Mayur Naik — Research Statement

Modern computing platforms comprising commercial clouds, multi-core laptops and desktops, and smartphones, tablets, and wearables pose significant challenges to developing high-quality software in aspects such as reliability, security, performance, scalability, and energy efficiency. My research in the area of programming languages and software engineering addresses this problem using a general technique called program analysis that concerns automatically discovering a wide range of useful artifacts about programs—specifications, proofs of correctness properties, witnesses of harmful bugs and vulnerabilities, and resource bounds.

I have proposed foundational techniques for program analysis, including client-driven abstraction refinement (POPL’11; PLDI’11; PLDI’13; PLDI’14a), inferring proofs from tests (OOPSLA’10; POPL’12), scalable inter-procedural analysis (ESOP’14; PLDI’14b; SAS’15), and user-guided program analysis (FSE’15a). I have also applied these techniques to diverse and important software engineering problems including testing and verifying concurrent programs (PLDI’06; POPL’07; PLDI’09a; PLDI’09b; ICSE’09; CAV’09; FSE’10), modeling and predicting program resource usage (NIPS’10; ATC’13), test input generation (FSE’12; FSE’13), statistical debugging and fault localization (POPL’03; PLDI’05; ICML’06), and compiling and optimizing programs on platforms for embedded computing (LCTES’02; ESOP’05), mobile-cloud computing (EuroSys’11; MobiHoc’14), and approximate computing (FSE’15b).

My research draws from, contributes to, and combines ideas from a variety of approaches including static (compile-time) analysis, dynamic (run-time) analysis, programming language design, and machine learning. My research typically involves both theoretical and empirical aspects. I devise techniques, principles, and methodologies that abstract away from the problem at hand, enabling to reason about formal properties and find applications to other unexpected problems. I also build and deploy software artifacts that translate techniques into tools, principles into systems, and methodologies into frameworks, which in turn can be used by people both within and outside my research community.

I next summarize my most significant contributions in reverse-chronological order and outline future directions.

Scalable Adaptive Program Analysis (2011-2015). As software becomes increasingly pervasive and complex, there is a growing need for diverse program analysis-based artifacts, such as vulnerability detectors for security analysts, configuration debuggers for network administrators, and resource-usage predictors for smartphone programmers.

The diversity of such artifacts necessitates adapting the underlying program analyses to client needs, in aspects of scalability, applicability, and accuracy. Today’s program analyses, however, do not provide useful tuning knobs. Thus, building useful artifacts atop current program analyses demands Herculean efforts or specialists with know-how of algorithmic and engineering aspects of program analysis.

I have developed general techniques that automatically adapt program analyses to diverse clients (POPL’11; PLDI’11; POPL’12; PLDI’13; ESOP’14; PLDI’14a; PLDI’14b). My main hypothesis is that it is feasible to cast adaptivity as a large-scale optimization problem, and that this problem can be solved effectively by means of new search algorithms and systems that achieve accuracy and scale by leveraging new kinds of data and large-scale computing resources.

In initial work (POPL’11), we proposed the concept of a minimal abstraction: the coarsest abstraction from a given family of abstractions of a given program that is capable of proving a given assertion in the program. We proposed four algorithms to infer minimal abstractions: deterministic vs. randomized and refinement-based vs. coarsening-based. We also showed that in practice, minimal abstractions are very simple even for large programs, making them suitable for humans to understand and for computers to infer.

The above algorithms are suited for the scientific study of minimal abstractions as opposed to for efficiently inferring them. In (POPL’12), we proposed a practical, combined dynamic/static analysis approach for finding minimal abstractions. The main insight is to efficiently compute from dynamic analysis of the given program, a necessary condition on the abstraction to prove the assertion, and thereby prune the space of abstractions that the static analysis must consider. We showed that this approach is effective in practice, requiring few runtime traces in the dynamic analysis to dramatically prune this space, and prove over 80% of queries posed by a static pointer analysis and a static thread-escape analysis in large Java programs. Previous work had only shown the ability of dynamic analysis to infer likely invariants; our work showed that dynamic analysis can additionally be used to guess the strategy for proving real invariants, and that this guess is correct for the vast majority of cases that arise in practice.

Proving the remaining 20% queries using the above approach needs program inputs that guide the dynamic analysis along desired control-flow paths. Such demand-driven input generation, however, is a well-known hard problem. In subsequent work (PLDI’13), we circumvented this problem and proposed a purely static approach based on counterexample guided abstraction refinement (CEGAR). Our key insight was to generalize, from abstract counterexample traces, the failure of the current abstraction to as many other abstractions in the family as possible, and try the cheapest of the remaining abstractions in the next iteration, until the assertion is proven or all abstractions are eliminated. This approach has two significant benefits over CEGAR approaches in the literature: it can avoid divergence by eliminating
and precise static analysis to find races in large, real-world, multi-threaded programs. They usually occur non-deterministically, under specific thread schedules. My doctoral research developed a sound approach to find races in large, real-world, multi-threaded programs. They usually occur non-deterministically, under specific thread schedules. Our approach lifts this limitation by automatically finding effective abstractions for analyses written in Datalog. Our approach is based on CEGAR: when a Datalog analysis run fails using an abstraction, it seeks to generalize the cause of the failure to other abstractions, and pick a new abstraction that avoids a similar failure. Our solution uses a boolean maximum satisfiability formulation (MaxSAT) that is general, complete, and optimal: it is independent of the Datalog solver, it generalizes the failure of an abstraction to as many other abstractions as possible, and it identifies the cheapest refined abstraction to try next. We demonstrated our approach on a pointer analysis and a typestate analysis, on Java benchmark programs of size 100-350 KLOC each. Our approach could resolve all client queries to these analyses compared to only 50% by the pointer analysis from [PLDI'11], and using 10X fewer CEGAR iterations than the typestate analysis from [PLDI’13].

Automated Testing of Mobile Apps (2011-2013). The changing computing landscape to deliver a more personalized and targeted user experience has led to a surge in applications on mobile devices such as tablets and smartphones. Unlike desktop apps, mobile apps serve a wide range of users in heterogeneous and demanding conditions. As a result, mobile app developers, testers, and ultimately end-users can benefit greatly from what-if analyses of mobile apps. A key hindrance to such analyses is obtaining inputs that adequately exercise app functionality. To this end, I developed new techniques to test mobile apps effectively [FSE’12,FSE’13]. These two papers have been cited 140 times. A popular technique for automatic and systematic input generation is dynamic symbolic execution. In [FSE’12], we proposed the first algorithm based on this technique for mobile apps. The central insight underlying our algorithm is two-fold: first, it views a mobile app as an event-driven program—a program whose input can be viewed as a sequence of events. Second and more significantly, it dynamically identifies subsumption between different event sequences, in order to avoid generating redundant event sequences and thereby alleviate the notorious path-explosion problem that afflicts conventional symbolic execution approaches. We implemented our algorithm for Android, the dominant mobile app platform, and showed that it scales significantly better than conventional approaches on real-world apps.

A key limitation of the above approach, however, is that it can only generate user-interface (UI) events, whereas mobile apps also react to system events—non-UI events such as an incoming SMS message, a request by another app for the device’s audio, or a notification of the device’s battery power running low. In subsequent work [FSE’13], we proposed a generalized approach for generating an interleaved sequence of both kinds of events. The key challenge in generating system events is that they are far more numerous than UI events due to the rich and dynamic environment of mobile apps. The main observation underlying our approach was that, despite this large number of possible events, a mobile app reacts to only a small set of events at any instant, and moreover, this set of events can be computed automatically by monitoring the app’s interaction with its environment. We implemented this approach in a tool Dynodroid and showed that it outperforms two prevalent approaches for testing mobile apps: manual testing and fuzz testing. For 50 open-source Android apps, Dynodroid could cover over 80% of the app code on average per app that humans could cover, and fuzz testing took over 20X more events to achieve peak code coverage on these apps. Dynodroid also found a number of bugs in these apps and in the top 1,000 free apps in the Google Play marketplace.

Reliability of Concurrent Programs (2006-2010). Concurrent programs are ubiquitous. Concurrency is the key to effective responsiveness, resource utilization, and throughput in software we interact with routinely, such as operating systems, web servers, databases, GUI applications, and games. The trend toward concurrent software has accelerated with the shift in the last decade from uniprocessors to multi-core processors. Concurrent programs, however, are significantly harder to write and debug than sequential ones. I developed scalable techniques to specify and check the correctness of large concurrent programs [PLDI’06,POPL’07,ICSE’09,PLDI’09a,FSE’10].

A particularly insidious concurrency programming error is a race: a condition wherein two threads simultaneously access the same memory location and at least one of those accesses is a write. Races are hard to detect and fix as they usually occur non-deterministically, under specific thread schedules. My doctoral research developed a sound and precise static analysis to find races in large, real-world, multi-threaded programs [PLDI’06,POPL’07]. The undecidability of exact race detection entails that any sound race analysis (one guaranteed to detect all races)
must be incomplete (misidentifying some non-races as races). The goal then is to devise a sound analysis that produces few false positives on real-world programs. Until my work, race detectors were predominantly based on dynamic analysis, which is inherently unsound. My key insight was to dissect races into five simpler sub-problems. Given a pair of potentially racing statements, these sub-problems ask questions such as: Can the pair of statements access the same memory location in some execution? Is that memory location accessible by different threads? Can those threads execute the pair of statements without holding a common lock? Each of these is a necessary condition for the pair of statements to be involved in a race; thus, if any of these is proven false then the pair of statements is proven race-free. My static race analysis built on advances in may-alias analysis, which is central to virtually all static analyses, not just those for concurrency. I developed new techniques to further advance the state-of-the-art in may-alias analysis, which answers whether a pair of statements is guarded by a common lock, is particularly hard: it requires must-alias analysis to answers whether a pair of locks is the same in all executions. Little progress has been made on must alias analysis in contrast to the vast literature on may alias analysis. I invented a more tractable analysis called conditional must not alias analysis that, instead of asking “is a pair of locks the same in all executions?” asks “whenever a pair of locks is different, are the memory locations guarded by them also different?” Assuming the pair of locks is different, a pair of threads may acquire them simultaneously, but then the pair of threads will also access distinct memory locations and hence cannot race. I implemented my race detector in a tool named Chord and applied it to several Java programs, many of which are mature and widely used, and found tens to hundreds of harmful races in them. Many of these races were fixed by developers often within a week of reporting. In my largest benchmark, Apache Derby, a popular relational database engine comprising 650 thousand bytecodes, Chord found over 300 distinct concurrency bugs. In contrast, the numerous publications on race detection in the past decade cumulatively reported less than 100 bugs. Chord, which began as part of my doctoral work in 2006, is now a versatile, open-source, program analysis platform for Java that my students and I continue to develop and maintain. It has been downloaded over 7,000 times by researchers world-wide.

I applied a similar approach to static deadlock detection. This work, which won an ACM SIGSOFT Distinguished Paper Award, was driven by the need for developers to ensure that any locks they introduce to eliminate races reported by Chord do not result in deadlocks. Since static analyses can be prone to false positives, we developed the idea of active testing described below, to automatically identify real deadlocks from among those reported by our static deadlock detector. In later work, we extended our techniques to not only detect deadlocks involving locks (resource deadlocks) but also those involving condition variables (communication deadlocks), thereby yielding a comprehensive solution for finding deadlocks.

I believe that effectively improving the reliability of concurrent programs requires augmenting program analysis with programming language extensions and restrictions. My work on Shoal is a step in this direction: it allows C programmers to specify, via type annotations, data sharing rules (read-only, protected by a lock, etc.), and enforces them using a combination of static and dynamic checks. We showed the practicality of Shoal on a wide range of concurrent C programs, the largest approaching a million lines of code.

**Debugging and Fault Localization (2003-2009).** Debugging is one of the most expensive activities in software engineering. Once a defect is identified, reproducing it, identifying its cause, and fixing it is highly tedious and error-prone. I developed new techniques that address this problem in three important and challenging contexts: software model checking, post-deployment software monitoring, and concurrent programming. I implemented these algorithms in industrial-strength tools (the SLAM software model checker, the Cooperative Bug Isolation framework, and the CalFuzzer concurrency testing framework) and applied them to identify significant crashing bugs in widely-used programs. These works also spawned new research areas, as evidenced by nearly 1,150 citations collectively to the above five papers.

Prior to my work, even state-of-the-art model checkers provided a poor user experience by reporting at most one lengthy error trace. My key insight was to formalize the notion of the cause of an error manifested in an error trace, which may itself be viewed as a symptom of the error. Once the notion of an error cause is formalized, generating multiple error traces amounted to generating error traces with distinct error causes; moreover, the error causes helped localize users’ inspection efforts to a few lines of code, even in traces with hundreds of statements.

In subsequent work, I formalized a similar problem but in the very different setting of post-deployment software monitoring: we proposed a statistical debugging algorithm that isolates bugs in programs containing multiple undiagnosed bugs. Earlier statistical algorithms that focus solely on identifying predictors that correlate
with program failure perform poorly when there are multiple bugs. Our technique separates the effects of different bugs and identifies predictors associated with individual bugs. These predictors reveal both the circumstances under which bugs occur and the frequencies of failure modes, allowing to prioritize debugging efforts.

In later work [PLDI'06, CAV'09], my colleagues and I introduced the idea of active testing to effectively debug concurrent programs. Active testing comprises two phases. It first uses off-the-shelf program analyses (e.g., PLDI'06, CSE'09) to identify potential concurrency bugs. In the second phase, it uses the reports from these analyses to explicitly control the underlying scheduler of the concurrent program to discover real concurrency bugs, if any, with high probability and low overhead. We demonstrated the effectiveness of active testing by applying it to a number of large multi-threaded Java programs and finding tens of previously known and unknown concurrency bugs in them.

Ongoing and Future Directions. I intend to pursue the following directions that I embarked upon recently.

1. Program Analysis for Big-Code. Most software development today leverages the world’s massive collection of open source software. There is significant room for program analyses to similarly leverage Big Code, the collective knowledge amassed from analyzing existing programs, to automatically infer or predict salient behaviors and vulnerabilities in new programs. As part of the DARPA program on Mining and Understanding Software Enclaves (MUSE), I am developing Petablox, a framework for automatically synthesizing use-cases of arbitrary declarative program analyses for Big Code tasks such as efficiently finding good abstractions, transferring analysis results across programs, and adapting analyses to user feedback. Despite their diversity, all these tasks entail solving large instances of MaxSAT, the maximum satisfiability problem which comprises a mix of hard (inviolable, logical) constraints and soft (violable, probabilistic) constraints. I plan to explore demand-driven, compositional, and learning-based MaxSAT optimizations in Petablox for scaling these tasks to large code bases.

2. User-Guided Program Analysis. Program analysis tools often produce undesirable output due to various approximations. In recent work [FSE'15], we proposed an approach and a system Eugene that allows user feedback to guide such approximations towards producing the desired output. We formulated the problem of user-guided program analysis in terms of solving a combination of hard rules and soft rules: hard rules capture soundness while soft rules capture degrees of approximations and preferences of users. Our technique solves the rules using an off-the-shelf MaxSAT solver in a manner that is sound (satisfies all hard rules), optimal (maximally satisfies soft rules), and scales to real-world analyses and programs. We evaluated Eugene on two different analyses with labeled output on a suite of seven Java programs of size 130–200 KLOC. Our experiments showed that Eugene reduced over 90% of misclassified reports upon providing limited amounts of feedback. I plan to investigate richer user feedback models, the theory of feedback generalization, and scalable MaxSAT solving to further advance user-guided program analysis.

3. Mobile-Cloud Computing. Building upon my existing research described above on analyzing concurrent programs, adapting program analyses to heterogeneous clients, and testing mobile apps, I intend to investigate new programming models, analyses, and runtimes for mobile-cloud computing—a new computing paradigm that combines mobile computing and cloud computing. As desktop computing continues to fade away and these two paradigms become increasingly mainstream, there is a dire need to bridge the gap between the “very small” mobile devices and the “very large” cloud. Mobile-cloud computing seeks to address this problem by seamlessly offloading compute-intensive portions of rich apps on resource-constrained mobile devices to the cloud. My preliminary work on this topic [EuroSys'11], cited over 250 times, is credited with pioneering this paradigm. However, much research remains to be done to make this paradigm viable for a wide variety of mobile apps, heterogeneous computing devices, and diverse network scenarios that occur in the real world.

Teaching Statement

I find teaching and advising a very fulfilling experience. Instructing students so that they not only grasp the material taught but also continue to learn more advanced material, or apply and extend it effectively on their own, gives me a deep sense of satisfaction.

I have taught three courses at Georgia Tech: graduate program analysis, undergraduate databases, and undergraduate compilers. I mentored six doctoral students as a research scientist at Intel Labs (2008-2011), and supervised four doctoral students, two Masters students, and an undergrad at Georgia Tech (2011-2015).

Teaching Approach and Philosophy. Most areas in computer science are evolving rapidly. This is especially true of programming languages and software engineering, which is broadly my area of research: new programming languages (e.g., dynamically-typed scripting languages like JavaScript and Python), models (e.g., map-reduce for cloud computing), and platforms (e.g., Android for mobile computing) are widely used today, enabling programmers to productively build and deploy software artifacts that accomplish tasks more sophisticated than ever before. I believe the computer science curriculum at universities should evolve at the same pace. While advanced graduate courses and
seminars are primarily devoted to emerging topics, I believe even an undergraduate course should include such topics. Not only does this enable undergraduate students to become well-informed engineers but it also kindles their interest in pursuing research and advancing the state-of-the-art. A significant fraction of all courses I have taught to date is devoted to topics on cutting-edge research in their respective areas. For instance, in my graduate program analysis course, I taught advanced software testing, debugging, and verification techniques for which no textbooks exist.

I believe in a style of teaching that has a mix of practice and theory. Not only does such a style cater to different kinds of students—those who are inclined to theory and those who understand the material better with implementation—but it also enables students of either kind to better appreciate the proficiency of the other. Such an approach is especially important in teaching courses in programming languages and software engineering. For instance, program analysis algorithms that have elegant theoretical properties can be ineffective when implemented and applied to real programs. At the same time, inculcating a solid theoretical basis in students is necessary. Formalizing a program analysis algorithm on a core language calculus as opposed to implementing it for a full-fledged language enables one to abstract away engineering details and reason about the algorithm’s correctness, complexity, limitations, and relationship to other algorithms for the same problem, or even find its applications to other unexpected problems.

To enable theoretically inclined students appreciate the benefits of implementation, I provide real-world software systems for students to use or extend in assignments. For instance, in my graduate program analysis course, I created assignments from scratch that gave students hands-on experience with state-of-the-art program analysis tools. My own experience in building real-world software systems has greatly facilitated this task. At the same time, I strive to inculcate formal concepts in students who are more inclined to experimentation. I use symbolic and intuitive notions to explain formal concepts and emphasize insights that would not be evident in an informal setting. For instance, in the undergraduate databases course, where almost 80% of the students were non-CS majors, I used such notions to explain theoretical concepts such as functional dependencies and normalization in database design.

**Outreach beyond my Community.** An important goal in my research and teaching activities has been to evangelize about program analysis to computer scientists beyond my own research community. I believe that other specializations of computer science can not only apply program analysis to solve their research problems but can also contribute to advancing the state-of-the-art in program analysis. This is the key inspiration behind Chord, which is used at many universities world-wide, not only by program analysis specialists but also by researchers in other fields, such as machine learning and systems. To further my goal, I gave a tutorial on Chord at PLDI 2011. I highlight two cases illustrating the progress I made toward my goal. In one case, Percy Liang, a PhD student specializing in machine learning at UC Berkeley (now a faculty at Stanford), who was auditing my lectures on pointer analysis, was intrigued by the possibility of systematically applying machine learning to scale pointer analysis. We worked together for a year and published three papers [OOPSLA’10, POPL’11, PLDI’11]. In another case, Ariel Rabkin, a PhD student specializing in systems at UC Berkeley built a software configuration checker atop Chord and deployed it at Cloudera Inc., a company that offers enterprises a map-reduce platform based on Apache Hadoop. I derived a great sense of achievement when Ariel published this work at ICSE 2011, a top-tier conference in software engineering. In both cases, the most satisfying part for me was the fact that I was able to transform a bright student who was a novice to program analysis into an independent researcher who advanced the state-of-the-art in the field.

**Student Advising and Mentoring.** Besides the two students mentioned above, I collaborated with four others at UC Berkeley and two at Tel-Aviv University. These collaborations resulted in 13 papers in top-tier conferences. At Georgia Tech, I am advising four PhD students (Xin Zhang, Ravi Mangal, Sulekha Kulkarni, and Xujié Si). I also supervised the Masters thesis of two students (Aravind Machiry and Rohan Tahiliani) and supervised a PhD student (Saswat Anand) who is now a research associate at Stanford. My advising experience has made me better appreciate the fact that each student has unique abilities, and that my role is to help each of them realize their strengths to the maximum potential. As a testament to this effort, these students have published nine papers as lead authors [FSE’12, FSE’13, PLDI’13, ESOP’14, PLDI’14a, PLDI’14b, MobiHoc’14, FSE’15a, FSE’15b]. They have also won several competitive awards. My first two PhD students, Xin Zhang and Ravi Mangal, won a Distinguished Paper Award [PLDI’14a]. Another of Ravi’s papers was a Best Paper Nominee [ESOP’14]. Xin won a 2015-2016 Facebook Fellowship and was also a finalist for the Qualcomm Innovation Fellowship in 2014. Sulekha won the Microsoft Research Graduate Women’s Scholarship in 2015. I also mentored undergraduate students at Georgia Tech who won awards and went to pursue higher studies. William Holton won the College of Computing’s Outstanding Undergrad Research Award in 2014 and is now pursuing a Masters at Georgia Tech, where he is building intelligent tutoring systems that will provide enhanced and personalized learning tools for students. Another of my Masters students, Aravind Machiry, won a Distinguished Artifact Award [FSE’13] for his thesis research on automated testing of mobile apps. Aravind also went on to win the College of Computing’s Outstanding MS Research Award in 2014.
References


