Bor-Yuh Evan Chang

Research Statement

Cooperative Program Analysis:
Tackling the Software Crisis by Bridging the Reasoning Gap between Users and Tools

Introduction

There is little dispute that software continues to become more and more pervasive in our society and our everyday lives. As such, the "software crisis"—that is, the significant difficulty in *developing software* that is functionally correct, highly secure, easily maintainable, and unequivocally verifiable—is as real today as it was when the term was first used in the late 1960s. The recent years have shown only an increase in the challenges for software developers with an explosion in the diversity of computing platforms: from data centers and large-scale distributed systems to ubiquitous mobile, personal devices to cyber-physical systems that use software to interact and control the physical environment.

> My research program focuses on *software quality* through *tool-assisted programmer productivity*—investigating the fundamental principles and techniques for using algorithmic tools throughout the software engineering process.

This work combines fundamental algorithm design based on formal methods and mathematical logic with tool building to evaluate designs on real-world software systems.

The vision that underlies the efforts in my research group is to empower every software developer with advanced program analysis tools that effectively assist them in overcoming the software crisis. The last decade has seen an explosion in the capabilities of program analyzers and software verification tools. The depth and breadth of the kinds of properties that they can discover about programs are now quite astonishing. Yet, when we consider the use of program analyzers in assisting software developers directly (e.g., to find and fix bugs and defects), it, while increasing, is still disproportionately rare. This situation is not due to a lack of effort in deployment but rather a relative lack of techniques that enable the human user to take advantage of the wealth of information derived by the analyzer.

My core expertise and research contributions lie at the intersection of the programming languages and software engineering disciplines. Driven by the needs of real-world applications, my research makes advances in fundamental program analysis and software verification techniques. My approach moves beyond the *singular* focus on the automated reasoning engine to develop innovative techniques that encompass the entire process of applying tools during software development. In program analysis research, the impasse that precise analysis techniques cannot scale to real programs while coarse methods give poor results is widely accepted. Drawing on my expertise in symbolic methods, my work seeks to get past this quagmire by reimagining how the full spectrum of analysis techniques can be used and combined for the ultimate goal of assisting software developers. For example, my research has led to advances in the capabilities of analysis tools by targeting alarm triage [1–3], leveraging testing code as specifications [4, 6], and taking type specifications further by permitting temporary violations [7, 13].

Developing as a Faculty Researcher

I see my role as a faculty researcher as being as much (if not more) about amplifying the efforts of those around me as advancing my own research program, beginning foremost with PhD students. To date, I have graduated three PhD students who are becoming increasingly influential in their own right: Arlen Cox who joined the Center for Computing Sciences at the Institute for Defense Analyses upon graduation, Devin Coughlin who joined Apple's programming language and analysis team, and Sam Blackshear

who joined Facebook's static analysis team. Prior to graduation, they received prestigious fellowships, including a Chateaubriand Fellowship in Science, Technology, Engineering, and Mathematics from the Embassy of France and a Facebook Graduate Fellowship. They have already made highly visible contributions upon graduation. For example, Coughlin was behind one of the most highly advertised features of Swift 2, and Blackshear has been involved in Facebook's highly visible Infer static analysis project.

In this spirit, I have spent significant energy in organizing with other University of Colorado (CU) faculty to collectively raise the profile of our research with the establishment and publicizing of the CU Programming Languages and Verification Group (CUPLV), initially with Profs. Sriram Sankaranarayanan, Amer Diwan (now at Google), and Jeremy G. Siek (now at Indiana University) and later successfully recruiting Profs. Pavol Černý and Matthew A. Hammer to the group.[1] While each faculty member has their own individual research trajectory, my efforts have been to establish a culture of community and shared success in CUPLV. Such efforts have paid dividends in the last two years with several joint grants from CUPLV, beginning with a $1.6M DARPA award in which I am serving as PI collaborating with Sankaranarayanan and Černý as well as Profs. Kenneth M. Anderson and Tom Yeh and later as co-PI on a $5.8M DARPA award (with Sankaranarayanan, Černý, and other collaborators) and a $450K NSF award (with Hammer and other collaborators). In total, I have been involved in sponsored research awards totaling approximately $9.4M (see CV for details). Since beginning my faculty appointment at CU, my research team and I have published nineteen papers in selective proceedings, two peer-reviewed invited papers, an invited paper accompanying a keynote, and three peer-reviewed workshop or short papers (see CV for details). I have contributed to significant tool building efforts, most notably with Thresher/Hopper [1–3], that have provided platforms for innovation in and validation of our research.

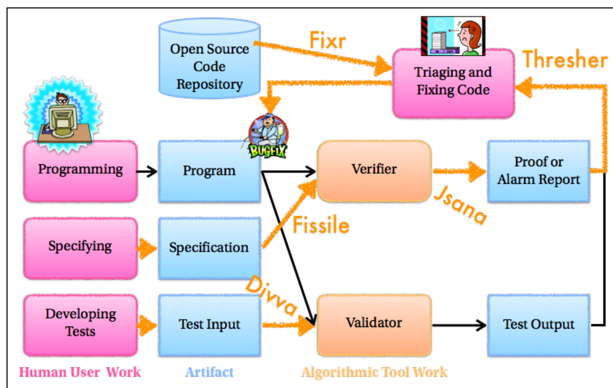

Figure 1: An overview of my research activities: *program analysis* in the *entire* bug mitigation *process*.

Figure 1 provides an overview of my research program in the context of applying program analysis during software development to minimize and mitigate bugs. I emphasize that mitigating bugs in software development is a *cooperative process* between human user work and algorithmic tool work. While the verifier plays a key role here, there is opportunity for algorithmic techniques to improve the entire process by looking more broadly. The orange arrows are labeled with some of my projects to address the challenges at particular places in the bug mitigation process, which I discuss in more detail below. I organize my efforts along two major thrusts across a number of application domains, including mobile and web applications, as well as several possible clients, including safety and security. While each thrust is driven by a different class of software systems, my team is making fundamental contributions to semantic program analysis techniques in general by revealing novel ways to mix specification of properties, automated reasoning engines, alarm triage, and mining source code repositories. In the following, I summarize each research thrust, highlight the most significant contributions, and expose their broader impact and application.

Thrust 1: Resolving Fixes and Making Improvements

A program has to be modified to incorporate an improvement—for example, to eliminate a bug. This thrust focuses on the user-tool interactive process of assessing potential defects, triaging reports, and finally resolving issues.

[1] CU Programming Languages and Verification: `http://plv.colorado.edu`.

The verification problem, whether all possible executions of a program satisfies a desired property, cannot be solved completely algorithmically (i.e., is in general undecidable). Because of these fundamental undecidability results, all automated analyzers must approximate in some way. For example, verifiers are typically designed to be one-sided in their approximation: they are certain when asserting that the program satisfies the property of interest (i.e., has no false negatives) but may raise false bug alarms (i.e., has false positives). While having no false negatives is a nice property of a verifier, the potential for false alarms confuses the user because now she has to decide whether each warning is a false alarm or a true bug. From the user's perspective, the presence of false alarms undermines the trustworthiness of the verifier.

Alarm Triage. Modern application software written in today's high-level programming languages make heavy use of the heap and concurrency. Thus, static reasoning about any non-trivial property of modern applications requires effective derivation of heap and concurrency properties. Unfortunately, conventional wisdom is that on real programs, the true bug/alarm ratio (precision) of static techniques will be unusably low. To overcome this impasse, our key insight is to move focus away from up-front precision and instead assist the programmer with precise alarm triage. Collaborating with researchers from Samsung, we developed the **Thresher**/Hopper tools that implement effective triage analysis algorithms for heap reachability properties based on goal-directed abstract interpretation that quickly filter out false alarms. Thresher/Hopper can precisely answer heap reachability queries on *real-world*, object-oriented programs [2] and event-driven programs [3]. These results were published at PLDI 2013 and OOPSLA 2015—top, highly selective venues for programming languages research (17% and 25% acceptance rates, respectively). With Thresher/Hopper, we have found 115 true memory leaks in open source Android applications and 11 crashing bugs due to misunderstanding or misusing the Android lifecycle. We also discovered defects in Android's core libraries; these were reported back to and fixed by Google. To achieve these results, we developed new fundamental analysis techniques (i.e., goal-directed heap analysis [2] and jumping analysis [3])—driven by novel ways of incorporating semantic program analysis into the bug mitigation process. Because heap properties are so fundamental in object-oriented programs, there are numerous potential clients including for safety (e.g., no crashes due to cast failures), security (e.g., secret objects are not leaked to untrusted code), and performance (e.g., non-escaping objects can be stack-allocated). My current work here includes investigating alarm triage in the challenging context of dynamic languages in collaboration with researchers from Aarhus University (Denmark), as well as for richer properties and other applications like for preventing resource-usage attacks on servers.

Mining Fixes. The basis for Thresher/Hopper is that the human user can more easily identify the true bugs in her program if the false alarms from the verifier can be filtered out in a directed manner. The human user must, however, still figure out *how* to modify her program to eliminate the bug.

While some bug fixes might always require human insight, other bug fixes may be more automatable—for example, instances of fixing common bug patterns in using software frameworks. Modern software frameworks enable creating rich applications with relative ease, but they also require client applications (apps) to follow complex protocols to get the desired behavior. At the same time, because such rich frameworks, like Android, have vibrant communities of open source app developers, there is an opportunity to apply algorithmic techniques to find, generalize, and apply bug fixes latent in source code repositories to other buggy apps—thereby *transferring* a bug fix from one app to another. The on-going **Fixr** project funded by DARPA that I lead seeks to find out whether this premise is possible.

Thrust 2: Identifying Defects and Verifying Fixes

This thrust focuses on identifying defects or verifying their absence in software. Verification requires powerful reasoning engines that are, at their core, some combination of invariant inference and specifi-

cation checking algorithms. Any particular combination trades off automation versus predictability.

Almost Everywhere Invariants. Traditional type systems are among the most successful application of formal methods to software engineering to date in large part because they focus on specifying or inferring global invariants that hold for all object instances on all executions—that is, everywhere. This restriction caps the complexity of reasoning that the user has to consider to understand the results of the analysis but also limits the kind of properties that can be checked. With my Ph.D. student Devin Coughlin, we created the **Fissile** framework that intertwines efficient, global refinement type analysis with precise, local symbolic analysis to check invariants that hold almost everywhere [7]. In particular, this work demonstrates that predictable, user-oriented global specifications can be strengthened by permitting temporary violations that are verified through execution-based, precise symbolic analysis. Similar to the previously mentioned projects, the underlying theme is a targeted application of different analysis techniques driven by a holistic view of a real-world application. We have applied Fissile to the problem of verifying reflective call safety in dynamic languages—an issue for which limited tool support exists, and we have tested our research prototype on a suite of large Objective-C libraries and applications with 461,000 lines of code and 23,000 methods. Overall, our tool was more accurate than type analysis alone with minimal cost in terms of user effort or analysis time. These results were published at POPL 2014, a highly-selective venue (23% acceptance rate). Because our framework is quite generic with respect to the type of properties that can be specified, there are many additional clients that one can consider for safety (e.g., data race freedom) or security (e.g., absence of buffer overruns).

Abstract Domain Combinators. Recent years have seen explosion in the use of dynamic languages, such as JavaScript, in software systems because of their ease of use for prototyping but also because of their flexibility. This flexibility has enabled numerous advanced and useful frameworks that simply cannot exist in traditional static languages. At the same time, this flexibility has come with a steep cost in tool support: current static analysis techniques designed primarily for traditional static languages are ineffective. Collaborating with researchers at ENS Paris, we have designed a number of abstract domain combinators that directly target dynamic language features in the **Jsana** project, such as sets and dictionaries [8–12]. This line of work asserts that dynamic languages are not beyond the scope of analysis techniques but require a rethinking of the way in which we apply them. I have also contribute to the design of several abstract domain combinators for reasoning about the heap of low-level C code. This line of work has addressed reasoning about multiple views to the same memory as with C-style unions [14], multiple inductive summaries of a recursive data structure [17, 18], and for unstructured sharing [15]. It has also addressed a fundamental problem in simultaneously reasoning about recursion in memory and recursion in execution [16] and was published at POPL 2011—a top, highly-selective venue (23%). A synthesis of this line of work was solicited as an invited paper [5].

Future Work

As suggested by Figure 1, my research approach is to look more broadly in the software development process where fundamental research in algorithmic approaches is needed to address the needs of today's real-world applications and to inform tomorrow's designs. Here, I discuss briefly the immediate trajectory of some projects that are just getting started.

Bug-Fix Trends in Event-Driven Protocols. In the Fixr project mentioned above, we are just beginning to produce results in the very challenging domain of mining protocol bug fixes in Android. Android is an event-driven framework, and Android apps are implemented by registering a large set of event callbacks. This framework design yields complex protocols that consist of a careful orchestration and sequencing of framework API calls across various different callbacks. Coupling the aforementioned challenges with

applying static analysis on "Big Code," I see several significant technical challenges and research trajectories from this point. For one, precise and deep static analysis of an event-driven system is challenging because the framework drives the application code to handle events but the execution of application code can affect what events are even possible. Here, we have some initial work on learning a model of the framework with respect to how events are enabled and disabled by app interactions by observing real app executions. Another set of challenges that we are investigating include how to identify bug fixing trends in source code repositories and the fundamental question of how to semantically analyze programs longitudinally across revisions.

Verification-Validation. Programs are executed, usually with unknown, hidden bugs. Testing or dynamic validation to expose possible bugs is typically completely disconnected from static verification, even though the two approaches to bug detection have fundamentally different trade-offs. The idea of combining the two approaches has been explored, but there are fundamental challenges in deeply intertwining what can be derived via verification versus validation. An on-going project called **Divva** seeks such a deep intertwining and has shown promise for verifying-validating data structure invariants. Based on these preliminary results, a newly-funded NSF project beginning this year seeks to design and implement run-time systems that make such incremental verifier-validators possible and effective.

References

[1] Sam Blackshear, **Bor-Yuh Evan Chang**, Sriram Sankaranarayanan, and Manu Sridharan. The flow-insensitive precision of Andersen's analysis in practice. In *Static Analysis (SAS)*, pages 60–76, 2011.

[2] Sam Blackshear, **Bor-Yuh Evan Chang**, and Manu Sridharan. Thresher: Precise refutations for heap reachability. In *Programming Language Design and Implementation (PLDI)*, pages 275–286, 2013.

[3] Sam Blackshear, **Bor-Yuh Evan Chang**, and Manu Sridharan. Selective control-flow abstraction via jumping. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 163–182, 2015.

[4] **Bor-Yuh Evan Chang** and Xavier Rival. Relational inductive shape analysis. In *Principles of Programming Languages (POPL)*, pages 247–260, 2008.

[5] **Bor-Yuh Evan Chang** and Xavier Rival. Modular construction of shape-numeric analyzers. In *Semantics, Abstract Interpretation, and Reasoning about Programs*, pages 161–185, 2013.

[6] **Bor-Yuh Evan Chang**, Xavier Rival, and George C. Necula. Shape analysis with structural invariant checkers. In *Static Analysis (SAS)*, pages 384–401, 2007.

[7] Devin Coughlin and **Bor-Yuh Evan Chang**. Fissile type analysis: Modular checking of almost everywhere invariants. In *Principles of Programming Languages (POPL)*, pages 73–86, 2014.

[8] Arlen Cox, **Bor-Yuh Evan Chang**, and Sriram Sankaranarayanan. QUIC graphs: Relational invariant generation for containers. In *Object-Oriented Programming (ECOOP)*, pages 401–425, 2013.

[9] Arlen Cox, **Bor-Yuh Evan Chang**, and Xavier Rival. Automatic analysis of open objects in dynamic language programs. In *Static Analysis (SAS)*, pages 134–150, 2014.

[10] Arlen Cox, **Bor-Yuh Evan Chang**, and Sriram Sankaranarayanan. QUICr: A reusable library for parametric abstraction of sets and numbers. In *Computer-Aided Verification (CAV)*, pages 866–873, 2014.

[11] Arlen Cox, **Bor-Yuh Evan Chang**, Huisong Li, and Xavier Rival. Abstract domains and solvers for sets reasoning. In *Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, pages 356–371, 2015.

[12] Arlen Cox, **Bor-Yuh Evan Chang**, and Xavier Rival. Desynchronized multi-state abstractions for open programs in dynamic languages. In *European Symposium on Programming (ESOP)*, pages 483–509, 2015.

[13] Yit Phang Khoo, **Bor-Yuh Evan Chang**, and Jeffrey S. Foster. Mixing type checking and symbolic execution. In *Programming Language Design and Implementation (PLDI)*, pages 436–447, 2010.

[14] Vincent Laviron, **Bor-Yuh Evan Chang**, and Xavier Rival. Separating shape graphs. In *European Symposium on Programming (ESOP)*, pages 387–406, 2010.

[15] Huisong Li, Xavier Rival, and **Bor-Yuh Evan Chang**. Shape analysis for unstructured sharing. In *Static Analysis (SAS)*, pages 90–108, 2015.

[16] Xavier Rival and **Bor-Yuh Evan Chang**. Calling context abstraction with shapes. In *Principles of Programming Languages (POPL)*, pages 173–186, 2011.

[17] Antoine Toubhans, **Bor-Yuh Evan Chang**, and Xavier Rival. Reduced product combination of abstract domains for shapes. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, pages 375–395, 2013.

[18] Antoine Toubhans, **Bor-Yuh Evan Chang**, and Xavier Rival. An abstract domain combinator for separately conjoining memory abstractions. In *Static Analysis (SAS)*, pages 285–301, 2014.