Computer systems pervade modern society. But these systems are not correct. They are buggy; they fail. In 2002 NIST estimated that software failures alone cost the U.S. economy up to $60 billion. Furthermore, “over half of these costs are borne by software users in the form of error avoidance and mitigation activities.” This state of affairs is unacceptable. My goal is to help programmers find, fix, and prevent software defects. This document summarizes four of my larger research projects in service of this goal. I mention pre-tenure work for context, but focus on advances made after submitting my tenure dossier on November 1, 2010.

Instrumentation and Postmortem Program Analysis

When software fails, static and dynamic analyses can help developers understand and fix what went wrong. My interest in this topic goes back for years (UCB TR 2002, UW TR 2005, ESOP 2006, TSE 2010). Recently my team and I codified two complementary challenges in this area. The first challenge is how to gather dynamic information for later analysis. The second challenge is how to reconstruct a failing run, using whatever dynamic clues are available.

To gather dynamic information, one must balance thoroughness against efficiency. We created instrumenting compilers for C and C++ (with Java in development) that collect a variety of dynamic information to help failure reconstruction (ASE 2013, ACM SIGSOFT Distinguished Paper Award). Available data includes several variations on binarized code coverage and a lightweight form of bounded path tracing, all collectible with overheads too small to reliably measure. We introduced and studied the problem of customized program coverage, and proved that optimal coverage instrumentation is NP-hard (ASE 2016). Our integer-programming encoding of this problem is intractable for large programs, so we also developed two scalable approximations. Empirical evaluations show that these approximations are close to optimal for real-world code. Our techniques reduce instrumentation while allowing the developer complete freedom in defining coverage requirements. Where naïve instrumentation is dense or expensive, our optimizations lower execution overheads.

The reconstruction challenge begins with incomplete run-time data. We then infer as much as possible about what may, must, or cannot have happened during the failing run. For example, we know that all code dominating the failure location must have executed. Incorporating more dynamic data narrows the set of possibilities, allowing more precise reconstructions (ASE Journal 2016, invited submission). As we learned how to instrument more types of run-time data, we improved our reconstruction algorithms accordingly (WODA 2015). One of our recent approaches represented observed behavior using regular languages, encoded as finite-state acceptors. Intersections across these automata constrain the possible behaviors of failing runs. Another approach encoded the problem as a set of Boolean constraints, and used answer set programming to solve these constraints. We are still plumbing the theoretical depths of this problem, recently focusing on an obscure group of sub-regular languages that may offer a better balance of precision and scalability. We have also been learning how to present recovered executions to developers (ETX 2015), and plan to conduct a user study within the next year. In general, our approach to postmortem analysis is imperfect by design. We do not aim for exact failure replay. Rather, we seek the sweet spots where a small amount of cheap data yields a large improvement in analysis results.
Automated Repair of Concurrency Bugs

My pre-tenure research concerned automated debugging, with particular focus on concurrency bugs (SC 2008, SC 2009, WODA 2009, OOPSLA 2010). Post-tenure, I have shifted from automated bug detection to automated bug repair. Many non-deadlock concurrency defects lead to a program failing under a small number of bad interleavings. Our repairs remove bad interleavings while leaving most good interleavings alone. The approach seems natural enough, but the devil is in the details. Fixing concurrency bugs is challenging, in part due to non-deterministic failures and tricky parallel reasoning. We want to synthesize a patch that fixes the original problem without introducing new bugs, degrading performance, or damaging software readability.

My collaborators and I enacted this repair-by-removal strategy in two stages. We began with one of the most common concurrency bugs: atomicity violations. Our automated repair system took reports from existing bug-detection tools as input. It used static analysis to insert lock operations, creating a suitable patch for each bug report. It further tried to combine patches across bugs for better performance and readability. Finally, we added a run-time testing component tuned to match each patch. Our automated repairs eliminated six out of eight real-world bugs and made failure much less likely in the other two cases. Our patches never introduced new bugs and usually had similar performance to developers’ patches. Our work on this system appeared at PLDI 2011. A SIGPLAN CACM Research Highlight nomination praised our work as “well motivated,” “well written,” “very promising,” “surprisingly effective,” and “one of the first papers to attack the problem of automated bug fixing, so it should be of wide interest.”

The second stage of this work broadened our scope (OSDI 2012, SCIS 2015 invited). We added repair strategies for common ordering relationships, such as “some thread must initialize before any thread accesses” and “no access after deallocation.” We simultaneously used reports from a wide variety of concurrency-bug detectors, including ones not created by us. However, multiple defect detectors lead to multiple possible fixes. Thus, we created a new meta-level strategy that uses both static analysis and testing to choose the best repair. Applied to real code, our approach fixed thirteen out of thirteen bugs without causing deadlocks or excessive performance degradation. Our synthesized patches were of similar quality to those written by developers.

Error Propagation Analysis

Any code can harbor bugs, but error-handling code is especially vulnerable. In collaboration with domain experts, my team developed analyses that focus on error management in file system code (FAST 2008). Our most advanced interprocedural, flow- and context-sensitive analysis digested 871,000 lines of Linux kernel code. It detected hundreds of cases of error codes that fall out of scope, are not saved, or are overwritten without being handled (PLDI 2009). Kernel developers confirmed many of these as destructive but previously unknown, with one developer remarking that “this is an excellent way of detecting bugs that happen rarely enough that there are no good reproduction cases, but likely hit users on occasion and are otherwise impossible to diagnose.” Adding manual pages to our analysis revealed over 1,700 mismatches between documented and actual error-reporting behavior (PASTE 2010). Post-tenure, our research culminated in one doctoral dissertation and two peer-reviewed publications described below.

Our ISSTA 2011 paper explored the common Linux practice of hiding integer error codes inside pointers, as in “(inode *) -5”. Error-valued pointers are not valid memory addresses. Misuse of invalid pointers can crash the kernel, corrupt data, yield unexpected results, etc. This is a clear abuse of the C type system, but I thrive on finding robust solutions to such messy, real-world
problems. We used static program analysis to find dozens of cases of bad dereferences, bad pointer arithmetic, and bad overwrites in the Linux kernel.

The Linux kernel whetted my appetite for scalable program analysis. Prior efforts here have often hit a wall when they exhaust available memory. My team and I smashed through this wall by treating error propagation analysis as a database problem (ICSE 2015). We represented the program under analysis as a Semantic Web database, with analyses encoded as graph algorithms. We could analyze multi-million-line programs quickly and in just a fraction of the memory required by prior approaches. When memory alone was not enough, secondary disk storage kept us going after others would give up. Our approach generalizes to any interprocedural finite distributive subset (IFDS) problem, raising the bar for future work on scalable static analysis. Note that our lead author on this work was an industry practitioner: an atypical but rewarding collaboration.

**Static Analysis to Create Language Bindings**

An ounce of prevention is worth a pound of cure. Besides finding bugs, my team helps developers understand and build software that is correct by construction. Library bindings allow reuse of low-level code in high-level scripting languages. Writing bindings by hand is error-prone and tedious to the point of madness. Popular automated tools only scan library headers, leading to cumbersome bindings. Our approach analyzes library source code to reconstruct deeper models of an interface’s behavior (PLDI 2009). We then use this to build idiomatic bindings suitable for high-level scripting languages. Post-tenure, our bindings work is still in progress. It has led to one doctoral dissertation and two peer-reviewed publications described below.

Our recent work focuses on models of library resource management. Bugs in resource management can lead to leaks and crashes, and are especially hard to debug in multi-language programs. We developed analyses to infer the ownership semantics of C libraries (ISMM 2013). We found that our analyses significantly reduced the manual annotation burden for a suite of fifteen open source libraries. We even enlisted the scripting language’s garbage collector to manage parts of the low-level C heap (ASE 2016). Our system annotated each array argument as (1) terminated by a special sentinel value, (2) fixed-length, or (3) of length determined by another argument. Knowing these properties yielded more idiomatic, efficient bindings. In experimental evaluation with real-world libraries, we annotated at least 70% of all arrays with correct length types. Our results were comparable to those created by human developers, but took far less time to produce.

**Summary**

Getting software right is hard. My team and I strive to make it easier. I worked as a software engineer for four years before beginning graduate study. This experience has shaped a research style that emphasizes practical, best-effort tools and techniques to mitigate the nasty complexities of real-world software development. My work is rooted in deep formalisms: dataflow analysis, type inference, numerical optimization, automata theory, and a variety of other mathematical constructs. Atop these formalisms, my team and I build robust tools that withstand all manner of abuse from the ugly practicalities of real software development. We put formal theories to work addressing the practical, messy problems that make real-world software so hard to get right. In a world full of software lemons, we combine theory and practice to make the best lemonade that we can.