising new control and dataflow in the solver codebase. Hence, mutating operators yields a simple and powerful technique for testing SMT solvers [7]. The key here is also that this mutation should be *type-aware*, to create well-typed formulas to stress-test the SMT solvers. We realized this idea in a testing tool called OpFuzz and conducted a large-scale testing campaign. *We found 1,200+ unique bugs in Z3 and CVC4*. Most notably, OpFuzz found soundness bugs in almost every theory of SMT. *This came as a great shock to the SMT community*. Despite its unusual effectiveness, OpFuzz also has limitations. Formulas will stay at roughly the same size, i.e., they cannot grow or shrink. In our OOPSLA '21 work, we extended OpFuzz, yielding another 237 bugs in Z3 and cvc4 [4].

RQ3: Have we tested enough? With so many bugs being found, this is a natural follow-up question. Existing SMT fuzzers cannot answer this question as they are unsystematic, randomly scattering large test formulas. If fuzzers find no bugs, we do not know whether there are truly none or whether we were just out of luck. Perhaps most importantly, fuzzers miss small-sized bugs. The small scope hypothesis [3] states that *most bugs of software trigger on small inputs*. We confirmed this hypothesis for bugs in SMT solvers. My idea to exploit this insight is to enumerate the smallest inputs from a context-free grammar. There are several unique advantages to this approach. (1) it is systematic (2) if it finds bugs, they are small in size (3) it provides bounded guarantees on formula spaces (4) we can measure the evolution of SMT solvers. We integrate this idea in a tool called ET [6]. It works as follows: given a grammar that describes a relevant formula space and a number of designated tests, it first compiles the grammar into algebraic datatypes in

Haskell. Using this representation, we then take an off-the-shelf enumerative testing library [1]. With ET, we conducted a validation campaign of Z3 and cvc5, the two most widely used and popular solvers. We found 100+ new bugs but perhaps more important than these bugs are the assurances that we gained: SMT solvers no longer have bugs on the smallest formulas anymore. We are working on making ET a part of SMT solver's GitHub CI/CD pipelines. *Did SMT solvers become better, historically?* This is an essential question that ET enables us to investigate. To approach it, we tested all stable releases of Z3 and cvc5 on 8 million formulas generated by ET, tracking the number of bugs being triggered. Our results show a clear trend: While early releases of Z3 and cvc5 exhibit bugs in many theories, later releases show progressively fewer bugs and the latest Z3 has no bugs at all. Have we tested enough? Yes, under the assumption of the small scope hypothesis, *i.e.*, most bugs of software trigger on small inputs.

Future Direction: Formal Methods Engineering — Towards Reliable Software Engineering

My long-term vision is a future in which critical software bugs no longer exist, thereby preventing dangerous threats to both security and safety. To bring this future to fruition, *I will devote my career to developing Formal Methods Engineering (FME)—a new dedicated field for driving Formal Methods' mainstream adoption.* By solidifying SMT solvers, I made a first fundamental step towards establishing this new field. In the following I will outline how I aim at developing FME by resolving three fundamental challenges. My mission is to make substantial progress on each of them by inventing widely-applicable techniques to improve software security and safety.

Challenge C1: Making Formal Methods more mature A key obstacle hindering the mainstream adoption of Formal Methods is insufficient maturity [2]. There are four key aspects to this challenge: (a) reliability, (b) stability, (c) performance, and (d) user-friendliness. Reliability is largely addressed for SMT solvers, but mostly open for many other Formal Methods such as model checkers, interactive theorem provers, symbolic execution engines, program synthesizers and refinement tools, *etc.* Instability is the problem when minor changes to code, input, or specifications lead to unpredictable behavior or proof construction, undermining robustness and maintainability. This is a longstanding open problem particular for SMT solvers and program verifiers. Performance has been addressed on a fine-grained level but is open on a system level. User-friendliness is an open problem for almost all Formal Methods.

To tackle reliability, I aim to develop unified testing approaches applicable across many Formal Methods. I further aim to increase the reliability of Formal Methods (i.e., the *trusted core*) and software in general through life-code debloating.[2] To tackle instability, I propose *metamorphic executions*, a novel technique to create many semantically equivalent but syntactically different program inputs and run them in parallel reducing instability. For performance, I aim at making Formal Methods significantly faster using techniques from compiler optimization and high performance computing. For user-friendliness, I plan to conduct empirical studies to help build more user-friendly Formal Methods. *Resolving C1 will lead to a new generation of much more mature and powerful Formal Methods, boosting mainstream adoption, leading to substantial impacts on academia and industry*.

Challenge C2: Best utilzing Formal Methods Decades of research in Formal Methods have led to a myriad of approaches, techniques, and tools. However, almost all of them have fundamentally the same goal: *to make software more reliable*.[3] A key challenge is to best utilize and consolidate these Formal Methods, gaining maximum software reliability at minimum cost.

To address this challenge, I will invent *a theory for reasoning about software reliability*, as well as validation portfolios maximizing software reliability and minimizing costs in practice. I aim to develop the theory based on the following two steps. Given a software system as a graph of interconnected components $\mathcal{S} = \{C_1, \cdots, C_n\}$ and different Formal Methods $\mathcal{M}$, the high-level idea is the following:

[2]To the best of my knowledge, this is a novel perspective on software debloating [5] which has so far not been focusing on rarely used life code but instead on dead and unused code.

[3]I acknowledge the of Formal Methods for hardware, however, solely focus on software in this research statement.

Step 1 Describe the costs and assurances for using a method $M \in \mathcal{M}$ on a single component $C \in \mathcal{S}$.

$$C, M \rightarrow \$\$_{C,M}, \%_{C,M}$$

where $\$\$_{C,M}$ quantifies the costs spent and $\%_{C,M}$ quantifies the achieved assurances.

Step 2 Propagation: given a fixed mapping of methods to components, determine the overall cost incurred and assurances achieved for the software system $\mathcal{S}$, i.e., $\mathcal{S} \rightarrow \$\$, \%$

The generic definition of the software system as a graph of interconnected components is chosen to allow instantiations describing everything from a standalone code snippet over a single software project to the software universe as a whole. Exploiting the theory, I aim at developing *validation portfolios* that given a software system and its specification, automatically selects the appropriate technique for each component C. As an output, this yields a set of bugs and a certificate to justify the thoroughness of the process. Both the theoretical framework and the validation portfolio will be developed in a hybrid manner where the theory informs the construction of the validation portfolio, and the validation results, in turn, are used to refine and test the theory. *Resolving C2 will provide a principled understanding for reasoning about software reliability and validation portfolios to maximize the benefits and minimize costs of Formal Methods on practical software systems.*

Challenge C3: Overcoming the specification crisis Another major challenge hindering the mainstream adoption is the specification crisis i.e.,most software is written without formal specifications. However most Formal Methods need a specifications to be applied effectively. To address this, we aim to incentivize developers to write specifications by making the benefits more immediate and tangible. One direction is to build tools that support gradual annotation and demonstrate clear advantages to users, like TLA+, dependent types, or property-based testing. Another is to create an infrastructure akin to GitHub, where developers can share and discover functions and modules along with their validation results. This would enable reuse of well-tested, reliable components and foster a culture of specification by integrating formal guarantees into mainstream software development. *Resolving C3 makes formal specifications in software development the rule rather than exception, substantially boosting the adoption of Formal Methods.*

References

[1] Jonas Duregard, Patrick Jansson, and Meng Wang. "FEAT: Functional Enumeration of Algebraic Types". In: *Haskell '12*. 2012, 61–72.

[2] Hubert Garavel, Maurice H. ter Beek, and Jaco van de Pol. "The 2020 Expert Survey on Formal Methods". In: *Formal Methods for Industrial Critical Systems (FMICS 2020)*. Springer, 2020, pp. 3–69.

[3] D. Jackson and C.A. Damon. "Elements of style: analyzing a software design feature with a counterexample detector". In: *IEEE Transactions on Software Engineering* (1996), pp. 484–495.

[4] Jiwon Park, Dominik Winterer, Chengyu Zhang, and Zhendong Su. "Generative Type-Aware Mutation for Testing SMT Solvers". In: *OOPSLA '21*. 2021, pp. 1 –19.

[5] Chenxiong Qian, Hong Hu, Mansour Alharthi, Pak Ho Chung, Taesoo Kim, and Wenke Lee. "RAZOR: A Framework for Post-deployment Software Debloating". In: *USENIX Security Symposium (USENIX Security 19)*. 2019, pp. 1733–1750.

[6] Dominik Winterer. "Grammar-based Enumeration of SMT Solvers for Correctness and Performance". In: *OOPSLA '24*. 2024.

[7] Dominik Winterer, Chengyu Zhang, and Zhendong Su. "On the Unusal Effectiveness of Type-Aware Operator Mutation". In: *OOPSLA '20*. 2020, pp. 1–25.

[8] Dominik Winterer, Chengyu Zhang, and Zhendong Su. "Validating SMT Solvers via Semantic Fusion". In: *PLDI '20*. 2020, pp. 718–730.