Research Statement

He Zhu

Despite decades of investigation, software systems remain vulnerable to code defects. In security-critical software systems, security bugs can enable malicious users to bypass access controls in order to obtain unauthorized privileges. For instance, the Heartbleed bug in the OpenSSL cryptography library exposed a large number of private keys to the Internet because of inadequate checks in the code. In safety-critical autonomous systems, vulnerabilities can have life-impacting consequences. For example, there have been ongoing reports of safety incidents when using machine-learning enabled surgical robots that negatively impact surgery outcomes by causing procedure interruptions [1]. A central reason why software systems cannot be constructed with the same degree of reliability as other engineered artifacts is the lack of automated tools that can offload the burden of correctness verification. Using existing verification methodologies is a time-intensive and intellectually challenging process, that often requires significant manual involvement to engineer correctness proofs.

Research Highlights

My research has addressed the aforementioned conundrum by exploring **data-driven** techniques to enable scalable automated **verification of software systems including machine-learning enabled autonomous systems**. My work has focused on (a) validating complex data structures properties, (b) discovering sound invariants of complex control-flow processes (*i.e.,* loops and recursion), and (c) reasoning over new computational models (*i.e.,* machine learning models) with complex non-linear behavior. My main approach consists in discovering high-level *abstract specifications* capturing the programmer's intent (*i.e.,* reachability specifications in the case of data structures; inductive invariants in the case of loops and recursion; and, deterministic programs in the case of machine learning models) to enable efficient verification that would be difficult or impossible otherwise. The key technical insight is the application of data-driven techniques to a formal methods toolchain to guide the automated discovery of high-level abstract specifications from code.

I briefly outline each of these three contributions below, with further elaboration in the following sections.

Modern-day software typically leverage libraries that define complex data structures (*e.g.,* trees and heaps); these libraries impose sophisticated requirements on their correct use. A particularly important question when using these structures deals with *reachability*, *e.g.,* does a tree traversal function reach the elements of a tree in a specific order? Questions such as this have proven difficult for traditional verification methods to handle but are critical to answer to ensure that clients use these libraries correctly. I have developed an expressive verification system, called DOrder [2, 3], that enriches simple type systems with first-order logic correctness properties that can precisely describe a variety of data structure specifications including reachability properties. Importantly, DOrder can automatically learn the most precise reachability specification for a data structure function, in a data-driven manner, from a finite number of input-output examples. DOrder's type system enforces correctness specifications at compile time, thereby nipping in the bud a large swath of code defects. DOrder supports modern language features such as higher-order control flow, side-effects and concurrency.

The central challenge to effectively reason about a computation's complex control-flow behavior is finding abstract specifications as inductive invariants that characterize the deep purpose of loops and recursion in a program. (An inductive invariant is an invariant whose structure is amenable to automated checking and validation.) Identifying such invariants automatically has proven to be tremendously difficult. To tackle this issue, I have developed an automated inference tool, called SynthHorn [4], an innovative scheme that allows data-driven methods (*e.g.,* machine learning algorithms) to interact with a verification engine to synthesize sophisticated invariants for real-world programs. The technique works by learning potential invariants from examples, refining a candidate invariant if the underlying verification engine discovers additional counterexamples to the invariant's validity. My research has demonstrated that such an automated framework can be used to efficiently learn unrestricted inductive invariants for large-size real-world `C` programs. The significance of this result was acknowledged by a PLDI 2018 Distinguished Paper award.

As a continued exploration of the data-driven inference theme found in DOrder and SynthHorn, I have recently leveraged data-driven methods for realizing trustworthy machine-learning enabled autonomous cyber-physical systems. SynthML [5] is a verification toolchain that synthesizes abstract specifications of complex machine learning models (*e.g.,* neural networks). SynthML treats a neural network as a black box and samples environment states that can be encountered by the neural network, synthesizing a deterministic program that closely approximates the behavior of the neural network on these states and that importantly satisfies desired safety constraints. Thanks to this abstraction mechanism, formal methods techniques typically used for traditional software systems can be leveraged for safety verification of neural network systems. SynthML is capable of guaranteeing safety properties of a neural network controller, even when it is deployed in unanticipated or previously unobserved environments.

In the following sections, I elaborate on these efforts and present some of my ongoing ideas for future work.

Data Structure Verification

My research has explored ways to better integrate language and analysis design to help make formal methods tools more effective. I developed DOrder [2, 3], a data-driven software verifier for OCaml programs that takes as input OCaml source code and checks whether the code satisfies sophisticated correctness specifications. In DOrder, such specifications are naturally added to OCaml's type system as *refinement types*, *i.e.,* types annotated with logical predicates. Given only refinement type specifications for top-level functions, DOrder can automatically synthesize refinement types for all intermediate terms [3]. DOrder is a useful software verifier because it supports programs with polymorphic data types and higher-order functions [6]. DOrder [7] can conservatively approximate **side-effects** that an expression may produce (*e.g.,* array in-place updates), yielding a sound flow-sensitive refinement type system. Combining side-effects and data structures, DOrder can also reason about linearizability, the de facto correctness condition for **concurrent data structures** [8].

DOrder is capable of proving useful invariant-based specifications of sophisticated data structure functions [2]. It solves a particularly challenging exercise for data structure analyses, *i.e.,* **reachability**. DOrder accurately reasons about sophisticated reachability specifications that relate the shape of a data structure (*e.g.,* a binary tree) with the values contained therein (*e.g.,* the in-order relation of the elements of a binary tree). Atomic reachability predicates (*e.g.,* a tree node can be reached by another tree node following its left-child field) are automatically extracted from the inductive type definition of a data structure. For example, DOrder can type-check that a balanced binary search tree insert function can preserve in its output tree, the in-order relation between the tree nodes of an input tree, formalized as a first-order logic specification given in a refinement type. DOrder's verification procedure preserves SMT-based decidability, because it encodes reachability relations using the decidable effectively propositional logic (with first-order axiomatizations of transitive closures to bound the shape of tree-like data structures). With this formalism, DOrder can describe modular properties of data structures and can reason about challenging invariants such as *sortedness and uniqueness of lists* or preservation of *red-black invariants or heap properties* on trees.

DOrder also enabled **Automatic Verification** by using a data-driven inference procedure that automatically discovers reachability specifications from code. Like a type system synthesizing type signatures from a set of basic and user-defined data types, the heart of DOrder is an abstract specification learning algorithm [2] that can effectively learn specifications from any hypothesis domain of atomic predicates given by the programmer. If the hypothesis domain is chosen as the set of reachability predicates, DOrder uses a counterexample guided guess-check loop, to synthesize (and verify) the most precise reachability specification of a data structure function from a finite number of input-output examples of the function. In [9], DOrder is extended to synthesize heap specifications in separation logic for `C` programs using neural network learning.

Real-world Data Structures have been verified using DOrder. We proved functional correctness of several libraries including the OCaml extensible array library in which DOrder found two real bugs, a Redblack tree library and the full implementation of OCaml's Set library.

Verification via Machine Learning

Despite its high level of automation, DOrder is limited by the programmer's choice of a finite hypothesis domain and thus cannot be applied to extract specifications from infinite domains (*e.g.,* arithmetic that is important for many security verification tasks). To reason about a computation's complex control-flow

behavior, my research has also explored the applicability of machine learning to discover high-quality invariant-based specifications from unbounded hypothesis domains (*e.g.,* the Polyhedra abstract domain). This turns out to be very challenging. Consider the verification of a loop program. A machine learning algorithm should identify a loop invariant by learning a classifier that separates safe program states and unsafe states (safety properties are violated by executions from such states) sampled at the loop head. However, machine learning algorithms strive to avoid over-fitting concepts to training data by relaxing the requirement that a proposed concept must be fully consistent with given samples. Verification tasks, on the other hand, aim to ensure that a concept is correct, even if this comes at the expense of generality. This tension between generality, central to machine learning, and safety, central to verification, is a primary obstacle to seamlessly adapting existing machine learning frameworks to solve general verification problems.

To address this challenge, I have proposed SynthHorn [4], a machine-learning based verification framework for programs containing unconstrained loops and recursion. SynthHorn does not strictly require a machine learning algorithm, *e.g.,* a linear classification algorithm, to always produce a perfect classifier. Instead, it exploits an off-the-shelf classification algorithm that may return a classifier which misclassifies either safe states or unsafe states or both for the sake of generalization. To recover the loss of precision, SynthHorn applies the classification algorithm iteratively on misclassified samples, learning a family of classifiers that separate all samples correctly in the aggregate; such classifiers are conceptually logically connected with appropriate boolean operators $\wedge$ and $\vee$. SynthML frames the invariant inference procedure in terms of a *counterexample guided abstract refinement* (CEGAR) loop that progressively samples representative safe and unsafe states from unproven verification conditions, which are fed to machine learning for discovering more qualified invariants.

Real-world Applications have been verified using SynthHorn, including the verification of critical safety properties in large `C` programs collected from the literature and SV-COMP benchmarks [10]. It has proven efficient in verifying program safety and effective in generating counterexamples when a program is unsafe.

Verification for Machine Learning

Beyond being effective in providing abstract specifications for loops and recursion, the data-driven automated reasoning theme can be adapted to abstractly specify machine learning models and realize fully trustworthy machine learning systems. My research has addressed two major challenges that has heretofore hindered the wider adoption of useful artificial intelligence in autonomous systems: (1) the lack of strong safety guarantees that can be made about machine learning model behavior and (2) the poor transferability of a machine learning model to unseen environments or environment changes that deviate from assumptions made at training-time.

To address these shortcomings, I have developed SynthML [5], a verification framework based on *program synthesis*, a powerful technique in programming languages research, whose task is to automatically construct a program that satisfies a given high-level specification. The core idea is to synthesize a simpler and more interpretable deterministic program that approximates a neural network controller (called an *oracle*). SynthML encodes the neural network's runtime environment as an infinite state transition system and is equipped with a verification procedure that guarantees control actions proposed by the synthesized program always lead to states consistent with an inductive invariant of the state transition system and a given safety specification. This invariant separates all reachable (safe) and unreachable (unsafe) states expressible in the state transition system. Rather than repairing the neural network directly to satisfy the constraints governing the synthesized program, the program and its invariant form a safety shield that operates in tandem with the high performing neural network controller, overriding network-proposed actions whenever such actions can be shown to lead to a state that violates the invariant. To avoid unnecessary shield interventions, the synthesis procedure is guided by a quantitative objective such that a synthesized program bears reasonably close resemblance to its oracle. To ensure safety, SynthML frames the synthesis procedure in terms of a *counterexample guided inductive synthesis* (CEGIS) loop that eliminates any counterexamples to safety in a synthesized program.

SynthML is critical to ensuring safety when a neural network is deployed in unanticipated or previously unobserved environments different from the one used during training, *e.g.,* the mass of its controlled object changes. To incorporate the additional constraints defined by the new environment, SynthML attempts to synthesize a new deterministic program guided by the existing network, a task that is substantially easier to achieve than training a new neural network. SynthML simply builds a new shield that is composed of the new program and the safety boundary captured by the inductive invariant of this program. The shield can ensure the safety of the existing neural network in the new environment, despite the fact that the neural network was

trained in a different environment context. Experimental results over a wide range of cyber-physical systems demonstrated that it can be used to realize trustworthy machine learning systems with low overhead [5].

Future Directions

While my research results thus far provide evidence on the power of data-driven methods, there are still many open questions and directions to pursue. The investigation of these issues forms my future research agenda.

Statistical Program Analysis. In future work, I aim to improve the ability and scalability of data-driven program verification to solve important verification tasks that have confounded existing techniques. Statistical program analysis is a new program analysis paradigm that leverages probabilistic models of code learned over large code corpora for reasoning about new programs with *statistical* guarantees. The methods are intriguing because of the large amount (many terabytes) of open-source programs that are available for analysis and inspection. As an example, consider how we might verify that a program uses a library API correctly in the absence of any clear specification that dictates proper usage. I intend to exploit data-driven techniques to build automated transformation tools that can represent an API calling context as a single fixed-length dense vector of real-valued numbers, inspired by embedding-based algorithms such as word2vec [11] or code2vec [12]. These vector embeddings can be automatically learned from large codebases by grouping similar API calling contexts in a vector space. The analysis will detect API misuses if an API's calling context in a code location exhibits significantly different vector embedding than that of the API elsewhere.

Statistical machine-learning inspired program analysis methods can be used to build *rigorous* verification tools to prove functional correctness of large-scale systems beyond the capability of current techniques. Over the years, the formal verification community has verified a large set of real-world systems, such as the TLS protocol, with manually supplied specifications as proofs. I plan to develop new specification inference engines trained using reinforcement learning on these verified systems, with the goal of having these engines learn domain knowledge and hidden strategies that capture how a human expert might infer useful specifications and properties from code. Given a new program, the engine would then be used to mimic the reasoning used by human experts thereby assisting formal verification tasks.

Verification of Distributed Systems. Building from these insights, my future research will also look at problems in domains for which automated verification has been considered extremely challenging if not impossible, *e.g.,* verifying correctness of the software stack defining a cloud infrastructure like AWS or Azure. For distributed systems, among other things, correctness proofs must account for network delays and message loss that may result in behaviors that are neither intended nor anticipated. Due to the asynchronous nature of communication, the programmer must supply a verifier with (inductive) invariants that explicitly enumerate possible schedules by splitting cases over the global states of all participants and reason about the contents of unbounded message buffers. Such invariants are complicated and hard to divine. In DOrder, we used machine learning to synthesize reachability invariants of tree-like data structures. The success of DOrder encourages me to explore the feasibility of this solution for reasoning about reachability properties over topologies and message ordering relations in distributed systems so as to efficiently synthesize adequate invariants to prove correctness properties. I plan to develop a decision procedure extending DOrder's reachability logic that can supply counterexamples reflecting configurations (*e.g.,* network failures and message loss) that contradict a predicted specification. The aforementioned learning assisted specification inference engine, trained from expert-provided proofs for sophisticated distributed systems such as the Raft protocol, will be used to automatically learn sophisticated correctness specifications for new systems, such as Azure, from counterexamples.

Verification of Large-Scale Machine Learning Systems. Beyond traditional computer systems, I will push the data-driven abstract specification inference theme to the verification of new computational platforms, especially, large-scale machine learning systems, *e.g.,* learning-enabled autonomous systems and computer version systems. Such systems are increasingly important with the ongoing renaissance of deep neural networks. For achieving scalability, the proposed data-driven verification theme must embrace the nature of machine learning systems, combining the best of search-based verification approaches (*i.e.,* symbolic proof search) with optimization-based methods from the machine learning community (*e.g.,* gradient guided optimization). For example, search-based approaches can symbolically generate an abstraction that approximates a neural network's behavior, *e.g.,* deriving the lower and upper bound of a neuron's output. Gradient descent optimization methods allow us to learn, from the precision loss between the estimated bounds of the neural network's output and its desired safety range, how to refine the abstraction so that safety-critical properties can be verified.

References

[1] Yohannes Kassahun, Bingbin Yu, Abraham Temesgen Tibebu, Danail Stoyanov, Stamatia Giannarou, Jan Hendrik Metzen, and Emmanuel Vander Poorten. Surgical robotics beyond enhanced dexterity instrumentation: a survey of machine learning techniques and their role in intelligent and autonomous surgical actions. *International journal of computer assisted radiology and surgery*, 2016.

[2] He Zhu, Gustavo Petri, and Suresh Jagannathan. Automatically learning shape specifications. In *PLDI*, 2016.

[3] He Zhu, Aditya V. Nori, and Suresh Jagannathan. Learning refinement types. In *ICFP*, 2015.

[4] He Zhu, Stephen Magill, and Suresh Jagannathan. A data-driven CHC solver. In *PLDI*, 2018.

[5] He Zhu, Zikang Xiong, Stephen Magill, and Suresh Jagannathan. An inductive synthesis framework for verifiable machine learning. In *under double-blind review*, 2019.

[6] He Zhu and Suresh Jagannathan. Compositional and lightweight dependent type inference for ml. In *VMCAI*, 2013.

[7] He Zhu, Aditya V. Nori, and Suresh Jagannathan. Dependent array type inference from tests. In *VMCAI*, 2015.

[8] He Zhu, Gustavo Petri, and Suresh Jagannathan. Poling: SMT aided linearizability proofs. In *CAV*, 2015.

[9] Marc Brockschmidt, Yuxin Chen, Byron Cook, Pushmeet Kohli, Siddharth Krishna, Daniel Tarlow, and He Zhu. Learning to verify the heap. Technical report, Microsoft Research Cambridge, 2016.

[10] SV-COMP. http://sv-comp.sosy-lab.org/2017/, 2017.

[11] Quoc Le and Tomas Mikolov. Distributed representations of sentences and documents. In *ICML*, 2014.

[12] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. In *POPL*, 2019.