Pranav Garg — Research Statement

My research agenda is to build verification technology that helps programmers write reliable, secure, and verified software. In particular, my research focuses on building automatic techniques that significantly lessen the burden on a programmer trying to prove her program secure or correct. The solutions I develop are learning based automatic software verification including machine learning algorithms for learning inductive program invariants, and reverse engineering a set of proof tactics from manual proofs to learn fully automatable *natural proofs*. My research impacts the building of verified software in the realms of software infrastructures and platforms that have many users, whose security and reliability is becoming increasingly important, and which include systems software such as operating systems, device drivers, mobile platforms, cloud infrastructures, and verification against specifications like race-freedom for parallel programs, memory safety and security.

Motivation and Overview of Research

Verified software addresses a very important problem given that the software industry today spends multiple billions dollars every year on software errors playing catch-up to new vulnerabilities, as and when they are discovered. The problem as it stands is intrinsically *hard* and the current state-of-the-art in software verification requires a considerable amount of manual effort on the part of the programmers to write machine-checkable proofs, and this impedes the practical utility of the current technology. The programmer, today, has to manually annotate inductive invariants of the program as well as annotate the program with proof tactics to guide the verification. I have successfully attacked the problem of the annotation burden on the programmer and automated both these steps using *learning*.

- **Machine learning inductive program invariants**. The primary focus of my work has been in building, adapting, and using learning techniques, including scalable machine learning algorithms, to aid program verification. It turns out that existing learning techniques for automatically synthesizing inductive program invariants are broken! In my research, I have proposed a new learning model called *ICE* (that stands for *learning using Implication Counter-examples*) for learning inductive invariants [6]. I have also developed robust, scalable, and true machine learning algorithms for learning these invariants for various concept classes including numerical invariants and quantified linear data-structure invariants [5, 6, 9, 12].
- **Natural Proofs for automatically validating undecidable verification problems**. For automating verification problems that are undecidable, my research philosophy is to reverse engineer, from manual proofs, *natural proof* tactics that are fully automatic and that work well for a large class of similar programs. I have developed natural proofs for the verification of data structures and verified complicated algorithms over most standard data structures like binary-search trees, red-black trees, avl-trees, and binomial heaps [3].
- **Combination of learning invariants and natural proofs in real verification**. Combining learning of invariants with natural proofs is very powerful and leads to complete automation of the verification problem without the programmer having to specify any proof annotations. For instance, augmenting natural proofs for data-structures [3] with a procedure for learning data-structure invariants can completely automate the verification of data-structure intensive systems software, like Android OS, for security [15]. Similarly, it can be used for automatic correction of student submissions for programming assignments in MOOCs. I have developed such a combination for concurrent systems that exhibit asynchronous communication, where natural proofs target explorations with bounded asynchrony and the invariant is simultaneously synthesized using model checking [7, 8]. I verified the responsiveness of Microsoft Windows phone USB driver stack using this combined scheme [7, 8].

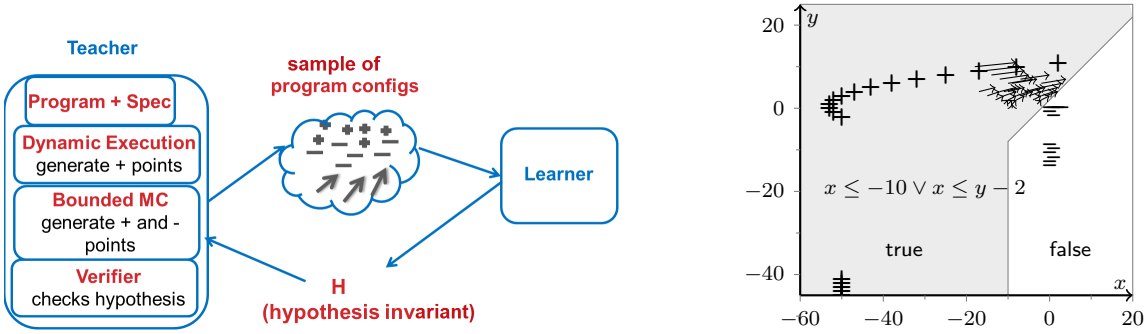
Real-world Applications. My research has paved the way for programmers to develop verified software in the realms of systems software, cloud infrastructures, mobile platforms, etc. In my work on natural proofs for asynchronous programs combined with learning invariants, I verified the full USB driver stack that ships with Microsoft Windows phone [7, 8]. Using natural proofs for data structures [3], I have verified algorithms in data-structure libraries from Linux and OpenBSD kernels, and modules pertaining to memory management in Linux and ExpressOS, which is a secure Android OS developed by my colleagues [15]. In ongoing work, I am applying scalable invariant learning algorithms to the verification of concurrent GPU kernels for race-freedom. Lastly, there is no declarative way today for a programmer to specify rich data-structure properties in her program. In recent work, I have developed an assertion logic for data-structures and a fast procedure for checking these assertions [13].

Taken together, my work makes fundamental contributions to automating verified software engineering using learning and automatable proofs that alleviate the annotation burden for a programmer verifying her program.

Machine Learning Inductive Program Invariants

The problem of synthesizing inductive invariants lies at the heart of automatic software verification. Once the programmer has annotated her program with inductive invariants, the deductive verifier can generate the verification conditions and discharge them *mostly* automatically using provers for various decidable theories. The manual task of annotating these inductive invariants is cumbersome and difficult, and is a huge impediment to the widespread adoption of deductive verification for writing and designing verified software. Synthesizing these invariants using learning is a good approach because machine learning algorithms, in general, are way more scalable than current logical or symbolic techniques [5,6,9,12]. There is another advantage to learning– for more complex programs with complicated language features such as pointers, objects, data-structures, etc., a black-box learning approach for synthesizing invariants is much more easier than constructing new logical mechanisms [5, 6, 9]. *It turns out, however, that current techniques for learning inductive invariants are broken! In my research, I propose a new, robust learning model called ICE for learning inductive program invariants [6].*

ICE: A Robust Framework for Learning Inductive Invariants. The problem of learning invariants can be reduced to learning a Boolean classifier over the configuration space of the program (for instance see Figure 1b). The ICE learning model (that stands for *learning using Implication Counter-examples*) [6] consists of an automatic teacher based on a deductive verifier that checks for the adequacy of the hypothesized invariant and returns a counter-example if the hypothesis is not an adequate inductive invariant of the program (see Figure 1a). However, traditional learning models that offer learning from positive or negative counter-examples do not suffice in this setting of deductive verification. This is because the teacher does not know the precise invariant that is being learned. And so, when the concept hypothesized is not inductive but is consistent with the initial states of the program and excludes the error states, the teacher has no positive or negative counter-example to return to the learner. The ICE model allows the teacher, in such a scenario, to return a pair of counter-examples (x_1, x_2) such that there is an execution of the program from configuration x_1 to x_2, and if the learner classifies x_1 as positive then it should also classify x_2 as positive (and similarly the contraposition). These are called implication counter-examples as the learner can label configurations x_1 and x_2 such that they are constrained by an implication relation (represented in Figure 1a and Figure 1b as an implication arrow in the sample). Note that in the earlier models, the teacher would resolve the implication counter-examples in some arbitrary manner and return them as positive or negative counter-examples to the learner. But this introduced bias in the learning framework that was hard to control, and led to divergence.



(a) (b)

Figure 1: (a) Learning inductive invariants in the ICE model. (b) A visualization of learning from an ICE sample.

The key question while building ICE learners is how to handle implication counter-examples. In [6], I develop a new learning algorithm that uses a constraint solver to determine labels for these implication counter-examples and at the same time learn a hypothesis that is consistent with the sample. I achieve generalization by constructing a hierarchy of concept classes, ordered by their complexity, and learning concepts in a given class only if there exists none in the simpler classes of concepts. This also ensures that the learning algorithm is *convergent* and is guaranteed to find an adequate invariant if there exists one!

Machine Learning Algorithms for ICE Learning. In collaboration with machine learning researchers (Dan Roth), I have developed scalable machine learning algorithms for learning program invariants by adapting decision trees, that use statistical measures such as information gain and shannon entropy as an inductive bias to learn smaller trees, to learn from an ICE sample [12] (the work is under submission to PLDI and the notification is due on February 5, 2015). I have developed several novel information gain measures that the learner uses to

construct a decision tree in the presence of positive, negative and implication counter-examples. The resulting machine learning algorithm is highly scalable, very robust and practically converges with an adequate inductive invariant for a large suite of benchmark programs.

Learning Quantified Invariants over Unbounded Data-structures. Invariants over unbounded data-structures typically require quantification. For example, expressing that an array is sorted involves universally quantifying over two successive array indices and checking that the array values at those indices are ordered. In [6], I extended the ICE model to learn such invariants and showcased new algorithms in this model for learning invariants over linear data-structures. As quantified invariants are typically more complex, learning them directly gets out of hand. I discovered a novel automata model called quantified data automata [4, 5, 9] that are compact representations for universally quantified invariants for arrays and list structures. I further adapted well-known automata learning algorithms– Angluin's L* algorithm and RPNI (regular positive negative inference), to the ICE model and applied them to learn these quantified invariants [5, 6, 9]. The resulting algorithm is convergent and is guaranteed to learn an adequate inductive invariant for array and list programs [6]! The learning algorithms are very scalable because of the chosen compact automata based representation and they break new ground in synthesizing quantified invariants for classes of array or list programs for which current logical techniques do not work.

Concurrency. The ICE learning model works seamlessly for learning inductive invariants for concurrent programs using sequentializations [1]. In ongoing work, I am investigating ICE learning algorithms to check race-freedom in GPU kernels. Preliminary experiments suggests a 5x speedup using more scalable learning algorithms.

My research has paved way for machine learning to be used directly in synthesizing program invariants for verifying software. I show that machine learning algorithms for learning invariants are upto two orders of magnitude more scalable than current symbolic or even prior learning approaches [5, 6, 12]. This is a fundamental advance and puts me in a position to leverage the developed learning algorithms to automate software verification to a significant extent.

Natural Proofs for Undecidable Verification Problems

Logically validating verification conditions for programs is, in general, *undecidable*. Even after annotating program invariants or automatically learning them, a programmer typically has to also annotate her program with proof tactics to guide the verification. Annotating proof tactics is very tedious even for expert users and requires too much knowledge from the programmer about the underlying proof systems or decision procedures. My natural proof philosophy is to reverse engineer from manual proofs, a set of *natural* proof tactics that are fully automatic and that generalize well to a large class of similar programs. I have developed natural poofs for recursive data-structures [3] and concurrent systems exhibiting asynchronous communication [7, 8].

Natural Proofs for Recursive Data-structures. Data-structure verification is extremely important and central to the verification of large, real-world software systems including systems software, cloud infrastructures and mobile platforms. However, verifying data-structure programs is particularly hard. A data-structure contains an unbounded number of heap locations and expressing even simple properties such as the binary-search property for tree data-structures requires quantifiers that makes the validity problem undecidable.

Natural proofs for recursive data-structures [3] is based on the insight that algorithms manipulating these data-structures are typically written recursively (or iteratively), where a function (respectively a loop iteration) locally modifies only a bounded portion of the data-structure, followed by a call to itself to carry out the modifications on the sub-structures. These algorithms are usually proved inductively by assuming the contract on the recursive call modifying the sub-structures, and precisely tracking the modifications by the current function call to prove its contract. I developed Dryad [3], a new logic suited to natural proofs, that allows no explicit quantification but expresses second order quantified data-structure properties using recursive definitions. Natural proofs now involve unfolding these recursive definitions precisely along the footprint as the program manipulates the data-structure, and at the site of a recursive call or a loop iteration it abstracts these definitions followed by strengthening them in accordance with the contracts assumed on the modifications to the sub-structures. Natural proof for data-structures is sound, is fully automatic and works well for most standard data-structure algorithms.

Impact. Natural proofs successfully verify standard data-structure algorithms including algorithms manipulating linked-lists, queues, trees, binary-search trees, avl trees, red-black trees, and binomial heaps [3]. Automatic data-structure verification forms an important cog in the verification of real systems code that is typically very data-structure intensive. I used natural proofs in [3] to verify algorithms in the data-structure libraries in Linux and OpenBSD kernels and modules pertaining to memory management in the Linux kernel and ExpressOS, which is a secure mobile platform cross-compatible with the Android API [3]. Verification of the memory management

in ExpressOS was subsequently used to ensure process isolation for security [15].

Combining Learning of Invariants and Natural Proofs

Combining learning of invariants with natural proofs is very powerful that can lead to complete automation of the verification problem in a real setting. I have developed such verification technology for concurrent systems that exhibit asynchronous communication, which includes co-located event-driven device drivers communicating with the kernel, web applications, or a distributed service in a cloud infrastructure [7, 8].

Learning Invariants and Natural Proofs for Proving Asynchronous Programs Responsive. My work builds up on top of P [14], which is a new domain-specific actor-based language from Microsoft for writing verified device drivers. An important safety specification for P programs is their *responsiveness* that checks that handlers have been written by the programmer for all events that can be triggered by the kernel in a particular state. In general, one might also be interested in checking the reachability of an error state by the program. However it turns out that these problems are undecidable. In fact, current state-of-the-art techniques such as partial order reduction fail miserably in proving asynchronous programs against these specifications [8]!

Natural proofs for asynchronous programs [8] is based on the observation that most asynchronous programs exhibit bounded asynchrony, that is, asynchronously executing components constituting the program are out of sync by only a few message exchanges at a time. I developed a reduction-based proof system that is sound and complete and prioritizes linearizations of interleavings where the components constituting the program communicate *mostly* synchronously. In combination with this natural proof tactic is a model checking procedure that explores these interleavings and constructs an *almost-synchronous invariant* of the program completely automatically. This results in a push-button procedure that is completely automatic and verifies asynchronous programs including event-driven device-drivers and distributed protocols against the responsiveness specification.

Impact. Natural proofs for asynchronous programs enable verified device drivers for the Windows operating system. It assures greater reliability where drivers, which were earlier only model-checked upto a certain depth, are now fully verified using invariants and natural proofs. One of the largest systems I verified using natural proofs was the USB driver that ships with Microsoft Windows phone [7, 8]. I also used natural proofs to verify several standard distributed protocols including the distributed time synchronization service and the German cache coherence protocol [7, 8]. Not only does natural proof for asynchronous programs assure greater reliability with complete verification, it is also upto two orders of magnitude faster than model checking for finding bugs [7, 8].

Bridging the Gap from Testing to Verification

It is important to realize that a sudden shift in emphasis from writing tests to verification is not easy for a software engineer. It is therefore up to us to facilitate this gradual shift from testing to verification. For instance, writing logical specifications might itself be a difficult task for an average programmer, specially if there is no easy way to check these specifications at run-time when they are written in a complex logic or involve quantifiers. In recent work, I have developed such *an assertion checking mechanism* for rich data-structure specifications [13]. Learning *likely* invariants or interface specifications is another interesting problem bridging software engineering and formal verification and my work on learning likely quantified invariants over arrays or list structures falls in this category [5, 9]. In the same spirit of testing and reliable computing, I have prior work on an automatic unit test generation scheme that combines random testing with symbolic execution [2] and an efficient distributed checkpointing system for enhanced reliability of critical computation in an environment with transient errors [11].

Future Directions of Research

My long-term research agenda is to build verification technology that enables building verified and secure software systems with very little manual effort. The problem of software security is attacked today from various directions and I feel that building secure systems using automated verification has its own place in this repertoire. Instead of a full-blown functional verification of the *entire* system, I believe that verification should be targeted towards more important modules or portions of the system and towards security properties that system designers really care about, for example high-level properties like authorization, authentication, integrity, and privacy. Such formal verification helps prove certain crucial parts of the system and formally capture the assumptions on the untrusted and unverified components, which can be checked using dynamic techniques, like [13], at runtime.

An example of such a verified system is ExpressOS [15], a secure mobile platform developed by my colleagues, that protects mobile users against more than 95% of the latest vulnerabilities reported in CVE, with modest performance and verification overheads. I want to build such secure systems in other domains including the browser

and the web infrastructure, the cloud infrastructure, operating systems, etc. By using learning based invariant generation that scales to real software and fully automatic natural proofs, I want to reduce the annotation burden on the programmer verifying such a system. Building these systems and verifying them with less manual effort will also drive my future research towards solving important problems in automated software verification.

Technique: Machine Learning for Reducing Annotation Burden. My research has mainly focused on applications of learning to reduce the annotation burden on the programmer verifying her program, by automatically inferring the required annotations. Machine learning algorithms are way more scalable than current logical or symbolic techniques for synthesizing these invariant annotations and I plan to continue pushing this research direction forward. I plan to adapt more machine learning algorithms for verification, including algorithms to learn linear classifiers such as support vector machines, the perceptron algorithm, the winnow algorithm that learns smaller concepts faster from a large set of attributes, and investigate their trade-offs and novel combinations. I want to understand the connections between ICE learning and semi-supervised learning, and the online mistake-bound learning model and take cues from these learning models to design new ICE algorithms. Additionally, I plan to develop new classifiers for specialized concept classes like the class of heap invariants, classifiers for separation logic, etc., and apply them to verification. All these algorithms will build on the ICE learning model. I am already collaborating with machine learning researchers (Dan Roth) and I would like to forge new collaborations to build these learning algorithms on ICE.

I want to apply the above advances in automatic invariant generation and natural proofs to building secure software systems. I see two complementary research directions for verifying programs for security– (1) verifying programs against generic code-level specifications like memory safety or data-race freedom, and (2) verifying programs against higher-level system-wide specifications like integrity or privacy, and both these domains pose interesting research questions for verification that I plan to investigate.

Application Domain: Software Security at Code-level against Generic Specifications. Source code-level vulnerabilities in the software are commonly exploited today for breaching security and this includes buffer overflows, dangling pointers, data-races in concurrent software, flawed input sanitization routines, etc. Since these specifications are not particular to a given program but are very generic, the programmer does not need to spend any effort in specifying them. However the programmer does need to annotate invariants that prove safety from these vulnerabilities. Static analysis tools today are either not precise in the absence of such annotations or require the programmer to write these annotations (this includes safe variants of unsafe languages that embrace more complicated type annotations). I plan to apply learning based invariant generation to learn these annotations and build precise, scalable and automatic tools for detecting these vulnerabilities. I am particularly interested in the memory-safety specification for programs; given the large amount of work on annotating programs for memory-safety, I believe that learning algorithms can be successfully applied to learn *most* annotations automatically. I am also currently investigating new learning algorithms for proving data-race freedom in the context of GPU kernels where preliminary experiments suggest a 5x speedup using more scalable learning algorithms.

Application Domain: System-level Security against Higher-level Specifications. System designers, typically, have a very high-level security specification that they want to assure, like integrity or privacy. While verifying systems against these high-level specifications, significant manual effort goes into refining these specifications to the level of modules in the system. Even formally stating these, now low-level, specifications often require the programmer to expose additional *auxiliary* state in the program, and then write auxiliary code annotations throughout that maintains this auxiliary state as the program executes. For instance, to check if an operating system ensures data-integrity for an application running on it, the programmer can introduce an *auxiliary* state that tracks the application that "owns" a given file, and check ownership whenever there is any modification to this file. Manually writing these auxiliary code annotations are very error-prone and cumbersome. In the future, I plan to study these refinement maps and automate synthesis of these auxiliary code annotations using program synthesis. I have preliminary work on machine learning algorithms for synthesizing programs from logical specifications [10] and I plan to build on top of this. To understand these refinements in a real context, I am also collaborating with system researchers (Indranil Gupta) to formally specify and verify such high-level specifications for distributed systems. In particular, I am working on verifying the hinted-handoff protocol in distributed key-value stores for eventual consistency. The challenge here is to help users lift code-level verification to high-level and system-wide security guarantees.

Verified software is finally within our reach and I believe we will start building verifiably secure systems routinely in the next few years. I believe that reducing annotation burden and using the powerful logic solvers we have today will lead this revolution, and I look forward to being a major contributor to this endeavor.

References

[1] **Pranav Garg** and P. Madhusudan. Compositionality Entails Sequentializability. In *Proceedings of the 17th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 26–40, 2011.

[2] **Pranav Garg**, Franjo Ivancic, Gogul Balakrishnan, Naoto Maeda, and Aarti Gupta. Feedback-Directed Unit Test Generation for C/C++ using Concolic Execution. In *Proceedings of the 35th International Conference on Software Engineering (ICSE)*, pages 132–141, 2013.

[3] Xiaokang Qiu, **Pranav Garg**, Andrei Stefanescu, and P. Madhusudan. Natural Proofs for Structure, Data, and Separation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 231–242, 2013.

[4] **Pranav Garg**, P. Madhusudan, and Gennaro Parlato. Quantified Data Automata on Skinny Trees: An Abstract Domain for Lists. In *Proceedings of the 20th International Static Analysis Symposium (SAS)*, pages 172–193, 2013.

[5] **Pranav Garg**, Christof Löding, P. Madhusudan, and Daniel Neider. Learning Universally Quantified Invariants of Linear Data Structures. In *Proceedings of the 25th International Conference on Computer Aided Verification (CAV)*, pages 813–829, 2013.

[6] **Pranav Garg**, Christof Löding, P. Madhusudan, and Daniel Neider. ICE: A Robust Framework for Learning Invariants. In *Proceedings of the 26th International Conference on Computer Aided Verification (CAV)*, pages 69–87, 2014.

[7] Ankush Desai, **Pranav Garg**, and P. Madhusudan. A New Reduction for Event-Driven Distributed Programs. In *7th International Workshop on Exploiting Concurrency Efficiently and Correctly (EC2)*, 2014.

[8] Ankush Desai, **Pranav Garg**, and P. Madhusudan. Natural Proofs for Asynchronous Programs using Almost-Synchronous Reductions. In *Proceedings of the 28th ACM International Conference on Object Oriented Programming Systems Languages & Applications (OOPSLA)*, pages 709–725, 2014.

[9] **Pranav Garg**, Christof Löding, P. Madhusudan, and Daniel Neider. Quantified Data Automata for Linear Data Structures. A Register Automaton Model with Applications to Learning Invariants of Programs Manipulating Arrays and Lists. Invited Paper, *Formal Methods in System Design (FMSD)*, 2014. Under Review.

[10] Rajeev Alur, Rastislav Bodik, Eric Dallal, Dana Fisman, **Pranav Garg**, Garvit Juniwal, Hadas Kress-Gazit, P. Madhusudan, Milo M. K. Martin, Mukund Raghothaman, Shamwaditya Saha, Sanjit A. Seshia, Rishabh Singh, Armando Solar-Lezama, Emina Torlak, and Abhishek Udupa. Syntax-Guided Synthesis. Invited Paper, *NATO Proceedings of the Maktoberdorf Summer School*, 2014. Under Review.

[11] Rishi Agarwal, **Pranav Garg**, and Josep Torrellas. Rebound: Scalable Checkpointing for Coherent Shared Memory. In *Proceedings of the 38th International Symposium on Computer Architecture (ISCA)*, pages 153–164, 2011.

[12] **Pranav Garg**, P. Madhusudan, Daniel Neider, and Dan Roth. Learning Invariants Using Decision Trees and Implication Counterexamples. Submitted to PLDI'15.

[13] Alex Gyori, **Pranav Garg**, Edgar Pek, and P. Madhusudan. Asbtraction-guided Runtime Checking of Separation Logic Assertions. Under Submission.

External References

[14] Ankush Desai, Vivek Gupta, Ethan K. Jackson, Shaz Qadeer, Sriram K. Rajamani, and Damien Zufferey. P: Safe Asynchronous Event-Driven Programming. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 321–332, 2013.

[15] Haohui Mai, Edgar Pek, Hui Xue, Samuel Talmadge King, and P. Madhusudan. Verifying Security Invariants in ExpressOS. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 293–304, 2013.