Research Statement
Xujie Si

My areas of interest are programming languages and artificial intelligence. The goal of my research is to help programmers build reliable software primarily by innovating at the intersection of these two exciting fields. Tremendous progress has been made in programming languages research, and the developed techniques have been adopted by companies like Amazon, Facebook, Google and Microsoft. Recent advances in program synthesis enable end-users to write programs by simply providing a few input and output examples. While these advances are promising, their underlying techniques are designed case by case, requiring non-trivial expertise from their designers and sometimes even their users.

My research has focused on developing general AI techniques to address programming challenges spanning program analysis, program synthesis, software verification, and testing. Below, I briefly summarize my technical contributions, research and industrial impact, and future research plan. Subsequently, I elaborate upon each of these aspects.

Technical Contributions

• First, my thesis work [NeurIPS18, ICLR19] developed a general end-to-end learning framework, which is inspired by recent AI advances in deep learning and reinforcement learning, so that correct proofs, desired programs, and efficient heuristics can be automatically learned without requiring any manually labeled data. This learning framework is applicable to address a range of programming challenges such as program verification, program synthesis, and software testing. My work has also advanced AI techniques, especially representation learning of highly structured data with rich semantics, and reinforcement learning with extremely sparse reward.

• Second, learning and inference are two key components in machine learning applications. My research has developed a scalable inference engine [CP16, CAV17] for Markov logic networks to perform various program analysis tasks [VMCAI18]. Since the quality of inference heavily depends on the choice of rules of the underlying static analysis, I also developed an efficient Datalog synthesis framework [FSE18], which learns interpretable logical rules from input and output relational data. The framework applies to many domains besides program analysis such as big-data analytics and software-defined networks. I further proposed a numerical relaxation technique [IJCAI19] that uses highly efficient gradient-based and stochastic approaches to learn logical rules.

• Third, my research has combined statistical and logical methods in program reasoning to overcome practical issues like missing specifications or even code. Specifically, my research has developed techniques to learn API specifications by applying statistical approaches to massive code bases [Security16] and to improve program analysis approximations by relaxing logical reasoning with probabilistic models [MAPL17, MLP18] and learning from user feedback [OOPSLA17, PLDI19].

Research and Industrial Impact

My research has resulted in a number of popular open-source tools, notably the deep reinforcement learning-based loop invariant generation framework Code2Inv, the APiSAN framework for API specification mining, and the Drake program analysis framework for continuous integration. Additionally, Code2Inv was highlighted as a spotlight paper (top 3%), APiSAN was nominated as a finalist for CSAW Best Applied Research Paper Award, and Drake won the ACM SIGPLAN Distinguished Paper Award. Companies and the open-source community benefited from and expressed interests in our work — Code2Inv won a Facebook research award, APiSAN discovered 76 previously unknown bugs in the Linux kernel and OpenSSL library, and Drake is being transitioned to the GitHub platform.

Future Research

I envision that future program reasoning techniques and underlying constraint solving techniques (e.g. SAT and SMT) will be equipped with a learning component, which can be automatically and continuously improved. My research aims to realize this vision and thereby make these techniques generally applicable to many domains. In one of my ongoing works, I am investigating how to introduce learning-based techniques into state-of-the-art software verification and testing frameworks such as StAHorN and KLEE. Also, I am interested in a number of machine learning problems in the context of improving software quality, such as reinforcement learning, transfer learning, representation learning of highly structured data, data-efficient learning, and neuro-symbolic reasoning.
Research Contributions

I. Combining Statistical and Logical Methods in Program Reasoning

Real-world programs are large and complex, which involves many third party libraries that may lack specification or even source code, and new features or patches are being introduced regularly. To address these complexities and uncertainties, my research has investigated how to use statistical approaches to learn API specifications from massive code bases [Security16], and how to improve program analysis approximations by learning from user feedback [OOPSLA17, PLDI19].

Automatically learning API specifications from “Big Code”. One crucial but often ignored problem in program analysis is the issue of missing specifications. A specification describes the expected behavior of a piece of code, without which it is impossible to reason about the correctness or bugginess of the given code. My research [Security16] has targeted learning specifications of APIs that are used in complex software systems such as the Linux kernel, OpenSSL, PHP and Python. The key observation is that majority usage patterns of an API suggest its normal behaviors, and usage patterns can be refined by including nearby contexts. This idea makes it feasible to infer the specification of an API without access to its source code or binary.

Improving analysis accuracy by learning from user feedback. Due to undecidability and many practical issues, a sound static analysis produces false alarms that are unavoidable by the analysis designer. We observe that the analysis user, who usually implements the test program, has much more information than the analysis designer. Incorporating feedback from the user opens up a new dimension to improve the analysis. Indeed, my research [OOPSLA17] shows that interactively asking the user a few yes/no questions suffices for resolving most false alarms. But some special care is needed as noisy feedback could misguide the analysis. My research [MAPL17, MLP18] has combined probabilistic reasoning approaches, particularly Markov logic networks and Bayesian networks, with rule-based logical reasoning approaches so that noise introduced occasionally can be tolerated.

My research has also developed a differential analysis technique [PLDI19], an extension of Bayesian network, that is particularly suited to continuous integration so that the analysis can adapt to a specific user's interests and only recommend truly relevant alarms. This is critical for an analysis framework to be usable in practice, because in the real world a user contributes to a small fraction of a large software system and only cares about whether his or her immediate commit introduces any new alarms.

II. Scalable Inference and Rule Learning

As my research has shown, potentially noisy user feedback can be incorporated into static analysis by first reducing the analysis to Markov logic network and then performing Maximum A Posteriori probability estimation, or MAP inference. My follow-up work improves this line of research in two fundamental ways. First, I built a scalable inference engine [CP16, CAV17] based on the observation that MAP inference is essentially a MaxSAT solving problem. Secondly, I have developed a logic program synthesis framework [FSE18, IJCAI19], which applies to many domains besides program analysis such as big-data analytics and software-defined networks.

Scalable inference via systematic constraint solving. The MAP inference problem for Markov logic network can be solved by grounding logical rules into a set of weighted constraints, which is then solved by an off-the-shelf MaxSAT solver. However, naively doing so is extremely inefficient. My research improves each phase of the inference significantly. From the perspective of the application, constraints do not have to be solved entirely, which allows us to have a combined top-down (exploits laziness) and bottom-up (exploits eagerness) grounding [CAV17]. From the perspective of the underlying solver, which is invoked many times instead of only once, incremental solving [CP16] becomes feasible and efficient.

Synthesizing rich declarative logic programs. Declarative logic programs, in particular Datalog programs, have witnessed promising applications in various domains due to its rich expressivity, which also makes synthesizing them a fairly challenging task. Prior work synthesizes Datalog programs following a set of pre-defined templates and may not guarantee soundness. My research [FSE18] eliminates this limitation by a systematic template augmentation process, which enables synthesis of arbitrary rules but without significantly enlarging the search space. Furthermore, our work synergistically combines a bi-directional search over version space with active learning so that efficient search is achieved and the number of required examples is minimized.
Scalable synthesis via numerical relaxation. Inspired by the fact that numerical relaxation techniques are widely used and remarkably successful in optimization problems, my recent work [IJCAI19] proposed a numerical relaxation technique for learning logic programs, which is fundamentally different from traditional search based approaches. The key idea is to associate candidate logical rules with random weights and then carefully design the semantics on propagating weights through logical rules to derived tuples, which reduces the original synthesis objective into a numerical optimization objective. Compared to the previous state-of-the-art, this work achieves up to three orders of magnitude improvement, e.g. reducing the synthesis time from half an hour to one second.

III. Deep Learning for Program Reasoning

Deep learning has sparked a remarkable revolution in many challenging fields of science and engineering. Arguably, the most exciting advancement is that deep learning combined with reinforcement learning (a.k.a deep reinforcement learning) significantly outperforms human experts on video and board games by directly learning from raw pixels and self-play. However, can this technique help to solve programming challenges? Specifically, can analysis, verification, and synthesis be automatically learned by neural networks without any human intervention? My doctoral research [NeurIPS18, ICLR19] aims to answer this question, which needs to address two fundamental AI challenges: 1) representation learning of structured data with rich semantics, and 2) the interplay between neural networks and symbolic reasoning (a.k.a neuro-symbolic reasoning).

Program verification by deep reinforcement learning. Program verification concerns formally validating properties of a program so that it will never crash once verified. The central challenge is to infer a strong enough loop invariant for a given loop (or recursion). This problem is undecidable in general, and specialized heuristics are needed in each domain from experts, which are usually lacking. To eliminate the necessity of strong expertise as well as laborious heuristic design process, my research [NeurIPS18] proposed an end-to-end learning framework, Code2Inv, which takes a piece of code as input, then automatically learns and improves a neural policy by interacting with a proof checker, and finally produces an invariant so that the verification can be done successfully.

In particular, I first proposed a novel representation learning technique for programs so that the semantics (rather than syntax) can be captured by neural networks, which leverages classic techniques in compilers (e.g. control and data flow analysis and static single assignment transformation) as well as recent advances in deep learning (e.g. graph neural networks, attention mechanism, and neural network with external memory). Then, I proposed a novel reinforcement learning technique for loop invariant generation. Surprisingly, both the neural representation and the invariant generation policy can be learned jointly from feedback given by a checker, no human intervention is needed except for a checker informing whether a generated invariant is desired or not.

Vanilla reinforcement learning cannot work on the verification task due to extremely sparse rewards, since the checker only provides a YES/NO feedback, and even worse, the checker almost always provides a NO feedback given that a YES feedback triggers the termination of verification. To address this issue, I proposed two novel reward mechanisms — a continuous reward mechanism, based on the observation that there is a counterexample associated with each NO feedback, and an early reward mechanism, which avoids trivial or useless predicates in a loop invariant as early as possible so that learning could be more sample efficient. These two reward mechanisms could serve as helpful examples for inspiring new insights on improving sample efficiency of reinforcement learning in general.

Program synthesis by deep reinforcement learning. Both representation learning techniques and reinforcement learning techniques I proposed in the Code2Inv [NeurIPS18] work are loosely coupled with the invariant checker. Except for a weak learning signal from the underlying checker, these learning techniques are completely independent from the checker, which can check arbitrary safety properties the user is interested in. What these learning techniques really capture are a neural representation of some structured data and a neural policy mapping the learned representation to another structured data. My recent work [ICLR19] adapts these learning techniques to a much broader domain, namely syntax guided program synthesis (SyGuS), where each synthesis task has its own domain specific language, making manually designing heuristics for each task prohibitive. This also makes learning across different tasks extremely difficult as generalizing across different languages (i.e. learn from one language and then test on a different language) is near the limit of machine learning and has been rarely explored so far. This recent work is an attempt in this meta-learning direction and demonstrates quite promising results — the learned neural representation as well as policy can be transferred to (i.e. help to accelerate) synthesis tasks that use similar but different grammars.
Future Directions

My doctoral research has focused on improving software quality by static analysis, verification, and synthesis via cutting-edge AI techniques ranging from symbolic reasoning, constraint solving, statistical and probabilistic models, to recent approaches for deep learning. Looking forward, I envision that future program reasoning techniques and the underlying constraint solving techniques (e.g. SAT and SMT) will be equipped with a learning component, which can be automatically and continuously improved. I briefly outline a few directions towards achieving this vision.

**Learning-aided reasoning.** Machine learning models, especially deep neural networks, are good at capturing patterns that are hard to describe in a formal way and implement in code, which is not only useful for perceptual tasks like vision and speech recognition, but also provides a new dimension on heuristics design. As my recent research shows it greatly helps to solve sophisticated logical reasoning tasks. This suggests a new insight on solving challenging tasks — instead of designing heuristics, design basic building blocks of heuristics, and introduce a learnable component into the original algorithm or system, which constructs effective heuristics through training. This insight applies to a broad range of problems from low-level constraint solving such as SAT solving, SMT solving, and combinatorial optimization to high-level program reasoning and testing such as software model checking, automated theorem proving, symbolic execution, fuzzing, and concolic testing.

I am very excited about exploring problems mentioned above. My recent internship at DeepMind involved how energy based models help SAT solving. I am also investigating how to introduce learning-based techniques into state-of-the-art software verification and testing frameworks such as SeaHorn and KLEE.

Furthermore, the underlying driving ideas of machine learning such as gradient-based approaches, numerical relaxation, and stochastic optimization are extremely effective on solving unexpected tasks like logical rule learning as demonstrated by my recent work [IJCAI19]. Similar successes have been recently observed in software fuzzing as well. Many new applications of these techniques could be exciting to explore in future.

**Learning from “small data”**. The great success of machine learning, especially deep learning, is due to the availability of enormous computational resources and training data; however, this could be the very reason hindering its further successes, since high-quality training data is usually lacking and expensive to collect, especially for program reasoning tasks. My verification and synthesis work [NeurIPS18, ICLR19] suggest reinforcement learning could be an effective approach to overcoming the issue of lacking data. However, this work is still data inefficient as the checker could provide more structural information that is not utilized by the current framework. New models are good at handling raw pixels or sequences of tokens. How to data-efficiently learn from highly structural data with rich semantics is another research direction I would like to pursue.

**Learning to build secure and correct system.** With the idea of correctness by construction, many system software like OS kernel, file system, and compiler have been fully verified, which is a remarkable achievement but requires developers to manually write machine checkable proofs, which is tedious and requires acrobatic skills. Ideally, these proofs should be automated, which creates unique challenges for machine learning. In particular, how to learn a good representation of known axioms and lemmas from past proofs so that they can effectively re-used in future? Also, how to abstract, modularize and invent new lemmas during seeking a valid proof? Graph neural network is a good starting point, but many innovations are presumably needed in order to learn layers of hierarchical abstractions. Solving these challenges will revolutionize the way we build provably correct systems.

**Interpretable learning and algorithm discovery.** The main concerns of deep learning are its generalizability and interpretability. I am interested in exploring solutions to these issues via an interesting problem called algorithm discovery, which synthesizes efficient algorithms by distilling deep learning models into interpretable programs. The insight is that, though deep learning model is not interpretable, it can provide interpretable and efficient execution traces after trained for a given task, and those execution traces are valuable demonstrations of some unknown algorithm to be learned. Once the algorithm is recovered from execution traces, the interpretability and generalizability will be guaranteed. This is an ongoing joint work with DeepMind that I am excited to explore further. Besides, I am working on setting up a benchmark suite, DATALOGBENCH, for interpretable rule learning tasks, which is expected to foster more future research in this direction.
References