Software controls every device, system, and infrastructure serving our society. Rapid growth of software complexity and high demand for human resources makes the task of building software tedious and error-prone. My research aims at making the reasoning about software automated, which helps to build trustworthy, secure, and compact programs. I develop algorithms for automated *formal methods* which nowadays rely on decision procedures and solvers for Satisfiability Modulo Theories (SMT). My results span the foundations behind SMT and scalable applications of SMT to program verification and synthesis.

In particular, to verify that a program meets a specification, my algorithms not only explore all possible program behaviors but also do it symbolically and efficiently. They are essential ingredients in *security verification*, which reveals information leakage and side channels; and automated *program synthesis*, which guarantees that the resulting programs are correct by construction. To enable these applications, I have improved SMT solvers with algorithms for solving formulas over a set of unknown predicates, also known as *Constrained Horn Clauses* (CHC), which have lately become the formula language of choice for problems in program verification. They offer the advantages of flexibility and modularity in designing verifiers for various systems and languages and are applicable even for verification of smart contracts and neural networks. I have also worked on backend solver techniques for incremental Craig-interpolation-based satisfiability solving and for synthesizing Skolem functions via lazy quantifier elimination. At the same time, I pursued a tight integration of SMT solvers with program verifiers and synthesizers, such that the SMT formulas by construction are smaller and easier to solve, and verifiers/synthesizers exploit the output of SMT solvers to a more significant extent.

In the future, I will work on improving the scalability of formal methods by applying recent advances of *machine learning* and developing new SMT-based algorithms. For instance, analysis of data gathered in user reports, code repositories, and log files will guide the verification and synthesis processes and make them scale better. It is equally important to develop foundational aspects behind SMT which will enable CHC-based verification of programs handling complex data structures.

1 Research Contributions

My area of expertise is in program verification and synthesis, existing approaches to which are in general resource-demanding and usually treat underlying SMT solvers as black box. To lower the cost of verification and synthesis, my algorithms leverage backend solver techniques and establish a tight integration of SMT solvers with verifiers and synthesizers. For instance, my lazy quantifier elimination procedure helped to discover inductive invariants, simulation relations, existential recurrence sets, and thus enabled scalable algorithms for proving safety, termination, and security properties of programs. Furthermore, my technique for extracting Skolem functions allowed a fast and practical approach to functional synthesis.

I acknowledge my collaborators from Princeton University, Carnegie Mellon Software Engineering Institute, King’s College London, Universities of Washington, Minnesota, Waterloo, and Lugano, and TCS Research Pune.

**Synthesizing proofs for program verification.** State-of-the-art techniques for automated program verification certify their outcomes with formal proofs. When verifying program safety, a proof is an *inductive invariant*, when proving program termination, a proof is a *ranking function*, and when proving program non-termination, a proof is a *recurrence set*. Synthesis of these proofs necessitates exploring a usually large search space and relies on SMT solvers. However, it is often the case that the search space could be biased by extensive use of the program’s source code. Intuitively, source code often gives useful information, e.g., of occurrences of variables, constants, and operators. I showed that source code could be used to automatically construct a formal grammar, which is often sufficient for an iterative probabilistic sampling of proofs. I developed an extensible verification framework called *FreqHorn* with applications for proving safety [14, 8], termination, and non-termination [6] of programs. The experimental evaluation confirmed that although the formal grammar is limited to the syntax of the program, in the majority of cases my algorithms are scalable and fast. More generally, *FreqHorn* is a CHC solver [7] which enables a broader class of applications including program synthesis and security verification (see in next paragraphs), solving word equations with length constraints, verification of deep neural networks and Ethereum smart contracts (planned for future work).

**Relational verification for security.** Many applications, including verification of security properties such as non-interference, require reasoning over multiple program executions. Different executions can be compared symbolically to check whether there is a leak of sensitive information. This is traditionally done via composing
two programs (or copies of the same program) in a so-called product program and verifying the result for safety. However, the challenge arises when a verifier analyzes a sequence of loops in a naively composed program: inductive invariants for them are difficult to find and often are inexpressible in the verifier’s backend language. I developed an approach that creates a product program in a lockstep manner [11, 10] and is further reducible to CHC solving. Instead of precise isolated invariants, it generates more general and easy-to-find relational invariants that are adequate for the verification task. Further, a similar technique was optimized, generalized, and implemented on top of Relational Hoare Logic [4].

**Enumerative synthesis for parallelization.** Being a special case of relational verification, a program equivalence problem also benefits from techniques to efficiently construct and verify product programs. In turn, effective and fast equivalence provers, which are at heart program synthesizers, make the synthesis technology applicable, e.g., to automatically parallelize loops. Given a sequential program, a parallelizer searches for a parallel program (in a pool of candidate programs) which is equivalent to the given sequential program. Thus, the synthesis procedure is reduced to a sequence of equivalence verification tasks. I developed a parallelizer GRASSP and applied it for distributed big-data analytics [19, 15]. In practice, Hadoop translations of the GRASSP solutions speed performance linearly on an AWS cluster.

**Fast functional synthesis.** The task of generating a function implementation from an input-output relation is commonly addressed by functional synthesis. Unlike the enumerative synthesis (discussed above), functional synthesis is driven by the mathematical structure of the specification and does not restrict the shape of the possible solution. In practice, such synthesizers scale much better than enumerative ones but produce large and redundant solutions. I developed a technique called AE-VAL [20, 3] which performs a lazy quantifier elimination and extracts witnessing Skolem functions for all quantified variables. The outputs of AE-VAL are often more compact than the ones produced by the competing tools. Further, AE-VAL was applied to the JSYN synthesizer for C implementations complying with specifications written in the Lustre dataflow language [5].

**Incremental program verification.** One way to lower the cost of verification is to exploit the observation that software is developed gradually. The verification could mirror the development process and should reuse efforts invested in the preceding stages of the software lifecycle. Aiming to support such scenario, verification tools should act in an incremental manner. I investigated various incremental techniques and extended state-of-the-art by approaches to deal with multiple program versions. In particular, a NIAGARA framework [25, 20, 18] aims to lift inductive invariants across program versions driven by the automatic synthesis of simulation relations among program versions. It uses AE-VAL on the backend. In cases when the lifting of inductive invariants is impossible, NIAGARA generates an explanation of what was changed in the program that breaks the specification via a so-called change impact certificate. A FUNFROG framework [30, 29, 28, 27, 26, 23, 24, 13, 22, 17, 16, 12, 9], orthogonal to NIAGARA, aims to generate function summaries utilizing interpolation and reuse them among program versions. Whenever a program is proven safe, FUNFROG constructs summaries from the safety proof; and whenever the program is modified, it re-validates the summaries locally, thus avoiding an expensive verification of the entire program from scratch.

**Mining program specifications.** The automation level of the process of formulating program specifications is crucial for many testing and formal-verification toolchains. Specifications can be obtained in the form of assertions from the source code or behavior observed during runtime. The assertions can then be formally verified against the source code using any of the existing tools. I developed an algorithm for identifying implication relations between assertions, filtering noise, and finding redundant assertions and implemented it on top of FUNFROG [21]. This knowledge becomes useful in an industrial environment when the number of assertions generated automatically grows large. I showed experimentally that this technique could be used to reduce the number of assertions that need to be checked formally, thus improving overall verification performance.

## 2 Ongoing and Future Research

Building on my past contributions and expertise in verification and synthesis, I plan to conduct new research that will ultimately deliver new applications and technologies. In particular, much can be gained by using the machine-learning methods in program synthesis and by extending the CHC solvers to handle complex
data structures. It will push the frontiers in automated program repair, functional synthesis, and security verification.

**Next generation of program synthesizers.** Approaches to synthesis are still not mature enough and often require a lot of user insights in forms of an implementable specification, a large set of examples, or fine-tuned formal grammars that describe the shape of desired implementations. Much in synthesis can be improved after applying recent advances of machine learning. One potential improvement is believed to be in a combination of functional or enumerative synthesis and synthesis by examples. While it is in general difficult for users to write full functional specifications, it is often easier to provide a set of concrete input-output pairs. Machine learning is a known method to discover meaningful relationships among the given examples. Such relationships can be taken by AE-VAL together with an *incomplete* (but easy-to-write) functional specification. I believe, such an addition will lower the efforts that the user should do to formulate the specification, and also will result in more compact and expressive solutions.

**Optimal crypto-synthesis.** The security of a crypto-system that handles private data can be compromised not only by leaking information via outputs but also by the presence of side channels via e.g., the execution time or the energy consumption of a program. Attacks that exploit timing side channels include the Lucky 13 attack on SSL/TLS and attacks on the RSA decryption implementation in OpenSSL. To resist timing-based side-channel attacks, implementations of modern crypto-systems should be *constant-time*. These security criteria should apply to bit-precise implementations of arithmetic operations, divergence due to control flow, and balanced termination conditions for loops. Because manually writing constant-time implementations is tedious and error-prone, I propose to automatically generate security patches for blocks of code that operate on sensitive data. Furthermore, since constant-time code usually introduces a significant performance overhead, it makes sense to *maximize* the number of replacements in order to have a high degree of security and to guarantee compliance with certain performance constraints. For the remaining code that is not replaced, I propose to synthesize *padding statements* that are guaranteed to not further increase the worst-case execution time. Initial experiments that use a symbolic program encoding and state-of-the-art Optimizing SMT Solvers have already been successfully conducted on a prototype tool called *ct-synt* [2].

**Automated program repair.** Another improvement of the synthesis technology is by relying on prior experience of humans who have written code before. Because software development is an iterative procedure, all previously implemented functionality of some program should remain available in a repository after a new feature has been implemented. It is useful when a new commit has introduced a bug in a previously developed and verified functionality. Then, a possible patch can be synthesized from the older revisions of the code. A machine-learning tool would maintain a large code base and would suggest candidate replacements. With this guidance, a formal verifier would perform a highly precise filtering of patches by proving or disproving their validity mathematically and fully automatically. Another guidance for bug localization is proposed to be via the output of a tool like *Niagara* which includes a valid counterexample and a change impact certificate. The certificate, obtained for each bug, intuitively identifies fragments of the source code (i.e., particular loops and function calls) likely responsible for the bug, and thus will bias the repair procedure.

**Exploring tradeoffs in program verification.** Existing approaches to discover inductive invariants are limited mainly to light-weight and imprecise theories in first-order logic (such as linear arithmetic or uninterpreted functions) or expensive and impractical bit-precise reasoning. Programs handling arrays, strings and inductively defined data structures provide a challenge for verification tools since invariants require universal quantification over the data elements contained in these structures. For arbitrary quantified formulas, SMT solvers are incomplete and inefficient, but they rarely take into account the exact use of data structures which frequently appears in the code. In fact, even incomplete solvers would be useful, if they can handle practically relevant formulas. Our preliminary experiments with arrays in *FREQHORN* [1] confirms it is promising to extract patterns automatically from the source code and to reduce the invariant generation task over arrays to a sequence of simpler tasks over linear arithmetic. In the future, I envision *FREQHORN* to support inductive data types and strings too.

The proposed research agenda will result in new contributions to the Programming Languages and Software Engineering communities, will give rise to new ideas and proposals, and will benefit society by making software more reliable, efficient, and secure.
Publications


