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.

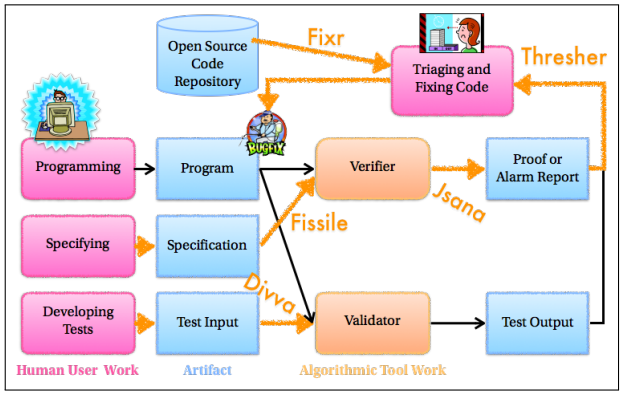


Figure 1: An overview of my research activities: *program analysis* in the *entire* bug mitigation *process*.

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.

Bor-Yuh Evan Chang

Teaching Statement

Teachers open the door. Students enter by themselves. — Proverb

Overview

The quote above sets the tone for my teaching: a teacher's primary role is to create a simultaneously enriching, challenging, and compassionate environment that supports the learning and growth of the individual student. This fundamental principle underlies both my classroom teaching—as exhibited by adopting a project-based approach for my courses—and my individual mentoring activities—as exemplified by my use of daily stand-up meetings.

Since joining the faculty at the University of Colorado (CU), I have taught courses across the graduate and undergraduate curricula. At the undergraduate level, I have invested in a significant refresh predicated on project-based and active learning principles for CSCI 3155 (Principles of Programming Languages), a foundational course required by all Computer Science majors. Without any explicit advertisement, a colleague at Tufts expressed interest in adopting my materials from this course, as well as a professor from National Chengchi University in Taiwan. I have also contributed to the improvement of CSCI 4555/5525 (Compiler Construction), an advanced undergraduate elective also commonly taken by master's students. At the graduate level, I have worked on revamping CSCI 5535 (Fundamentals of Programming Languages), an introductory graduate course to the formal semantics of programming languages. I have also developed a course on Program Analysis (CSCI 7135) that both strengthens the department's graduate offerings on formal methods and complements my research activities. The success of the first iteration of CSCI 7135 in Fall 2010 led to a request to offer a follow-on, advanced practicum course on the subject in Spring 2011. I discuss my approach and work on these classes in greater detail below.

With respect to outreach, in what was a particularly fun event, I led a "What is Computer Science?" workshop as part of the ASPIRE Summer Bridge program for newly admitted first-year engineering students organized by CU's BOLD (Broadening Opportunity through Leadership and Discovery) Center. The goal of the workshop was to dispel any misconceptions about the study of computer science by putting forth my assertion, "Computer science is about solving puzzles with social impact."

I have graduated three Ph.D. students, placing them in strong teams at the Center for Computing Sciences at the Institute for Defense Analyses (Arlen Cox), Apple (Devin Coughlin), and Facebook (Sam Blackshear). They have made highly visible contributions after graduation. For example, Coughlin was behind a highly advertised feature of the Swift 2 programming language—the feature was discussed at Apple's Worldwide Developers Conference 2015. I have also graduated four B.S. thesis students, am currently mentoring two postdoctoral research associates, three Ph.D. students, one B.S. thesis student, one M.S. student, and two B.S. students on independent research. In total, I am (or have been) the primary advisor for six Ph.D. students and five B.S. thesis students, as well as working with five M.S. students and six B.S. students. Of these eleven B.S. students, six of them started in my group as Discovery Learning Apprentices, a program through the College of Engineering and Applied Sciences.

As a faculty advisor, I see my role as amplifying the efforts and facilitating the success of those around me, beginning with my Ph.D. students. Of my twenty-six publications since joining CU, fifteen have been with my direct advisees and in total twenty-two have been with student co-authors from CU and elsewhere. Two of my Ph.D. students have won prestigious graduate fellowships: Chateaubriand Fellowship in Science, Technology, Engineering, and Mathematics from the Embassy of France (Arlen Cox) and a

Facebook Graduate Fellowship (Sam Blackshear). I have also placed Ph.D students at top research labs for summer internships, including Microsoft Research Redmond, Cambridge (UK), and India, NEC Labs, IBM Research Tokyo, Apple, and Facebook.

Integrating my teaching activities with my research activities has been an important way to recruit students into my group. All five B.S. thesis students along with three other B.S. students became interested in programming languages research through their experiences in my offerings of CSCI 3155.

I spend a significant portion of my energy in organizing the CU Programming Languages and Verification (CUPLV) Group[1] to create a collaborative and supportive environment that fosters excellence including but also beyond my own advisees. Currently with Profs. Sriram Sankaranarayanan, Pavol Černý, and Matthew A. Hammer, we seek to create a collaborative and supportive community with a critical mass that would be difficult to achieve within any individual faculty member's advisees. I credit the strong community among our graduate students for the successes of our research group at least as much as the individual faculty mentoring. I also credit the group in our ability to attract strong Ph.D. students and postdoctoral research associates. The recruiting environment in computer science has become particularly competitive for potential Ph.D. students and postdocs because the competition is not only with other universities but also with technology companies offering six-figure entry-level salaries.

In the following, I describe in further detail the application of my teaching philosophy, the challenges that I have faced, and the strategies that I have employed to improve.

Developing as a Teacher

My development as an instructor has benefited from leveraging the resources and workshops offered by the Faculty Teaching Excellence Program (FTEP). In Spring 2010, I took part in the "Teaching in a Nutshell" workshop facilitated by Prof. Lee Potts. This experience was invaluable in shaping and ultimately improving my classroom style. I have also attended FTEP seminars on the first day of class and the changes coming with on-line lectures and courses. When I taught my first large, 100+ student class, I received guidance and encouragement through the video consultation service, and I have followed up by engaging in the Classroom Learning Interview Process (CLIP) in Fall 2015. These experiences are part of my effort to continuously reflect on and improve my teaching. I have taken the opportunity to discuss and sit in on the classes of my colleagues. These experiences have informed how I have structured my courses, including abandoning the use of PowerPoint slides in favor of a conversational classroom and the introduction of short, frequent quizzes to provide students with prompt feedback on their level of understanding. The conversational style has led to deeper student engagement, and frequent quizzes have helped students better assess their progress. I attended a very helpful workshop organized by the College of Engineering and Applied Science and facilitated by Prof. Michael Prince (Bucknell) on active learning in the same semester in which I was revamping CSCI 3155 (Principles of Programming Languages at the undergraduate level), as well as a recent workshop on graduate advising organized by the Graduate School.

Classroom Teaching Philosophy

Reflecting on the courses from which I learned most and the instructors who I admire most, I have observed that quality teaching begins with a recognition of different ways of learning, deliberate and careful organization of learning goals, and a genuine enthusiasm for both the subject matter and the act of sharing it. While definitely challenging, I strive to create such an environment for *all* students.

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

Project-Based Course Design and the Conversational Classroom. Following the principle that my job is not to deliver tidbits of knowledge but rather to create an enriching environment for learning, I have embraced project-based courses and active learning ideas. With the design of every course, I begin by first making clear the learning goals of the course. These goals then drive the development of *hands-on homework projects* that students tackle through the course of the semester. The classroom meetings are then driven by dialogue about the thinking and background needed to tackle the project. As a concrete example, my undergraduate courses are typically structured into a sequence of two-week homework projects, each with clear and definitive learning goals. My lecture preparation consists of understanding the typical stumbling blocks in the project and creating discussion material to guide students past these stumbling blocks. The content of a lecture is never set but instead is driven by the questions that the students have on the project that day. While the discussion points for a given two-week project are the same from semester to semester, the order in which those points are covered is typically different. I have found this approach extremely effective in driving an interactive, engaging classroom.

The most common positive feedback that I hear from students on my courses are how much they learned and how much they were challenged by the homework projects. My impression of the reputation of my courses is that students will need to work very hard but if they do, they will get a lot of benefit from the experience.

> *I have never been so challenged in a course, and I am thankful for it.*
> *— Student in Fall 2015 CSCI 3155*

Continuous and Personalized Feedback. I have found a particularly challenging aspect for students in a project-based course is figuring out if they achieved the learning goals—whether they have been able to synthesize and internalize the higher-level concept from the detail work on the projects. To address this concern, I have employed three different strategies: (1) in-class exercises, (2) on-line drills, and (3) interview grading. In the subsequent class meeting after a project deadline, I administer an exercise or "quiz" consisting of questions covering the learning goals of the project and representative of exam questions. I create on-line drills to offer practice in basic skills and feedback on conceptual understanding. Finally, homework projects are graded through an interview process with a teaching assistant. I prepare a rubric and a set of questions for the teaching assistants, and the purpose of the interview is to uncover any misunderstandings the student might have about the project in an individualized setting. I have received feedback that the interview process, while intimidating at first, is one of the most helpful aspects of the course.

Continuous Monitoring. One of the difficulties with a heavy emphasis on homework projects is making sure that the course load is appropriate. One common issue is that students are sometimes unclear about the deliverable and spend more time delivering something beyond what I had expected of them. While I encourage interested students to explore, I also understand that students are balancing many obligations and should have a work load commensurate with the credit hours for the course. To address this issue, I include a survey on each assignment where I ask students to estimate the number hours spent, a qualitative assessment of the difficulty, and comments on the parts that gave them the trouble. I then review the results in a subsequent class to address any disconnect in expectations and to emphasize my commitment to a reasonable work load.

Research Mentoring Philosophy

Beginning from the foundation that my role is to facilitate and engender the success of my advisees, I have developed several strategies to promote a transition to independent thinking while providing a

structure and foundation for that development.

Providing a Structure and Foundation for Research Independence. With any student (or postdoc), I discuss what I believe it means to be a successful Ph.D. student (or undergraduate researcher, etc.) and ask them about their goals after graduation. I explain that my advice may differ depending on their goals and that I wish to stay connected with respect to their goals as they might change. With a new Ph.D. student, I emphasize getting active in research from day one—not after taking classes. I typically have them get started on an ongoing project with another student where they are expected to begin contributing on all aspects (brainstorming, implementing, writing, revising) right away to the best of their ability. Simultaneously, I encourage them to start thinking about how they can transition to more of a leadership role where they are the clear owner of a project.

Driven by the goal of supporting students while promoting ownership of their own research, I have continuously tweaked the way I conduct research meetings with students. I no longer have regularly scheduled weekly meetings with each student as is common—instead, I meet with them much more frequently in a much more fluid manner. Each day, the most important meeting of the day for me is a fifteen minute stand-up meeting with all of my students together to review their plan for the week. It is at this meeting that I can assess how a student is progressing and then schedule lengthier research meetings. This process helps me maximize the times I reserve for student research meetings: longer or shorter discussions as beneficial to the student.

Fostering a Collaborative and Supportive Community. I have also placed a strong emphasis on community, understanding the significant impact a dynamic group of peers can have on the success of the student. To support this effort for our collective students in the CUPLV group (across multiple faculty advisors), I organize a weekly student seminar series, a weekly reading group, and a twice-weekly, "stand-up status meeting." Each Ph.D. student is asked to give one slide-based presentation and one whiteboard-based presentation each semester in the seminar series. The stand-up status meetings have been fantastic for getting students into the lab, connected, and helping each other. I also run "critique meetings" where we get together—as a group—to critique and offer constructive suggestions on paper drafts and presentations.

While technical skills to solve research problems are important, I place even greater emphasis on *justifying* that they are addressing the right problems. I challenge students to continuously refine their problem statement and their justification of why that problem is *important*, *hard*, and *interesting*. I take every opportunity to support this thinking collectively as a group, including for example a group meeting devoted to delivering "elevator pitches" to be prepared when meeting important people at conferences.

Graduate Courses

Previously, the graduate offerings in the programming languages were not coordinated, resulting for example in three graduate courses being offered in one semester and none in the next. I organized an effort to coordinate schedules such that (1) we balance the graduate course offerings in this area across semesters and (2) schedule the foundational courses in the Fall and the more advanced courses in the Spring.

At the graduate level, I have alternated between teaching the foundational course on Fundamental Concepts of Programming Languages (CSCI 5535; in Spring 2009, Spring 2010, and Fall 2013) and a specialized course on Program Analysis (CSCI 7135; in Fall 2010, Spring 2013, Fall 2014, and Spring 2016). CSCI 5535 serves to provide a solid foundation to students going into programming languages research but is also a breadth course for students in other research areas. CSCI 7135 is a specialized graduate course. While many students are those engaged in the CU Programming Languages and Verification

group, there are often a few other graduate students with a secondary interest in developer tools. Advanced undergraduates are also welcome: a few undergraduates take the course each semester. In this course, I have structured the course project in a unique manner that is both collaborative and independent. Each student (or pair) undertakes a research project of their choosing, but all students contribute to a common infrastructure for building their analyzers. Particularly in the Fall 2010 version, this collaboration led to a lot excitement around the project. So much so that some students lobbied me to offer a second semester of the course to continue the project, which I obliged by offering a practicum course in Spring 2011 on top of my normal teaching duties.

Video Lectures and Distance Learning. I have offered most of my graduate courses through the distance learning program at CU. This program provides distance learning to working professionals by offering recordings of on-campus classes on-line. With distance students, the most significant challenge I have faced is managing the lower engagement level, particularly given my conversational classroom. To address this, I have offered "on-line office hours" via instance messaging, phone, or video conferencing. Then, in all of my classes (distance or not), I record the classroom discussion on a tablet computer (instead of on the blackboard) so that I can post pdf "whiteboards" after class. I have also promoted heavy use of class-wide on-line discussion forums, which has been beneficial for both distance and in-class students. I was one of the earliest adopters and one of the heaviest users of the on-line engagement tool Piazza at CU. I first used this technology in Fall 2011 and students liked it so much that we generated 1,253 posts.[2] I believe this challenge in engaging with distance students contributes to lower instructor ratings in these distance sections compared to the on-campus sections (3.7 versus 5.4 in Spring 2012, 4.3 versus 4.7 in Fall 2011, 4.3 versus 5.4 in Spring 2015), though these averages may also have been affected by low enrollment with sometimes one or two disengaged students (e.g., of the three distance students from Spring 2010, two rated the course with a 1 or a 2 but one student gave a 6 rating). While the trend is upwards, I continue to work on improving engagement with distance students.

Undergraduate Courses

My primary effort in undergraduate education has been developing and teaching CSCI 3155 (Principles of Programming Languages). I have taught this course five times in Fall 2009, Spring 2012, Fall 2012, Spring 2014, and Fall 2015. This course is both theoretically foundational and practically oriented. The programming languages in use are constantly changing, so a highly-qualified software engineer must be able to adapt to new languages and technologies quickly. This course elucidates the underlying principles of programming languages and computational models so that one can easily see how a new language is an evolution from existing ones. I am very explicit about this goal to the students: I say that the primary goal is to help them become "non-outsourceable software engineers."

Experience Revamping CSCI 3155. In Fall 2009, I followed closely existing materials for the course while having many discussions with students on how to improve the course. Based on that experience, I undertook a revamping of the course in Spring 2012 following project-based principles. While the pedagogical research and my own anecdotal experience have shown that students are much more engaged and end up more likely to understand and retain the subject matter using project-based approaches, I was surprised by the difficulties and resistance that I encountered that semester. This was in the end reflected in the lowest FCQ instructor rating in my pre-tenure period (3.7), whereas my average instructor rating in all other on-campus courses is 5.1. Fortunately, I was able to seek the support and advice of my senior colleagues, including one of our department's Presidential Teaching Scholars. Based on those discussions, I saw there were a confluence of factors and mistakes that led to this outcome. First, I severely underestimated the difficulty in addressing the needs of a large, 100+ student class in contrast

[2] Courses at CU using Piazza: `https://piazza.com/school/colorado/`.

to the ~40 student classes I had taught previously, and this was magnified by my discussion-oriented style. Second, a project-based course requires setting expectations for a certain level of independence. Third, this course is the last required course for computer science majors, and thus unfortunately there are some students who just do the bare minimum to graduate. Finally, I allowed the extreme negative feelings of one student to spread through the class even though they were not universally held. Given this experience, I asked to teach the course again in the next semester (Fall 2012) and took this experience into account. In particular, I was conscious about making clear the learning goals, the reasons for the project-based course, and the expectations required of them. I also added quizzes to help students better assess their own progress, and I paid close attention to students who I thought were falling behind by inviting them to meet with me. This resulted in a significant FCQ instructor rating improvement to 5.3. Note that both semesters had students who were extremely positive about the class: 3 students later joined my research group, and 3 others asked me for letter of recommendations. Multiple students have come by later to tell me how much they appreciated what they learned in the class, including one student from Spring 2012 who was rather unhappy during the semester. Overall, I have been happy with this turnaround without sacrificing the goals I had set out with incorporating project-based principles. According to Prof. Michael Prince, an expert in active learning techniques from the workshop I attended, this initial negative reaction from students is quite common for courses transitioning to project-based techniques.

What has crystallized in the subsequent semesters is that I must set expectations about the project-based structure clearly at the beginning of the semester. I must be clear that the course is technically deep and challenging, but at the same time, the course is structured so there are many opportunities to get past stumbling blocks before taking the exams. My project-based course design philosophy is largely informed by my experiences in developing this course.

Future Work. Grounded by my teaching philosophy, I continue working to improve as an instructor through self-reflection and foremost listening to students. To do so, I will continue to follow my existing approaches for surveying students, as well as finding new ways to get feedback (e.g., through FTEP's CLIP). I am actively searching for ways to offer students more continuous and personalized feedback through, for example, videos, interactive and guided exercises, and other hands-on mediums, as I believe doing so has the most potential for improving student learning.

Bor-Yuh Evan Chang

Service Statement

I am honored to be a faculty member at the University of Colorado Boulder and feel privileged to represent my department, college, and university in the scholarly community and society at large. I see service as an essential part of being a faculty member to contribute to the common good of society. My service efforts have been driven always by considering how I, at the current stage in my career, can best represent CU and help the university succeed in its educational and research missions.

Service to the CU Programming Languages and Verification Group

I have placed strong emphasis on organizing a community around the faculty whose research interests are most closely related to my own with the following goals: (1) collectively raise the profile of research at CU, (2) build a cooperative, collaborative, and supportive community for our graduate students, and (3) coordinate efforts to maximize the use of our resources. To this end, I have coordinated maintaining the public-facing website;[1] the internal mailing list, wiki, real-time chat (Slack), and version control services; weekly seminars; weekly reading group; and twice-weekly "stand-up" status meetings. This organization has led to a tight-knit community, for example, with students self-organizing practice talks and other meetings by themselves without faculty involvement. This community has also been a strong selling point in recruiting new Ph.D. students—in at least four cases, we were able to successfully recruit a student from competition at higher-ranked departments.

Service to the Department of Computer Science

My approach to department service has been to take on roles where I can best contribute to the collective good of the department.

I am presently serving on the Executive Committee (since Fall 2015). In this role, I am representing the junior faculty in discussing issues that concern the department leadership. I sought this experience to gain insight into the decision making process as I prepare to potentially take on future leadership roles in the department.

For the past two academic years, I have also served on the Faculty Search Committee. In this role, I have taken an active role in the reviewing, screening, interviewing, and recruiting of candidates, and I have taken a particular interest in increasing the engagement among our Ph.D. students in faculty recruitment. I worked with a Ph.D. student committee chair to develop process for student engagement in faculty recruitment modeled after a process from Berkeley.

I have also served as the Colloquium Chair for the department from Fall 2012 to Spring 2016. The primary role of this position is to coordinate the scheduling and advertising of our weekly colloquiums. In this role, I have also tried to raise the profile of our colloquium series by recording the sessions and making the content available on-line.

Earlier, I served on the Graduate Committee for five consecutive semesters (Spring 2009 to Spring 2011). One of the most important responsibilities of the Graduate Committee is graduate admissions and graduate recruiting. In 2009 and 2011, I co-organized the Ph.D. Student Recruiting Weekend. This event is hugely important for recruiting the most promising Ph.D. students. It is at this event that we are able to showcase the depth and breadth of exciting research taking place in our department coupled with displaying Boulder as a great place to live for their next five or six years. Organizing this event is a significant undertaking for the entire department, as we host 30–40 prospective students for two full days, requiring careful coordination between faculty, staff, and current Ph.D. students. Anecdotally,

[1] CUPLV Website: `http://plv.colorado.edu`.

students who we successfully recruited often say that their experience at the recruiting weekend is what became the deciding factor for choosing to come to CU.

Service to the College of Engineering and Applied Science and the University

During the 2011-2012 academic year, I served on the Faculty Search Committee in the Department of Electrical, Computer, and Energy Engineering (ECEE). One goal of this search was to foster stronger connections between ECEE and Computer Science (CS), which has been a component in my efforts for the CU Programming Languages and Verification group. This search resulted in the hiring of two new faculty members: Profs. Pavol Černý and Eric Keller. Both of these hires have strengthened the connections and collaboration opportunities between ECEE and CS, not only for me but for other faculty. Prof. Černý has collaborated with me as part of the CUPLV group, and Prof. Keller has collaborated with Prof. Rick Han in Computer Science.

For the College, I evaluate transfer credit for computer science courses, and for the University, I have taken part in reviewing proposals for the Innovative Seed Grant Program.

Service to the Research Community

I have provided external reviews to over thirty conferences and journals (see my CV for a complete list). Since joining CU, I have served on twelve program committees, including the most major conferences in my field—Principles of Programming Languages (POPL) in 2016 and Programming Languages Design and Implementation (PLDI) in 2017. Serving on the program committee of POPL or PLDI is an honor with great responsibility. Because these publication venues are so important to the community, the process expects program committee members to review 20–25 dense, full length papers without subreviewers, to interact with authors through one round of question-response, to deliberate in an on-line forum with other reviewers for a period of a few weeks, and to finally meet in person as a program committee to make and agree upon the acceptance or rejection decisions.

I have also served my research community by sitting on three NSF in-person proposal review panels. The process for NSF proposal review panels is similarly rigorous: reviewing approximately 7–10 proposals and then meeting in DC to deliberate and rank proposals.

Also for the research community, I have served on the external review committee at major conferences, and I have supported junior researchers in judging student research competitions. In 2013, I was asked to chair the Tools for Automated Program Analysis Workshop and since then have served on its steering committee.

I also served as co-treasurer or treasurer for the Principles of Programming Languages Conference from 2010 to 2014. The role of this position is to work with the general chair of the conference to manage a budget with more $200,000 in expenditures. The position is multi-year to provide continuity for the conference from one year to the next and is intended for a junior, untenured researchers to enable them to interact with the most senior members of the community. In this role, I have worked with Dr. Thomas Ball (Microsoft Research), Dr. John Field (Google, formerly IBM Research), Prof. Roberto Giacobazzi (University of Verona), Prof. Suresh Jagannathan (Purdue), and Dr. Sriram Rajamani (Microsoft Research).

Bor-Yuh Evan Chang

Curriculum Vitæ

bec@cs.colorado.edu
http://www.cs.colorado.edu/~bec/
+1 (303) 492-8894 (phone)
+1 (303) 492-2844 (fax)

Department of Computer Science
University of Colorado Boulder
430 UCB
Boulder, CO 80309-0430 USA

Education

PhD **University of California, Berkeley**, Computer Science 2008
Advisor: Prof. George C. Necula

Thesis: ***End-User Program Analysis***
Dissertation Committee: Prof. George C. Necula (chair), Prof. Koushik Sen, and Prof. Jack Silver

MS **University of California, Berkeley**, Computer Science 2005
Advisor: Prof. George C. Necula

Thesis: ***Type-Based Verification of Assembly Language***

BS **Carnegie Mellon University**, Computer Science, 4.0 GPA 2002
University and College Honors
Minors: Biological Science and Mathematical Science
Advisors: Prof. Robert Harper and Prof. Frank Pfenning

Thesis: ***Iktara in ConCert: Realizing a Certified Grid Computing Framework from a Programmer's Perspective***

Academic Appointments

University of Colorado Boulder January 2009–present
Assistant Professor, Department of Computer Science
Assistant Professor, Department of Electrical, Computer, and Energy Engineering (by courtesy).

University of Maryland, College Park September 2008–November 2008
Postdoctoral Researcher, Department of Computer Science
Advisor: Prof. Jeffrey S. Foster

Research Interests

Algorithms, tools, and techniques for building, understanding, and ensuring reliable computational systems.

Keywords: software quality, programmer productivity, program analysis, automated reasoning, programming languages, semantics, logic, formal methods.

Awards and Honors

Professional

National Science Foundation CAREER Award December 2010

Graduate

College of Engineering Graduate Student Prize December 2008
University of California, Berkeley

Siebel Scholar, offered but declined because of graduation August 2008
University of California, Berkeley

National Science Foundation Graduate Research Fellowship 2004–2007

California Microelectronics Fellowship 2002–2003

Undergraduate

Phi Kappa Phi Honor Society, inducted May 2002

Andrew Carnegie Society Presidential Scholar, selected December 2001

Phi Beta Kappa Honor Society, inducted October 2001

Lambda Sigma Honor Society, inducted September 1999

Carnegie Mellon University Presidential and Institutional Scholarships 1998–2002

Award Papers

ECOOP 2013: Distinguished Artifact Award July 2013
QUIC Graphs: Relational Invariant Generation for Containers

TACAS 2012: Journal Special Issue Invitation October 2012
A Bit Too Precise? Bounded Verification of Quantized Digital Filters

GRANTS

I have been a PI or co-PI on sponsored research totaling in awards of approximately $9.4M of which $2.3M have been led by me as PI. My portion of these awards has totaled approximately $2.3M. A summary of these figures is given below:

Total Awarded:	$9,422,144
Awarded for CU Boulder:	$5,809,166
Awarded, my portion, approximate:	$2,284,076
Awarded as PI:	$2,308,927

NSF CCF-1619282, "SHF: Small: Collaborative Research: Online Verification-Validation," $449,999 with $309,990 for CU Boulder, 09/01/2016–08/31/2019, Matthew Hammer (PI); Bor-Yuh Evan Chang (co-PI); David Van Horn (PI at University of Maryland, College Park).

DARPA FA8750-15-2-0096, "STAC: Auditr: Securing Space/Time Defenses in Java Bytecode," $5,797,751 with $2,891,227 for CU Boulder, 04/22/2015–04/11/2019, Pavol Cerny (PI); John Black, Bor-Yuh Evan Chang, and Sriram Sankaranarayanan (co-PIs); Isil Dillig (PI at the University of Texas at Austin); Marijn Heule (co-PI); Henny Sipma (PI at Kestrel Technology).

DARPA FA8750-14-2-0263, "MUSE: Mining and Understanding Bug Fixes to Address Application-Framework Protocol Defects," $1,599,343, 09/24/2014–09/13/2018, Bor-Yuh Evan Chang (PI); Kenneth M. Anderson, Pavol Cerny, Sriram Sankaranarayanan, and Tom Yeh (co-PIs).

DARPA via Maryland Z8140001, "APAC: Specialized Binary Analysis for Vetting Android Apps Using GUI Logic," $733,734 with $202,774 for CU Boulder, 09/01/2013–10/31/2015 subcontract from University of Maryland, College Park, Atif Memon (PI at University of Maryland, College Park); Tom Yeh and Bor-Yuh Evan Chang (co-PIs at CU Boulder).

NSF CCF-1218208, "SHF: Small: Modular Reflection," $250,000, 10/01/2012–09/30/2015, Bor-Yuh Evan Chang (PI); Jeremy G. Siek (co-PI).

NSF CCF-1055066, "CAREER: Cooperative Program Analysis: Bridging the Gap Between User and Tool Reasoning," $459,584, 06/01/2011–05/31/2017, Bor-Yuh Evan Chang (PI).

Joint Institute for Strategic Energy Analysis (JISEA) UGA-0-41026-04, "Verifiable Decision-Making Algorithms for Reconfiguration of Electric Microgrids," $49,985 with $14,500 for CU Boulder, 07/01/2010–06/30/2011, Siddharth Suryanarayanan (PI at Colorado State University); Sriram Sankaranarayanan (PI at CU Boulder); Bor-Yuh Evan Chang and Dirk Grunwald (co-PIs). JISEA is affiliated with the National Renewable Energy Laboratory (NREL).

NSF CCF-0939991, "EAGER: Exploratory Research on Gradual Programming," $81,748, 08/01/2009–07/31/2010, Jeremy G. Siek (PI); Bor-Yuh Evan Chang and Amer Diwan (co-PIs).

ADVISING

Graduated PhD Dissertation Advisees

1. Sam Blackshear, Fall 2010–Spring 2015, defended May 27, 2015. Computer Science. University of Colorado Boulder *Flexible Goal-Directed Abstraction*.
2. Devin Coughlin, Fall 2008–Spring 2015, defended January 7, 2015. *Type-Intertwined Separation Logic*. Computer Science. University of Colorado Boulder.
3. Arlen Cox, co-advised with Xavier Rival (ENS Paris) and Sriram Sankaranarayanan, Fall 2008–Fall 2014, defended November 17, 2014. *Parametric Heap Abstraction for Dynamic Language Libraries*. Electrical, Computer, and Energy Engineering. University of Colorado Boulder.

Graduated Senior Thesis Advisees

1. Evan Roncevich (BS 2016). Senior Thesis, Fall 2015–Spring 2016, defended May 6, 2016. Discovery Learning Apprentice, Fall 2014–Spring 2015. Computer Science. University of Colorado Boulder.
2. Ross Holland (BS/MS 2016). Senior Thesis, Fall 2014–Spring 2015, defended April 30, 2015. Computer Science. University of Colorado Boulder.
3. Nick Vanderweit (BS 2014). Senior Thesis, Fall 2013–Spring 2014, defended May 7, 2014. Computer Science. University of Colorado Boulder.
4. Alexander Beal (BS 2013). Senior Thesis, Fall 2012–Spring 2013, defended May 9, 2013. Computer Science. University of Colorado Boulder.

Current Postdoctoral Advisees

1. Edmund S.L. Lam, Summer 2016–present. Computer Science. University of Colorado Boulder.
2. Sergio Mover, Fall 2015–present. Computer Science. University of Colorado Boulder.

Current PhD Dissertation Advisees

1. Benno Stein, Fall 2015–present. Computer Science. University of Colorado Boulder.
2. Tianhan Lu, Fall 2015–present. Computer Science. University of Colorado Boulder.
3. Shawn Meier, Fall 2014–present. Computer Science. University of Colorado Boulder.

Current Senior Thesis Advisees

1. Maxwell Russek (BS), Fall 2014–present. Senior Thesis. Computer Science. University of Colorado Boulder.

Past Research Advisees

I have advised or co-advised independent studies or semester research projects for the following students.

1. Aleksandar Chakarov (PhD), advised by Sriram Sankaranarayanan, Summer 2010, Summer 2016. Computer Science. University of Colorado Boulder.
2. Alexandra Gendreau (PhD), advised by Ben Shapiro, Fall 2014–Spring 2015. Computer Science. University of Colorado Boulder.
3. Christoph Reichenbach (PhD 2009), advised by Amer Diwan, Spring 2009–Fall 2009. Computer Science. University of Colorado Boulder.
4. Nicholas Dronen (PhD 2016). Independent Study, Fall 2009. Computer Science. University of Colorado Boulder.
5. Krishna Chaitanya Sripada (MS 2016), co-advised with Pavol Cerny, Summer 2015–Spring 2016. Electrical, Computer, and Energy Engineering. University of Colorado Boulder.
6. Yi-Fan Tsai (MS 2013), Fall 2011–Spring 2014. Computer Science. University of Colorado Boulder.
7. Robert Frohardt (MS), co-advised with Sriram Sankaranarayanan, Fall 2009–Fall 2010. Computer Science. University of Colorado Boulder.
8. Daniel Stutzman (MS 2010), Spring 2010–Fall 2010. Computer Science. University of Colorado Boulder.
9. Kyle Howell (BS/MS 2016), Fall 2014–Spring 2016. Discovery Learning Apprentice, Fall 2014–Spring 2015. Computer Science. University of Colorado Boulder.
10. Parker Evans (BS 2016), co-advised with Pavol Cerny. Discovery Learning Apprentice, Fall 2013–Spring 2014. Computer Science. University of Colorado Boulder.
11. Kira Quan (BS/MS 2014), Fall 2012–Spring 2013. Computer Science. University of Colorado Boulder.
12. Chris Bubernak (BS 2013). Discovery Learning Apprentice, Fall 2011–Spring 2012. Computer Science. University of Colorado Boulder.
13. James Holley (BS 2012). Discovery Learning Apprentice, Fall 2009–Spring 2010. Electrical, Computer, and Energy Engineering. University of Colorado Boulder.

Current Research Advisees

1. Athithyaa Panchapakesan Rajeswari (MS), Summer 2016–present. Computer Science. University of Colorado Boulder.
2. Rhys Braginton Pettee Olsen (BS), co-advised with Sriram Sankaranarayanan, Spring 2016–present. Computer Science. University of Colorado Boulder.
3. Peilun Zhang (BS). Discovery Learning Apprentice, Fall 2016–present. Computer Science. University of Colorado Boulder.

Current Research Projects

University of Colorado Boulder 2016–

Ovv: Online Verification-Validation

Today's programmers essentially face a choice between creating software that is extensible and software that is verifiable. Languages either emphasize open extensibility or closed enforcement of abstractions. In this project, we explore language and virtual machine design techniques to offer verification in an extensible world by permitting analyses that freely interpose abstract interpretation with concrete execution in an incremental runtime system. With Matthew A. Hammer and David Van Horn.

University of Colorado Boulder 2015–

Auditr: Securing Space/Time Defenses in Java Bytecode

Recent work has highlighted vulnerabilities in server applications using resourage usage attacks that can cause denial of service and/or leakage of secret information. In this project, we investigate analysis techniques that enable a security analyst to more effectively secure an application to such attacks. Our premise is that such a technique must carefully orchestrate components that the detect, refine, and witness possible resource usage-based exploits. With Tianhan Lu, Pavol Černý, Sriram Sankaranarayanan, Ashutosh Trivedi, Işıl Dillig, Marijn Heule, and Henny Sipma.

University of Colorado Boulder 2014–

Fixr: Mining and Understanding Bug Fixes to Address Application-Framework Protcol Defects

With modern software frameworks like Android, there are millions of client applications (apps) that must carefully follow the complex and implicit protocols prescribed by the framework. And thus the same bug patterns occur over and over in apps. In this project, we search for algorithmic techniques for finding and generalizing bug fixes latent in source code repositories of some apps that can be automatically transferred to other apps. With Shawn Meier, Max Russek, Aleksandar Chakarov, Sergio Mover, Edmund S.L. Lam, Pavol Černý, Sriram Sankaranarayanan, Kenneth M. Anderson, and Tom Yeh.

University of Colorado Boulder 2011–

Thresher: Refutations via Goal-Directed Abstract Interpretation

Sound static analyzers over-approximate the concretely possible behaviors of programs. This statements means that they are certain when a given program satisfies a property of interest, but the trade-off they make is that they may raise false bug alarms. From a user's perspective, the presence of false alarms undermines the trustworthiness of the analyzer. In this project, we approach the false alarm problem by empowering the user with tools for alarm triage. Rather than focusing on guessing up-front what precision is necessary in the analyzer, we investigate approaches by which a user can direct further analysis of suspious alarms to quickly filter out false ones. We have effectively applied this approach to prove heap reachability properties and event-order safety properties of Android applications, and we are currently exploring applying this approach to the static analysis of dynamic languages. With Sam Blackshear, Benno Stein, Manu Sridharan, and Anders Møller.

Selected Past Research Projects

University of Colorado Boulder 2012–2014

Fissile Type Analysis: Modular Checking of Almost Everywhere Invariants

Reflection is a language feature that reduces boilerplate and affords flexible and powerful frameworks. In current languages, however, the safety of reflection is checked only at run-time. In this project, we examine a modular, dependent type analysis that ensures compile-time reflection safety by checking refinement predicates relating mutable storage locations. With Devin Coughlin.

University of Colorado Boulder 2012–2014

Jsana: Abstract Domain Combinators for Dynamic Languages

A hallmark of dynamic languages is the presence of libraries that emulate other language features using underlying dynamic features (e.g., class systems using JavaScript's open objects). In this project, we investigate abstract domain combinators to enable modular reasoning of the dynamic language features, such as open,

dynamically-extendable objects, indirect property lookup, and property iteration. With Arlen Cox and Xavier Rival.

University of Colorado Boulder 2011–2012

Measuring Enforcements for Data-Driven Static Analysis Design

Static analysis design is incredibly expensive, as many long iterations are required to determine whether or not a design is sufficient. We say that a static analysis design is sufficient if it can prove the property of interest with an acceptable number of false alarms. Ultimately, the only way to confirm that an analysis design is sufficient is to implement it and run it on real-world programs. If the evaluation shows that the design is insufficient, the designer must return to the drawing board and repeat the process. In this project, we observe that developers embed information in the form of enforcements that are both important to static analysis design and measurable with dynamic instrumentation. With Devin Coughlin, Jeremy G. Siek, and Amer Diwan.

University of Maryland, College Park 2008–2011

Mix: Mixing Program Analyses

Program analysis design is an exercise in tradeoffs. A precise analysis verifies deeper properties but may become prohibitively expensive to use, while a coarse analysis is efficient but suffers from high false alarm rates. In this project, we examine how to mix radically different analysis algorithms of varying precision that enables the analysis user (rather than the analysis designer) to make such tradeoffs. As a case study, we investigate the mixing of type inference (efficient) and symbolic evaluation (precise). With Khoo Yit Phang and Jeffrey S. Foster.

University of California, Berkeley 2006–2010

Xisa: Extensible Inductive Shape Analysis

Shape analyses are unique in that they can capture detailed aliasing and structural information that is typically beyond the ability of other static program analyses. To do so, they rely on specialized data structure descriptions to build and decompose summaries of memory regions. Unfortunately, existing approaches suffer from usability and scalability issues that make them impractical to apply broadly. Typically, they either are insufficiently extensible or require low-level, expert interaction. Instead, our project focuses first on practicality by designing an extensible shape analysis based around high-level, program developer-oriented specifications. In particular, we observe that data structure checking code (e.g., used in testing or dynamic analysis) provides shape information that can also be used effectively in static analysis. With Xavier Rival and George Necula.

Refereed Publications in Proceedings

In many areas of Computer Science, including Programming Languages, the first and primary publication venues are full length papers in the proceedings of selective conferences. Many conferences have extensive two-phase review processes with author responses. The two main outlets are proceedings published by ACM (Association for Computing Machinery) and the Lecture Notes in Computer Science (LNCS) series published by Springer. The DBLP service (`http://dblp.uni-trier.de/db/`) maintains an up-to-date index of Computer Science literature. In the following, the † mark indicate students that I formally advise, and the ‡ mark indicate other student co-authors.

Arlen Cox†, **Bor-Yuh Evan Chang**, Huisong Li‡, and Xavier Rival. 2015. Abstract Domains and Solvers for Sets Reasoning. In *International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR)*, volume 9450 of *Lect Notes Comput Sc*, pages 356-371, November 2015. Acceptance: 46.7%.

Sam Blackshear†, **Bor-Yuh Evan Chang**, and Manu Sridharan. 2015. Selective Control-Flow Abstraction via Jumping. In *ACM SIGPLAN Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, volume 50, number 10 of *ACM SIGPLAN Notices*, pages 163-182, October 2015. Acceptance: 25.2%.

Huisong Li‡, **Bor-Yuh Evan Chang**, and Xavier Rival. 2015. Shape Analysis for Unstructured Sharing. In *International Static Analysis Symposium (SAS)*, volume 9291 of *Lect Notes Comput Sc*, pages 90-108, September 2015. Acceptance: 40.1%.

Arlen Cox†, **Bor-Yuh Evan Chang**, and Xavier Rival. 2015. Desynchronized Multi-State Abstractions for Open Programs in Dynamic Languages. In *European Symposium on Programming (ESOP)*, volume 9032 of *Lect Notes Comput Sc*, pages 356-371, April 2015. Acceptance: 28.7%.

Arlen Cox[†], **Bor-Yuh Evan Chang**, and Xavier Rival. 2014. Automatic Analysis of Open Objects in Dynamic Language Programs. In *International Static Analysis Symposium (SAS)*, volume 8723 of *Lect Notes Comput Sc*, pages 134-150, September 2014. Acceptance: 37.7%.

Antoine Toubhans[‡], **Bor-Yuh Evan Chang**, and Xavier Rival. 2014. An Abstract Domain Combinator for Separately Conjoining Memory Abstractions. In *International Static Analysis Symposium (SAS)*, volume 8723 of *Lect Notes Comput Sc*, pages 285-301, September 2014. Acceptance: 37.7%.

Arlen Cox[†], **Bor-Yuh Evan Chang**, and Sriram Sankaranarayanan. 2014. QUICr: A Reusable Library for Parametric Abstraction of Sets and Numbers. In *International Conference on Computer Aided Verification (CAV)*, volume 8859 of *Lect Notes Comput Sc*, pages 866-873, July 2014. Acceptance: 24.9%.

Devin Coughlin[†] and **Bor-Yuh Evan Chang**. 2014. Fissile Type Analysis: Modular Checking of Almost Everywhere Invariants. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, volume 49, number 1 of *ACM SIGPLAN Notices*, pages 73-86, January 2014. Acceptance: 23.2%.

Arlen Cox[†], **Bor-Yuh Evan Chang**, and Sriram Sankaranarayanan. 2013. QUIC Graphs: Relational Invariant Generation for Containers. In *European Conference on Object-Oriented Programming (ECOOP)*, volume 7920 of *Lect Notes Comput Sc*, pages 401-425, July 2013. Acceptance: 25.0%. **Distinguished Artifact Award**.

Sam Blackshear[†], **Bor-Yuh Evan Chang**, and Manu Sridharan. 2013. Thresher: Precise Refutations for Heap Reachability. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 48, number 6 of *ACM SIGPLAN Notices*, pages 275-286, June 2013. Acceptance: 17.2%.

Antoine Toubhans[‡], **Bor-Yuh Evan Chang**, and Xavier Rival. 2013. Reduced Product Combination of Abstract Domains for Shapes. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 7737 of *Lect Notes Comput Sc*, pages 375-395, January 2013. Acceptance: 37.5%.

Alejandro Sánchez[‡], Sriram Sankaranarayanan, César Sánchez, and **Bor-Yuh Evan Chang**. 2012. Invariant Generation for Parametrized Systems using Self-Reflection. In *International Static Analysis Symposium (SAS)*, volume 7460 of *Lect Notes Comput Sc*, pages 146-163, September 2012. Acceptance: 40.3%.

Devin Coughlin[†], **Bor-Yuh Evan Chang**, Amer Diwan, and Jeremy G. Siek. 2012. Measuring Enforcement Windows with Symbolic Trace Interpretation: What Well-Behaved Programs Say. In *International Symposium on Software Testing and Analysis (ISSTA)*, pages 276-286, July 2012. Acceptance: 28.7%.

Arlen Cox[†], Sriram Sankaranarayanan, and **Bor-Yuh Evan Chang**. 2012. A Bit Too Precise? Bounded Verification of Quantized Digital Filters. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 7214 of *Lect Notes Comput Sc*, pages 33-47, March 2012. Acceptance: 24.5%.

Sam Blackshear[†], **Bor-Yuh Evan Chang**, Sriram Sankaranarayanan, and Manu Sridharan. 2011. The Flow-Insensitive Precision of Andersen's Analysis in Practice. In *International Static Analysis Symposium (SAS)*, volume 6887 of *Lect Notes Comput Sc*, pages 60-76, September 2011. Acceptance: 32.8%.

Xavier Rival and **Bor-Yuh Evan Chang**. 2011. Calling Context Abstraction with Shapes. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, volume 46, number 1 of *ACM SIGPLAN Notices*, pages 173-186, January 2011. Acceptance: 23.4%.

Robert Frohardt[†], **Bor-Yuh Evan Chang**, and Sriram Sankaranarayanan. 2011. Access Nets: Modeling Access to Physical Spaces. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 6538 of *Lect Notes Comput Sc*, pages 184-198, January 2011. Acceptance: 33.8%.

Khoo Yit Phang[‡], **Bor-Yuh Evan Chang**, and Jeffrey S. Foster. 2010. Mixing Type Checking and Symbolic Execution. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, volume 45, number 6 of *ACM SIGPLAN Notices*, pages 436-447, June 2010. Acceptance: 20.1%.

Vincent Laviron[‡], **Bor-Yuh Evan Chang**, and Xavier Rival. 2010. Separating Shape Graphs. In *European Symposium on Programming (ESOP)*, volume 6012 of *Lect Notes Comput Sc*, pages 387-406, March 2010. Acceptance: 24.8%.

Bor-Yuh Evan Chang and Xavier Rival. 2008. Relational Inductive Shape Analysis. In *ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, volume 43, number 1 of *ACM SIGPLAN Notices*, pages 247-260, January 2008. Acceptance: 16.5%.

Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. 2007. Shape Analysis with Structural Invariant Checkers. In *International Static Analysis Symposium (SAS)*, volume 4634 of *Lect Notes Comput Sc*, pages 384-401, August 2007. Acceptance: 30.6%.

Bor-Yuh Evan Chang, Matthew Harren, and George C. Necula. 2006. Analysis of Low-Level Code Using Cooperating Decompilers. In *International Static Analysis Symposium (SAS)*, volume 4134 of *Lect Notes Comput Sc*, pages 318-335, August 2006. Acceptance: 28.8%.

Bor-Yuh Evan Chang, Adam Chlipala, and George C. Necula. 2006. A Framework for Certified Program Analysis and Its Applications to Mobile-Code Safety. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3855 of *Lect Notes Comput Sc*, pages 174-189, January 2006. Acceptance: 48.3%.

Bor-Yuh Evan Chang and K. Rustan M. Leino. 2005. Abstract Interpretation with Alien Expressions and Heap Structures. In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 3385 of *Lect Notes Comput Sc*, pages 147-163, January 2005. Acceptance: 29.3%.

Refereed Journal Publications

Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, **Bor-Yuh Evan Chang**, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, and Dimitrios Vardoulakis. 2015. In Defense of Soundiness: A Manifesto. Commun. ACM, 58(2), pages 44-46, February 2015.

Arlen Cox[†], Sriram Sankaranarayanan, and **Bor-Yuh Evan Chang**. 2014. A Bit Too Precise? Verification of Quantized Digital Filters. Int J Softw Tools Technol Transfer, 16(2), pages 175-190, 2014.

Peer-Reviewed Invited Publications

Xavier Rival, Antoine Toubhans[‡], and **Bor-Yuh Evan Chang**. 2014. Construction of Abstract Domains for Heterogeneous Properties (Position Paper). In *International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA)*, volume 8803 of *Lect Notes Comput Sc*, pages 489-492, October 2014.

Bor-Yuh Evan Chang and Xavier Rival. 2013. Modular Construction of Shape-Numeric Analyzers. In *Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday (SAIRP)*, volume 129 of *Electr P in Theor Comput Sc*, September 2013.

Mike Barnett, **Bor-Yuh Evan Chang**, Robert DeLine, Bart Jacobs, and K. Rustan M. Leino. 2005. Boogie: A Modular Reusable Verifier for Object-Oriented Programs. In *International Symposium on Formal Methods for Components and Objects (FMCO)*, volume 4111 of *Lect Notes Comput Sc*, pages 364-387, 2005.

Invited Publications

Bor-Yuh Evan Chang. 2014. Refuting Heap Rechability (Extended Abstract). In *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, volume 8318 of *Lect Notes Comput Sc*, pages 137-141, January 2014. **Invited contribution**.

Refereed Workshop Publications and Peer-Reviewed Short Papers

Workshops in Computer Science vary in whether they have formal proceedings (i.e., are considered publications) or only informal ones. Formal proceedings are published by ACM, in LNCS by Springer, or in Electronic Notes in Theoretical Computer Science (ENTCS) by Elsevier.

Sam Blackshear[†], Alexandra Gendreau[†], and **Bor-Yuh Evan Chang**. 2015. Droidel: A General Approach to Android Framework Modeling. In *ACM SIGPLAN Workshop on State of the Art in Program Analysis (SOAP)*, pages 19-25, June 2015.

Khalid Alharbi[‡], Sam Blackshear[†], Emily Kowalczyk[‡], Atif Memon, **Bor-Yuh Evan Chang**, and Tom Yeh. 2014. Android Apps Consistency Scrutinized. In *Extended Abstracts at ACM SIGCHI Conference on Human Factors in Computing Systems (CHI-EA)*, pages 2347-2352, April 2014.

Bor-Yuh Evan Chang, Amer Diwan, and Jeremy G. Siek. 2009. Gradual Programming: Bridging the Semantic Gap (Position Paper). In *Fun Ideas and Thoughts at ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI-FIT)*, 2 pages, June 2009.

Bor-Yuh Evan Chang and K. Rustan M. Leino. 2005. Inferring Object Invariants. In *International Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL)*, volume 131 of *Electr Notes Theor Comput Sc*, pages 63-74, January 2005. Acceptance: 84.6%.

Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. 2005. Type-Based Verification of Assembly Language for Compiler Debugging. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI)*, pages 91-102, January 2005. Acceptance: 43.5%.

Bor-Yuh Evan Chang, Adam Chlipala, George C. Necula, and Robert R. Schneck. 2005. The Open Verifier Framework for Foundational Verifiers. In *ACM SIGPLAN International Workshop on Types in Language Design and Implementation (TLDI)*, pages 1-12, January 2005. Acceptance: 43.5%.

Bor-Yuh Evan Chang and Manu Sridharan. 2003. PML: Toward a High-Level Formal Language for Biological Systems. In *Workshop on Concurrent Models in Molecular Biology (BioConcur)*, volume 180, number 3 of *Electr Notes Theor Comput Sc*, pages 15-30, September 2003.

Bor-Yuh Evan Chang, Karl Crary, Margaret DeLap, Robert Harper, Jason Liszka, Tom Murphy VII, and Frank Pfenning. 2002. Trustless Grid Computing in ConCert. In *International Workshop on Grid Computing (GRID)*, volume 2536 of *Lect Notes Comput Sc*, pages 112-125, November 2002.

Andreas Abel, **Bor-Yuh Evan Chang**, and Frank Pfenning. 2001. Human-Readable Machine-Verifiable Proofs for Teaching Constructive Logic. In *Workshop on Proof Transformations, Proof Presentations, and Complexity of Proofs (PTP)*, 14 pages, June 2001.

Editing

Bor-Yuh Evan Chang. 2013. *Workshop on Tools for Automatic Program Analysis (TAPAS)*, volume 311 of *Electr Notes Theor Comput Sc*.

Technical Reports

Yi-Fan Tsai[†], Devin Coughlin[†], **Bor-Yuh Evan Chang**, and Xavier Rival. 2015. Synthesizing Short-Circuiting Validation of Data Structure Invariants. Technical Report arXiv:1511.04846, University of Colorado Boulder, 18 pages.

Alejandro Sánchez[‡], Sriram Sankaranarayanan, César Sánchez, and **Bor-Yuh Evan Chang**. 2012. Invariant Generation for Parametrized Systems using Self-Reflection. Technical Report CU-CS-1094-12, University of Colorado Boulder, 31 pages.

Devin Coughlin[†], **Bor-Yuh Evan Chang**, Amer Diwan, and Jeremy G. Siek. 2012. Measuring Enforcement Windows with Symbolic Trace Interpretation: What Well-Behaved Programs Say. Technical Report CU-CS-1093-12, University of Colorado Boulder, 18 pages.

Sam Blackshear[†], **Bor-Yuh Evan Chang**, Sriram Sankaranarayanan, and Manu Sridharan. 2011. The Flow-Insensitive Precision of Andersen's Analysis in Practice. Technical Report CU-CS-1083-11, University of Colorado Boulder.

Robert Frohardt[†], **Bor-Yuh Evan Chang**, and Sriram Sankaranarayanan. 2011. Access Nets: Modeling Access to Physical Spaces. Technical Report CU-CS-1076-10, University of Colorado Boulder, 23 pages.

Khoo Yit Phang[‡], **Bor-Yuh Evan Chang**, and Jeffrey S. Foster. 2010. Mixing Type Checking and Symbolic Execution. Technical Report CS-TR-4954, University of Maryland, College Park, 19 pages.

Bor-Yuh Evan Chang. 2008. End-User Program Analysis. Technical Report UCB/EECS-2008-161, University of California, Berkeley.

Bor-Yuh Evan Chang, Xavier Rival, and George C. Necula. 2007. Shape Analysis with Structural Invariant Checkers. Technical Report UCB/EECS-2007-80, University of California, Berkeley.

Bor-Yuh Evan Chang, Matthew Harren, and George C. Necula. 2006. Analysis of Low-Level Code Using Cooperating Decompilers. Technical Report UCB/EECS-2006-86, University of California, Berkeley.

Bor-Yuh Evan Chang, Adam Chlipala, and George C. Necula. 2006. A Framework for Certified Program Analysis and Its Applications to Mobile-Code Safety. Technical Report UCB/ERL M05/32, University of California, Berkeley.

Bor-Yuh Evan Chang. 2005. Type-Based Verification of Assembly Language. Technical Report UCB/EECS-2008-186, University of California, Berkeley.

Bor-Yuh Evan Chang and K. Rustan M. Leino. 2005. Abstract Interpretation with Alien Expressions and Heap Structures. Technical Report MSR-TR-2004-115, Microsoft Research.

Bor-Yuh Evan Chang and Manu Sridharan. 2003. PML: Toward a High-Level Formal Language for Biological Systems. Technical Report UCB/CSD-03-1251, University of California, Berkeley.

Bor-Yuh Evan Chang, Kaustuv Chaudhuri, and Frank Pfenning. 2003. A Judgmental Analysis of Linear Logic. Technical Report CMU-CS-03-131R, Carnegie Mellon University.

Bor-Yuh Evan Chang. 2002. Iktara in ConCert: Realizing a Certified Grid Computing Framework from a Programmer's Perspective. Technical Report CMU-CS-02-150, Carnegie Mellon University.

PRESENTATIONS

Fixr: Mining and Understanding Bug Fixes for App-Framework Protocol Defects. DARPA MUSE Site Visit. University of Colorado Boulder. Boulder, Colorado, USA. — May 3, 2016

Fixr: Mining and Understanding Bug Fixes for App-Framework Protocol Defects. DARPA MUSE Demonstration Workshop. Arlington, Virginia, USA. — February 1, 2016

Cooperative Program Analysis for Reliable Mobile Applications. Huawei Vision Forum. Santa Clara, California, USA. — October 21, 2015

Goal-Directed Program Analysis with Jumping. Max Planck Institute for Software Systems. Kaiserslautern, Germany. — October 3, 2015

Goal-Directed Program Analysis with Jumping. École Normale Supérieure. Paris, France. — October 1, 2015

Goal-Directed Program Analysis with Jumping. Google. Mountain View, California, USA. — July 24, 2015

Fixr: Mining and Understanding Bug Fixes for App-Framework Protocol Defects. DARPA MUSE PI Meeting. SRI International. Menlo Park, California, USA. — July 22, 2015

Type-Intertwined Heap Analysis. Aarhus University. Aarhus, Denmark. — May 18, 2015

Fixr: Mining and Understanding Bug Fixes for App-Framework Protocol Defects. DARPA MUSE Site Visit. University of Colorado Boulder. Boulder, Colorado, USA. — February 25, 2015

Cooperative Program Analysis. Computer Science Colloquium. Colorado State University. Fort Collins, Colorado, USA. — September 22, 2014

Fissile Type Analysis: Modular Checking of Almost Everywhere Invariants. Japan Advanced Institute of Science and Technology. Nomi, Japan. — August 6, 2014

Cooperative Program Analysis. National Taiwan University. Taipei, Taiwan. — August 5, 2014

Fissile Type Analysis: Modular Checking of Almost Everywhere Invariants. National Taiwan University. Taipei, Taiwan. — August 1, 2014

Refuting Heap Reachability. National Chiao Tung University. Hsinchu, Taiwan. — July 31, 2014

Fissile Type Analysis: Modular Checking of Almost Everywhere Invariants. Academia Sinica. Taipei, Taiwan. — July 30, 2014

Fissile Type Analysis: Modular Checking of Almost Everywhere Invariants. Carnegie Mellon University. Pittsburgh, Pennsylvania, USA. — April 23, 2014

Fissile Type Analysis: Modular Checking of Almost Everywhere Invariants. University of Maryland, College Park. College Park, Maryland, USA. — February 26, 2014

Refuting Heap Reachability. Invited Keynote: Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI). San Diego, California, USA. — January 20, 2014

Cooperative Program Analysis. Computer Science Colloquium. University of Colorado Boulder. Boulder, Colorado, USA. — October 3, 2013

Precise Heap Reachability by Refutation Analysis. Université Paris Diderot. Paris, France. — July 22, 2013

Precise Heap Reachability by Refutation Analysis. École Normale Supérieure. Paris, France. — July 19, 2013

Precise Heap Reachability by Refutation Analysis. Dagstuhl Seminar 13162: Pointer Analysis. Wadern, Germany. — April 16, 2013

Witnessing Heap Reachability for Resource Leaks in Android. Google. Mountain View, California, USA. — August 29, 2012

Measuring Enforcement Windows with Symbolic Trace Interpretation: What Well-Behaved Programs Say. Danish Static Analysis Symposium (DANSAS). Odense, Denmark. — August 24, 2012

Xisa: Extensible Inductive Shape Analysis. Aarhus University. Aarhus, Denmark. — August 23, 2012

Modular Reflection Checking using Relationship Refinements. Aarhus University. Aarhus, Denmark. — August 20, 2012

The Flow-Insensitive Precision of Andersen's Analysis in Practice. University of California, Berkeley. Berkeley, California, USA. — June 10, 2011

Xisa: Extensible Inductive Shape Analysis. Carnegie Mellon University. Pittsburgh, Pennsylvania, USA. — March 16, 2011

Calling Context Abstraction with Shapes National Taiwan University. Taipei, Taiwan. — December 17, 2010

Mixing Type Checking and Symbolic Execution. Front Range Architecture, Compilers, Tools, and Languages Workshop (FRACTAL). Boulder, Colorado, USA. — December 5, 2009

End-User Program Analysis for Data Structures. National Taiwan University. Taipei, Taiwan. — August 12, 2009

Using Checkers for End-User Shape Analysis. National Taiwan University. Taipei, Taiwan. — August 11, 2009

End-User Shape Analysis. National Taiwan University. Taipei, Taiwan. — August 11, 2009

Reduction in End-User Shape Analysis. Dagstuhl Seminar 09301: Typing, Analysis, and Verification of Heap-Manipulating Programs. Wadern, Germany. — July 24, 2009

Gradual Programming: Bridging the Semantic Gap. Fun Ideas and Thoughts Session (FIT) at the 2009 Conference on Programming Language Design and Implementation (PLDI'09). Dublin, Ireland. — June 16, 2009

End-User Program Analysis for Data Structures. Computer Science Department Colloquium. University of Virginia. Charlottesville, Virginia, USA. — November 24, 2008

End-User Program Analysis. Dissertation Talk. University of California, Berkeley. Berkeley, California, USA. — August 28, 2008

Extensible Shape Analysis by Designing with the User in Mind. Open Source Quality Project Retreat. Santa Cruz, California, USA. — May 16, 2008

Precise Program Analysis with Data Structures. Job Talk. — February–April 2008

Relational Inductive Shape Analysis. Thirty-Fifth International Symposium on Principles of Programming Languages (POPL'08). San Francisco, California, USA. — January 11, 2008

Materialization in Shape Analysis with Structural Invariant Checkers. Copenhagen Programming Language Seminar. IT University of Copenhagen. Copenhagen, Denmark. — August 27, 2007

Shape Analysis with Structural Invariant Checkers. Fourteenth International Static Analysis Symposium (SAS'07). Kongens Lyngby, Denmark. — August 24, 2007

Shape Analysis with Structural Invariant Checkers. Open Source Quality Project Retreat. Santa Cruz, California, USA. — May 10, 2007

Analysis of Low-Level Code Using Cooperating Decompilers. Thirteenth International Static Analysis Symposium (SAS'06). Seoul, Korea. — August 31, 2006

Inferring Object Invariants. First International Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL'05). Paris, France. — January 21, 2005

Abstract Interpretation with Alien Expressions and Heap Structures. Sixth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'05). Paris, France. — January 18, 2005

Type-Based Verification of Assembly Language for Compiler Debugging. Second International Workshop on Types in Language Design and Implementation (TLDI'05). Long Beach, California, USA. — January 10, 2005

Extensible Verification of Untrusted Code. Open Source Quality Project Retreat. Santa Cruz, California, USA. — May 13, 2004

PML: Toward a High-Level Formal Language for Biological Systems. First Workshop on Concurrent Models in Molecular Biology (BioConcur'03). Marseille, France. — September 6, 2003

Human-Readable Machine-Verifiable Proofs for Teaching Constructive Logic. Workshop on Proof Transformations, Proof Presentations and Complexity of Proofs (PTP'01). Siena, Italy. — June 19, 2001

Teaching Experience

University of Colorado Boulder — Fall 2016, Fall 2015, Spring 2014, Fall 2012, Spring 2012, and Fall 2009
Principles of Programming Languages, CSCI 3155, *Instructor*
Undergraduate-level course on the concepts of programming languages. In Spring 2012, I revised course using project-based principles.

University of Colorado Boulder — Spring 2016, Fall 2014, Spring 2013, and Fall 2010
Program Analysis: Theory and Practice, CSCI 7135, *Instructor*
Graduate seminar on the theoretical foundations and practical implementations of program analysis.

University of Colorado Boulder — Spring 2015 and Fall 2011
Compiler Construction, CSCI 4555/5525, ECEN 4553/5523, *Instructor*
Combined undergraduate and graduate-level course on an introduction to compiler construction concepts and techniques.

University of Colorado Boulder — Spring 2012
Readings in Programming Languages, CSCI 7900, *Instructor*
Advanced graduate reading seminar on current research topics. Discussions are primarily led by advanced graduate students.

University of Colorado Boulder — Spring 2011
Program Analysis Practicum, CSCI 7135, *Instructor*
Graduate practicum on program analysis design and implementation.

University of Colorado Boulder — Fall 2013, Spring 2010, and Spring 2009
Fundamentals of Programming Languages, CSCI 5535, *Instructor*
Core graduate-level course on the fundamental ideas behind modern programming language design and analysis.

University of California, Berkeley — Spring 2004
Programming Languages and Compilers, CS164, *Graduate Student Instructor*
Upper division course on programming language principles and compiler design, assisting Prof. George Necula. Also, applied research ideas to develop Coolaid, an assembly-level type-checking tool, to help students with compiler development and understanding.

Carnegie Mellon University — Fall 2000
Principles of Programming, 15-212, *Teaching Assistant*
Lower division course on abstraction and reasoning about programs and functional programming (taught in Standard ML), assisting Prof. Karl Crary and Prof. John Lafferty.

Carnegie Mellon University — Spring 1999
Fundamentals of Computer Science I, 15-211, *Teaching Assistant*
Lower division course on data structures and algorithms in C++, assisting Prof. Klaus Sutner.

Carnegie Mellon University — Fall 1999
Mathematical Foundations of Computer Science, 15-151, *Teaching Assistant*
Lower division course on fundamental concepts of discrete mathematics using Mathematica, assisting Prof. Edmund Clarke and Prof. Klaus Sutner.

Professional Activities

Program Chair

- Fifteenth Asian Symposium on Programming Languages and Systems (APLAS'17)
- Fourth International Workshop on Tools for Automatic Program Analysis (TAPAS'13)

Program Committees

- Thirty-First European Conference on Object-Oriented Programming (ECOOP'17)
- The 2017 Conference on Programming Language Design and Implementation (PLDI'17)
- Fourteenth Asian Symposium on Programming Languages and Systems (APLAS'16)
- Twenty-Third International Static Analysis Symposium (SAS'16)
- Seventeenth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'16)
- Forty-Third Symposium on Principles of Programming Languages (POPL'16)
- Seventh IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'15)
- Third International Workshop on Tools for Automatic Program Analysis (TAPAS'12)
- Thirteenth International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'12)
- Tenth Workshop on Program Analysis for Software Tools and Engineering (PASTE'11)
- Twenty-Fourth International Workshop on Languages and Compilers for Parallel Computing (LCPC'11)
- Third International Workshop on Numerical and Symbolic Abstract Domains (NSAD'11)
- First Workshop on Abstract Interpretation of Object-Oriented Languages (AIOOL'05)

Steering Committees

- International Workshop on Tools for Automatic Program Analysis (TAPAS)

External Review Committees

The 2015 Conference on Programming Language Design and Implementation (PLDI'15)

Forty-Second Symposium on Principles of Programming Languages (POPL'15)

The 2012 Conference on Programming Language Design and Implementation (PLDI'12)

Thirty-Ninth Symposium on Principles of Programming Languages (POPL'12)

The 2009 Conference on Programming Language Design and Implementation (PLDI'09)

External Reviews

Science of Computer Programming (2016)

Transactions on Programming Languages and Systems (2016)

Theoretical Computer Science (2015)

Computing Surveys (2015)

Transactions on Programming Languages and Systems (2014)

Computing Surveys (2014)

Acta Informatica (2014)

Journal of Automated Reasoning (2014)

The 2014 Conference on Programming Language Design and Implementation (PLDI'14)

Twenty-Fourth European Symposium on Programming (ESOP'14)

Forty-First Symposium on Principles of Programming Languages (POPL'14)

Eleventh Asian Symposium on Programming Languages and Systems (APLAS'13)

Semantics, Abstract Interpretation, and Reasoning about Programs: Essays Dedicated to David A. Schmidt on the Occasion of his Sixtieth Birthday (SAIRP'13)

Transactions on Programming Languages and Systems (2013)

Twenty-Third European Symposium on Programming (ESOP'13)

Fortieth Symposium on Principles of Programming Languages (POPL'13)

Second International Conference on Certified Programs and Proofs (CPP'12)

Transactions on Programming Languages and Systems (2012)

Transactions on Programming Languages and Systems (2011)

Twenty-Second European Symposium on Programming (ESOP'12)

Ninth Asian Symposium on Programming Languages and Systems (APLAS'11)

Eighteenth International Static Analysis Symposium (SAS'11)

Twelfth Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'11)

Thirty-Eighth Symposium on Principles of Programming Languages (POPL'11)

Nineteenth European Symposium on Programming (ESOP'10)

Thirteenth Conference on Foundations of Software Science and Computation Structures (FoSSaCS'10)

Thirty-Seventh Symposium on Principles of Programming Languages (POPL'10)

Eighteenth European Symposium on Programming (ESOP'09)

Fourteenth Static Analysis Symposium (SAS'07)

Twenty-First Symposium on Logic in Computer Science (LICS'06)

The 2006 Conference on Programming Language Design and Implementation (PLDI'06)

The 2006 Symposium on Security and Privacy (Oakland'06)

Thirty-Third Symposium on Principles of Programming Languages (POPL'06)

Thirty-Second Symposium on Principles of Programming Languages (POPL'05)

Ninth Conference on Functional Programming (ICFP'04)

Professional Service

Selection Committee, Student Reserach Competition, the 2015 Conference on Object Systems, Programming, Languages and Applications: Software for Humanity (SPLASH'15)

Selection Committee, Student Reserach Competition, the 2015 Conference on Programming Language Design and Implementation (PLDI'15)

Co-Treasurer, Forty-First Symposium on Principles of Programming Languages (POPL'14)

Treasurer, Fortieth Symposium on Principles of Programming Languages (POPL'13)

Treasurer, Thirty-Ninth Symposium on Principles of Programming Languages (POPL'12)

Treasurer, Thirty-Eighth Symposium on Principles of Programming Languages (POPL'11)

Co-Treasurer, Thirty-Seventh Symposium on Principles of Programming Languages (POPL'10)

Organizer, Front Range Architecture, Compilers, Tools, and Languages Workshop (FRACTAL), Fall 2009.

Student Committees

PhD Dissertation Committee

Aditya Zutshi (PhD), advised by Sriram Sankaranarayanan. Electrical, Computer, and Energy Engineering. University of Colorado Boulder.

Magnus Madsen (PhD 2015), advised by Anders Møller. Department of Computer Science. Aarhus University, Denmark.

Weiyu Miao (PhD 2013), advised by Jeremy G. Siek. Electrical, Computer, and Energy Engineering. University of Colorado Boulder.

Graham Price (PhD 2011), advised by Manish Vachharajani. Electrical, Computer, and Energy Engineering. University of Colorado Boulder.

Christoph Reichenbach (PhD 2009), advised by Amer Diwan. Computer Science. University of Colorado Boulder.

PhD Proposal/PhD Comprehensive Examination Committee

Hadi Ravanbakhsh (passed May 12, 2016), advised by Sriram Sankaranarayanan. Computer Science. University of Colorado Boulder.

Aditya Zutshi (passed December 16, 2015), advised by Sriram Sankaranarayanan. Electrical, Computer, and Energy Engineering. University of Colorado Boulder.

Aleksandar Chakarov (passed February 16, 2015), advised by Sriram Sankaranarayanan. Computer Science. University of Colorado Boulder.

Graham Price (passed 2010), advised by Manish Vachharajani. Electrical, Computer, and Energy Engineering. University of Colorado Boulder.

Joseph Blomstedt (passed December 9, 2009), advised by Dirk Grunwald and Jeremy Siek. Electrical, Computer, and Energy Engineering. University of Colorado Boulder.

PhD Preliminary Examination Committee

Hadi Ravanbakhsh, advised by Sriram Sankaranarayanan. Computer Science. University of Colorado Boulder.

Aleksandar Chakarov, advised by Sriram Sankaranarayanan. Computer Science. University of Colorado Boulder.

Chris Wailes, advised by Dirk Grunwald and Jeremy Siek. Computer Science. University of Colorado Boulder.

Weiyu Miao, advised by Jeremy Siek. Electrical, Computer, and Energy Engineering. University of Colorado Boulder.

MS Thesis Committee

Erin Duggan (MS 2015), advised by Tom Yeh. Computer Science. University of Colorado Boulder.

Shashank Bharadwaj (MS 2012), advised by Jeremy Siek. Electrical, Computer, and Energy Engineering. University of Colorado Boulder.

Erik Silkensen (MS 2012), advised by Jeremy Siek. Computer Science. University of Colorado Boulder.

Huxley Bennett (MS 2012), advised by Sriram Sankaranarayanan. Computer Science. University of Colorado Boulder.

Moss Prescott (MS 2010), advised by Jeremy Siek. Computer Science. University of Colorado Boulder.

Scott Mackey (MS 2009), advised by Ken Anderson. Computer Science. University of Colorado Boulder.

ME Report Committee

Jesse Bowes (ME 2016), advised by Ken Anderson. Computer Science. University of Colorado Boulder.

Mark Grebe (ME 2013), advised by Ken Anderson. Computer Science. University of Colorado Boulder.

Russell Winkler (ME 2012), advised by Ken Anderson. Computer Science. University of Colorado Boulder.

BS Thesis Committee

Trystan Binkley-Jones (BS 2015), advised by Ken Anderson. Computer Science. University of Colorado Boulder.

Emily Bertelson (BS 2015), advised by Tom Yeh. Computer Science. University of Colorado Boulder.

Eric Horacek (BS 2013), advised by Ken Anderson. Computer Science. University of Colorado Boulder.

Pavol Zelinsky (BS 2012), advised by Elizabeth R. Jessup. Computer Science. University of Colorado Boulder.

Erik Silkensen (BS 2011), advised by Jeremy Siek. Computer Science. University of Colorado Boulder.

Robert Stimpfling (BS 2010), advised by Ken Anderson. Computer Science. University of Colorado Boulder.

Departmental Service

Department of Computer Science. University of Colorado Boulder

Executive Committee: 8/2015–present

Faculty Search Committee: 8/2014–5/2016

Colloquium Chair: 6/2012–present

Graduate Committee: 1/2009–8/2011

PhD Recruiting Weekend: co-organized 2009, 2011

Organizes PhD Preliminary Exam in Programming Languages: 2010–present

Department of Electrical, Computer, and Energy Engineering. University of Colorado Boulder

Faculty Search Committee: 8/2011–5/2012

Computer Science Division. University of California, Berkeley.

Computer Science Graduate Student Association (CSGSA) Faculty Candidate Committee: 2007 (chair), 2006, and 2005

College Service

Transfer Credit Evaluator for Computer Science courses on behalf of the College of Engineering and Applied Sciences: 04/2015–present.

University Service

Innovative Seed Grant Proposal Reviews: 2011

Professional Affiliations

Association for Computing Machinery (ACM)

Special Interest Group on Programming Languages (SIGPLAN)

Special Interest Group on Software Engineering (SIGSOFT)

Special Interest Group on Logic and Computation (SIGLOG)

CURRENT COLLABORATORS

CU: Profs. Kenneth M. Anderson, Matthew Hammer, Sriram Sankaranarayanan, and Tom Yeh (Computer Science); Prof. Pavol Černý (Electrical, Computer, and Energy Engineering).

National: Prof. Isil Dillig, Dr. Marijn J. H. Heule (UT Austin); Dr. Henny Sipma (Kestrel Technology); Prof. Samuel Z. Guyer (Tufts); Prof. Amer Diwan (Google); Prof. Jeremy Siek (Indiana); Profs. Jeffrey S. Foster, Atif Memon, David Van Horn (Maryland); Dr. Manu Sridharan (Samsung Research).

International: Dr. Xavier Rival (INRIA/ENS/CNRS, France); Prof. Anders Møller (Aarhus University, Denmark).

CITIZENSHIP

United States of America

Selective Control-Flow Abstraction via Jumping

Sam Blackshear
University of Colorado Boulder, USA
samuel.blackshear@colorado.edu

Bor-Yuh Evan Chang
University of Colorado Boulder, USA
evan.chang@colorado.edu

Manu Sridharan
Samsung Research America, USA
m.sridharan@samsung.com

Abstract

We present *jumping*, a form of selective control-flow abstraction useful for improving the scalability of goal-directed static analyses. Jumping is useful for analyzing programs with complex control-flow such as *event-driven* systems. In such systems, accounting for orderings between certain events is important for precision, yet analyzing the product graph of all possible event orderings is intractable. Jumping solves this problem by allowing the analysis to selectively abstract away control-flow between events irrelevant to a goal query while preserving information about the ordering of relevant events. We present a framework for designing sound jumping analyses and create an instantiation of the framework for performing precise inter-event analysis of Android applications. Our experimental evaluation showed that using jumping to augment a precise goal-directed analysis with inter-event reasoning enabled our analysis to prove 90–97% of dereferences safe across our benchmarks.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords Android static analysis; event-driven systems; control-flow abstraction

1. Introduction

We consider the problem of selective flow/path-sensitive static analysis of *event-driven* systems. These systems are becoming increasingly important due to their prevalent use in web and mobile applications. In event-driven systems, control-flow occurs via invocation of event callbacks that may or may not be ordered. *Inter-event* flow/path-sensitive reasoning is often important for precision, but such reasoning can be prohibitively expensive due to the large number of possible event orderings that must be considered.

OOPSLA'15, October 25–30, 2015, Pittsburgh, PA, USA

http://dx.doi.org/10.1145/2814270.2814293

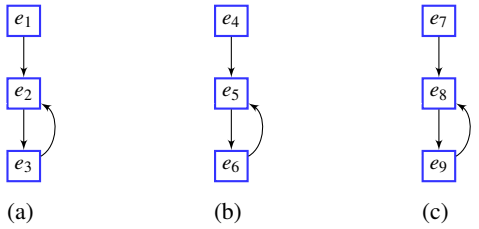


Figure 1: A simple event system containing three components with independent lifecycles.

To illustrate this problem, consider the simple event system in Figure 1. This system has three components (a), (b), and (c) with independent *event lifecycle graphs*. Events within an individual lifecycle graph are ordered by directed edges: $e_1 \rightarrow e_2$ specifies that e_1 must execute before event e_2. Otherwise, the events are unordered with respect to events in other lifecycle graphs. For example, the system specifies that any of events e_2, e_4–e_9 can execute immediately after e_1, but e_3 cannot. Event interleavings across lifecycle graphs are important to consider since all events may access shared mutable state.

The challenge in performing a flow/path-sensitive analysis of such systems is respecting intra-lifecycle ordering constraints while soundly accounting for interleavings of inter-lifecycle events. The obvious approach to achieving this result is to compute and analyze the product graph of all event lifecycle graphs in the event system. However, the number of edges in the product graph will be exponential in the number of components, and all such edges must be considered to perform a flow/path-sensitive analysis (even for the tiny system of Figure 1, the product graph contains 27 edges). For practical event systems with tens of components and hundreds of events like the Android applications we consider in Section 7, this graph quickly becomes intractable to represent—let alone analyze.

In practice, additional complications arise that make this problem even more difficult. Analyzing an individual event can be quite expensive because each event is essentially a standalone program—it may call thousands of procedures and contain loops and recursion. Component lifecycles can execute more than once, so the analysis may have to visit each edge in the product graph multiple times in order to compute a

fixed point. Finally, systems like Android implement lifecycle components and events via objects and instance methods, so the analysis may need to consider an unbounded number of instances of each lifecycle component.

Our insight is that although inter-event flow-sensitive reasoning is required to prove many properties of event-driven systems, most of these properties can be proven without considering all of the possible interleavings across component lifecycles. To leverage this insight to improve scalability, we need a *selective* form of control-flow abstraction flexible enough to apply flow/path-sensitive reasoning within an event lifecycle, but not across all edges in the lifecycle product graph. Though previous approaches to control-flow abstraction have effectively addressed the problem of selective context/object-sensitivity [26, 29, 30, 33], we are not aware of any previous work that can vary flow/path-sensitivity in the manner desired here. Previous flow-sensitive approaches to analyzing Android applications (e.g., [18]) have avoided this issue by assuming the lifecycles of different components cannot interleave. This is unsound and (as we will see in Section 2) can cause the analysis to miss real bugs.

In this paper, we tackle the challenge of selective flow/path-sensitive abstraction by introducing *jumping*, a form of goal-directed control-flow abstraction that enables the analysis to jump directly to code relevant to a particular goal query. Our key idea is that if we can identify the set of events that may affect the query at hand, we only need to reason about all possible orderings between these events in order to be sound. We have found that since the number of relevant events for a given query is typically small in practice, this approach is tractable even for large event systems. Our jumping framework allows us to limit the analysis to relevant events while retaining flow/path-sensitivity.

Our approach chooses locations to jump to by considering both data dependencies using a *data-relevance relation* and control dependencies using a notion of *control-feasibility*. The data-relevance relation enables the analysis to identify commands that may affect the current query, while control-feasibility information allows the analysis to consider event lifecycle constraints to preserve flow/path-sensitivity while jumping.

Though this paper focuses primarily on applying jumping to enable scalable analysis of event-driven systems, jumping is a general framework for selective control-flow abstraction whose utility goes beyond analyzing event-driven systems. Our framework provides an expressive way to describe a wide variety of sound control-flow abstractions that can be applied to any goal-directed backward abstract interpretation.

This paper makes the following contributions:

- We present a framework for *jumping analysis*, an approach for improving the effectiveness of goal-directed abstract interpretation via selective control-flow abstraction. We define soundness conditions for the relevance relation that enable jumping and prove the soundness of an analysis that performs jumps based on such a relation (Section 3).
- We devise an efficient points-to based technique for computing precise data-relevance information in heap-manipulating programs (Section 4).
- We give a semantics to the lifecycle graphs used by Android documentation to represent inter-event control-flow constraints, and we show how to specialize a general lifecycle graph to a specific subclass and object instance (Section 5).
- We use our jumping framework to design HOPPER, a practical jumping analysis for verifying safety properties in event-driven, heap-manipulating Android programs. HOPPER leverages our precise data-relevance relation along with control-feasibility information from Android lifecycle graphs to efficiently preclude consideration of concretely infeasible event orderings (Section 6).
- We evaluated HOPPER on the challenging client of checking null dereferences in event-driven Android programs (Section 7). Our results showed that adding jumping to a path-, flow-, and context-sensitive backward analysis decreased the number of unproven dereferences by an average of 54%. In addition, we found 11 real bugs in four different Android applications, nine of which have already been fixed via our submitted patches.

2. Overview

In this section, we motivate the difficulty and importance of verifying the absence of null dereferences in event-driven Android programs (Section 2.1) and show how jumping allows our analysis to prove the safety of dereferences without reasoning about an intractable number of event orderings (Section 2.2). Our running example (distilled from buggy code in the ConnectBot[1] app) will be attempting to prove the safety of the dereferences at lines 7 and 8 in Figure 2—that is, we want to show that `mService` and `mHostDb` (respectively) are always non-null at these lines. However, only the dereference at line 8 can be proven safe; the dereference at line 7 is buggy.

2.1 Motivation: Verifying Dereference Safety in Android

Null dereference errors (a Java `NullPointerException`, or NPE) are a major cause of failures in Android applications. In a search of the commit logs of the ten open-source Android apps used in our evaluation, we found 738 distinct commits containing the string "NPE" or "null," roughly 3% of all commits. Further, a recent paper on Facebook's INFER static analyzer reported that their internal database of production Android app crashes contained many null dereference errors [11]. Such errors cause crashes that stop the app and degrade user experience. Unlike crashes in web

[1] `https://github.com/connectbot/connectbot/pull/60`

```
class HostActivity extends onClickListener {
  // in method HostActivity.<init>
1   ManagerService mService = null;
2   HostDatabase mHostDb = null;

    void onCreate() {
      ServiceConnection cxn =
3       new ServiceConnection() {
          void onConnected(@Nonnull Service s) {
4           mService = (ManagerService) s;
          }
        };
      bindService(..., cxn);
5     findViewById(...).setOnClickListener(this);
6     mHostDb = new Database();
    }

    void onClick(View v) {
7     Host host = mService.getHost();
8     mHostDb.saveHost(host);
    }

    void onDestroy() {
9     mHostDb = null;
10    mService = null;
    }
}
```

Figure 2: A simple Android app with two components whose lifecycles are shown in Figure 3. The programmer uses correct reasoning about the lifecycle to ensure that the dereference at line 8 is safe, but mistaken assumptions about the lifecycle make the dereference at line 7 unsafe.

application code that can be fixed on the backend and pushed to the user when the page is refreshed, an app crash cannot be fixed until (1) an update fixing the bug is released and approved by the app store, and (2) the user elects to download the update, a process that can take weeks or even months [11]. Below, we discuss how subtleties of the Android app lifecycle can lead to null deference errors.

Bugs, Safety, and Lifecycles One reason that null dereferences in Android apps are easy to create and difficult to reason about is the complicated Android lifecycle. Though most events within the lifecycle of a single component are ordered, the lifecycles of different components can interleave arbitrarily and cause unexpected behavior. As a concrete example, Figure 3 shows the lifecycle graphs of the HostActivity and ServiceConnection classes from Figure 2. As in Section 1, a directed edge from event e_1 to e_2 means that e_1 must execute before e_2 is allowed to execute. The ε event represents a special "skip" event to soundly model the fact that user interaction events such as `onClick` may not be triggered. The edges between `onClick` and ε model the fact that a user can trigger the callback an arbitrary number of times.

The HostActivity and ServiceConnection components have independent lifecycles, but (as we can see from the code in Figure 2) they share the `mService` object. This leads to a null dereference at line 7 in the case that the `onClick` event fires before the `onConnected` event (since `mService` will still be null). This bug is due to faulty reasoning about the event-driven lifecycle of Android—the developer does not account for all possible interleavings between the HostActivity and ServiceConnection lifecycles.

On the other hand, the dereference at line 8 is safe because of ordering constraints in the HostActivity lifecycle. The developer delays initializing the `mHostDb` field to the heavyweight Database object until line 6 of the `onCreate` callback to avoid incurring the memory footprint of this object until it is needed. In addition, the developer assigns null to `mHostDb` at line 9 of the `onDestroy` event in order to relieve memory pressure as soon as possible (the enclosing HostActivity object may not become unreachable for some time after this event). These optimizations are safe because the lifecycle for HostActivity dictates not only that the `onCreate` event always executes after the constructor and before the `onClick` event, but also that the `onDestroy` event can only execute after all invocations of the `onClick` event.

Finding lifecycle sensitive bugs via testing is difficult given that (a) real apps have hundreds or thousands of events, (b) the developer must find the right combination of events that lead to a bug, and (c) exercising the app in a way that triggers the right events in the proper order is a tedious process. Thus, an effective static approach to this problem has the potential to significantly improve the state of affairs for Android app developers.

2.2 Proofs and Bug-Finding with Jumping Analysis

Though numerous static approaches to proving the absence of null dereferences have been proposed (e.g., [15, 22, 24, 25]), the key challenge in analyzing our motivating example does not concern the client of null dereferences specifically. Even a type-based approach with programmer-written nullness annotations would likely not work well. In Android, the nullness or non-nullness of a reference is frequently not a flow-insensitive invariant that holds at every program point or even an almost-everywhere invariant [12] that holds at nearly every program point. Instead, non-nullness (along with many other properties) holds during some phases of the lifecycle and not others.

Thus, the challenge for analyses (as we have explained in Section 1) is to perform precise reasoning about event orderings within a lifecycle without incurring the cost of reasoning about *all* event orderings. In what follows, we use the example in Figure 2 to demonstrate how jumping meets this challenge. This example has been simplified to contain only the events and instructions relevant to the two queries, but a real app would have many other lifecycle components whose event orderings the analysis might need to consider. In essence, the power of jumping is that it allows the analysis

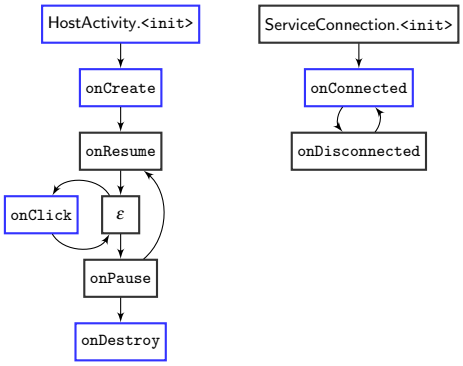


Figure 3: Lifecycle graphs for the HostActivity and ServiceConnection classes of Figure 2.

to soundly reduce a complex real-world event system into a simple system like the one in Figure 2.

Anatomy of a Jumping Analysis We consider extending THRESHER [7] (a path-, flow-, and context-sensitive backward analysis for *refuting* queries written in the symbolic heap fragment of separation logic [4] without inductive predicates) with the ability to jump, but jumping can just as easily be combined with any form of backward abstract interpretation (e.g., [9, 28]). At the intra-event level, the analysis behaves like THRESHER. When the backward analysis reaches an event boundary (that is, the entry of an application method that is invoked by the Android framework), the analysis chooses to compute a set of relevant events to jump to rather than continuing to follow backward control flow into the complex Android framework code.

From the entry of the current event e_{cur}, the analysis executes a jump by performing the following steps:

(1) **Identify important commands using data-relevance** For each constraint in the query, the analysis computes the set of *data-relevant* program commands whose concrete execution may produce a configuration in the concretization of the constraint. This process is similar to computing a partial slice that only considers immediately relevant commands (see Section 8 for a full discussion of the differences between our technique and slicing). Finding the set of relevant commands makes use of a global view of the program from a points-to graph computed by an up-front analysis.

(2) **Associate relevant commands to events** The analysis walks backward in the program's call graph from the containing method of each relevant command identified in step (1) and stops each time it hits an event boundary. This yields the set of events that may lead to the execution of relevant commands.

(3) **Order relevant events using control-feasibility information.** Though it would be sound to jump to all relevant events or even to jump directly to each relevant command, doing so loses information about the ordering of relevant events/commands, which is bad for analysis precision. In order to be precise, the analysis must account for the fact that only certain events are *control-feasible* with respect to the current event e_{cur} according to the Android lifecycle.

(4) **Jump to each control-feasible event.** The analysis forks a case split for each event that is both data-relevant and control-feasible, jumps to the exit of the event, and continues backward analysis for each case.

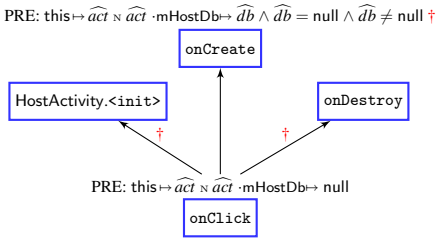


Figure 4: Proving safety of the dereference at line 8 of Figure 2 using jumping analysis.

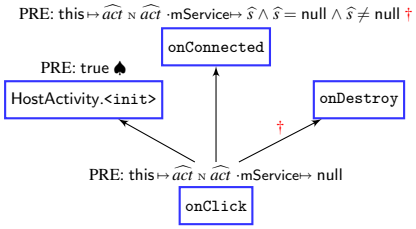


Figure 5: Failed safety proof for buggy dereference at line 7 of Figure 2 using jumping analysis.

Analyzing the Example App We now demonstrate how our jumping analysis uses the process described above to prove the safety of the dereference at line 8 and identify the bug at line 7. The analysis works in the style of THRESHER: it takes an initial query R that is a necessary precondition [13] for the bug to occur and attempts to prove safety (*refute* the query) by propagating the query backward from its initial program point ℓ in an attempt to derive a contradiction. This analysis is a form of proof by contradiction: it computes an over-approximation of the backward reachable states from the program point/query and refutes the query when it has derived the unreachability of (R, ℓ) (e.g., $\bot$—no possible concrete states) at a set of locations that together control-dominate the initial program point ℓ. For the dereference at line 8 to fail, the initial query is $\widehat{act}$ ·mHostDb↦ null[2] at program point 8, where $\widehat{act}$ is a symbolic variable representing a non-null HostActivity object and the ↦ edge denotes an *exact* points-to constraint in the sense of separation logic [27].

[2] All separation logic queries we write should be interpreted as describing a sub-heap (i.e., spatially conjoined with the predicate describing any heap).

The diagram in Figure 4 visualizes the process of proving the safety of this dereference by refuting the query. The analysis first uses THRESHER's transfer functions to propagate this precondition backward to the beginning of the `onClick` event, yielding the necessary bug precondition $\mathsf{this} \mapsto \widehat{act} \ast \widehat{act}.\mathsf{mHostDb} \mapsto \mathsf{null}$ shown in the figure.

At this point, the analysis chooses to perform a jump because it has reached an event boundary. The analysis decides which events to visit next using the four steps outlined above: first, it computes the data-relevant commands that may change the current bug precondition. This step yields the commands at lines 2, 6, and 9. Second, it uses the call graph to associate these relevant commands with their calling events, which yields the set of events HostActivity.`<init>`, `onCreate`, and `onDestroy`.

Third, the analysis uses the lifecycle graph for HostActivity in Figure 3 to perform control-feasibility filtering. The analysis determines that `onDestroy` is not control-feasible with respect to the current event `onClick` because `onDestroy` is not backward-reachable from `onClick` in the lifecycle graph for HostActivity. The analysis also determines that HostActivity.`<init>` is not control-feasible because it is *postdominated* by the relevant event `onCreate`—every feasible concrete execution reaching `onClick` visits `onCreate` between HostActivity.`<init>` and `onClick`.

Thus, the analysis concludes that it only needs to jump to `onCreate`. Figure 4 represents the decision not to jump to the data-relevant, but control-infeasible events HostActivity.`<init>` and `onDestroy` by marking the edges to these events with †'s. The directed edge from `onClick` to `onCreate` indicates that the analysis performs a jump from the entry of `onClick` to the exit of `onCreate` with the precondition shown for `onClick` as the abstract state.

When the analysis encounters the assignment at line 6 of the `onCreate` event, it refutes the query because there is an inconsistency between this command and the current abstract state: the points-to constraint $\widehat{act}.\mathsf{mHostDb} \mapsto \mathsf{null}$ says that the `mHostDb` field must hold the value null, but the command assigns a non-null Database value to this field. The analysis has therefore shown the safety of the dereference at line 8.

Figure 5 shows how the same analysis process (correctly) fails to prove the safety of the dereference at line 7. The analysis determines that the dereference is safe if the `onConnected` event executes before `onClick` and that the relevant event `onDestroy` is not control-feasible with respect to `onClick`, so it marks these paths as refuted (†). However, in the case that HostActivity.`<init>` is the last relevant event to fire before `onClick`, the command `mService = null` at line 1 discharges the precondition for `onClick`, leaving a necessary bug precondition of true. The analysis cannot hope to find a refutation given this precondition, so it gives up and reports the dereference at line 7 as a possible bug (as indicated by the ♠ symbol).

programs	$P, T ::= \{t_1, \ldots, t_n\}$
transitions	$t ::= \ell_1 \dashv[c]\vdash \ell_2$
commands	$c ::= \texttt{skip} \mid \texttt{assume}\ e \mid \texttt{call}\ \ell \mid \texttt{return}\ \ell \mid \ldots$
program labels	$\ell \in \textbf{Label}$
call strings	$L \in \textbf{Strings} ::= [] \mid \ell :: L$
abstract call strings	$\hat{L} \in \hat{\textbf{Strings}}$
concrete stores	$\rho \in \textbf{Store}$
concrete states	$\sigma \in \textbf{State} ::= (\rho, L)$
abstract stores	$\hat{\rho} \in \hat{\textbf{Store}}$
abstract states	$R \in \hat{\textbf{State}} ::= \top \mid \bot \mid (\hat{\rho}, \hat{L}) \mid R_1 \vee R_2$
command semantics	$\langle \sigma, c \rangle \Downarrow \sigma'$
abstract semantics	$\vdash \{ R_{\text{pre}} \}\ c\ \{ R_{\text{post}} \}$
concretization	$\gamma : \hat{\textbf{State}} \to \wp(\textbf{State})$
invariant map	$I : \textbf{Label} \to \hat{\textbf{State}}$

Figure 6: A language composed of atomic commands connected by unstructured control flow.

3. Jumping Analysis Framework

In this section, we formalize a framework for jumping analyses. The framework provides a mechanism by which any goal-directed, over-approximate, backward abstract interpretation can soundly perform "jumps" over irrelevant code given a *relevance relation* that meets certain soundness conditions. We will show how to instantiate this framework to perform tractable analysis of event-driven programs in Section 6.

3.1 Preliminaries: Commands and Transitions

We consider the imperative programming language of commands and unstructured control-flow presented in Figure 6. Our framework is parametric in the sub-language of commands and abstraction of concrete states chosen. A program in our language consists of a finite set of transitions t. We use P for the program of interest and T for a set of transitions in P. A transition $\ell_1 \dashv[c]\vdash \ell_2$ consists of a pre-label ℓ_1, a command c, and a post-label ℓ_2.

We assume that the concrete semantics of the commands are provided via a judgment form $\langle \sigma, c \rangle \Downarrow \sigma'$ that specifies how c transforms a concrete state σ to another state σ'. A transition relation for a small-step operational semantics of transition systems is given by a judgment form

$$\langle \sigma, \ell \rangle \underset{t}{\rightarrow} \langle \sigma', \ell' \rangle \,,$$

defined by applying the concrete command semantics $\langle \sigma, c \rangle \Downarrow \sigma'$ to transition $t : \ell \dashv[c]\vdash \ell'$. A judgment $\langle \sigma, \ell \rangle \underset{t}{\rightarrow} \langle \sigma', \ell' \rangle$ is well-formed only if the pre- and post-labels of t are ℓ and ℓ', respectively.

Our model encodes conditional branching using an `assume` e command that blocks unless e evaluates to true and encodes looping using `assume` along with back edges in the transition relation. We represent procedure calls using call and return commands that are linked to (respectively) callee procedures and caller sites in the program transitions.

$$\boxed{I \vdash \ell_1 \dashv[c]\vdash \ell_2}$$

$$\textsc{A-Transition}\quad \frac{I(\ell_2) \models R \quad \vdash \{\, R' \,\}\ c\ \{\, R \,\} \quad R' \models I(\ell_1)}{I \vdash \ell_1 \dashv[c]\vdash \ell_2}$$

$$\boxed{I \vdash \ell}$$

$$\textsc{A-Jump}\quad \frac{I(\ell_{\text{post}}) \models R \quad \langle R, \ell_{\text{post}} \rangle \rightsquigarrow T_{\text{rel}} \quad I \vdash t \text{ for all } t\colon \ell_i \dashv[c_{ij}]\vdash \ell_j \in T_{\text{rel}} \quad R \models I(\ell_j) \text{ for all } \ell_j \quad I \vdash \ell_i \text{ for all } \ell_i}{I \vdash \ell_{\text{post}}}$$

Figure 7: Jumping analysis. The key A-JUMP rule expresses the ability to skip code based on a relevance relation.

Both commands manipulate a call string L composed of program labels. The concrete semantics for these commands are standard (they are given in Blackshear [6, Chapter 5]). For simplicity in presentation, we assume that all variables in the program are globally scoped and that parameter binding is accomplished via ordinary assignment.

We write R for an abstract state that over-approximates a set of concrete states defined by a concretization γ. Notationally, we use a semantic entailment relation $R_1 \models R_2$ defined over concretization as $\gamma(R_1) \subseteq \gamma(R_2)$. We write $\top$ for the state such that $\gamma(\top) \stackrel{\text{def}}{=} \textbf{State}$ and $\bot$ for the state such that $\gamma(\bot) \stackrel{\text{def}}{=} \emptyset$. Otherwise, a state is a finite disjunction of pairs of a store abstraction $\hat{\rho}$ and a call string abstraction $\hat{L}$. We leave the particular store and call string abstractions of interest unspecified. An abstract semantics for commands is given by the judgment form $\vdash \{\, R_{\text{pre}} \,\}\ c\ \{\, R_{\text{post}} \,\}$: a backward Hoare triple stating that for all concrete post-states in R_{post} in which c terminates for some concrete pre-state, then that concrete pre-state is in R_{pre}. More formally, the abstract semantics must satisfy the following soundness condition:

$$\text{If } \vdash \{\, R_{\text{pre}} \,\}\ c\ \{\, R_{\text{post}} \,\} \text{ and } \langle \sigma_{\text{pre}}, c \rangle \Downarrow \sigma_{\text{post}} \text{ such that } \sigma_{\text{post}} \in \gamma(R_{\text{post}}), \text{ then } \sigma_{\text{pre}} \in \gamma(R_{\text{pre}}). \quad (1)$$

As an informal shorthand, we say R_{post} is *may-witnessed* by executing the command c from R_{pre}. As a corollary of this soundness condition, if the analysis *refutes* an input query R_{post} (i.e., derives $\bot$ on all backward paths originating from R_{post}), then R_{post} represents a set of unreachable concrete states. However, the analysis may over-approximate by failing to refute R_{post} even if R_{post} does not represent any concretely reachable states.

3.2 Control-Flow Abstraction with Jumping

To describe static analysis of transition systems, we define an invariant map $I : \textbf{Label} \to \hat{\textbf{State}}$ that maps each program label ℓ to candidate invariants at ℓ given by an abstract state R. Our jumping analysis is defined by the judgment form $I \vdash \ell$ that asserts, "I over-approximates the concrete states from which ℓ can be reached in a state satisfying $I(\ell)$." As a shorthand, when the judgment $I \vdash \ell$ holds, we say that I *may-witnesses* $I(\ell)$, or simply I *may-witnesses* ℓ.

This judgment form relies on an auxiliary judgment form $I \vdash t$ that asserts, "For a transition $t\colon \ell_1 \dashv[c]\vdash \ell_2$, $I(\ell_1)$ over-approximates the concrete states from which executing c yields a state satisfying $I(\ell_2)$." As above, we say that I *may-witnesses* transition t when the judgment $I \vdash t$ holds.

In Figure 7, we define these two judgment forms. The A-TRANSITION rule defines $I \vdash t$, which is analogous to the consequence rule of standard Floyd-Hoare logic. The rule says that I may-witnesses $\ell_1 \dashv[c]\vdash \ell_2$ if there is a triple $\vdash \{\, R' \,\}\ c\ \{\, R \,\}$ that satisfies soundness condition (1), such that $I(\ell_2)$ is stronger than R and $I(\ell_1)$ is weaker than R'. This rule is essentially just a wrapper that lifts an abstract semantics for commands to an abstract semantics for transitions that is constrained by our invariant map I.

The key rule for our jumping analysis is A-JUMP, which decides the transitions that the analysis should visit next. This rule relies on a relevance relation written using the judgment form $\langle R, \ell_{\text{post}} \rangle \rightsquigarrow T_{\text{rel}}$ that asserts, "Given an abstract state R at program label ℓ_{post}, the set of relevant program transitions is T_{rel}." Intuitively, the rule says to perform a backward jump from the current label ℓ_{post} to the post-label of each relevant transition in T_{rel}, skipping all transitions in between.

The rule's first two premises $I(\ell_{\text{post}}) \models R$ and $\langle R, \ell_{\text{post}} \rangle \rightsquigarrow T_{\text{rel}}$ state that we compute a set of relevant transitions T_{rel} using some weakening of the query $I(\ell_{\text{post}})$. Allowing this weakening of the state abstraction is crucial, as weakening of the state can only make the set of relevant transitions T_{rel} smaller. The "$R \models I(\ell_j)$ for all ℓ_j" premise constrains the post-state of each relevant transition to be weaker than the current state R. Together, these two premises can be seen as consequence for the transitions skipped by the jump.

The premise "$I \vdash t$ for all $t\colon \ell_i \dashv[c_{ij}]\vdash \ell_j \in T_{\text{rel}}$" checks that I may-witnesses each relevant transition $t \in T_{\text{rel}}$—that is, it uses the auxiliary judgment form to abstractly execute each relevant transition that was jumped to. Finally, the remaining premise "$I \vdash \ell_i$ for all ℓ_i" recursively continues the backward analysis by checking that I may-witnesses the pre-label ℓ_i of each relevant transition that was jumped to.

Inference, Loops, and Recursion While the judgment form $I \vdash \ell$ is most easily read as a checking system for judging when I may-witnesses ℓ for a given I, we can obtain an inference system that computes an invariant map I with a standard post-fixed-point computation via abstract interpretation. We begin the abstract interpretation from a map I_0 initialized with the initial query at the start label ℓ and all other labels mapping to $\bot$. The analysis applies the A-JUMP rule to that start label and updates the invariant map with the inferred values for R. A weakening (as in premise $R' \models I(\ell_1)$ of A-TRANSITION) corresponds to an update to the invariant map with a join (i.e., $I_{i+1}(\ell_1) = I_i(\ell_1) \sqcup R'$) or widen $\triangledown$ as appropriate to break loops in the abstract interpretation. This

process continues with additional labels via the recursive invocation "$I \vdash \ell_i$ for all ℓ_i" in the A-JUMP rule until the invariant map computation reaches a fixed point.

In the analysis, an arbitrary context- and object-sensitivity policy can be implemented by the choice of the call string abstraction $\hat{L} \in \widehat{\textbf{Strings}}$ and the state abstraction $R \in \widehat{\textbf{State}}$. For example, a simple k-callstring context-sensitivity policy could keep a disjunct for distinct call strings up to length k (joining or widening abstract stores $\hat{\rho}$ as needed).

In our implementation, we uniformly handle all sources of looping and recursion by widening at targets of back edges. Our widening operator bounds the length of the materialized prefix of the abstract call string $\hat{L}$ (i.e., program labels $\ell_1 :: \cdots :: \ell_k :: \mathsf{anystring}$) and the number of materialized heap locations (i.e., $\mapsto$ constraints) in the abstract store $\hat{\rho}$.

3.3 Identifying Relevant Transitions

The A-JUMP rule is an extremely general rule that allows a wide variety of strategies for choosing the transitions that the analysis should jump to. All transitions not jumped to are skipped. But what transitions can the analysis soundly skip? A-JUMP allows the analysis to skip any transitions except for the set of transitions T_{rel} returned by the relevance relation, so the burden of ensuring soundness falls squarely upon this relation. In this subsection, we will first build intuition for what transitions can and cannot be skipped before formally defining the soundness conditions that must imposed on a relevance relation in order to ensure sound jumping.

Data-Relevance and Control-Feasibility For the relevance relation to be sound, it must not omit any important transitions that could be involved in may-witnessing the query of interest. There are many different strategies that the relevance relation can use to ensure this soundness property. We will show that each of these strategies can be thought of as (1) choosing a set of *data-relevant* transitions that would be sound to return on their own, then (2) soundly filtering this set using *control-feasibility* information. As a first consideration, consider a relevance relation that returns all transitions in the program as data-relevant and performs no filtering. This relevance relation is trivially sound: it cannot skip any important transitions because it does not skip any transitions at all. This corresponds to a fully flow- and context-insensitive view of the program, as every transition will be a jump target from every other transition.

The above strategy of taking all program transitions can be improved by considering a simple form of control-feasibility: postdominance in the control-flow graph. Intuitively, if transition t' postdominates transition t, there is no need to consider both t and t' as relevant, as all backward paths to t must go through t'. Hence, it is sufficient to only consider t' as relevant. Via this reasoning, we can conclude that instead of treating all transitions in the program as relevant, one can instead use just the immediate predecessor transitions of the current program label while remaining sound. We have simply recovered the standard approach taken by flow/path-sensitive analyses.

Treating just the immediate predecessors as relevant is quite precise, but it does not utilize the key strength of jumping: the ability to skip irrelevant transitions entirely. We can make better use of jumping by refining the set of transitions returned by the data-relevance step using information about the program's data-flow. We can leverage data-flow information by modifying the data-relevance step to return all transitions that may affect the query state as data-relevant. For example, if the abstract state R is $\mathtt{x} \geq 0$ for a program variable $\mathtt{x}$, then writes to any variable other than $\mathtt{x}$ clearly cannot affect the query state and can be soundly skipped. An interesting aspect of our framework is that it admits a more restrictive strategy in which only transitions that *weaken* the abstract state are considered relevant (further discussion in Blackshear [6, Chapter 5]).

This strategy of taking the set of data-relevant transitions is less precise than taking the set of the immediate predecessors. An analysis that uses the data-relevance strategy will lose flow-sensitivity while jumping because it does not take the program's control-flow into account. On the other hand, the data-relevance strategy is likely to be more efficient because it considers only the (typically small) set of transitions that may affect the query without reasoning about any of the other transitions in the program or the control-flow between them.

A very powerful strategy is combining the previous two: first identify a set of data-relevant transitions for the current query, then use control-feasibility information such as postdominance in the control-flow graph to filter this set as much as possible. Doing this allows us to get the best of both worlds while jumping: we can skip vast swaths of irrelevant code by limiting our consideration to the data-relevant transitions, and we can maintain flow-, path-, and context-sensitivity while jumping by filtering away control-infeasible transitions using information about the control-flow between data-relevant transitions. This is the approach that we take with the relevance relation we will define in Section 6.

Defining Relevance Soundness Motivated by the preceding discussion of sound strategies for selecting relevant transitions to explore, we define our soundness condition for a relevance relation in a way that permits all strategies to be thought of as computing data-relevant transitions, then filtering these conditions using control-feasibility:

Condition 1 (Relevance soundness)**.**
If $\langle R, \ell_{\text{post}} \rangle \rightsquigarrow T_{\text{rel}}$, $\langle \sigma, \ell_{\text{pre}} \rangle \xrightarrow[T]{}{}^{*} \langle \sigma', \ell_{\text{post}} \rangle$,
$t_{\text{irrel}} : \ \ell_1 \dashv [c] \mapsto \ell_2 \in P - T_{\text{rel}}$, *and* $\vdash \{ R' \}\ c\ \{ R \}$, *then either*
(a) $R' \models R$, *or*
(b) $\exists\, T_1, T_2$ *s.t.* $T = T_1 \hat{} T_2$, $t_{\text{irrel}} \notin T_2$ *and* $T_{\text{rel}} \cap T_2 \neq \emptyset$.

We write $\langle \sigma, \ell \rangle \xrightarrow[T]{}{}^{*} \langle \sigma', \ell' \rangle$ for the judgment form of multi-step concrete evaluation, that is, the reflexive-transitive closure of single-step concrete evaluation. This multi-step

concrete evaluation records each transition it visits between ℓ_{pre} and ℓ_{post} in the trace T. We denote trace concatenation with $T_1 \hat{} T_2$.

Condition 1(a) captures the soundness of returning data-relevant transitions by imposing restrictions on a transition t_{irrel} that is *not* returned by the relevance relation. It states that for any program transition t_{irrel} not included in the set of relevant transitions for state R, the pre-state R' with respect to the transition command c is at least as strong as R. From the analysis perspective, this means that it is sound to exclude t_{irrel} from the jump targets because it cannot possibly move R closer to being witnessed. This is a very general notion of data-relevance, as it captures relevance based on both modification and weakening.

Condition 1(b) captures the soundness of filtering data-relevant transitions based on control-feasibility information. It says that if a transition t_{irrel} is not included in the set of relevant transitions for program point ℓ_{post}, then we can decompose the trace of visited transitions T into a pre-trace T_1 and a post-trace T_2 such that the post-trace contains a relevant transition, but does not contain t_{irrel}. This means then some relevant transition $t_{\text{rel}} \in T_{\text{rel}}$ must always happen *between* t_{irrel} and ℓ_{post}. From the perspective of our backward analysis, this means that it is sound to exclude t_{irrel} from the set of jump targets because some relevant transition t_{rel} that will be jumped to postdominates t_{irrel} in the program control-flow.

Using data dependence analysis to skip analysis of clearly irrelevant code is an effective and commonly used technique in program analysis. The novelty of our framework for sound jumping analysis based on Condition 1 is the ability to *simultaneously* reason about data-relevance and control-feasibility information to filter away a larger set of transitions (i.e., transitions that are data-relevant, but not control-feasible). Our relevance soundness condition is both permissive and general: it allows all of the control-flow abstraction strategies we have discussed so far and opens the door for any strategy whose structure can be described as computing data-relevant transitions, then filtering using control-feasibility.

Soundness of Jumping Analysis Next, we state and prove a soundness theorem demonstrating that any relevance relation satisfying this soundness condition can be used to define a sound jumping analysis.

Theorem 1 (Soundness of jumping analysis)**.**
If $\langle \sigma_{\mathsf{dummy}}, \ell_{\mathsf{dummy}} \rangle \xrightarrow[T]{}{}^{*} \langle \sigma_{\text{post}}, \ell_{\text{post}} \rangle$ *and* $I \vdash \ell_{\text{post}}$ *such that* $\sigma_{\text{post}} \in \gamma(I(\ell_{\text{post}}))$*, then* $\sigma_{\mathsf{dummy}} \in \gamma(I(\ell_{\mathsf{dummy}}))$.

The theorem says that if the concrete state σ_{post} at program point ℓ_{post} is in the concretization of the abstract state stored at label ℓ_{post} of the invariant map, then σ_{dummy} is in the concretization of the abstract state stored at the pre-entry label ℓ_{dummy}. In the theorem, we write concrete state σ_{dummy} for a distinguished element of **State** that represents the uninitialized, "junk" state before beginning the execution of the program. The proof of this theorem can be found in Blackshear [6, Appendix A]. The primary challenge of this proof was in formulating Condition 1 to be strong enough to prove the theorem, but weak enough to permit a wide variety of strategies for control-flow abstraction (including all of the other strategies described in the "Data-Relevance and Control-Feasibility" discussion on page 7).

Instantiating the Framework This section has presented a general framework for performing control-flow abstraction in a goal-directed backward abstract interpretation. In subsequent sections, we will present a practical instantiation of this framework for effective goal-directed analysis of event-driven Android programs. Section 4 explains how we compute precise data-relevance information in the presence of heap-manipulating commands, Section 5 explains how we represent inter-event control in Android in a way that enables control-feasibility filtering, and Section 6 combines the previous two sections to define a relevance relation for efficient event-driven analysis that satisfies relevance soundness (Condition 1).

4. Computing Precise Data-Relevance Information for Separation Logic Constraints

In this section, we explain how we compute precise data-relevance information for heap constraints in a way that satisfies the data-relevance condition of relevance soundness (Condition 1(a)). The key challenge of computing data-relevance information in the presence of the heap is to compute a precise approximation of relevant writes (i.e., precisely identify commands that may write to portions of the heap described in abstract state). If the analysis is not effective at precisely identifying such commands, it may report too many commands as data-relevant and negate the scalability benefits of jumping. Our approach is to leverage a combination of up-front points-to information, instance-from constraints [7] relating object instances to abstract locations, and explicit disaliasing information to precisely identify heap dependencies.

4.1 Command Language, Abstract State, and Abstract Semantics

As explained in Section 3.1, our framework for jumping analysis can be used with any representation of abstract state, command language, and abstract semantics. Here, we will consider computing data-relevance information for the separation logic-based abstract representation of THRESHER [7]. Specifically, the relevance relation we define here complements the abstract semantics of THRESHER extended with the abstract semantics for performing procedure calls presented in Figure 10. We will carefully explain the the command language and abstract state used by THRESHER as well as the abstract semantics of procedure calls before defining our data relevance relation in Section 4.2.

commands	$c ::= x := e \mid x := \texttt{new}_a\,\tau() \mid x := y.f \mid x.f := y$
	$\mid \texttt{assume}\ e \mid \texttt{call}\ \ell \mid \texttt{return}\ \ell$
expressions	$e ::= x \mid \cdots$
abstract locations	a
program variables	x, y
object fields	f
types	τ

Figure 8: An imperative command language with dynamic memory allocation and heap reads/writes.

abstract states	$R ::= \top \mid \bot \mid (\hat{\rho}, \hat{L}) \mid R_1 \vee R_2$
abstract call strings	$\hat{L} ::= \mathsf{anystring} \mid \ell :: \hat{L}$
abstract stores	$\hat{\rho} ::= M \wedge F \mid \mathsf{false}$
memories	$M ::= \mathsf{any} \mid x \mapsto \widehat{v} \mid \widehat{v}.f \mapsto \widehat{u} \mid M_1 \ast M_2$
pure formulæ	$F ::= \mathsf{true} \mid F_1 \wedge F_2 \mid \widehat{v}\ \mathsf{from}\ \mathring{r} \mid \cdots$
points-to regions	$\mathring{r}, \mathring{s} ::= \{a_0, \ldots, a_n\}$
instances	$\widehat{v}, \widehat{u}, \widehat{o}$

Figure 9: Components of abstract state.

Command Language We consider a simple imperative command language with objects and dynamic memory allocation (Figure 8). Like in Figure 6, concrete states σ are pairs of a concrete store ρ and a concrete call string L. Commands interact with the program heap via reads, writes, and allocations. Programmers can create aliasing relationships using imperative assignment. We leave our language of expressions unspecified, but we assume that expressions are pure and include reads of variables. An abstract location a is named by a syntactic allocation site $x := \texttt{new}_a\,\tau()$ and represents a potentially unbounded number of object instances of type τ allocated from its naming site. The language also contains `call` and `return` commands for performing procedure calls (as explained in Section 3.1).

Abstract State The components of THRESHER's state are enumerated in Figure 9. The top-level analysis unit is an abstract state R which consists of an (abstract store, abstract call string) pair, a disjunction of abstract states, or the special $\top/\bot$ states representing all concrete states and an unreachable state (respectively). A symbolic instance $\widehat{v}$ represents a *single* instance of an object (unlike an allocation site, which may represent an unbounded number of objects).

Abstract call strings implement a simple call-site sensitive form of context-sensitivity. Call strings consist of either the special anystring call string representing any possible call string, or a known label ℓ prepended to an abstract call string.

A memory M consists of an arbitrary memory any, an exact points-to constraint, or a separating conjunction of two memories. We write any for an arbitrary memory instead of using the more traditional true because we use true to denote the boolean truth value. We interpret memory M as $M \ast \mathsf{any}$, but typically omit the any when writing a memory for the sake of conciseness.

Points-to edges in our analysis state are *exact* points-to constraints of the form $x \mapsto \widehat{v}$ or $\widehat{v}.f \mapsto \widehat{u}$. The form $x \mapsto \widehat{v}$ means that the program variable x contains a value represented by the symbolic instance $\widehat{v}$. Similarly, the form $\widehat{v}.f \mapsto \widehat{u}$ means the symbolic instance $\widehat{v}$ holds the value represented by the symbolic instance $\widehat{u}$ at the offset specified by its f field.

The spatial constraints in a memory M are conjoined with a pure formula F consisting of either the truth value true, a conjunction of pure formulæ, or a special instance-from constraint. Instance-from constraints of the form "$\widehat{v}$ from $\mathring{r}$" state that the instance $\widehat{v}$ must have been allocated from the region $\mathring{r}$ (a region $\mathring{r}$ is a set of allocation sites). The constraint $\widehat{v}$ from $\emptyset$ is equivalent to false since it means that $\widehat{v}$ could not have been allocated from any allocation site. Tracking instance-from constraints explicitly in the analysis enables deriving such contradictions, and have been shown to significantly decrease the number of aliasing case splits that a backward separation logic-based analysis must perform [7]. We leave the remaining forms of pure formulæ unspecified, but our implementation handles other pure constraints that can be easily represented and solved by an SMT solver (inequality, arithmetic constraints, etc.).

$\boxed{\vdash \{R'\}\ c\ \{R\}}$

A-RETURN

$$\frac{}{\vdash \{(\hat{\rho}, \ell :: \hat{L})\}\ \texttt{return}\ \ell\ \{(\hat{\rho}, \hat{L})\}}$$

A-CALL-OK

$$\frac{\ell = \ell_1}{\vdash \{(\hat{\rho}, \hat{L})\}\ \texttt{call}\ \ell_1\ \{(\hat{\rho}, \ell :: \hat{L})\}}$$

A-CALL-REF

$$\frac{\ell \neq \ell_1}{\vdash \{\bot\}\ \texttt{call}\ \ell_1\ \{(\hat{\rho}, \ell :: \hat{L})\}}$$

A-CALL-ANY

$$\frac{}{\vdash \{(\hat{\rho}, \mathsf{anystring})\}\ \texttt{call}\ \ell\ \{(\hat{\rho}, \mathsf{anystring})\}}$$

Figure 10: Abstract semantics of `call` and `return` commands.

Abstract Semantics for Procedure Calls Figure 10 defines the abstract semantics for `call` and `return` commands. We explained the meaning of the backward Hoare triple $\vdash \{R'\}\ c\ \{R\}$ in Section 3. The abstract semantics for commands follows those defined previously in Blackshear et al. [7, Figure 4], which include all command forms other than `call` and `return`. While the abstract semantics are not strictly nec-

essary for understanding the data-relevance relation that we will present in Section 4.2, we define the abstract semantics for `call` and `return` here for completeness.

The A-RETURN rule says that when the analysis moves backward across the statement `return` ℓ, the return label ℓ is prepended to the abstract call string. This constrains the abstract call string to reflect that any concrete execution could only have reached this program point if it previously visited a matching `call` ℓ instruction that pushed ℓ onto the call string. The A-CALL-OK rule expresses the case where the analysis subsequently encounters this matching call. If the label ℓ on top of the call string matches the label ℓ_1 of the call command, the analysis weakens the state by popping the label off of the call string. By contrast, the A-CALL-REF rule expresses the case where the analysis subsequently encounters a non-matching call. If the label on top of the call string ℓ does not match the label ℓ_1 of the call command, the analysis *refutes* the current path (derives $\bot$) since no concrete state could have a non-ℓ_1 label on top of its call string immediately after executing the command `call` ℓ_1. Finally, the A-CALL-ANY rule says that an unconstrained call string anystring can be propagated backward across a `call` command without any changes.

4.2 Creating a Data-Relevance Relation for Separation Logic Constraints

Figure 11 defines a data-relevance relation for the abstract state and semantics explained in Section 4.1. The top-level judgment form $R \rightsquigarrow T_{\text{rel}}$ asserts that the transitions in T_{rel} may be relevant to the abstract state R. The auxiliary judgment forms $\hat{\rho} \rightsquigarrow T_{\text{rel}}$ and $\hat{L} \rightsquigarrow T_{\text{rel}}$ assert the same thing for the two sub-components of an abstract state: an abstract store $\hat{\rho}$ and an abstract call string $\hat{L}$.

They key challenge in computing data-relevance information for separation logic is determining what commands might be relevant to a points-to constraint $\widehat{v}.f \mapsto \widehat{u}$ stating that an object instance $\widehat{v}$'s field f contains the value $\widehat{u}$. For this constraint, we begin by considering commands that syntactically update field f (commands of the form $x.f := y$) as being possibly-relevant. Then, we can use the pure constraints in the abstract memory to further restrict relevant commands. In particular, if we have the instance-from constraint $\widehat{v}$ from $\mathring{r}$, we can restrict the relevant commands to those that satisfy the condition $\text{pt}(x) \cap \mathring{r} \neq \emptyset$. The function $\text{pt}(x)$ denotes the points-to set of x as computed by an up-front points-to analysis on the program P. The above condition ensures consistency between the points-to sets of x and the corresponding region $\mathring{r}$, rejecting any command that could not possibly yield the $\widehat{v}.f \mapsto \widehat{u}$ points-to constraint because it writes to an object different from $\widehat{v}$.

This process of leveraging instance-from constraints and up-front points-to information to precisely identify heap writes is encoded in the R-WRITE rule of Figure 11. We can further restrict relevant writes based on disaliasing constraints (either explicitly given $\widehat{v} \neq \widehat{o}$ or implied by separation $\widehat{v}.g \mapsto - \ast \widehat{o}.g \mapsto -$), though these additional restrictions are not expressed in the rule for presentation. In other words, we consider all transitions that may modify $\widehat{v}.f \mapsto \widehat{u}$ (syntactically) to be relevant and exclude a transition when we can prove that it does not modify $\widehat{v}.f \mapsto \widehat{u}$ (semantically). This strategy satisfies the data-relevance condition of relevance soundness (Condition 1(a)), and we adopt a similar strategy to ensure soundness for the remaining rules.

The remaining rules for the judgment form $\hat{\rho} \rightsquigarrow T_{\text{rel}}$ define data-relevance for other store-manipulating command forms. The rules R-READ, R-NEW, and R-ASSIGN compute the relevant commands for a local points-to constraint $x \mapsto \widehat{v}$ by using program syntax to identify all possible writes to x. These rules essentially encode a flow-insensitive variation of reaching definitions. The R-SEP rule gives structure to the judgment form by recursively applying the relevance relation to each sub-memory of the store to find the relevant transitions for the entire store. It says that the set of relevant transitions for the store $\hat{\rho}$ is the union of the relevant transitions for each of its sub-memories.

The R-BOT, R-TOP, and R-ANY rules defines the base cases of relevance for an unreachable state, a state representing all concrete states, and the separation logic predicate any that is satisfied by any heap (respectively). Each of these rules returns only the special initial transition t_{init} representing the first transition in the program. We can think of these rules as saying that nothing is relevant to each of these states; the reason each rule returns t_{init} is that proving relevance soundness is much easier if we can maintain the invariant that the set of relevant transitions is never empty (more specifically, that it always contains t_{init}).

The R-CASES rule says that for a disjunction of abstract states $R_0 \vee R_1$, the set of relevant transitions is the union of the relevant transitions for R_0 and R_1. The R-SPLIT rule decomposes an abstract state into its store component and call string component, computes the relevant transitions for each component using the auxiliary relevance judgments, and returns the union of the relevant transitions.

Finally, the R-CALL rule says that a call command with return label ℓ_1 must be considered relevant to a call string with a label $\ell = \ell_1$ as its first label. In our backward analysis, the abstract semantics for `call` can weaken the abstract state by popping a label off of the call string, thereby creating a less constrained call string. Thus, we must consider all `call` instructions with labels matching the top of the call string to be relevant in order to be sound. However, at an event boundary our implementation always chooses to weaken the abstract call string to anystring at before computing data-relevance/control-feasibility information and jumping, so this rule is never applied in practice. Instead, R-ANYSTRING will always be applied. This rule is simply the call string analog of the R-ANY and R-TOP rules.

$\boxed{R \rightsquigarrow T_{\text{rel}}}$

$$\textsc{R-Cases}\ \frac{R_0 \rightsquigarrow T_0 \quad R_1 \rightsquigarrow T_1}{R_0 \vee R_1 \rightsquigarrow T_0 \cup T_1} \qquad \textsc{R-Bot}\ \frac{}{\bot \rightsquigarrow \{t_{\text{init}}\}} \qquad \textsc{R-Top}\ \frac{}{\top \rightsquigarrow \{t_{\text{init}}\}} \qquad \textsc{R-Split}\ \frac{\hat{\rho} \rightsquigarrow T_1 \quad \hat{L} \rightsquigarrow T_2}{(\hat{\rho}, \hat{L}) \rightsquigarrow T_1 \cup T_2}$$

$\boxed{\hat{\rho} \rightsquigarrow T_{\text{rel}}}$

$$\textsc{R-Sep}\ \frac{\hat{\rho} = M_1 \ast M_2 \wedge F \quad M_1 \wedge F \rightsquigarrow T_1 \quad M_2 \wedge F \rightsquigarrow T_2}{\hat{\rho} \rightsquigarrow T_1 \cup T_2} \qquad \textsc{R-Assign}\ \frac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \dashv[x := y]\vdash \ell_j \,\}}{x \mapsto \widehat{v} \wedge \widehat{v}\ \mathsf{from}\ \mathring{r} \wedge F \rightsquigarrow T_{\text{rel}}}$$

$$\textsc{R-New}\ \frac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \dashv[x := \mathtt{new}_a\, \tau()]\vdash \ell_j \,\}}{x \mapsto \widehat{v} \wedge \widehat{v}\ \mathsf{from}\ \mathring{r} \wedge F \rightsquigarrow T_{\text{rel}}} \qquad \textsc{R-Read}\ \frac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \dashv[x := y.f]\vdash \ell_j \,\}}{x \mapsto \widehat{v} \wedge \widehat{v}\ \mathsf{from}\ \mathring{r} \wedge F \rightsquigarrow T_{\text{rel}}}$$

$$\textsc{R-Write}\ \frac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \dashv[x.f := y]\vdash \ell_j \text{ and } \mathrm{pt}(x) \cap \mathring{r} \neq \emptyset \,\}}{\widehat{v}.f \mapsto \widehat{u} \wedge \widehat{v}\ \mathsf{from}\ \mathring{r} \wedge \widehat{u}\ \mathsf{from}\ \mathring{s} \wedge F \rightsquigarrow T_{\text{rel}}} \qquad \textsc{R-Any}\ \frac{}{\mathsf{any} \wedge F \rightsquigarrow \{t_{\text{init}}\}}$$

$\boxed{\hat{L} \rightsquigarrow T_{\text{rel}}}$

$$\textsc{R-Call}\ \frac{T_{\text{rel}} = \{\, t \mid t \in P \text{ and } t = \ell_i \dashv[\mathtt{call}\ \ell_1]\vdash \ell_j \text{ and } \ell = \ell_1 \,\}}{\ell :: \hat{L} \rightsquigarrow T_{\text{rel}}} \qquad \textsc{R-AnyString}\ \frac{}{\mathsf{anystring} \rightsquigarrow \{t_{\text{init}}\}}$$

Figure 11: A data-relevance relation that uses an up-front points-to analysis and instance-from constraints to precisely identify relevant commands for separation logic constraints.

4.3 Cost of Computing Data-Relevance

To conclude, we briefly comment on the cost of computing data-relevance information. Clearly, this process needs to be efficient in order for control-flow abstraction via jumping to enhance the scalability of the analysis. Our data-relevance relation's use of a precomputed points-to analysis typically allows us to compute relevance quite quickly.

One potential scalability concern is than many rules in Figure 11 quantify over every command in the program P. We note that in practice, we can often compute relevance much more efficiently by exploiting procedural abstraction and up-front points-to information. The R-Assign, R-New, and R-Read rules compute the relevant statements for a constraint on some local variable x, so we only need to inspect each command in the method that x belongs to.

Shrinking the number of commands that the R-Write rule must consider is slightly more challenging, but we can do so using the points-to graph and instance-from constraints. Let $\mathring{E}$ be the edge set of the points-to graph, and let $x \Mapsto a$ denote a may-points to edge from the graph. Further, let the containing method of a local variable x be given by $method(x)$. Assuming that we are interested in determining the relevant write commands of the form $x.f := y$ for a heap constraint $\widehat{v}.f \mapsto \widehat{u} \wedge \widehat{v}\ \mathsf{from}\ \mathring{r} \wedge \widehat{u}\ \mathsf{from}\ \mathring{s}$, we can compute two sets of procedures: $P_{\widehat{v}} = \{\, method(x) \mid (x \Mapsto a) \in \mathring{E} \wedge a \in \mathring{r} \,\}$ and $P_{\widehat{u}} = \{\, method(y) \mid (y \Mapsto a) \in \mathring{E} \wedge a \in \mathring{s} \,\}$. These are the sets of methods containing locals that may point to $\widehat{v}$ and $\widehat{u}$ (respectively). Any method containing a write command that discharges the constraint $\widehat{v}.f \mapsto \widehat{u}$ must have a local variables pointing to both $\widehat{v}$ and $\widehat{u}$, so we only need to look at write commands from methods in the set $P_{\widehat{v}} \cap P_{\widehat{u}}$. In practice, this set is typically small enough to investigate efficiently.

5. Representing Inter-Event Control-Flow

In this section, we explain how we use *lifecycle graphs* to represent the inter-event control-flow in Android applications in a way that allows our analysis to address the challenges laid out in Section 1. We will make use of this information to check inter-event control-feasibility when we define a practical relevance relation for Android in Section 6.

For Android programs, we must consider two distinct kinds of control-flow information: intra-event control flow and inter-event control flow. Handling intra-event control-flow is the same as handling interprocedural control flow in an ordinary Java program, which is a well-understood problem. Control-flow between methods can be represented

using a call graph and control-flow within a method can be represented using a control-flow graph for the method.

Representing *inter-event* control-flow is more difficult because this information is not directly represented in the call graph. In fact, the logic for maintaining orderings among events lives in native code in the Android framework, so ordering information cannot be inferred by analyzing the Java portion of the framework alone.

Our approach to representing inter-event control-feasibility constraints is to formally define the meaning of the event ordering information that programmers have access to: the lifecycle documentation for Android components (e.g., the Activity lifecycle[3]). This documentation takes the form of *lifecycle graphs* where nodes are lifecycle event methods and directed edges express ordering constraints among the events. We have already seen how such graphs are useful in Section 2: Figure 3 specified the ordering of lifecycle events for the components used in the example and allowed the analysis to filter the set of relevant events to jump to.

To go from the documentation to a graph to a representation that can provide control-feasibility information during static analysis, we need the following:

(1) A well-defined *semantics* for Android lifecycle graphs. Our analysis can then use these graphs to filter out irrelevant transitions based on the control-feasibility condition of relevance soundness (Condition 1(b)).

(2) A specialization of generic lifecycle graphs of core Android components (e.g., Activity, Service) to a lifecycle graph for a specific *application subclass* of that component. This specialization resolves Java method overriding to make explicit the method code for each *application subclass*, and for precision, it incorporates other callbacks, such as those for handling user interface widgets.

(3) A way for the analysis to resolve lifecycle events on *object instances*. Since events in Android are methods on lifecycle objects, we need to prove that object instance $\widehat{o}_1$ *must-aliases* $\widehat{o}_2$ for two events $\widehat{o}_1.m_1$ and $\widehat{o}_2.m_2$ in order to show that $\widehat{o}_1$ and $\widehat{o}_2$ are constrained by the same lifecycle graph. If we cannot prove this fact, it is unsound to do any control-feasibility filtering because $\widehat{o}_1$ and $\widehat{o}_2$ could be different instances of the same lifecycle class (and therefore would have independent lifecycles).

Once a lifecycle graph has been specialized for a particular subclass and its events have been resolved to a particular object instance, it can be used directly by the analysis to perform control-feasibility filtering.

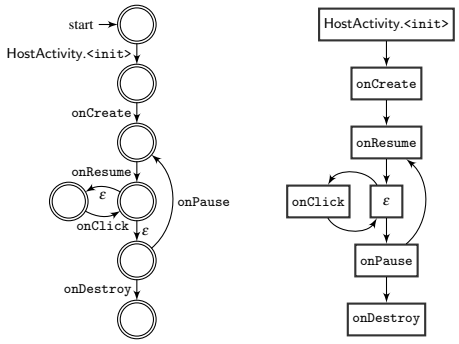
Giving Semantics to Android Lifecycle Graphs Consider the lifecycle graphs in Figure 3. These graphs specify the sequence of possible event traces for a particular Android lifecycle component (though the Android documentation never explicitly explains their meaning). If we think of the nodes of a lifecycle graph as labels for their outgoing edges,

[3] `http://developer.android.com/guide/components/activities.html#Lifecycle`

$$\begin{array}{ll} \text{lifecycle graphs} & G ::= \{\ldots, e_1 \to e_2, \ldots\} \\ \text{events} & e ::= C.m \mid \widehat{o}.m \mid \varepsilon \\ \text{classes} \quad C \qquad \text{methods} \quad m \end{array}$$

Figure 12: Components of lifecycle graphs.

we can interpret a lifecycle graph as a nondeterministic finite automata (NFA) that accepts the language of all feasible concrete event traces for its lifecycle component. In order to account for partial traces (e.g., traces ending in an exception that interrupts the lifecycle), every node must be an accepting state. For example, the lifecycle graph for the `HostActivity` class from Figure 3 (reproduced below right for convenience) corresponds to the following NFA:



In order to connect the meaning of a lifecycle graph G to our model of concrete program execution, let us consider labeling an NFA edge not with the name of its corresponding event e, but with the entry transition of the event method, which we write as $\text{entry}(e)$. This means that the strings accepted by the lifecycle NFA (which we write as $\ulcorner G \urcorner$ for a lifecycle graph G) are strings of transitions t (i.e., traces T) rather than strings of events e. We can now state a soundness condition for lifecycle graphs.

Condition 2 (Lifecycle graph soundness)**.** *If concrete execution can reach event $e \in G$, the lifecycle graph G accepts the concrete trace projected onto the transitions of the lifecycle graph. More formally, if $\langle \sigma, \ell_{\mathsf{dummy}} \rangle \xrightarrow{T \,\hat{}\, t}{}^{*} \langle \sigma', \ell' \rangle$ and event $e \in G$ where $t = \text{entry}(e)$, then $\ulcorner G \urcorner$ accepts $\text{events}(T \,\hat{}\, t, G)$.*

The function $\text{events}(T, G)$ simply projects a concrete trace T onto the transitions of a lifecycle graph G:

$$\text{events}(T, G) \stackrel{\text{def}}{=} \begin{cases} t \,\hat{}\, \text{events}(T_1) & \text{if } T = t \,\hat{}\, T_1 \text{ and } \exists\, e \in G.\ \text{entry}(e) = t \\ \text{events}(T_1) & \text{if } T = t \,\hat{}\, T_1 \text{ and } \nexists\, e \in G.\ \text{entry}(e) = t \\ [] & \text{if } T = [] \end{cases}$$

We assume that the lifecycle graphs specified in the Android documentation are sound.

Static Lifecycle Graphs: Specializing Lifecycle Graphs to Application Classes Android applications hook into the framework by subclassing special Android core components like Activity. Thus far, we have discussed events rather

abstractly, but events in Android correspond to methods on Java objects. We make this explicit by considering events as pairs of the method and the class in which it is defined (i.e., $C.m$) or as pairs of the method and the receiver object on which it is invoked (i.e., $\widehat{o}.m$) as shown in Figure 12. A well-formed lifecycle graph can consist of class-method events or object-method events; we call the former version a *static lifecycle graph* and the latter a *dynamic lifecycle graph*. We will explain the special event ε shortly.

The Android lifecycle documentation specifies the ordering of methods for core components, but we would like static lifecycle graphs specialized for the classes in the application under analysis. The specialization of lifecycle methods is straightforward by following the method resolution semantics of Java given a class hierarchy. Suppose we wish to specialize a general lifecycle graph G describing an Android core component C_{core} for an application subclass C_{app} (i.e., $C_{\text{app}} <: C_{\text{core}}$). For each event method node $C_{\text{core}}.m$ in G, we replace the node with $C.m$ where C is the class from which C_{app} inherits method m (e.g., $C = C_{\text{app}}$ if C_{app} overrides m).

An application class can also register custom callback methods that are triggered by external events such as user interaction. For example, the HostActivity class from Figure 2 extends the `OnClickListener` interface, overrides the `onClick` method, and registers itself as the listener for `onClick` events by calling `setOnClickListener(`**this**`)` at line 5. For soundness, we need to account for all such callback methods, which we could do simply by treating them as independent lifecycle components that have no ordering constraints. However, for precision it is important for the analysis to associate these callback methods with the appropriate component. The analysis should also understand that these user-triggered events can only occur during the "active" phase of the registering lifecycle component when the user can interact with the component. For Activity components, this active phase is the interval between `onResume` and `onStop`.

We incorporate custom callback events into the lifecycle graph with a simple flow-insensitive analysis. For an application class C_{app}, we consider its reachable methods in the call graph to determine what custom callbacks it may register. To represent the active phase of a class $C <:$ Activity, we introduce an ε event between `onResume` and `onClick` events as we saw in Figure 3. An ε event is a no-op event that translates to an ε-transition in the NFA formulation.

Once we have identified the set of callbacks $\{e_{\text{cb}}, \ldots\}$ that can execute during the active phase of the registering component, we "attach" each custom callback event e_{cb} to the active phase with edges $\varepsilon \to e_{\text{cb}}$ and $e_{\text{cb}} \to \varepsilon$. This models the fact that the user may or may not trigger an interaction event and that interaction events can be triggered an arbitrary number of times. This analysis is flow-insensitive because we do not consider the program point where registering methods like `setOnClickListener` are called. We also do not consider orderings between core lifecycle components (e.g., modeling the launching order of Activity's). Incorporating this information via techniques like those presented in Yang et al. [32] could improve the precision of our static lifecycle graphs.

Callback-registering methods like `setOnClickListener` may register any object with the appropriate method defined as the callback object (not just the **this** object). A common pattern is to use anonymous inner classes to implement these callbacks, as the anonymous ServiceConnection object created at line 3 of Figure 2 does. As a consequence, a lifecycle graph may need to contain methods invoked on multiple object instances (e.g., the **this** object and the anonymous inner class object). We consider this issue next.

Dynamic Lifecycle Graphs: Resolving Lifecycle Events on Object Instances A significant challenge in leveraging lifecycle information in a flow/path-sensitive analysis is to soundly account for the fact that the lifecycle applies to object instances at run time. Our approach is to resolve static lifecycle graphs to object instances in order to create a dynamic lifecycle graphs that can be directly used by the analysis. We perform this resolution on-the-fly during analysis.

To describe this approach more concretely, suppose our abstract memory is $(\mathsf{this} \mapsto \widehat{o}_1) \ast (x \mapsto \widehat{o}_2) \ast M$ for some memory M while currently analyzing code in some event method $C.m_2$; that is, we are in the lifecycle event $\widehat{o}_1.m_2$ in the corresponding dynamic lifecycle graph with some facts about objects $\widehat{o}_1$ and $\widehat{o}_2$. We would like to leverage an event-ordering constraint $C.m_1 \to C.m_2$ in the static lifecycle graph for C, but for soundness, we have to consider both $\widehat{o}_1.m_1$ and $\widehat{o}_2.m_1$ as possible events.

Our analysis handles this problem by performing an eager case split on aliasing (if we have no existing aliasing information on $\widehat{o}_1$ and $\widehat{o}_2$). That is, just before considering the event-ordering constraint, we split the abstract state into an aliased case $(\mathsf{this} \mapsto \widehat{o}_1) \ast (x \mapsto \widehat{o}_2) \ast M \wedge \widehat{o}_1 = \widehat{o}_2$ and a disaliased case $(\mathsf{this} \mapsto \widehat{o}_1) \ast (x \mapsto \widehat{o}_2) \ast M \wedge \widehat{o}_1 \neq \widehat{o}_2$. The aliased case gives us the must-alias fact that we need to soundly leverage the event-ordering constraint for control-feasibility filtering.

The eager case split means that we have a separate proof obligation for the disaliased case where we cannot use the event-ordering constraint in the static lifecycle graph. However, as we will see in more detail in Section 6, applying data-relevance often allows us to quickly rule out this case. In the common case that the relevant commands in $C.m_1$ and $C.m_2$ are writes to **this** of the lifecycle object, we can use data-relevance to rule out $\widehat{o}_2.m_1$ (in the disaliased $\widehat{o}_1 \neq \widehat{o}_2$ case). Even if the relevant writes are through non-**this** pointers (e.g., `p.f = ⋯`), our precise reasoning about aliasing and strong updates typically handles this disaliased case quickly.

6. A Jumping Analysis for the Heap and Events

In this section, we bridge the gap between theory and practice by combining the jumping framework from Section 3, the data-relevance relation for the heap from Section 4, and the formalization of Android lifecycle graphs in Section 5 to design HOPPER, a practical jumping analysis for analyzing event-driven Android programs. We achieve this by devising a sound relevance relation for THRESHER [7], a precise backward analysis that tightly integrates the results of an up-front points-to analysis to refute separation logic queries.

We have given the intuition for the jumping strategy that our relevance relation implements in Sections 1 and 2: when the analysis reaches an event boundary, it identifies events that contain data-relevant commands, filters the set of data-relevant events using control-feasibility constraints based on Android lifecycle information, then jumps to the remaining events. To realize this vision, we need to address one remaining issue: using the semantics of lifecycle graphs to perform control-feasibility filtering. We explain how we solve this issue in Section 6.1 before presenting an algorithm for computing relevant transitions that utilizes our solution in Section 6.2.

6.1 Using Lifecycle Graphs for Control-Feasibility Filtering

To utilize Android lifecycle information in our relevance relation, we must connect the meaning of lifecycle graphs from Section 5 (Condition 2) to the control-feasibility condition of relevance soundness (Condition 1(b)). To maintain precision while jumping from one event to another, we must ensure that we only perform jumps that respect the ordering constraints encoded in Android lifecycle graphs (while simultaneously considering the necessary interleavings in order to be sound). Our solution is to utilize the reachability and *post-dominance* information encoded in the lifecycle graph. Since our analysis is backward, only jumps from the current program point to a preceding transition in a concrete execution trace ending at the current program point are control-feasible. Thus, we can filter a set of possibly relevant transitions using control-feasibility by considering backward reachability in the lifecycle graph.

If some event e is not backward-reachable from the current event e_{cur}, then we know that no concrete trace ending in e_{cur} can possibly have visited e first (following the semantics of lifecycle graphs as concrete trace-accepting NFAs in Condition 2). Thus, we can prune all events that are not backward-reachable from e_{cur} in the lifecycle graph G to produce a pruned lifecycle graph G' where e_{cur} is a leaf node.

For example, we can use this technique to reason that if the analysis is currently in the `onClick` event of Figure 3, the `onDestroy` event has not yet occurred in the current lifecycle. If we prune nodes and edges not backward reachable from the current node `onClick`, we can prune the `onDestroy` event. We do not need to consider jumps from e_{cur} to pruned events.

The analysis can further refine the possible jump targets using postdominance on the pruned graph G'. Consider the postdominance tree rooted at e_{cur}; that is, a tree where each node is the immediate postdominator of its children. For any set of potentially relevant events E, we only need to consider the smallest set $E' \subseteq E$ such that E' postdominates E. As a consequence, for all $e \in E$, there is an $e' \in E'$ such that e' is between e_{cur} and e in the postdominator tree rooted at e_{cur}. The correctness of this reasoning follows directly from the meaning of lifecycle graphs and the definition of postdominance: if e_{cur} postdominates e' and e' postdominates e, we can conclude that every trace accepted by the lifecycle NFA that visits e_{cur} always visits e' beforehand without visiting e in between.

To give a more concrete example using the HostActivity lifecycle graph in Figure 3, we would like to able to determine that if the analysis is currently in the `onClick` event and we know that only the `onCreate` and HostActivity.`<init>` events are relevant, then we only need to jump to `onCreate`. We can derive this fact by demonstrating that `onClick` postdominates `onCreate` and `onCreate` postdominates HostActivity.`<init>` in G'.

6.2 An Effective Relevance Relation for Android

Finally, we present our algorithm for computing Android-specialized relevance information (Figure 13) and argue that our algorithm is sound with respect to relevance soundness (Condition 1). The algorithm implements the $\langle R, \ell \rangle \rightsquigarrow T_{\text{rel}}$ judgment form for relevance relations and is executed each time the A-JUMP rule is applied.

In the usual case where the current program label ℓ_{cur} is not the entry label of an event, the algorithm behaves like a standard path-sensitive backward analysis by choosing to visit the predecessor labels of the current program label next (lines 2–4). Clearly, this satisfies relevance soundness by satisfying the control-feasibility condition (Condition 1(b)).

In the case that the current program label is the entry label of an event, we perform jumps to a computed set of relevant transitions using the data-relevance and control-feasibility constraints described previously. Specifically, the algorithm computes the set of data-relevant events that may write to the current abstract state (lines 5–20) and then filters this set of events using control-feasibility information from the lifecycle graph of the current event (lines 22-37).

First, the algorithm computes the set of data-relevant transitions T_{rel} for R_{cur} using the points-to analysis, as we have explained in Section 4.2. In principle, the algorithm could return the set T_{rel} and still be sound by satisfying relevance soundness Condition 1(a), but this would be imprecise because it would not take the ordering of events in the lifecycle graph into account. The algorithm thus walks backward from the calling method of each relevant transition t_{rel} (given by method(t_{rel})) in the call graph until it reaches an event

Require: Current abstract state R_{cur}
Require: Current label ℓ_{cur}
Require: Program transition relation P
Require: Call graph CG
Ensure: Returned transition set T_{rel} satisfies Condition 1
1: // not at event entry, follow predecessors
2: **if** ℓ_{cur} is not the entry label of an event **then**
3: **return** preds(ℓ_{cur}, P)
4: **end if**
5: // at event entry, get data-relevant events
6: $T_{\text{rel}} \leftarrow$ dataRel(R_{cur}) // compute $R_{\text{cur}} \rightsquigarrow T_{\text{rel}}$
7: $E_{\text{rel}} \leftarrow \emptyset$ // events leading to a relevant transition
8: **for all** $t_{\text{rel}} \in T_{\text{rel}}$ **do**
9: $W \leftarrow$ [method(t_{rel})] // method worklist
10: $V \leftarrow \emptyset$ // track visited methods to handle CG cycles
11: **while** $W \neq \emptyset$ **do**
12: Remove m from W
13: **if** m is event **then**
14: $E_{\text{rel}} \leftarrow \{ m \} \cup E_{\text{rel}}$
15: **else if** $m \notin V$ **then**
16: Add preds(m, CG) to W
17: **end if**
18: $V \leftarrow \{ m \} \cup V$
19: **end while**
20: **end for**
21: // filter data-relevant events with a lifecycle graph
22: $e_{\text{cur}} \leftarrow$ event(ℓ_{cur})
23: $G \leftarrow$ specializeLifecyleGraph(class(e_{cur}), CG)
24: $E_{\text{inG}} \leftarrow \{ e \mid e \in E_{\text{rel}} \wedge e \in G \}$
25: $E_{\text{notinG}} \leftarrow \{ e \mid e \in E_{\text{rel}} \wedge e \notin G \}$
26: $E_{\text{feas}} \leftarrow \emptyset$ // data-relevant/control-feasible events in G
27: $W \leftarrow$ [e_{cur}] // lifecycle graph event worklist
28: $V \leftarrow \emptyset$ // track visited events to handle cycles in G
29: **while** $W \neq \emptyset$ **do**
30: Remove e from W
31: **if** $e \in E_{\text{inG}}$ **then**
32: $E_{\text{feas}} \leftarrow \{ e \} \cup E_{\text{feas}}$
33: **else if** $e \notin V$ **then**
34: Add preds(e, G) to W
35: **end if**
36: $V \leftarrow \{ e \} \cup V$
37: **end while**
38: **return** exitTrans($E_{\text{feas}} \cup E_{\text{notinG}}$)

Figure 13: An algorithm for selecting relevant transitions to visit in event-driven, heap-manipulating Android programs.

on each path (loop from lines 8–20). The resulting set of data-relevant events E_{rel} contains the set of all events whose execution might lead to a relevant transition. Returning the exit transition of each of these events E_{rel} would also satisfy relevance soundness via a combination of Condition 1(b) and (a) because by construction of E_{rel}, these exit transitions collectively postdominate all relevant transitions.

However, we can gain additional precision by removing events from T_{rel} based on control-feasibility information from the lifecycle graph, which is what the algorithm does next. Lines 22 and 23 compute a lifecycle graph specialized for the class of the current event e_{cur}, as we have described in Section 5. The algorithm then partitions the set of relevant events E_{rel} based on their presence in the lifecycle graph (lines 24–25). It does this because only the events E_{inG} that are in the lifecycle graph should be filtered in the subsequent step—the events in E_{notinG} are unordered with respect to e_{cur} and the algorithm must return all of them for soundness.

The loop from lines 29–37 performs control-feasibility filtering on nodes in the lifecycle graph. This loop computes a subset E_{feas} of E_{inG} that must be returned for soundness. The loop walks backward from the current event e_{cur} in the lifecycle graph G and stops each time it reaches a relevant event. The construction of E_{feas} ensures that at the end of the loop, relevant events that are not backward reachable from e_{cur} will be excluded from the set E_{feas}, and as will events postdominated by both e_{cur} and some other relevant event. We have argued for the soundness of excluding events based on backward reachability and postdominance in the lifecycle graph in Section 6.1.

Finally, the algorithm takes the union of the lifecycle graph control-feasible relevant events E_{feas} and the unordered relevant events E_{notinG} and returns their exit transitions as the set of transitions that must be visited (line 38). The set $E_{\text{feas}} \cup E_{\text{notinG}}$ is a subset of the set E_{rel} whose exit transitions already satisfy relevance soundness. We have only removed events from this set by soundly filtering based on lifecycle control-feasibility information, so returning the exit transitions of $E_{\text{feas}} \cup E_{\text{notinG}}$ also satisfies relevance soundness.

7. Empirical Evaluation

In order to evaluate the effectiveness of jumping analysis, we sought to test the following experimental hypothesis:

> *Jumping is a scalable approach to flow/path-sensitive inter-event analysis.*

We hypothesize that augmenting a state-of-the-art path-sensitive analysis with jumping increases precision by allowing the analysis to reason about event orderings, yet limits the number of event orderings that must be considered enough to make analysis tractable.

Experimental Setup In order to test our hypotheses, we chose to evaluate jumping analysis by attempting to prove the absence of null dereferences in event-driven Android programs. We chose this client because null dereferences are a common problem in real-world Android apps, and the event-driven nature of Android makes precisely verifying the absence of null dereferences a significant challenge for analyses (see Section 2.1 for a more detailed discussion of this client). We implemented the practical jumping analysis described in Section 6 in the HOPPER [1] tool, a variant of the THRESHER [7] tool that builds on the WALA [3] analysis infrastructure and the Z3 [14] SMT-solver. HOPPER extends THRESHER by adding the ability to perform jumps, but the tools are otherwise identical.

The core of THRESHER is an engine for refuting queries written in separation logic. Clients are implemented as lightweight add-ons that take a program as input and emit separation logic queries for the core refuter to process. We extended THRESHER/HOPPER with a new client for checking null dereferences. The client leverages `@Nonnull` annotations inferred by the NIT [2] tool to eliminate easy cases where non-nullness of fields, function return values, or function parameters is a flow-insensitive property. For each non-static field read/write `x.f` or function call `x.m()` in the the program, the client emits the necessary bug precondition $x \mapsto \mathsf{null}$ as a query to refute in order to prove dereference safety.

We gave THRESHER and HOPPER a maximum budget of 10 seconds to refute each query. This budget includes all aspects of analysis time except for the initial call graph construction (including HOPPER-specific aspects such as computing data-relevance, specialization of lifecycle graphs, etc.). We chose this budget through trial and error—we found that larger budgets did not allow either tool to find appreciably more refutations, whereas smaller budgets caused too many timeouts for both tools. In the case that a tool cannot refute a query within the budget, a timeout was declared and the dereference was reported as a potential bug. We ran all experiments in single-threaded configuration on a Mac desktop machine running Mac OS 10.10.2 with 64GB of RAM and 3.5GHz Intel Xeon processors.

Android applications can make use of concurrency—events execute atomically if they run on the same thread, but the execution of events can interleave if the events execute on separate threads. In addition, app developers can use Java threads for multithreaded execution in the usual way. THRESHER and HOPPER do not soundly account for either of these features, as both tools assumes that all events execute atomically on a single thread. Both tools also do not soundly handle reflection and native code for which we do not have handwritten stubs—these constructs are treated as no-ops.

Representing Android Event Dispatch Instead of analyzing a synthetic harness that models the event dispatch performed by the Android framework, THRESHER and HOPPER analyze the actual logic for event dispatch in the Android framework source code. To allow this, we pre-process each app we analyzed with DROIDEL [8], a tool that explicates key reflective calls in the Android framework by replacing them with calls to automatically generated, application-specific stub methods. We then use the `ActivityThread.main` method of the framework as a single entrypoint for call graph construction. There are several advantages to analyzing the actual event dispatch code instead of using a harness: (1) we do not have to worry about soundly and precisely modeling the execution context of events, which can be a significant challenge, and (2) generating a harness that precisely represents all ordering constraints is impractical, as we have already argued in Section 1.

7.1 Proving Dereferences Safe with Jumping

We ran both THRESHER and HOPPER on the corpus of ten open-source Android apps shown in Figure 14. The apps range in size from 3K source lines of code to 57K source lines of code. Since the primary challenge of analyzing these apps comes from considering interleavings of their component lifecycles, we also report the number of core lifecycle components (i.e., Application, Activity, Fragment, Service, and ContentProvider subclasses) and the total number of events in each app. Our analysis must consider the possibility of interleavings between events of different components for soundness, but must preserve the ordering of events within the lifecycle of a single component for precision. Recall from Section 1 that the size of a reified harness that considers the interleavings between just a single instance per component is exponential in the number of components.

The "Unsafe Derefs" columns of Figure 14 summarize the results of proving null dereference safety on our benchmark apps with NIT, THRESHER and HOPPER. Each column reports the number of unproven dereferences after running the tool (where 0 represents proving all dereferences safe, so lower is better). The results show that although about a third of the dereferences can be proven safe using the flow-insensitive analysis of NIT, the path-, flow-, and context-sensitive THRESHER analysis was significantly more precise (providing evidence that precision beyond flow-insensitivity is necessary for proving dereference safety in Android apps). The **Hop Impr** column gives the percentage reduction in unsafe dereferences achieved by running HOPPER (where 100% represents proving all remaining dereferences safe, so higher is better). HOPPER substantially improved on the already-significant precision of THRESHER—on average, HOPPER reduced the number of dereferences unproven by THRESHER by more than half.

The difference between HOPPER and THRESHER is that the jumping capability of HOPPER enables precise inter-event analysis, as we predicted in our experimental hypothesis. We noticed that when THRESHER reaches an event boundary without proving safety, it continues precise backward analysis of the event dispatch code of the Android framework and (almost always) times out without finding a proof. By contrast, HOPPER jumps from an event boundary to a (typically) small set of relevant events and is frequently able to prove safety based on precise and tractable inter-event reasoning.

The final **Total Hop Proven** column shows that for every benchmark, HOPPER proved at least 90% of the dereferences safe (92% safe on average). We note that previous state-of-the-art work in null dereference checking for ordinary, non-event-driven Java programs (e.g., [22–25]) reports proving 84-91% of dereferences safe on average. Achieving similar precision results in the presence of the formidable scalability challenges introduced by an event-driven setting is a significant advance.

Benchmark Size				Unsafe Derefs				HOPPER Effectiveness		
Bench	**KLOC**	**Com**	**Evt**	**Deref**	**Nit**	**Thr**	**Hop (Impr %)**	**Proven (%)**	**Time (s)**	**Time/deref (s)**
DrupalEditor	3	10	127	790	574	157	66 (58)	92	717	0.9
NPR	5	14	120	829	617	181	50 (72)	94	691	0.8
Last.fm♠	11	12	174	4840	3528	950	474 (50)	90	5922	1.2
DuckDuckGo	13	34	272	1901	1277	514	144 (72)	92	1751	0.9
GitHub	19	70	572	3603	2520	583	284 (51)	92	3749	1.0
SeriesGuide♠	32	80	871	8184	5438	987	625 (37)	92	13929	1.7
ConnectBot♠	33	13	201	2190	1562	316	75 (76)	97	880	0.4
TextSecure	38	63	588	5921	3643	621	272 (56)	95	2306	0.4
K-9Mail	55	52	750	19032	11968	3067	1988 (35)	90	27197	1.2
WordPress♠	57	98	1325	15000	9735	2402	1341 (44)	91	18596	0.9
Total	266	446	5000	62290	40862	9778	5319 (53)	92	75738	0.9

Figure 14: Proving dereference safety in event-driven Android apps with HOPPER. The "Benchmark Size" column grouping gives the number of (thousands of) lines of application source code (**KLOC**), lifecycle components (**Com**), and events (**Evt**) for each benchmark. The "Unsafe Derefs" column grouping lists the number of possibly-unsafe dereferences in each app before analysis (**Deref**) followed by the number remaining after running NIT (**Nit**), THRESHER (**Thr**), and HOPPER (**Hop**). The HOPPER column also lists the percentage reduction in unproven derefs of HOPPER over THRESHER (**Impr %**). The final column grouping gives the percentage of derefs proven safe by HOPPER (**Proven (%)**), the time taken to process all derefs (**Time (s)**), and the time taken per deref (**Time/deref (s)**). The "Total" row gives the sum of all numeric rows and the geometric mean of the **(Impr %)**, **Proven (%)**, and **Time/deref** rows. ♠'s indicate benchmarks where our partial manual triaging revealed a true bug.

7.2 Manual Triaging of Alarms

To understand why HOPPER sometimes fails to prove safety, we manually triaged a sample of 20 unproven dereferences from each of our 10 benchmark applications (a total of 200 alarms). We classified the unproven dereferences into three categories (a) true bugs, (b) scalability issues, or (c) precision issues. We placed a dereference into the true bugs category if we found a concretely feasible sequence of events would lead the application to throw a NullPointerException. We classified a dereference as a scalability issue if we determined that HOPPER possessed the necessary precision to prove the dereference safe, but was not able to do so within the 10 second budget. Finally, we labeled a dereference as a precision issue if HOPPER did not have the precision required to prove the query correct. This category includes both analysis imprecision (e.g., loop invariant inference, container abstraction) as well as modeling imprecision (e.g., Android UI models, Android/Java reflection and native code).

The results from our manual triaging are shown inset. In the 200 alarms we examined, most dereferences that cannot be proven safe are due to precision issues (172). Of these 172 alarms, 132 would require more precise modeling of the Android framework and 41 are due to more fundamental analysis imprecision. Many of Android modeling issues are additional constraints on the interaction between different lifecycle components that we do not account for. For example, proving safety of some dereferences required understanding details such as the order in which Activity's launch each other or the fact that a callback on a Button cannot be invoked if the `visible` attribute of the Button is set to false. Handling all of the corner cases of the complex Android framework is a challenging task that we leave to future work.

(a) Bug	(b) Scalability	(c) Precision
11	17	172

Nearly all of the the analysis imprecision issues stem from imprecise abstraction of containers and strings. Both of these precision problems are orthogonal to HOPPER's approach to analysis of event-driven programs and could in principle be addressed by enhancing HOPPER with better abstractions or solvers (e.g., [16] for containers and [20] for strings).

The fact that only 17 of the 200 unproven dereferences we examined could not be proven safe due to scalability issues strengthens our conviction that jumping is an effective approach for tractable analysis of event-driven systems. Though HOPPER is not perfect, it proves an impressive 92% of the dereferences it encounters. The vast majority of proof failures are due to our incomplete modeling of Android rather than scalability issues.

Bugs Found We found eleven bugs in four different apps: Last.fm (1), Seriesguide (5), ConnectBot (4) and WordPress (1). The bug in WordPress had already been eliminated by the developers, though in an indirect way (replacing the functionality in the buggy class with an entirely new class). We sent pull requests fixing the bugs in each of the remaining projects. The developer of SeriesGuide and ConnectBot accepted all of our pull requests. The developers of Last.fm have not yet responded to our pull requests. This project is updated infrequently and has a backlog of pending pull requests.

Of the eleven bugs that we found, five of them involved misunderstanding or misusing the Android lifecycle in some way. This strengthens our belief that the lifecycle is a source of confusion for developers that would be well-served by better analysis tools. We further note that four of the five bugs involved interactions between the lifecycles of different components. These bugs could not be found by an unsound approach that models the lifecycle of each component, but does not consider interleaving lifecycles of different components.

Other Instantiations of the Framework In addition to using jumping for effective analysis of event-driven programs, we have also considered an alternate instantiation of the jumping framework for analysis of ordinary (that is, non-event-driven) Java programs. This instantiation relies on data-relevance information alone and chooses to jump only when the safety of the query may rely on a flow-insensitive invariant established earlier in the program. We applied this instantiation to the problem of proving downcast safety in the DaCapo2006 [5] benchmarks and observed a similar precision improvement of HOPPER over THRESHER. See Blackshear [6, Chapter 6] for more details on this instantiation of the framework.

8. Related Work

Static Analysis of Android Applications Numerous techniques have considered static analysis of Android apps, but to the best of our knowledge, few have tackled the problem that we address in this paper: soundly modeling the interleaving of different component's lifecycles. The harness method generated by the state-of-the-art FLOWDROID [18] tool soundly models the sequential execution of component lifecycles, but not their interleaving. This unsound modeling avoids the cost of computing a product graph as described in Section 1, but will miss bugs like the one explained in Section 2 along with the five lifecycle-sensitive bugs we found in Section 7.

ANADROID [21] is the only tool we are aware of that explicitly claims to handle interleavings between lifecycles of different components. Their *entry point saturation* technique efficiently computes a fixed point over all possible event ordering. However, this computation does not take intra-lifecycle event orderings into account and thus will lose precision. We found this kind of precision to be crucially important in Section 7—HOPPER's improvement over THRESHER comes entirely from more precise and scalable inter-event reasoning.

Analysis of Asynchronous and Event-Driven Programs Identifying a small set of commands relevant to the query and their corresponding events using data-relevance exploits the fact that the data dependencies of a program are often less complex than its control dependencies in practice. Recent techniques for concurrent program verification [17] and bug finding [10] have used a similar insight: an effective way to prevent the complexity of a concurrent program analysis (static or dynamic) from growing exponentially in the number of threads is to design the analysis around tracking data dependencies rather than control dependencies. Jumping based on a relevance relation allows the analysis to exploit both data-relevance and control-feasibility information to improve scalability, and jumping can be applied in sequential, concurrent, and event-driven settings.

Jhala et al. show that the IFDS framework can be extended to enable analysis of event-driven programs and present a goal-directed algorithm for proving safety properties in their extended framework [19]. Their focus is on handling unordered events whose execution may interleave, whereas we focus on the problem of preserving the ordering between lifecycle events whose execution is atomic.

Program Slicing Identifying commands that may affect a query using a data-relevance relation is closely related to program slicing [31]. Our approach is most closely related to semantic slicing [9, 28], since we perform a slice with respect to an abstract state rather than a seed command. A key difference between our data-relevance relation and semantic slicing is that we only compute the first step of a slice (i.e., the *immediately* relevant commands) rather than computing a transitive closure of relevant commands as a complete slice does. In many cases, taking a complete slice includes the majority of the program and is prohibitively expensive to compute. Our approach is much more efficient than the obvious approach of taking a full slice with respect to the query and analyzing the sliced program.

9. Future Work

In future work, we plan to demonstrate the generality of our framework for jumping analyses by instantiating it with new data-relevance/control-feasibility relations and applying it to different problem domains. A logical next step would be trying to adapt jumping analysis to the analysis of concurrent programs (both event-driven and threaded). For example, we could combine the data-relevance relation from Section 4 with a control-feasibility filter that leverages information about **synchronized** blocks and graphs of thread spawning structure (in a manner similar to our utilization of Android lifecycle graphs in Section 6). We are hopeful that such a strategy would greatly enhance the scalability of the analysis by decreasing the number of thread interleavings that must be considered while jumping, just as the analysis described in this paper significantly reduces the number of event orderings that the analysis needs to consider.

10. Conclusion

We have presented *jumping*, a general framework for selective control-flow abstraction that can be applied to any goal-directed backward abstract interpretation. The key idea behind jumping is an intertwining of data-relevance and control-feasibility reasoning that enables an analysis to soundly jump

to a small set of relevant program transitions. We instantiated our jumping framework with HOPPER, a tool for performing precise and scalable reasoning of event-driven Android apps by leveraging heap dependencies for data-relevance and Android lifecycle graphs for control-feasibility. HOPPER proved the safety of 90–97% of dereferences in the apps it analyzed and allowed us to isolate eleven real bugs.

Acknowledgments

We thank Pavol Černý and the University of Colorado Programming Languages and Verification Group (CUPLV) for insightful discussions, as well as the anonymous reviewers for their helpful comments. This material is based on research sponsored in part by DARPA under agreement numbers FA8750-14-2-0039 and FA8750-14-2-0263, the National Science Foundation under CAREER grant number CCF-1055066, and a Facebook Graduate Fellowship. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

[1] Hopper. `https://github.com/cuplv/hopper`, 2015.

[2] Nit: Nullability inference tool. `http://nit.gforge.inria.fr/`, 2015.

[3] T.J. Watson Libraries for Analysis (WALA). `http://wala.sf.net`, 2015.

[4] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2005.

[5] Stephen M. Blackburn, Robin Garner, Chris Hoffmann, Asjad M. Khan, Kathryn S. McKinley, Rotem Bentzur, Amer Diwan, Daniel Feinberg, Daniel Frampton, Samuel Z. Guyer, Martin Hirzel, Antony L. Hosking, Maria Jump, Han Bok Lee, J. Eliot B. Moss, Aashish Phansalkar, Darko Stefanovic, Thomas VanDrunen, Daniel von Dincklage, and Ben Wiedermann. The DaCapo benchmarks: Java benchmarking development and analysis. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, 2006.

[6] Sam Blackshear. *Flexible Goal-Directed Abstraction*. PhD thesis, University of Colorado Boulder, 2015.

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

[8] Sam Blackshear, Alexandra Gendreau, and Bor-Yuh Evan Chang. Droidel: A general approach to Android framework modeling. In *State of the Art in Program Analysis (SOAP)*, 2015.

[9] François Bourdoncle. Abstract debugging of higher-order imperative languages. In *Programming Language Design and Implementation (PLDI)*, 1993.

[10] Sebastian Burckhardt, Pravesh Kothari, Madanlal Musuvathi, and Santosh Nagarakatte. A randomized scheduler with probabilistic guarantees of finding bugs. In *Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2010.

[11] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods (NFM)*, 2015.

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

[13] Patrick Cousot, Radhia Cousot, Manuel Fähndrich, and Francesco Logozzo. Automatic inference of necessary preconditions. In *Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2013.

[14] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008.

[15] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Programming Language Design and Implementation (PLDI)*, 2008.

[16] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *Principles of Programming Languages (POPL)*, 2011.

[17] Azadeh Farzan, Zachary Kincaid, and Andreas Podelski. Inductive data flow graphs. In *Principles of Programming Languages (POPL)*, 2013.

[18] Christian Fritz, Steven Arzt, Siegfried Rasthofer, Eric Bodden, Alexandre Bartel, Jacques Klein, Yves le Traon, Damien Octeau, and Patrick McDaniel. FlowDroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps. In *Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[19] Ranjit Jhala and Rupak Majumdar. Interprocedural analysis of asynchronous programs. In *ACM Symposium on Principles of Programming Languages (POPL)*, 2007.

[20] Adam Kiezun, Vijay Ganesh, Shay Artzi, Philip J. Guo, Pieter Hooimeijer, and Michael D. Ernst. HAMPI: A solver for word equations over strings, regular expressions, and context-free grammars. *ACM Trans. Softw. Eng. Methodol.*, 21(4), 2012.

[21] Shuying Liang, Andrew W. Keep, Matthew Might, Steven Lyde, Thomas Gilray, Petey Aldous, and David Van Horn. Sound and precise malware analysis for Android via pushdown reachability and entry-point saturation. In *Security and Privacy in Smartphones and Mobile Devices (SPSM@CCS)*, 2013.

[22] Alexey Loginov, Eran Yahav, Satish Chandra, Stephen Fink, Noam Rinetzky, and Mangala Gowri Nanda. Verifying dereference safety via expanding-scope analysis. In *Software Testing and Analysis (ISSTA)*, 2008.

[23] Ravichandhran Madhavan and Raghavan Komondoor. Null dereference verification via over-approximated weakest preconditions analysis. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, 2011.

[24] Amogh Margoor and Raghavan Komondoor. Two techniques to improve the precision of a demand-driven null-dereference verification approach. *Sci. Comput. Program.*, 98, 2015.

[25] Mangala Gowri Nanda and Saurabh Sinha. Accurate interprocedural null-dereference analysis for Java. In *International Conference on Software Engineering (ICSE)*, 2009.

[26] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[27] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Logic in Computer Science (LICS)*, 2002.

[28] Xavier Rival. Understanding the origin of alarms in Astrée. In *Static Analysis (SAS)*, 2005.

[29] Yannis Smaragdakis, George Kastrinis, and George Balatsouras. Introspective analysis: context-sensitivity, across the board. In *Conference on Programming Language Design and Implementation (PLDI)*, 2014.

[30] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *Programming Language Design and Implementation (PLDI)*, 2006.

[31] Frank Tip. A survey of program slicing techniques. *J. Prog. Lang.*, 3(3), 1995.

[32] Shengqian Yang, Dacong Yan, Haowei Wu, Yan Wang, and Atanas Rountev. Static control-flow analysis of user-driven callbacks in Android applications. In *International Conference on Software Engineering (ICSE)*, 2015.

[33] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in Datalog. In *Conference on Programming Language Design and Implementation (PLDI)*, 2014.

Fissile Type Analysis: Modular Checking of Almost Everywhere Invariants

Devin Coughlin Bor-Yuh Evan Chang
University of Colorado Boulder
{devin.coughlin,evan.chang}@colorado.edu

Abstract

We present a generic analysis approach to the *imperative relationship update problem*, in which destructive updates temporarily violate a global invariant of interest. Such invariants can be conveniently and concisely specified with dependent refinement types, which are efficient to check flow-insensitively. Unfortunately, while traditional flow-insensitive type checking is fast, it is inapplicable when the desired invariants can be temporarily broken. To overcome this limitation, past works have directly ratcheted up the complexity of the type analysis and associated type invariants, leading to inefficient analysis and verbose specifications. In contrast, we propose a *generic lifting* of modular refinement type analyses with a symbolic analysis to efficiently and effectively check concise invariants that hold *almost everywhere*. The result is an efficient, highly modular flow-insensitive type analysis to *optimistically* check the preservation of global relationship invariants that can fall back to a precise, disjunctive symbolic analysis when the optimistic assumption is violated. This technique permits programmers to temporarily break and then re-establish relationship invariants—a flexibility that is crucial for checking relationships in real-world, imperative languages. A significant challenge is selectively violating the global type consistency invariant over heap locations, which we achieve via *almost type-consistent heaps*. To evaluate our approach, we have encoded the problem of verifying the safety of reflective method calls in dynamic languages as a refinement type checking problem. Our analysis is capable of validating reflective call safety at interactive speeds on commonly-used Objective-C libraries and applications.

Categories and Subject Descriptors F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs

Keywords almost everywhere invariants; dependent refinement types; almost type-consistent heaps; reflection; Objective-C

1. Introduction

Modular verification of just about any interesting property of programs requires the specification and inference of invariants. One particularly rich mechanism for specifying such invariants is dependent *refinement types* [38], which have been applied extensively to, for example, checking array bounds [12, 31, 32, 37]. Refinement types are compelling because they permit the specification of *relationships* in a type system framework that naturally admits modular checking. For example, a modular refinement type system can relate an array with an in-bounds index or a memory location with the lock that serializes access to it.

A less well-studied problem that also falls into a refinement type framework is modularly verifying the safety of reflective method call in dynamic languages. Reflective method call is a dynamic language feature that enables programmers to invoke a method via a run-time string value called a *selector*. In many languages, these calls have become commonplace in libraries as a mechanism to decouple client and framework code. Yet while they are powerful and convenient, reflective method calls introduce a new kind of run-time error: the method named by the selector may not exist.

Our key observation about modularly verifying reflective call safety is that the essential property to capture and track through the program is the relationship between a *responder* (the object on which the method is invoked) and a valid selector. In particular, the verifier does not need to determine the actual string value as long as it can ensure that the "responds-to" relationship between the object and the selector holds at the reflective call site. This observation is crucial because the point where this relationship invariant is established and where the selector is known (usually in client code) is likely far removed from the point where the reflective call is performed and at which the invariant is relied upon (usually in framework code).

The imperative relationship update problem. A significant challenge in checking relationship invariants in dynamic, imperative programming languages is reasoning precisely about destructive updates of related storage locations, such as local variables on the stack and object fields on the heap. To illustrate this issue, consider an object `o` with an (object) invariant specifying a relationship between two fields `o.f` and `o.g`. For example, `o.f` could point to an object that should respond to a selector stored in `o.g`. We can then use this invariant to show that a reflective method call on `o.f` using selector `o.g` is safe. Again, there are many other interesting properties that require specification of relationships (e.g., `o.f` is an array with an in-bounds index stored in `o.g`, or `o.f` is a monitor object guarded by the lock stored in `o.g`).

Now consider a call `o.updateRelation(p)` to a simple method `updateRelation` that updates fields `o.f` and `o.g` with the corresponding fields from the argument `p`:

def `updateRelation(x) =` ❶ **self**`.f = x.f;` ❷ **self**`.g = x.g;` ❸

where three program points ❶ to ❸ are marked explicitly. Suppose that fields `p.f` and `p.g` of object `p` have the corresponding relationship with each other as those fields of `o` (e.g., `p.f` responds to `p.g`). Then it is clear that after a call to `o.updateRelation(p)`,

POPL '14, January 22–24, 2014, San Diego, CA, USA.

ACM 978-1-4503-2544-8/14/01…$15.00.
http://dx.doi.org/10.1145/2535838.2535855

the relationship invariant between `o.f` and `o.g` should still hold. This property seems obvious, but two issues make it challenging to verify: (1) a standard, flow-insensitive type analysis by itself is insufficient and (2) sound symbolic reasoning about potential aliasing requires an expensive disjunctive analysis.

Issue 1 arises because the first assignment violates the type refinement on **self**`.f`. In our reflective call safety example, after the assignment, the new **self**`.f` (holding the value passed in `x.f`) might not respond to the old **self**`.g`. Switching the order of assignments does not help: then after the first assignment, the old **self**`.f` might not respond to the new **self**`.g`.

Problems arising from imperative relationship updates have been studied in the context of array bounds checking [12, 32, 37]. To deal with Issue 1, these type systems either require simultaneous relationship updates [12] or drop flow-insensitivity altogether and move to a flow-sensitive treatment of typing [32, 37]. In contrast to a flow-insensitive invariant, the type of a location in a flow-sensitive treatment is not fixed globally and is permitted to vary.

We argue that for the relationship update problem, computing new types at every program point is *overly pessimistic*. Although the ideal flow-insensitive type invariant imposed by the refinement on the **self**`.f` field is broken at program point ❷, it is restored almost immediately at point ❸. While the blocks of code between violations and restorations (e.g., points ❶ to ❸) require at least flow-sensitivity, a fast flow-insensitive analysis can still be effective in checking that the relationship invariant is preserved everywhere else.

Issue 2 arises in the above example because we must reason about (potentially) two concrete objects: the object pointed to by **self** and the (possibly same) object pointed to by `x`. While this code satisfies the desired property regardless of whether or not **self** and `x` are aliases, the analysis must consider both cases. Consider a slight variant where, for example, an assignment **self**`.f = null` occurs before point ❶. A sound analysis must detect a bug in this variant (when **self** and `cb` are aliases).

Prior approaches (e.g., [1, 2, 18, 32]) enable strong updates in some situations by reasoning about concrete objects one-at-a-time. While one-at-a-time reasoning is sound, it is insufficiently precise for even this unextraordinary example. In particular, a disjunctive analysis (e.g., a path-sensitive analysis) must reason about two cases: (1) where there is one concrete object—**self** and `x` are aliases—and (2) where there are two objects—**self** and `x` are not aliases.

We also argue that directly augmenting a type system to handle temporary invariant violations or other flow-analysis issues is *overly specific*. The imperative relationship update problem applies to checking just about any property that requires relational invariants. In contrast to the aforementioned works, the essence of our approach is to mix a property-specific refinement type system with a generic, symbolic flow analysis to handle temporary violations of the type invariant. The symbolic analysis performs execution-based reasoning and decouples issues like flow- and path-sensitivity from the particular type abstraction of interest.

Fissile type analysis. We propose flow-insensitive storage location (FISSILE) type analysis—a framework for enriching a dependent refinement type system to tolerate temporary violations via a generic, symbolic, separation logic-based, flow analysis. The result is an analysis that soundly checks a type invariant that holds almost everywhere. When the fast, flow-insensitive analysis detects an invariant violation, FISSILE type analysis *splits* the type environment (which relates storage locations to storage locations) into two components: (1) relationships between symbolic values (i.e., refinement types lifted to values) and (2) the locations where those values are stored (i.e., a symbolic memory). The symbolic analysis permits *bounded violation* of the storage location property and takes over until the type invariant is restored, at which point the fast flow-insensitive analysis resumes. As illustrated in the above example, we must support bounded relationship violations not only among local variables but also among a number of heap locations. Supporting bounded violations of heap locations is a significant challenge.

When applied to relationships that hold almost everywhere, FISSILE type analysis addresses the imperative relationship update problem while still maintaining two key benefits of flow-insensitivity:

1. It is nearly as fast as a flow-insensitive analysis (5,000 to 38,000 lines of code per second—see Section 4), since it begins with the *optimistic* assumption that the invariant holds everywhere and then only has to reason precisely about the relatively few program locations in which any relationships are violated.
2. It permits the vast majority (99.95%) of method type signatures to be flow-insensitive. Flow-insensitive type signatures do not need to specify the effect of the method on the heap. Contrast this to a modular, flow-sensitive analysis that has summaries specifying the effects on all methods, in essence, to rule out the *pessimistic* assumption that any callee could violate the invariant.

In summary, we make the following contributions:

- We describe the FISSILE type analysis framework for intertwined fast type and precise symbolic analysis of almost flow-insensitive storage location properties and instantiate it to check reflective call safety. Our framework is built on the observation that flow-insensitive type invariants on mutable storage locations really serve two roles: specifying facts about the values stored in them and constraining what values are allowed to be stored in them. We define two generic operations to *handoff* between the type and symbolic analyses that essentially either decouples or relinks these roles (Section 3.2).
- We introduce the notion of *almost type-consistent heaps* that (at the concrete level) splits the overall heap into two regions: one explicit region where locations are permitted to be type inconsistent and one implicit region where locations are type consistent up to encountering a location that is potentially inconsistent. We then define type-consistent materialization and summarization operations that permit transitioning an arbitrary number of locations with any connectivity relation between the two regions. With materialized locations, we thus enable strong update reasoning in our disjunctive symbolic analysis. These capabilities are critical for supporting bounded relationship violations among an *arbitrarily* bounded number of heap locations (Sections 2.2, 3.3, and 3.4).
- A key challenge in verifying re-establishment of violated refinement relationships is that the code that breaks a relationship may be in one module while the storage for that relationship is in another. We introduce symbolic summaries that enable programmers to specify encapsulated relationship storage and thus retain modular checking for *cross-module bounded violations* (Sections 2.3 and 3.5).
- We evaluate the effectiveness of our approach to checking reflective call safety on a suite of Objective-C libraries and applications and on code snippets posted by inexperienced developers on public forums (Section 4).

2. Overview

In this section, we present an example that illustrates the main challenges in permitting temporary violations of type consistency, particularly with respect to heap-allocated objects. Our examples are drawn from verifying reflective call safety in real-world Objective-C code, which require such temporary violations to be able to use simple, global type invariants. Objective-C, like C++, is an object-oriented layer on top of C that adds classes and methods. We will describe its syntax as needed.

```
1  @interface Button
2  - (void)drawState:(String * (| in {'Up', 'Down'} |))state {
3    String *m = ...
4    CustomImage *image = ...
5    m = ["draw" append:state];
6    [image setDelegate:self selector:m];
7    [image draw];
8  }
9  - (void)drawUp { ... }
10 - (void)drawDown { ... }
11 @end
12 @interface CustomImage {
13   Object * (| respondsTo selector()→void |) delegate; String *selector;
14 }
15 - (void)setDelegate:(Object * (| respondsTo s()→void |))d
16             selector:(String *)s {
17   self->delegate = d;
18   self->selector = s;
19 }
20 - (void)draw { [self->delegate performSelector:self->selector]; }
21 @end
```

Figure 1. Verifying reflective call safety requires knowing responds-to relationships between delegates and selectors.

The example in Figure 1, adapted from the ShortcutRecorder library, demonstrates how programmers use reflective method call to avoid boilerplate code and to decouple components. Ignore the annotations in double parentheses $(\!|\cdot|\!)$ for now—these denote our additions to the language of types. The `Button` class (lines 1–11) contains a `drawState:` method (lines 2–8) that draws the button as either up or down, according to whether the caller passes the string `"Up"` or `"Down"` as the `state` argument. A class is defined within `@interface...@end` blocks; an instance method definition begins with `-`. Methods are defined and called using infix notation (Smalltalk-inspired syntax). For example, the code at line 6 calls the `setDelegate:selector:` method on the `image` object with `self` as the first argument and `m` as the second. This call is analogous to `image->setDelegateSelector(self,m)` in C++. Now, a `Button` object draws itself by using the `CustomImage` to call either `drawUp` or `drawDown`. The `CustomImage` sets up a drawing context and *reflectively* calls the passed in selector on the passed in delegate at line 20—the delegate and selector pair form, in essence, a callback. This syntax `[`*o* `performSelector:`*s*`]` for reflective call in Objective-C is analogous to *o*`.send(`*s*`)` in Ruby, `getattr(`*o*`,`*s*`)()` in Python, and *o*`[`*s*`]()` in JavaScript. In this case, the delegate is set to the `Button` object itself, and the selector is constructed by appending the passed in `state` string to the string constant `"draw"` (lines 5–6). Constructing the selector dynamically reduces boilerplate by avoiding, for example, a series of `if` statements inspecting the `state` variable. Using reflection for callbacks also improves decoupling—`CustomImage` is agnostic to the identity of the delegate. This *delegate idiom* is one common way responder-selector pairs arise in Objective-C and other dynamic languages. The use of reflection in this example comes at a cost: while the Objective-C type system statically checks that directly called methods exist, it cannot do so for reflective calls—these are only checked at run time. In this work we present an analysis that enables modular static checking of reflective call safety while still maintaining the benefits of reduced boilerplate. To prove that the program is reflection-safe, we use refinement types [19, 20, 31] to ensure that the responder does, in fact, respond to the selector.

To see how these "responds-to" relationships arise, consider the reflective call at line 20. It will throw a run-time error if the receiver does not have a method with the name specified in the argument—conversely, to be safe, it is sufficient that `self->delegate` responds to `self->selector`. There is an unexpressed invariant that requires that for every instance of `CustomImage`, the object stored in the `delegate` field must respond to the selector stored in the `selector` field. We capture this invariant by applying the **respondsTo** selector()→`void` refinement to the `delegate` field at line 13. This refinement expresses the desired *relationship* between the `delegate` field and the `selector` field. The method signature ()→`void` states that the `selector` field holds the name of a method that takes zero parameters with return type `void`. This expresses an intuitive invariant that, unfortunately, does not quite hold everywhere. Still, our framework is capable of using this *almost* everywhere invariant to check that the required relationships hold when needed—we revisit this point in Section 2.1.

Working backward, we see that the `setDelegate:selector:` method updates the `delegate` and `selector` fields with the values passed as parameters—this demonstrates the need, in a modular analysis, for **respondsTo** refinements to apply to parameters as well as fields. We annotate parameter `d` to require that it responds to `s`. Any time the method is called, the first argument must respond to the second. Thus at the call to `setDelegate:selector:` at line 6, we must ensure that `self` responds to `m`. In the caller (i.e., the client of `CustomImage`), we know a precise type, `Button`, for the first argument (whereas from the callee's point of view it is merely `Object`). This means we know that, from the caller's point of view, the delegate will respond to the selector if the selector is a method on `Button`—so if we limit the values `m` can take on to either `"drawUp"` or `"drawDown"` the **respondsTo** refinement in the callee will be satisfied. We write **in** for the refinement that limits strings to one of a set of string constants (i.e., a union of singletons). For simplicity in presentation, our only refinement for string invariants is **in**, although more complex string reasoning is possible.

2.1 Insufficient: Flow-Insensitive Typing of Refinements

Restricting values stored in variables, parameters, and fields with **respondsTo** relationships and **in** refinements captures the essential invariants needed to verify reflective call safety. Here, we show why a flow-insensitive type analysis with these refinements is insufficient in the presence of imperative updates. We focus on the most interesting case: updates to fields of heap-allocated objects.

Refinement types $\{\, v : B \mid R(v) \,\}$ consist of two components: the base type B, which comes from the underlying type system, and the (optional) refinement formula R, which add restrictions on the value v beyond those imposed by the base type. As a notational convenience, we write them without the bound variable and assume the bound variable is used as the first argument of all atomic relations, writing for example, Int $\restriction \geq 0$ instead of $\{\, v : \text{Int} \mid v \geq 0 \,\}$.

Subtyping with refinement types. We assume the refinement type system of interest comes equipped with a subtyping judgment $\Gamma \vdash T_1 <: T_2$ that is a static over-approximation of semantic inclusion (i.e., under a context Γ, the concretization of type T_1 is contained in the concretization of T_2). As an example, we consider informally subtyping with the **respondsTo** and the **in** refinements for reflective call safety. For **in** refinements, this is straightforward: an **in**-refined string is a subtype of another string if the possible constant values permitted by the first are a subset of those required by the second. The situation for subtyping the **respondsTo** refinement is complicated by the fact that a relationship refinement can refer to the *contents* of related storage locations. Consider the `Button *` type in the `drawState:` method. The type environment, Γ, limits the local variable `m` to hold either `"drawUp"` or `"drawDown"`. Since `Button *` is a subtype of `Object *` in the base Objective-C type system and `Button` has both a `drawUp` and a `drawDown` method, `Button *` is a subtype of `Object *` $\restriction$ **respondsTo** m *in this environment*. If, for example, $\Gamma(\mathsf{m})$ instead had refinement **in** {'Fred'} the above subtyping relationship would not hold. Note that for presentation we have elided the method type on the **respondsTo** here; we do so whenever it is not relevant to the discussion.

Type checking field assignments. We check field assignments flow-insensitively with a weakest-preconditions–based approach similar to the Deputy type system [12] (although extended to handle subtyping). Here, we focus on why flow-insensitive typing raises an alarm after the field assignment `self->delegate = d` at line 17 (see Section 3 for more details on how checking proceeds). To check this assignment, we first augment the type environment with fresh locals representing the fields of the assigned-to object and then check the assignment as if it were a local update. Conceptually we *temporarily* bring field storage locations into scope and give them local names. Let this augmented type environment be

$$\Gamma_a \quad = \quad \Gamma[\mathsf{d} : \texttt{Object *} \upharpoonright \textbf{respondsTo}\,\mathsf{s}] \\ [\mathsf{delegate} : \texttt{Object *} \upharpoonright \textbf{respondsTo}\,\mathsf{selector}]$$

for some Γ and where we explicitly show the two **respondsTo** refinements (for presentation, we use the field names as the fresh locals). Checking the assignment, we end up with the subtyping check $\Gamma_a \vdash \Gamma_a(\mathsf{d}) <: \Gamma_a(\mathsf{delegate})$, which expresses the invariant that the relationship between delegate and selector should be preserved across this assignment. Note that while this subtyping check is what is prescribed for assignment in a standard, non-dependent type system, the weakest-preconditions–based approach ensures that this same check is also required (and thus also fails) for the next line (line 18) where `selector` is updated. Although these two assignments are each unsafe in isolation, considered in combination, they are safe. The first assignment breaks the invariant that the `delegate` field should respond to `selector`, but the second restores it.

These temporary violations cannot be tolerated by a flow-insensitive type analysis because, in imperative languages, flow-insensitive types on storage locations really perform two duties. First, they express facts about values: any value read from a variable with type **respondsTo** m can be assumed to respond to the value stored in location m. But second, they express constraints on mutable storage: for the fact to universally hold, the type system must disallow any write to that variable of a value that does not respond to m. These constraints are fine for standard types but are problematic for relationships that are established or updated in multiple steps.

2.2 Tolerating Violation of Relationship Refinements

As we previewed in Section 1, we argue that moving to a flow-sensitive treatment of typing is *too pessimistic*. A flow-sensitive type analysis drops the global constraints on mutable storage locations that enable concise specification. Instead it focuses on tracking facts on values as they flow through the program, which then require more verbose summaries. Other possible alternatives are to change the programming language to disallow temporary violations (e.g., by requiring simultaneous updates) or to somehow generalize the type invariant to capture the temporary violations of the simpler invariant; we argue that these approaches are *too specific*.

Our work is motivated by the observation that although programmers do sometimes violate refinement relationships, most of the time these relationships hold—they are *almost* flow-insensitive. To set up our notion of *almost type-consistent* heaps, we first make explicit a standard notion of type-consistency.

Definition 1 (Type-Consistency)**.** A storage location is *type-consistent* if the values stored in it and all locations in its reachable heap conform to the requirements imposed by their flow-insensitive refinement type annotations. Thus, a storage location is *type-inconsistent* if either the value stored in it *immediately* without pointer dereferences violates a type constraint or there is a type-inconsistent location *transitively* in its reachable heap. We distinguish these two cases of *immediately type-inconsistent* versus only *transitively type-inconsistent*.

In this work we rely on two premises about how programmers violate refinement relationships over storage locations on the heap:

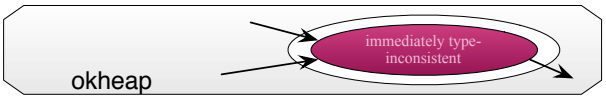
Premise 1 *All* of the heap is type-consistent *most* of the time.

Premise 2 *Most* of the heap is not immediately type-inconsistent *all* of the time. In other words, only a few locations are responsible for breaking the global type invariant at any time.

Following **Premise 1**, we apply type analysis when the heap is type-consistent and switch to symbolic analysis when the type invariant is violated. Under this premise, these periods of violation are bounded in execution—and short enough that the path explosion from precise symbolic analysis is manageable.

To enable the intertwining of a type and a symbolic analysis, we require two conceptually inverse operations, symbolization and typeification, that are applied when switching between the two kinds of analyses. *Symbolization* splits a type environment Γ (which expresses relationship constraints between storage locations) into a symbolic fact map $\widetilde{\Gamma}$ and a symbolic memory $\widetilde{M}$. The fact map expresses relationships between symbolic *values* (i.e., refinement types lifted to the symbolic domain) and the memory indicates where those values are stored (symbolic local variables and heap). So, for example, splitting a type environment Γ with $[\mathsf{d} : \texttt{Object *} \upharpoonright \textbf{respondsTo}\,\mathsf{s}]$ will result in a fact map $\widetilde{\Gamma}$ with fact $[\widetilde{v_1} : \texttt{Object *} \upharpoonright \textbf{respondsTo}\,\widetilde{v_2}]$ and a memory $\widetilde{M}$ which maps d to $\widetilde{v_1}$ and s to $\widetilde{v_2}$. Here $\widetilde{v_1}$ and $\widetilde{v_2}$ are arbitrary symbolic names for values initially stored in d and s upon symbolization. In order for a symbolization to be sound, the symbolized fact map must be "no stronger" than the original type environment. A type environment Γ is not directly comparable to a symbolic fact map $\widetilde{\Gamma}$, so we capture the required relationship with a "subtyping under substitution" judgment $\Gamma <:_{\tilde{\theta}^{-1}} \widetilde{\Gamma}$ that, in essence, converts the symbolic fact map to a type environment before performing the comparison. We perform this conversion with a substitution $\tilde{\theta}^{-1}$, constructed from the symbolic memory, that maps symbolic values to the local variables that store them. *Typeification* fuses $\widetilde{\Gamma}$ and $\widetilde{M}$ back into a type environment Γ. In this case, the type environment under the forward memory substitution must be no stronger than the fact map: $\widetilde{\Gamma} <:_{\tilde{\theta}} \Gamma$. We describe these relations fully in Section 3.3.

Premise 2 is at the core of our approach to soundly handling temporary type violations on heap locations. The key idea is a view of the heap as being made up of two separate regions: (a) a small number of individual locations that are allowed to be immediately type-inconsistent and (b) an *almost type-consistent* region consisting of (fully) type-consistent or only transitively type-inconsistent locations, which we illustrate intuitively below:



In the illustration, the dark node represents one location that is immediately type-inconsistent, while the light rectangle around it represents the almost type-consistent region. Note that there may be pointers (shown as arrows) to the this location where such objects are not type-consistent but only transitively type-inconsistent. In the analysis, the locations in the almost type-consistent region are summarized and represented by an atomic assertion okheap, while the possibly immediately type-inconsistent locations are materialized and explicitly given in the symbolic memory $\widetilde{M}$.

During symbolic analysis, we can materialize explicit storage locations from the almost type-consistent region okheap and thus update them to hold values that do not match their expected types (i.e., to become immediately type-inconsistent). When the materialized storage is again consistent with its expected types, we can summarize it back into okheap. When the heap consists *entirely* of

okheap, we know that the entire heap is fully type-consistent and we can return to type checking. Given this mechanism, we can easily allow for more than one materialization at a time as long as we account for possible aliasing with already materialized locations—Premise 2 suggests that this explosion is also manageable. As an aside, we see okheap as a specialization of separating implication $-\!\!*$ from separation logic [30] for type-consistency. Materializing from okheap corresponds to removing a location from its concretization and introducing an implication awaiting type-consistency of the location (intuitively, the hole in the above illustration). Summarizing into okheap corresponds to putting a location into its concretization and eliminating an implication (see Section 3.3).

2.3 Modular Checking and Symbolic Summaries

Flow-insensitive type signatures are effective for achieving method-level modularity, but they do not summarize the effect that a method has on the heap. Such heap effect summaries are needed when the code that breaks and restores a relationship invariant is spread between multiple methods. Consider again the temporary invariant violation at line 17 in Figure 1, but suppose that instead of updating the delegate and selector fields directly, the programmer used the accessor idiom with setDelegate: and setSelector: methods:

```
17 [self setDelegate:d];
18 [self setSelector:s];
```

Each of these methods in isolation break the relationship invariant, but they are safe when used in combination. Flow-insensitive type signatures specify neither alias information nor the effect of the method on the heap—a richer specification is needed to check such cross-method relationship violations.

Rather than complicate the type analysis with flow-sensitive method type signatures or effect annotations, we enrich the symbolic analysis with the ability to summarize methods via pre- and post-conditions describing the structure of the heap (in separation logic [30]). For example, the symbolic summary for the setDelegate: method with a parameter named p would have input heap $\mathsf{self} \mapsto \{\mathsf{delegate} : -\}$ and output heap $\mathsf{self} \mapsto \{\mathsf{delegate} : \mathsf{p}\}$. This summary says that (1) regardless of its contents on method entry, on method exit the delegate field contains the passed in the parameter and (2) the method does not touch any fields other than delegate. When symbolically executing the caller of one of these methods, we can modularly apply the summary, as long we ensure conformance to the summary when checking the method body (Section 3.5).

Uniquely, we need symbolic method summaries only as a rare escape hatch because of an intertwined approach: within symbolic analysis, we can apply a nested type analysis with a different local type invariant. This enables using flow-insensitive method types within symbolic analysis and other ways to leverage symbolic reasoning outside of the type system (see Section 3.5). On one hand, flow-insensitive type signatures are imprecise but simple to express and fast to check. On the other, symbolic method summaries can precisely describe method heap effects but are complex to express and slow to check. We therefore can obtain the same precision as a fully symbolic analysis but still take advantage of the simpler, more efficient type analysis where the global type invariant holds. In contrast to flow-sensitive approaches (e.g., [32]), we do not need heap effect summaries for all methods—but rather only for those very few (0.05%) that leave a relationship invariant violated.

3. Storage Type and Symbolic Value Analysis

In this section, we provide an example-driven description of how FISSILE type analysis intertwines flow-insensitive type analysis and path-sensitive symbolic analysis to modularly check refinement relationships between storage locations. For reference, we give the full details of the type and symbolic analyses in our TR [13]; here we focus on the core contributions of our framework and illustrate them via an instantiation to check reflective call safety.

Preliminaries: Syntax of language and types. We describe FISSILE type analysis over a core imperative programming language of expressions e with objects and reflective method call. For presentation purposes, we have only three types of values: unit, strings, and objects. We assume disjoint syntactic classes of identifiers for program variables x, y, z, field names f, method names m, and parameter names p, as well as a distinguished identifier 'self' that stands for the receiver object in methods. Program expressions include literals for unit $\langle\rangle$, strings c, objects $\{\overline{\mathrm{var}\, f : T\ =\ e}, \overline{\mathrm{def}\, m(\overline{p : T_p}) : B_{\mathrm{ret}}[S]\ =\ e}\}$. The 'var' declarations specify mutable fields f of types T, and the 'def' declarations describe methods m with parameters p of types T_p and with return type B_{ret}. The return type is a base type, which does not itself have refinements (but could have refinements on its fields in the case of an object base type). An overline stands for a sequence of items. Methods can also be optionally annotated with symbolic summaries S, which we revisit in Section 3.5. Objects are heap allocated. Local variable binding 'let $x : T = e_1$ in e_2' binds a local variable x of type T initialized to the value of e_1 whose scope is e_2. We include one string operation for illustration: string append x_1 @ x_2. Then, we have reads of locals x and fields $x.f$, writes to fields $x.f := y$, basic control structures for sequencing $e_1; e_2$ and branching $e_1 \,[]\, e_2$. For presentation, we use non-deterministic branching, as the guard condition of an 'if-then-else' expression has no effect on flow-insensitive type checking and can be reflected in the symbolic analysis in the usual way (i.e., by strengthening the symbolic state with the guard condition). Finally, we have two method call forms: one for direct calls $z.m(\bar{x})$ and one for reflective calls $z.[y](\bar{x})$. A call allocates an activation record for the receiver object z and parameters $\bar{x}$; it then dispatches with the direct name m or the reflective selector y. Types T are a base type B for either unit Unit, strings Str, or objects $\{\overline{\mathrm{var}\, f : T_f},\ \overline{\mathrm{def}\, m(\overline{p : T_p}) \to B_{\mathrm{ret}}}\}$ with a set of refinements R, which are interpreted conjunctively.

The framework is parametrized by the language of refinements R needed to specify the invariants of interest, and refinements should be parametric with respect to the syntactic class of identifiers ι. We decorate with a superscript R^{L}, R^{F}, or R^{S} when we want to emphasize or make clear over which syntactic class of identifiers the refinement ranges: locals x, fields f, or symbolic values $\widetilde{v}$ (see Section 3.2), respectively. Because types include refinements, types are parametrized as well, written T^{L}, T^{F}, or T^{S}, as are type environments Γ.

3.1 Instantiation: The Reflection Checking Type Refinement

To verify reflective call safety, we have seen that the key property is the '$\mathsf{respondsTo}\ \iota(\overline{p : T_p}) \to B_{\mathrm{ret}}$' refinement that says an object must respond to the value named by ι with the given method type. As a symbolic fact, it says that an object must respond to the value named by ι. But as a type invariant on storage locations, the refinement also constrains *both* the storage location on which the refinement is applied and the storage location named by ι to hold a responder and a correspondingly valid selector for it. We also need some refinements on string values, such as the union of singletons: '$\mathsf{in}\ \{c_1, \ldots, c_n\}$' which says the value is one of the following (string) literals $c_1, \ldots, c_n$.

To type expressions, we use the standard typing judgment form $\Gamma \vdash e : T$ that says in a type environment Γ, expression e has type T. A type environment Γ is a finite map from identifiers to types, which we view as the types assigned to program variables (i.e., $\Gamma^{\mathrm{L}} \vdash e : T^{\mathrm{L}}$ for emphasis). The standard typing judgment form emphasizes that Γ is a flow-insensitive invariant.

$$\textsc{T-Reflective-Method-Call}\quad \frac{\Gamma(z) = \{\cdots\} \upharpoonright \mathsf{respondsTo}\ y(\overline{p:T_p}) \to B_{\text{ret}} \quad \Gamma <:_{[\overline{p:x},\text{self}:z]} \overline{p:T_p}, \text{self} : \Gamma(z)}{\Gamma \vdash z.[y](\overline{x}) : B_{\text{ret}} \upharpoonright}$$

$$\textsc{Sub-Obj-RespondsTo-Refl}\quad \frac{\Gamma \vdash \Gamma(x) <: \text{Str} \upharpoonright \mathsf{in}\ \{c_1, \ldots, c_n\} \quad \forall_{i \in 1..n}\ B \text{ has a method named } c_i \text{ with signature } (\overline{p:T_p}) \to B_{\text{ret}}}{\Gamma \vdash B \upharpoonright \cdots <: B \upharpoonright \mathsf{respondsTo}\ x(\overline{p:T_p}) \to B_{\text{ret}}}$$

Figure 2. Flow-insensitive typing for reflective method calls. The respondsTo refinement is checked on reflective dispatch.

We provide the full type system for reflective call safety in our TR [13]—here we focus on the rules needed to check reflective calls given the desired type invariant (Figure 2). The T-Reflective-Method-Call rule for checking a reflective method is itself not complicated: we require that the responder object z has a refinement guaranteeing that it responds to the selector y with method type signature $(\overline{p:T_p}) \to B_{\text{ret}}$. The arguments to call are checked against the specified types of the parameters via $\Gamma <:_{[\overline{p:x},\text{self}:z]} \overline{p:T_p}, \text{self} : \Gamma(z)$. We write $\Gamma <:_\theta \Gamma'$ as the lifting of subtyping to type environments under a substitution θ from variables on the right to variables on the left. The type of the call is then the return base type of the method (as expected) without any refinements $B_{\text{ret}} \upharpoonright$.

The respondsTo refinement is introduced via subtyping. The subtyping judgment $\Gamma \vdash T <: T'$ says that in typing environment Γ, type T implies type T'. The Sub-Obj-RespondsTo-Refl subtyping rule combines information from base object types and the environment to introduce respondsTo. This rule says that for any location x that is one of a set of selector strings $c_1, \ldots, c_n$, then any object of base type B with methods of the appropriate signature for all $c_1, \ldots, c_n$ responds to the method named by x with that signature. We also have subtyping rules expressing component-wise implication of refinements, conjunctive weakening of the set of refinements, and disjunctive weakening of in refinements. For our purposes, it does not matter how subtyping is checked as long as it is a sound approximation of semantic subtyping. We could, for example, use an SMT solver as in Liquid Types [31] and also replace the type rules for string operations with an off-the-shelf string solver.

Another Instantiation: Typing for array refinements. The Fissile type analysis framework is parametrized by the language of refinements and a decision procedure for semantic inclusion. To provide context for our approach, we sketch another instantiation that checks array-bounds safety—a property considered by many prior works—with a few refinements and rules. Suppose we augment our programming language to include array allocation and array access and add two refinements: (1) hasLength, which indicates that an array has the specified (non-zero) length and (2) indexedBy, which indicates that the specified index is a valid index for the array—that is, that the index is in bounds. When analyzing an array access $e[x]$, we check that x is a valid index into the array e by requiring that e's type has the refinement indexedBy x. We introduce this refinement via subtyping with the following rule, which says that x is a valid index for an array of length y if the environment Γ restricts x and y such that $x \geq 0$ and $x < y$:

$$\frac{[\![\Gamma]\!] \vdash_{\text{SMT}} x \geq 0 \wedge x < y}{\Gamma \vdash B \upharpoonright \mathsf{hasLength}\ y <: B \upharpoonright \mathsf{indexedBy}\ x}$$

We verify that this condition holds by encoding the environment into a linear arithmetic formula (written $[\![\Gamma]\!]$) and checking entailment with an SMT solver (written as the judgment $\phi_1 \vdash_{\text{SMT}} \phi_2$ for formulas ϕ_1 and ϕ_2). Here we map meta-variables x and y in the typing judgment to logical variables of the same name in the SMT entailment checking judgment.

$\widetilde{\Sigma}$	::=	$(\widetilde{\Gamma}, \widetilde{H})$	symbolic states
$\widetilde{E}$	::=	$\cdot \mid \widetilde{E}[x : \widetilde{v}]$	symbolic environments
$\widetilde{H}$	::=	$\mathsf{emp} \mid \widetilde{a} : \tilde{o} \mid \widetilde{H}_1 * \widetilde{H}_2 \mid \mathsf{okheap}$	symbolic heaps
$\widetilde{\Gamma}$	::=	$\cdot \mid \widetilde{\Gamma}[\widetilde{v} : T^{\text{S}}]$	symbolic facts
$\widetilde{P}$	::=	$\widetilde{\Sigma} \downarrow \widetilde{v} \mid \widetilde{P}_1 \vee \widetilde{P}_2 \mid \mathsf{false}$	symbolic paths
$\tilde{o}$	::=	$\mathsf{emp} \mid f : \widetilde{v} \mid \tilde{o}_1 * \tilde{o}_2$	symbolic objects
$\widetilde{v}, \widetilde{a}, \widetilde{x}, \widetilde{y}, \widetilde{z}$			symbolic values

Figure 3. The symbolic analysis state splits type environments into types lifted to values and the locations where values are stored.

3.2 Symbolic Analysis State and Handoff

The type analysis is efficient but coarse. It is flow-insensitive—constraining all storage locations to be a fixed type and the heap to be always in a consistent state. When these constraints hold, we get a simple and fast analysis. When they are temporarily violated, our overall analysis can switch to a path-sensitive symbolic analysis that continues until the constraints again hold.

We now walk through a modified version of the example from Section 2 to describe the key components of this switch: (1) we describe our symbolic analysis state and how we convert ("symbolize") a type environment to a symbolic state; (2) we describe type-consistent materialization and summarization from the almost type-consistent heap in Section 3.3; (3) we describe the proofs of soundness for handoff, materialization/summarization, and our overall analysis in Section 3.4; and (4) we explore the interaction between modular symbolic analysis and the almost type-consistent heap in Section 3.5.

Symbolic analysis state. In the symbolic analysis, we split a type environment Γ into a symbolic environment $\widetilde{E}$ and a symbolic state $\widetilde{\Sigma}$ (Figure 3). A symbolic environment $\widetilde{E}$ provides variable context: it maps variables to symbolic values $\widetilde{v}$ that represent their values. A symbolic state $\widetilde{\Sigma}$ consists of two components: a symbolic fact context $\widetilde{\Gamma}$, mapping symbolic values to the facts (symbolic types) known about them and a symbolic heap $\widetilde{H}$. A symbolic heap $\widetilde{H}$ contains a partially-materialized sub-heap that maps addresses ($\widetilde{a}$) to symbolic objects ($\tilde{o}$), which are themselves maps from field names (f) to symbolic values. We write symbolic objects and heaps using the separating conjunction $*$ notation borrowed from separation logic [30] to state that we refer to disjoint storage locations.

Symbolic values $\widetilde{v}$ correspond to existential, logic variables. For clarity, we often use $\widetilde{a}$ to express a symbolic value that is an address and similarly use $\widetilde{x}, \widetilde{y}, \widetilde{z}$ for values stored in the corresponding program variables x, y, z. Relationship refinements in $\widetilde{\Gamma}$ are expressed in terms of types lifted to symbolic values (T^{S})—that is, the refinements state relationship facts between values and not storage locations (like the refinements in Γ for typing). Our overall analysis state is a symbolic path set $\widetilde{P}$, which is a disjunctive set of singleton paths $\widetilde{\Sigma} \downarrow \widetilde{x}$. A singleton path is a pair of a symbolic state and a symbolic value corresponding to the return state and value, respectively.

The symbolic heap $\widetilde{H}$ enables treating heap locations much like stack locations, capturing relationships in the symbolic context $\widetilde{\Gamma}$, though certainly more care is required with the heap due to aliasing. A symbolic heap $\widetilde{H}$ can be empty emp, a single materialized object $\widetilde{a} : \tilde{o}$ with base address $\widetilde{a}$ and fields given by $\tilde{o}$, or a separating conjunction of sub-heaps $\widetilde{H}_1 * \widetilde{H}_2$. Lastly and most importantly, a sub-heap can be okheap, which represents an arbitrary but *almost type-consistent heap*. This formula essentially grants permission to materialize from the almost type-consistent heap and, as discussed in Section 2.2, is the key mechanism for soundly transitioning between the type and symbolic analyses.

Handoff from type checking to symbolic execution. Consider the formal language version of the callback example from Section 2.2:

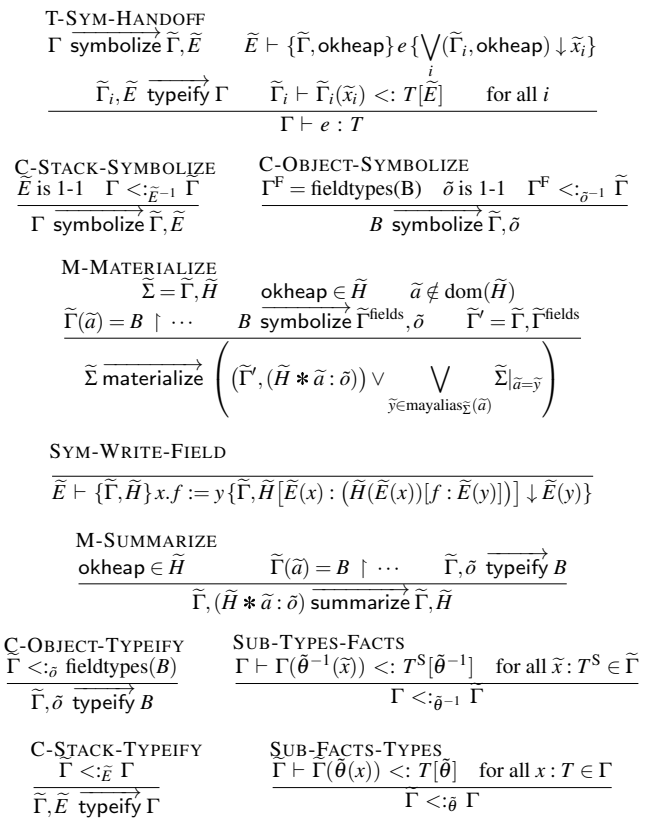


Figure 4. Selected rules demonstrating how FISSILE type analysis switches to a symbolic analysis to tolerate bounded violations.

```
1  { var del: {} ⊦ respondsTo sel, var sel: Str,
2    def update(d: {} ⊦ respondsTo s, s: Str): Unit =
3      self.del := d;
4      self.sel := s  }
```

Here the `update` method updates the `del` and `sel` fields in sequence. Recall that the assignment at line 3 breaks the type invariant and the assignment at line 4 restores it. We illustrate the core operations (Figure 4) behind FISSILE type analysis by walking through this example. When checking this method, the type analysis will produce a flow-insensitive type error for the assignment at line 3 and so will switch to symbolic execution. To do so, it will (1) "symbolize" a suitable symbolic analysis state from the type environment, (2) symbolically execute the two field writes, and (3) attempt to "typeify" the resultant symbolic analysis state back to the original type environment. If this succeeds, then the type analysis can continue.

The T-SYM-HANDOFF rule formalizes this process. It says the type checker can switch to the symbolic analysis to check an expression e in a type environment Γ by creating ("symbolizing") a symbolic state representing Γ and symbolically executing e in that state. The judgment form $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e\, \{\widetilde{P}\}$ says that in the context of a given symbolic local variable environment $\widetilde{E}$ and with a symbolic state $\widetilde{\Sigma}$ on input, expression e symbolically evaluates to a disjunction of state-value pairs $\widetilde{P}$ on output. Here the input facts and environment are obtained from Γ (via the $\Gamma \xrightarrow{\text{symbolize}} \widetilde{\Gamma}, \widetilde{E}$ judgment form), while the input heap is initialized to a fully type-consistent heap okheap. After symbolic execution, the resultant symbolic state must be consistent with ("typeify to") the original Γ, and the symbolic facts about the resulting symbolic value must be consistent with the inferred type T of the expression. Note that here we lift the subtyping judgment $\cdot \vdash \cdot <: \cdot$ to symbolic types in the expected way. Here $T[\tilde{\theta}]$ converts the standard type T to a symbolic type (fact) T^{S} with a substitution $\tilde{\theta}$ that replaces all variable references in T's refinements with symbolic values. Both the initially symbolized heap *and* the finally typeified heap must consist solely of okheap—the heap is assumed consistent on entry and must be restored on exit. The key aspect of this handoff is that although the symbolic execution is free to violate any of the flow-insensitive constraints imposed by Γ, it must restore them to return to type checking. We will discuss restoration in detail later—first we describe symbolization.

Symbolizing type environments to symbolic states. Symbolization splits a type environment Γ (which expresses type constraints on local variables) into a symbolic fact map $\widetilde{\Gamma}$ (expressing facts about symbolic values) and a symbolic local variable state $\widetilde{E}$ (expressing where those values are stored). For example, consider the type environment above at line 3, immediately before the first write:

$$\Gamma = [\mathsf{d} : \{\} \restriction \mathsf{respondsTo\ s}][\mathsf{s} : \mathsf{Str}][\mathsf{self} : \mathrm{T}_{\mathrm{Image}}]$$

where $\mathrm{T}_{\mathrm{Image}} = \{\mathsf{var\ del} : \{\} \restriction \mathsf{respondsTo\ sel}, \mathsf{var\ sel} : \mathsf{Str}\}$. We can symbolize this environment to create a symbolic environment where $\widetilde{E} = [\mathsf{d} : \widetilde{\mathsf{d}}][\mathsf{s} : \widetilde{\mathsf{s}}][\mathsf{self} : \widetilde{\mathsf{self}}]$. Here we have created fresh symbolic names to represent the values stored on the stack: $\widetilde{\mathsf{d}}$ is the name of the value stored in local d, $\widetilde{\mathsf{s}}$ in local s, etc. These symbolic values represent concrete values from a type environment in which the storage location refinement relationships hold, so we can safely assume that *values* initially stored in those locations have the equivalent relationships, expressed as lifted types:

$$\widetilde{\Gamma} = [\widetilde{\mathsf{d}} : \{\} \restriction \mathsf{respondsTo}\ \widetilde{\mathsf{s}}][\widetilde{\mathsf{s}} : \mathsf{Str}][\widetilde{\mathsf{self}} : \mathrm{T}_{\mathrm{Image}}]$$

Note that the refinement on $\widetilde{\mathsf{d}}$ refers to symbolic value $\widetilde{\mathsf{s}}$ and not storage location s, but that the refinements on the types of the *fields* of the base type of $\widetilde{\mathsf{self}}$'s fact $\mathrm{T}_{\mathrm{Image}}$ still refer to (field) storage locations. These field refinements on the base object type are, in essence, a "promise" that if any explicit storage for those fields is later materialized, it must be consistent with $\mathrm{T}_{\mathrm{Image}}$ when summarized back into okheap.

We formalize type environment symbolization in rule C-STACK-SYMBOLIZE, which captures the requirement that the symbolized state must over-approximate the original type environment. We note that $\widetilde{E}^{-1}$ forms a substitution map from symbolic values to local variable names and require that the symbolized fact map $\widetilde{\Gamma}$ under that substitution be an over-approximate environment of the original type environment Γ (rule SUB-TYPES-FACTS). In essence, any assumptions that the symbolic analysis initially makes about the symbolic facts must also hold in original type environment. That $\widetilde{E}$ is one-to-one ensures that the inverse exists but more importantly encodes the requirement that the newly symbolized environment makes no assumptions about aliasing between values stored on the stack in local variables. Note that when symbolizing a local variable with type $B \restriction R^{\mathrm{L}}$ in a type environment, we do not lift the base type B to the symbolic domain nor do we create storage for any of B's fields. That is, refinements on the *fields* of an object base type remain refinements over fields, expressing both facts about the field contents *and* constraints on those storage locations. This interpretation is what permits materialized, immediately type-inconsistent objects to point back into okheap (i.e., the almost type-consistent region). As we detail next, with this interpretation, our analysis *materializes* storage for objects from okheap *on demand*, which is not only more efficient but is required in the presence of recursion.

3.3 Materialization from Type-Consistent Heaps

Returning to the callback example, recall that the analysis has symbolized a state corresponding to the type environment immediately before line 3. A symbolic heap $\widetilde{H}$ consists of two separate regions:

(1) the materialized heap, a precise region with explicit storage that supports strong updates and allows field values to differ from their declared types (i.e., permits immediate type-inconsistency) and (2) the okheap, a summarized region in which all locations are either type-consistent or only transitively type-inconsistent. In a newly symbolized analysis state, $\widetilde{H}$ consists of solely okheap. Before the field write at line 3 can proceed, the analysis must first materialize storage for the $\mathrm{T}_{\mathrm{Image}}$ object pointed to by self to get:

$$\widetilde{H} = \mathsf{okheap} * \widetilde{\mathsf{self}} \mapsto \{\mathsf{del} : \widetilde{\mathsf{del}}, \mathsf{sel} : \widetilde{\mathsf{sel}}\}$$

and a new fact map $\widetilde{\Gamma}$ that contains the additional facts: $\widetilde{\mathsf{del}} : \{\} \upharpoonright \mathsf{respondsTo}\ \widetilde{\mathsf{sel}}$ and $\widetilde{\mathsf{sel}} : \mathsf{Str}$.

We formalize type-consistent materialization with the C-OBJECT-SYMBOLIZE and M-MATERIALIZE rules. Creating symbolic storage for an object type is very similar to symbolizing a type environment. As rule C-OBJECT-SYMBOLIZE defines, the analysis can symbolize a type B to a symbolic object $\tilde{o}$ (mapping field names to fresh symbolic values) and a fact map $\widetilde{\Gamma}$ (facts about those values) if the assumed facts about the values are no stronger than those guaranteed by the object's field types. Once the analysis has symbolized an object, it adds the new object storage to the explicit heap and facts about the fresh symbolic values to the fact map in rule M-MATERIALIZE.

For the symbolic analysis to perform strong updates, it must maintain the key invariant that any two objects' storage locations in the explicit heap are definitely separate. When materializing an arbitrary object, the evaluator must consider whether any of the already materialized objects aliases with the newly materialized object and case split on these possibilities. The split is required because any two distinct symbolic values may in fact represent the same concrete value. In M-MATERIALIZE, for any input state $\widetilde{\Sigma}$ in which the heap contains okheap, the symbolic analysis is free to materialize the object stored at a symbolic address $\widetilde{a}$ from the type-consistent heap. For the case where the materialized symbolic address does not alias any address already on the explicit heap, we symbolize a new symbolic object $\tilde{o}$ with fresh symbolic values as described above from the base type of $\widetilde{a}$. In the case where the address may alias some address $\widetilde{y}$ on the materialized heap, we must assert that $\widetilde{a} = \widetilde{y}$. We write $\widetilde{\Sigma}|_{\widetilde{a}=\widetilde{y}}$ for any sound constraining of $\widetilde{\Sigma}$ with the equality (we implement it by substituting one name for the other and applying a meet $\sqcap$ in the symbolic facts $\widetilde{\Gamma}$). We also leave unspecified the mayalias$_{\widetilde{\Sigma}}(\widetilde{a})$ that should soundly yield the set of addresses that may-alias $\widetilde{a}$; our implementation uses a type-based check to rule out simple non-aliases.

This rule is quite general. It permits an arbitrary number of locations to be immediately type-inconsistent without any constraints on connectivity, ownership, or non-aliasing. To simplify the formalization of type-consistent materialization, we restrict relations expressed by refinements in the heap to be among fields within objects. Relations between fields are captured because all of fields of the object are symbolized at the same time (see C-OBJECT-SYMBOLIZE). Supporting cross-object relations would merely require materializing multiple objects while disjunctively considering all possible aliasing relationships and then symbolizing their fields simultaneously within each configuration. It would also be possible to just materialize the fields corresponding to the specific relationships that we wish to violate by using a field-split model [26, 28] of objects.

Symbolic execution. With the symbolization and materialization complete, the analysis now executes the field writes at lines 3 and 4. The SYM-WRITE-FIELD rule describes symbolic execution of writing the value of a local variable y to a field f of a base address x. It requires that the object at the base address $E(x)$ already be materialized and updates the appropriate field in the symbolic heap $\widetilde{H}$. We give the rest of our symbolic executions rules in our TR [13]—they are as expected. Unlike traditional symbolic analysis, our mixed approach can soundly ensure termination by falling back to type checking. In practice, we switch to types at the end of loop bodies to cut back edges and cut recursion with method summaries.

Summarizing symbolic objects back into types. After execution of the field writes, the symbolic heap at line 4 is:

$$\widetilde{H} = \mathsf{okheap} * \widetilde{\mathsf{self}} \mapsto \{\mathsf{del} : \widetilde{\mathsf{d}}, \mathsf{sel} : \widetilde{\mathsf{s}}\}.$$

That is, the fields now contain the values passed in as parameters. But recall that $\widetilde{\Gamma}(\widetilde{\mathsf{d}}) = \{\} \upharpoonright \mathsf{respondsTo}\ \widetilde{\mathsf{s}}$ and $\widetilde{\Gamma}(\widetilde{\mathsf{self}}) = \mathrm{T}_{\mathrm{Image}}$. In this state, the value stored in field del again responds to the value stored in field sel—the flow-insensitive type invariant ($\mathrm{T}_{\mathrm{Image}}$) promised by $\widetilde{\Gamma}(\widetilde{\mathsf{self}})$ again holds—and thus the object can be safely summarized back into the okheap. We describe this process in rule M-SUMMARIZE, which says that a symbolic address $\widetilde{a}$ pointing to a materialized object $\tilde{o}$ can be summarized (i.e., removed from the explicit heap) if the object is consistent with (i.e., can be "typeified" to) the base type required of the address in the fact map. Typeifying a symbolic object $\tilde{o}$ to an object type B (rule C-OBJECT-TYPEIFY) is analogous to symbolization except that it goes in the other direction. We require that the symbolic fact map be over-approximated by the field types of B, nicely converting it to the symbolic domain using $\tilde{o}$ as the substitution. Note that $\tilde{o}$ does not need to be one-to-one; the observation that this constraint is irrelevant for typeification captures that types are agnostic to aliasing.

Once all materialized objects have been summarized (and thus $\widetilde{H} = \mathsf{okheap}$), the checker can end the handoff to symbolic analysis and resume type checking (back to rule T-SYM-HANDOFF) as long as the symbolic locals are consistent with the original Γ for all symbolic paths (rule C-STACK-TYPEIFY) and the returned symbolic values have facts consistent with the return type of the expression.

3.4 Concretization and Soundness

An important concern for materialization and summarization is whether information is transferred soundly between the type analysis and the symbolic analysis (i.e., we have a sound reduced product [15]). In particular, materialization "pulls" information from the heap type invariant on demand during symbolic execution and then permits temporary violations of the global heap invariant in some locations. We take an abstract interpretation-based approach [14] to soundness, which is critical for expressing almost type-consistent heaps and connecting the soundness of type checking with the soundness of symbolic analysis. In this section, we describe, via concretization, the different meanings of object types and their associated reachable heaps in the two analyses. Further, we present the properties of these concretizations that are key to proving soundness of handoff and materialization/summarization and also state a theorem of intertwined analysis soundness. We provide complete concretization functions and a full proof of soundness in our TR [13].

Concretization. A concrete state consists of a concrete environment E, mapping variables x to values v, and a concrete heap H, which is a finite map from addresses a to concrete objects o bundled with their allocated base types B. We overload $*$ to indicate both the disjoint heap union of two concrete subheaps and the separating conjunction of two static symbolic subheaps.

Concretization functions give meaning to abstract constructs by describing how they constrain the set of possible values and states. As is standard, we write γ for concretization and overload it for all constructs, except base types and field types. Object base types and field types, crucially, have different meanings in the type domain and symbolic domain. To disambiguate, we write $\gamma(B)$ for concretization in the type analysis and $\widetilde{\gamma}(B)$ for the symbolic analysis.

Concretization in the type analysis. In the type analysis, the concretization of an object type is fairly standard: it constrains

$$\gamma(B) \triangleq \left\{ (H,a) \;\middle|\; \begin{array}{l} \text{exists } o \text{ where } H(a) = \langle o, B\rangle \text{ and} \\ \quad ① \text{ for all methods } m \\ \quad\quad o(m) \in \gamma(B, (\overline{p : T_p}) \to B_{\text{ret}}) \text{ and} \\ \quad ② \text{ for all fields } f \\ \quad\quad (H, o, o(f)) \in \gamma(T_f^{\text{F}}). \end{array} \right\}$$

$$\gamma(B \restriction R_1^{\text{F}}, \cdots, R_n^{\text{F}}) \triangleq \left\{ (H,o,v) \;\middle|\; \begin{array}{l} (H,v) \in \gamma(B) \text{ and} \\ \text{for all refinements } R_i \\ \quad (H,o,v) \in \gamma(R_i^{\text{F}}) \end{array} \right\}$$

(a) Types domain

$$\widetilde{\gamma}(B) \triangleq \left\{ (H^{\text{ok}}, H^{\text{mat}}, a) \;\middle|\; \begin{array}{l} \text{exists } o \text{ where } H^{\text{ok}} * H^{\text{mat}}(a) = \langle o, B\rangle \text{ and} \\ \quad ① \text{ for all methods } m \\ \quad\quad o(m) \in \gamma(B, (\overline{p : T_p}) \to B_{\text{ret}}) \text{ and} \\ \quad ② \text{ for all fields } f \\ \quad\quad f \in \text{dom}(o) \text{ and if } a \in \text{dom}(H^{\text{ok}}) \text{ then} \\ \quad\quad\quad (H, o, o(f)) \in \widetilde{\gamma}(T_f^{\text{F}}). \end{array} \right\}$$

$$\widetilde{\gamma}(B \restriction R_1^{\text{F}}, \cdots, R_n^{\text{F}}) \triangleq \left\{ (H^{\text{ok}}, H^{\text{mat}}, v) \;\middle|\; \begin{array}{l} (H,v) \in \widetilde{\gamma}(B) \text{ and} \\ \text{for all refinements } R_i \\ \quad (H^{\text{ok}} * H^{\text{mat}}, o, v) \in \gamma(R_i^{\text{F}}) \end{array} \right\}$$

(b) Symbolic domain

Figure 5. Concretization of an object base type $B = \{\overline{\text{var } f : T_f},\ \overline{\text{def } m(\overline{p : T_p}) \to B_{\text{ret}}}\}$ in the types and symbolic domains. In the symbolic domain, values stored in the fields of an object in H^{ok} must not be immediately type-inconsistent; values stored in the fields of an object in H^{mat} are not constrained. The shaded region highlights the key difference between the concretizations.

not only the value but also the entire heap reachable from that object. As we show in Figure 5a, the concretization $\gamma(B)$ of an object type $B = \{\overline{\text{var } f : T_f},\ \overline{\text{def } m(\overline{p : T_p}) \to B_{\text{ret}}}\}$ yields a set of pairs of heaps and values (addresses). The concretization requires that the address point to an object o which ① has suitable method implementations (i.e., constrained by the concretization of the method signature $(\overline{p : T_p}) \to B_{\text{ret}}$ on an object of type B) for each method m in the object type and ② has suitable field values for each declared field f of type T_f^{F}. We adorn the type with its storage class, F, to make clear that it is a field type dependent on other fields.

The key property of concretization in the type domain is that the concretization of a field type $T^{\text{F}} = B \restriction R_1^{\text{F}}, \cdots, R_n^{\text{F}}$ is mutually inductively defined with that for base types and thus *constrains the entire heap reachable from that field*, in addition to the other fields of the object. This concretization yields a heap-object-value triple where the heap and value are constrained by the concretization of the base type and the entire triple is constrained by the concretization of each of the field refinements R_i^{F}. Because these refinements are dependent refinements, they may constrain *other* fields of the object in addition to the value of the field in question. For example, a heap-object-value triple in the concretization of a field refinement $\mathsf{respondsTo}\ g$ would constrain the g field of the object to store a string with name m that is a valid method for the (potentially different) object pointed to by the value. The concretization of a type environment $\gamma(\Gamma)$ is a set of concrete environment-heap (E,H) pairs where the values stored in local variables are consistent with the declared types and reachable heaps from the type bindings in Γ.

Concretization in the symbolic analysis. In contrast to the type analysis, in the symbolic world part of the heap may be explicitly materialized—and thus immediately type-inconsistent, leaving the rest of the heap almost type-consistent. To capture this difference, the symbolic concretization of an object type yields *two* heaps: H^{ok} and H^{mat}, corresponding to a non-deterministic choice of which objects are in the almost type-consistent heap and which objects are materialized and thus may have field values differing from their declared types. We write $\widetilde{\gamma}(B)$ here to emphasize concretization of base types in the symbolic domain. The shaded region of Figure 5b illustrates the key difference: in the symbolic domain, an object's fields are only constrained by the concretization of the field types if the object is in H^{ok}. If the object is in H^{mat}, the fields are guaranteed to exist, but the values stored in them are not constrained. Because concretization of object types is defined inductively, the concretization can "opt out" of type constraints for the reachable subheap at each field dereference, depending on the partitioning of the full heap into H^{ok} and H^{mat}. Crucially, this definition of concretization permits pointers from H^{ok} to H^{mat} and vice-versa. Note that regardless of which heap an object resides in, its method implementations are still constrained by their signatures.

The concretization $\gamma(\widetilde{E}, \widetilde{\Sigma})$ of a symbolic state $\widetilde{\Sigma} = (\widetilde{\Gamma}, \widetilde{H})$ with respect to a symbolic environment $\widetilde{E}$ yields a environment-heap pair (E,H). There must exist a valuation $V : \mathsf{SymVal} \to \mathsf{Values}$ mapping symbolic values to concrete values and a partitioning of the heap $H = H^{\text{ok}} * H^{\text{mat}}$ such that the concretizations of $\widetilde{E}$, $\widetilde{\Gamma}$, and $\widetilde{H}$ all agree upon. Symbolic fact maps $\widetilde{\Gamma}$ and heaps $\widetilde{H}$ both concretize to a set of valuation-heap-heap tuples where emp in the symbolic heap requires both H^{ok} and H^{mat} be empty whereas okheap requires that H^{mat} be empty but H^{ok} can be any heap. The singleton heap formula concretizes to a singleton in H^{mat} and $*$ is the heap disjoint union in both H^{ok} and H^{mat} A symbolic path is a disjunction of singleton paths $\widetilde{P} = \widetilde{\Sigma} \downarrow \widetilde{x}$; its concretization is similar to that of a symbolic state, except that it yields a triple (E,H,v) where $V(\widetilde{x}) = v$.

Soundness. The soundness of FISSILE type analysis—and in particular of handoff and materialization/summarization—depends on the following key properties of concretization.

At handoff, the analysis requires that the explicitly materialized heap be empty. The following lemma states that under those conditions (i.e., when the entire heap is H^{ok}), the meaning of base types in the symbolic domain is the same as the meaning of base types in the type domain:

Lemma 1 (Equivalence of Typed and Symbolic Base Types)**.** $\gamma(B) = \{(H,v) \mid (H,\cdot,v) \in \widetilde{\gamma}(B)\}$

We rely on this result to show that that the T-SYM-HANDOFF and SYM-TYPE-HANDOFF (Section 3.5) rules are sound.

To show soundness of the M-MATERIALIZE rule, we rely on a property about the meaning of base types in the symbolic domain:

Lemma 2 (Type-Consistent Materialization for Types)**.** If $(H^{\text{ok}} * a : \langle o_a, B_a\rangle, H^{\text{mat}}, v) \in \widetilde{\gamma}(B)$ then $(H^{\text{ok}}, H^{\text{mat}} * a : \langle o_a, B_a\rangle, v) \in \widetilde{\gamma}(B)$.

This lemma considers a value v of type B and a heap containing an object o_a of allocated type B_a stored at address a (informally, a is in v's reachable heap, otherwise it is uninteresting). If there is one concretization of B where a is in the almost type-consistent heap H^{ok}, then the configuration with a moved to the materialized H^{mat} is also in its concretization. In essence, moving the storage for a to the materialized heap will not cause the type of v to change from the perspective of the symbolic analysis.

$$\textsc{Sym-Type-Handoff}\quad \frac{\widetilde{\Gamma},\widetilde{E} \xrightarrow{\text{typeify}} \Gamma \qquad \Gamma \vdash e : T \qquad \Gamma \xrightarrow{\text{symbolize}} \widetilde{\Gamma}',\widetilde{E} \qquad \widetilde{z} \notin \mathrm{dom}(\widetilde{\Gamma}')}{\widetilde{E} \vdash \{\widetilde{\Gamma},\mathsf{okheap}\}\, e\, \{\widetilde{\Gamma}'[\widetilde{z} : T[\widetilde{E}]],\mathsf{okheap} \downarrow \widetilde{z}\}}$$

$$\textsc{Sym-Method-Call-Summary}\quad \frac{(\overline{p})[h/h'] : p^{\mathrm{ret}} = \mathrm{summary}(\widetilde{\Gamma}(\widetilde{E}(x)),m) \qquad \widetilde{H} \vdash h \circledast \widetilde{H}^{\mathrm{fr}} \backslash \tilde{\theta}^{\mathrm{h}} \qquad \tilde{\theta} = \tilde{\theta}^{\mathrm{h}} \uplus [p : E(y)]}{\widetilde{E} \vdash \{\widetilde{\Gamma},\widetilde{H}\}\, x.m(\overline{y})\, \{\widetilde{\Gamma},(\widetilde{H}^{\mathrm{fr}} * h'[\tilde{\theta}]) \downarrow \tilde{\theta}(p^{\mathrm{ret}}))\}}$$

Figure 6. Symbolic execution rules to switch to the typed analysis and to apply symbolic summaries.

A related lemma describes summarization:

Lemma 3 (Soundness of Type-Consistent Summarization for States)**.** If $\widetilde{\Gamma},\tilde{o} \xrightarrow{\text{typeify}} B$ and $\mathsf{okheap} \in \widetilde{H}$ and $\widetilde{\Gamma}(\widetilde{a}) = B \uparrow \cdots$ then for all $\widetilde{E}$ we have $\gamma(\widetilde{E},(\widetilde{\Gamma},\widetilde{H} * \widetilde{a} : \tilde{o})) \subseteq \gamma(\widetilde{E},(\widetilde{\Gamma},\widetilde{H}))$.

That is, if the symbolic execution determines that a materialized symbolic object is not immediately type inconsistent, then it can summarize the symbolic storage for that object back into okheap (rule M-SUMMARIZE) without unsoundly dropping concrete states.

We rely on the above lemmas to prove soundness of the intertwined analysis:

Theorem 1. FISSILE *Type Analysis is sound. That is, if* $E \vdash [H]\, e\, [r]$ *then*

1. *If* $\Gamma^{\mathrm{L}} \vdash e : T^{\mathrm{L}}$ *and* $(E,H) \in \gamma(\Gamma^{\mathrm{L}})$ *then* $r = H' \downarrow v'$ *where* $(E,H') \in \gamma(\Gamma^{\mathrm{L}})$ *and* $(E,H',v') \in \gamma(T^{\mathrm{L}})$*; and*
2. *If* $\widetilde{E} \vdash \{\widetilde{\Sigma}\}\, e\, \{\widetilde{P}\}$ *and* $(E,H) \in \gamma(\widetilde{E},\widetilde{\Sigma})$ *then* $r = H' \downarrow v'$ *where* $(E,H',v') \in \gamma(\widetilde{E},\widetilde{P})$.

This is a fairly standard statement of soundness of for both analyses. Here, $E \vdash [H]\, e\, [r]$ says that in a concrete environment E and with a concrete heap H, an expression e big-step evaluates to a result r. For types, if the initial concrete state is described by the static type environment and the expression type checks, then the expression evaluates to a heap-value pair (that is, not to an error) where the final state still conforms to the type environment and the value conforms to the static type. Similarly, the symbolic analysis soundly over-approximates the concrete execution and rules out a faulting error. The proof proceeds by induction on the derivation of concrete execution and covers an additional judgment form describing generalized path-to-path symbolic analysis—we provide full details in our TR [13].

3.5 Fully-Intertwined Type-Symbolic Analysis

As described in Section 3.3, the type analysis can hand off checking of an expression to the symbolic analysis. We also would like to perform the opposite handoff from symbolic to type (i.e., apply type analysis within symbolic analysis). For example, we would like do this handoff when we encounter a method call during symbolic analysis to enable modular analysis with only type specifications.

Fortunately, we can employ the same consistency mechanisms to allow handoff in the other direction, that is, to switch from symbolic analysis to type checking (rule SYM-TYPE-HANDOFF in Figure 6). The key constraint is that the entire heap must be fully type-consistent (okheap). In general, this requirement means summarization should have been applied so that no locations are materialized. The entire state is checked type consistent via "typeify" to Γ before type checking, and then symbolic analysis can resume by "symbolize"-ing a new symbolic state from Γ and adding an assumption from the type of the expression.

Suppose this symbolic analysis is the context of an outer flow-insensitive type analysis. Observe that type environment Γ is derived solely from the current symbolic execution state and could be more precise than the outer flow-insensitive invariant. A common scenario is to leverage symbolic reasoning about guard conditions in an `if` but switch to type analysis inside the `if` body with a stronger invariant without needing to the make any changes to the type analysis (cf. occurrence typing [35]). For the reflective call safety client, the "checked delegate" idiom is quite common where a reflective call is performed after checking if an object responds to a particular string—`respondsToSelector:` in Objective-C.

Symbolic summaries for cross-module bounded violations. On the other hand, we have seen a handful of cases where a further precision refinement is needed: when programmers break relationship refinements across module boundaries, such as when they abstract the heap with getters and setters. To see how this is a problem, consider the slightly modified version of the update function in which the direct field access is replaced the following setter functions:

```
1    def setDel(d)[del : −/del : d] : − = self.del := d
2    def setSel(s)[sel : −/sel : s] : − = self.sel := s
```

This small change exacerbates the problem of checking relationship refinement safety because now the invariant violation crosses module (function) boundaries and thus cannot go to a fully type-consistent heap on method call. To support this scenario, we enable the programmer to supply additional checked annotations, symbolic method summaries of the form $(\overline{p})[h/h'] : p^{\mathrm{ret}}$, that permit symbolic checking across module boundaries. Here $\overline{p}$ is a sequence of method parameter names, h and h' are summaries of the input heap and output heap respectively, and p^{ret} specifies the method return value. The input summary gives the form of the part of the heap the method will operate upon (the *footprint*) and gives names to the values stored in the fields of the input heap. The output summary gives the form of the heap after the method has finished executing, in terms of the names given in the input heap and the parameters. For example, the method summary annotation at line 1 says that whatever the `del` field stores before `setDel` is called, after the method is executed the field stores the value from the `d` parameter. Since the method is a setter, the return value is irrelevant. The annotation for `setSel` is analogous. Applying the summary for `setDel` leaves the heap in an inconsistent state, but then the summary for `setSel` restores it.

With these annotations, checking the calls to the setters is analogous to checking the field writes as before, except that rather than applying the symbolic transfer function for field writes, we apply the method summary. We formalize this in the SYM-METHOD-CALL-SUMMARY rule in Figure 6. The auxiliary function $\mathrm{summary}(T,m)$ looks up the summary of a method m on a static type T. The judgment $\widetilde{H} \vdash h \circledast \widetilde{H}^{\mathrm{fr}} \backslash \tilde{\theta}^{\mathrm{h}}$ splits the heap $\widetilde{H}$ into a footprint (specified by h) and the left-over frame ($\widetilde{H}^{\mathrm{fr}}$) that the method is guaranteed not to touch. This frame inference also produces a symbolic map $\tilde{\theta}^{\mathrm{h}}$ that captures the symbolic variables that the parameters in the h match to during the splitting. We then apply this map (combined with the mapping from parameter names to the symbolic values passed in as parameters) to the output summary h' to get the footprint on method exit and add it back to the frame to get the entire heap on method exit. We also consult this map $\tilde{\theta}$ to look up the value returned as specified by the summary. We write $\uplus$ for disjoint union of maps where the result is undefined if the union is not a map. The frame inference is a simple application of subtraction [3] but makes this rule quite flexible. The developer can choose to refuse access to okheap in a symbolic summary (as above) to provide a stronger guarantee to the method's callers or request it to get a stronger assumption for checking the method implementation.

We note that the particular kind of symbolic summary that we describe above (essentially, standard pre-/post-conditions in separation logic) is not the most interesting point. Other symbolic analysis techniques could be applied, such as context-sensitive, non-

modular reasoning. Rather, we argue that the interest lies in that heavier-weight symbolic analysis machinery can be applied when needed without the cost of applying it everywhere, all the time.

Checking Method Implementations. For modular checking, we type check each method implementation separately according to its type signature (as is standard). We guarantee that the implementation of a symbolically-summarized method conforms to its summary with our symbolic analysis infrastructure, although the summary checking technique is orthogonal to that for checking relationship refinements. One could substitute a different approach (e.g. abstract interpretation or interactive proofs) if desired.

4. Case Study: Checking Reflective Call Safety

We implemented our FISSILE type analysis approach to checking almost flow-insensitive invariants in a prototype method reflection safety checker for Objective-C. We evaluate our prototype, a plugin to the clang static analyzer, by investigating the following questions: What is the increased type annotation cost for checking reflection safety? How much does the mixed FISSILE approach improve precision? Do our premises about how programmers violate relationships hold in practice? Is our intertwined "almost type" analysis as fast as we hope? We also discuss a bug found by our tool—surprising in a mature application. The bug fix that we proposed was accepted by the application developer.

Table 1 describes our prototype's performance on a benchmark suite of 6 libraries and 3 large applications. The OmniFrwks are noteworthy because they are very large and have been in continuous development since 1997. The fact that our tool can run on them provides evidence for the kind of real world Objective-C that we can handle—something that would be challenging for a purely symbolic analysis. We discuss these results throughout the rest of this section.

The developer cost to add modular reflection checking. To seed potential type errors, we first annotated the reflection requirements on 76 system library functions (i.e., with **respondsTo** refinements). These are requirements imposed by the system API enriched to check for method reflection errors. Then, for each benchmark we report the total number of developer annotations required, as well as the average number required per reflective call site, to give an indication of how much work it would be for developers to modularly check their use of reflection (column "Total Annotations"). All annotations are *checked*—they emit a static type error if their requirements are not met. Based on this column, we observe that our benchmarks fall into three categories, depending on how they use reflection. *Clients* of reflective APIs, such as Sparkle and ZipKit, have a very low (essentially zero) annotation burden. In contrast, benchmarks that *expose* reflective interfaces, such as SCRecorder and OAuth have a higher annotation burden. This is perhaps not surprising, since annotations are the mechanism through which interfaces expose requirements to clients. In the middle are those that use reflection in both ways: parts of OmniFrwks do expose a reflective API, but they also use internal reflection quite significantly. Our application benchmarks also fall in this category: they are structured into modular application frameworks and a core application client.

Over our entire benchmark suite we find that we need 0.10 annotations and 0.01 symbolic summaries per reflective callsite (row "Combined," columns "Total Annotations" and "Symbolic Annotations"). In other words, on average, the programmer should expect to write one annotation for every 10 uses of reflection and a single symbolic heap effect summary for every 100 uses of reflection. Importantly, note that almost all of annotations are extremely lightweight refinement annotations, like **respondsTo**—only 0.05% of methods required a symbolic summary. Even there, the summaries were very simple because they were on leaf methods, such as setters. This overall low annotation burden highlights a key benefit of our optimistic mostly flow-insensitive approach: whenever the reflection relationship is preserved flow-insensitively, no method summary is needed. Contrast this to a modular flow-sensitive approach where a summary is needed on all methods to describe their potential effects on reflection-related fields.

Improved precision. We verified reflection safety on our benchmarks using two configurations: a completely flow-insensitive analysis (no switching) and our mixed FISSILE approach. We then compared the number of static type errors reported by each (columns "FI Type Errors" and "FISSILE Type Errors"). FISSILE type analysis sometimes significantly reduces the number of static type alarms (e.g., ASIHTTP)—and by 29% in our combined benchmark suite.

The number of FISSILE static type alarms ranges from 0 (for SCRecorder and ZipKit) to 74 (for OmniFrwks, our most challenging benchmark). Pessimistically viewing our tool as a post-development analysis, we manually triaged all the reported static type errors to determine if they could manifest at run-time as true bugs (see discussion on bugs below) or otherwise are false alarms due to static over-approximation. The single biggest source of false alarms were reflection calls on objects pulled from collection classes. Retrofitting Objective-C's underlying type system for parametric polymorphism (like, what has been for Java with generics) would directly improve precision for this case. At the same time, as discussed below, the efficiency of FISSILE makes it feasible to instead consider it as a development-time type checker where a small number code rewritings or cast insertions are not unreasonable (especially if most casts would go away altogether with generic types).

Premises. We designed FISSILE type analysis around two core premises (Section 2.2): (1) that most of the program can be checked flow-insensitively and (2) that even when a flow-insensitive relationship between heap storage locations is violated, most other relationships on the heap remain intact. As Table 1 shows, the number of times the analysis switches to symbolic execution and back (column "Successful Symbolic Sections") is quite low, even for large programs—Premise 1 appears to hold empirically. The maximum number of simultaneous materializations (column "Max. Mats.) is also low—Premise 2 holds as well empirically. Note that we need more than the single materialization that would be possible with a non-disjunctive flow-sensitive analysis.

Modular reflection checking at interactive speeds. Our two core premises hold, enabling FISSILE type analysis to soundly verify "almost everywhere" invariants quickly. Analysis times range from less than a second for our smaller (around 1,000 lines of code) benchmarks to around 9 seconds (for our largest, about 180,000 lines of code). These results (column "Analysis Time") include only the time to run our analysis: they do not include parsing or clang's base type checker. Our goal with these measurements is to determine the additional compile-time penalty a developer would incur when adding our analysis to her existing work-flow. Expressed as a rate (thousands of lines of code per second), our analysis ranges from about 5 kloc/s to around 38 kloc/s, with a weighted average of 23.0 kloc/s. In general, the larger benchmarks show a faster rate because they amortize the high cost of checking system headers (which are typically more than 100 kloc) over larger compilation units.

Finding bugs. When running our tool on the Vienna benchmark, we found a real reflection bug in a mature application:

```
NSNotificationCenter *nc = [NSNotificationCenter defaultCenter];
[nc addObserver:self selector:"autoCollapseFolder"
          name:"MA_Notify_AutoCollapseFolder" object:nil];
```

Here an object registers interest in being notified whenever any code in the project auto-collapses a folder. This notification takes the form of a reflective callback: the `autoCollapseFolder` method of self will be called. Unfortunately, self has no such method. Our

Benchmark	Lines of Code	Refl. Call Sites	Methods	Total Annotations / Per Refl. Site	Symbolic Annotations / Per Refl. Site	Check Sites	FI Type Errors	FISSILE Type Errors (% Reduced)	Successful Symbolic Sections	Max. Mats.	Analysis Time (Rate)
OAuth	1248	7	92	5 / 0.71	0 / 0.00	11	7	2 (- 71%)	7	1	0.24s (5.3 kloc/s)
SCRecorder	2716	12	200	9 / 0.75	4 / 0.33	15	2	0 (-100%)	2	2	0.28s (10.8 kloc/s)
ZipKit	3301	28	165	0 / 0.00	0 / 0.00	28	0	0 (–)	0	0	0.10s (33.0 kloc/s)
Sparkle	5290	40	320	0 / 0.00	0 / 0.00	40	4	1 (- 75%)	3	1	0.67s (7.9 kloc/s)
ASIHTTP	13565	68	707	2 / 0.03	2 / 0.03	68	50	10 (- 80%)	59	2	0.50s (27.2 kloc/s)
OmniFrwks	160769	192	7611	49 / 0.26	2 / 0.01	259	82	74 (- 10%)	9	1	4.25s (37.8 kloc/s)
Vienna	37348	186	2261	24 / 0.13	4 / 0.02	207	59	38 (- 36%)	28	2	2.79s (13.4 kloc/s)
Skim	60211	207	3010	7 / 0.03	0 / 0.00	212	43	43 (- 0%)	0	0	2.49s (24.1 kloc/s)
Adium	176632	587	8723	40 / 0.07	0 / 0.00	648	87	70 (- 20%)	17	1	8.79s (20.1 kloc/s)
Combined	461080	1327	23089	136 / 0.10	12 / 0.01	1488	334	238 (- 29%)	125	2	20.09s (23.0 kloc/s)

Table 1. The "Lines of Code" count includes project headers but excludes comments and whitespace; "Methods" indicates the total number of methods; "Refl. Call Sites" gives the number of calls to system library methods that perform reflection, either directly or as part of some other operation; "Total Annotations" lists the total number of annotations and the average number of annotations required per reflective callsite; "Symbolic Annotations" lists gives the number of symbolic summaries required; "Check Sites" gives the number of program sites where some annotation was checked; "FI Type Errors" indicates the number of check sites where a flow-insensitive type analysis produces a type error; "FISSILE Type Errors" indicates the number of check sites where we emit a static type error and the corresponding percent reduction from the flow-insensitive approach; "Successful Symbolic Sections" gives the number of times our analysis successfully switched from type checking to symbolic execution and back again; "Max. Mats." gives the maximum number of materialized objects ever present in the explicit heap (this includes unsuccessful symbolic sections); and "Analysis Time" indicates the speed of our analysis on each benchmark, in both absolute terms and in lines of code per second. Our benchmarks include OAuth, which performs OAuth Consumer authentication; SCRecorder, which records custom keyboard shortcuts and is the source of our motivating example in Section 2; ZipKit, which reads and writes compressed archives; Sparkle, a widely-used automatic updater; ASIHTTP, which performs web services calls; and the OmniFrwks, which provide base functionality to the widely used OmniGraffle application; Vienna, an RSS newsreader; Skim, a PDF reader; and Adium, an instant message chat client. The "Combined" row treats all of the benchmarks together as a combined workload. Experiments were performed on a 4-core 2.6 GHz Intel Core i7 laptop with 16GB of RAM running OS X 10.8.2. We used clang 3.2 (trunk 165236) compiled in "Release+Asserts" mode to perform the analysis and xcodebuild 4.6/4H127 to drive the build.

analysis detects this error and issues an alarm. We reported the bug to the developers; they acknowledged it as a bug and fixed it (see `https://github.com/ViennaRSS/vienna-rss/pull/85`).

Our tool was also useful in finding bugs in beginner Objective-C code. We used it to statically detect run-time errors in 12 code snippets culled from mailing lists and discussion forums. These novice reflective errors fell into three different categories: (1) typos in selector names, (2) intending to reflectively call a method with a selector stored in a variable but instead passing in a constant selector with the *name* of the variable, and (3) passing the wrong responder into a reflective call, typically a field of self instead of self itself. These results show that our tool can statically detect a common class of novice errors; they provide evidence in favor of including reflective call checking with FISSILE type analysis in the compiler.

5. Related Work

Dependent refinement types [20, 38] enable programmers to restrict types based on the value of program expressions and thus rule out certain classes of run-time errors, such as out-of-bounds array accesses. Extending dependent types to imperative languages [34, 37] has generally led to flow-sensitive type systems because mutation may change the value of a variable referred to in a type. The high burden that flow-sensitive type annotations impose on the programmer motivates sophisticated inference schemes [31], of which CSOLVE [32] is perhaps the closest work to ours. In contrast to CSOLVE, which performs flow-sensitive checking of inferred flow-sensitive types with at most one materialization, we use path-sensitive checking of flow-insensitive annotations [12] and support arbitrary materialization with a disjunctive symbolic analysis, as opposed to proving non-aliasing for one materialization (e.g., [1, 2, 18]). DJS [11] checks dependent refinements in JavaScript, including the safety of dynamic field accesses—a problem similar to reflective method call safety—but supports only single materialization and employs a flow-sensitive heap.

There has been a recent explosion in techniques (e.g., [7, 22]) that have significantly improved the effectiveness of symbolic execution. The SMPP approach [24] leverages SMT technology combined with abstract interpretation on path programs to lift a symbolic-execution–based technique to exhaustive verification. This technique can be seen as applying a fixed one level of analysis switching between a top-level symbolic executor and an abstract interpreter for loops. Our approach of switching between type checking and symbolic execution is similar to the MIX system [25] for simple types. A significant difference is that our approach enables the symbolic executor to leverage the heap-consistency invariant enforced by the type analysis through a type-consistent materialization operation, which is critical for our rich refinement relationship invariants, whereas the symbolic and type analyses in MIX interact minimally with respect to the heap. The notion of temporary violations of an invariant is also reminiscent of the large body of work on object invariants (see [17] for an overview). We remark on two perspective differences that make FISSILE complementary to this work. First, the points where the invariant is assumed and where they may be violated is not based on the program structure (e.g., inside a method or not) but instead is based on the analysis being applied (i.e., type or symbolic). Second, the symbolic analysis takes a more global view of the heap and decides specifically which objects may violate the global type invariant. Issues like reentrancy and multi-object invariants are not as salient in FISSILE, but are possible at the cost of separate symbolic summaries or more expensive, disjunctive analysis in certain complex situations.

On materialized heap locations, our symbolic analysis works over separation logic [3, 30] formulas. We define an on-demand materialization [33] that is universal in separation-logic–based analyzers [4, 9, 16]. However, our materialization operator pulls out heap cells that are summarized and validated *independently* using a refinement type analysis. Bi-abductive shape analyses [8, 23] are modular analyses that try to infer a symbolic summary for each method. Our analysis is modular using a fast, flow-insensitive type analysis with few uses of symbolic summaries. Bi-abduction and our technique could complement each other nicely in that (1) we do not require symbolic summaries on all methods—only those that violate type consistency across method boundaries—and (2) bi-abduction could be applied to generate candidate symbolic summaries.

Most prior work on reflection analysis has focused on whole-program *resolution*: determining, at a reflective site, what method is called (either statically [6, 10, 36] or dynamically [5, 21]).

We address the problem of modular static checking of reflective call safety: ensuring that the receiver responds to the selector, in languages with imperative update. Politz et al. [29] describe a type system that modularly checks reflection safety by combining occurrence typing [35] with first-class member names specified by string patterns. In contrast, we treat the "responds-to" *relationship* as first-class (i.e., we permit the user to specify it with a dependent refinement), allowing us to (1) check relationships between mutable fields and (2) express that an object responds to two completely unknown (i.e. potentially identical) selectors. Livshits et al. [27] *assume* reflection safety and leverage this assumption to improve precision of callgraph construction.

6. Conclusion

We have described FISSILE type analysis, which intertwines fast type analysis over storage locations and precise symbolic analysis over values to efficiently, effectively, and modularly prove relationship properties. We have evaluated FISSILE using an interesting safety property—reflective method call safety. The key technical enabler for our analysis is *materialization from an almost type-consistent heap*. On a benchmark suite consisting of commonly-used Objective-C libraries (6) and applications (3), we find that our approach is capable of finding confirmed bugs in both production and beginner code. It has a balanced annotation burden that is negligible for clients of reflection and moderate (up to 0.75 annotations per reflective call site) for reflective interfaces.

FISSILE type analysis starts with the optimistic assumption that global flow-insensitive relationship invariants hold almost everywhere—it only has to use precise and expensive reasoning for those few program locations that violate a relationship. Contrast this to traditional modular flow-sensitive analyses, which require *all* methods to specify their effect on the heap to rule out the pessimistic assumption that relationship invariants could be violated anywhere. Our approach permits the vast majority (99.95%) of methods to avoid any annotations related to heap effects. Checking most of the program flow-insensitively allows FISSILE to validate reflective method call safety at interactive speeds (5 to 38 kloc/s with an overall rate of 23.0 kloc/s over our benchmark suite).

Acknowledgments

We thank Sriram Sankaranarayanan, Amer Diwan, Jeremy Siek, the CUPLV group, and Ranjit Jhala for insightful discussions, as well as the anonymous reviewers for their helpful comments. This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-1218208 and CCF-1055066.

References

[1] A. Ahmed, M. Fluet, and G. Morrisett. L^3: A linear language with locations. *Fundam. Inform.*, 77(4), 2007.

[2] A. Aiken, J. S. Foster, J. Kodumal, and T. Terauchi. Checking and inferring local non-aliasing. In *PLDI*, 2003.

[3] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.

[4] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.

[5] E. Bodden, A. Sewe, J. Sinschek, H. Oueslati, and M. Mezini. Taming Reflection: Aiding static analysis in the presence of reflection and custom class loaders. In *ICSE*, 2011.

[6] M. Braux and J. Noyé. Towards partially evaluating reflection in Java. In *PEPM*, 2000.

[7] C. Cadar, D. Dunbar, and D. R. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[8] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.

[9] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.

[10] A. S. Christensen, A. Møller, and M. I. Schwartzbach. Precise analysis of string expressions. In *SAS*, 2003.

[11] R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *OOPSLA*, 2012.

[12] J. Condit, M. Harren, Z. R. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *ESOP*, 2007.

[13] D. Coughlin and B.-Y. E. Chang. Fissile Type Analysis: Modular checking of almost everywhere invariants (extended version), 2013.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[15] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.

[16] D. Distefano, P. W. O'Hearn, and H. Yang. A local shape analysis based on separation logic. In *TACAS*, 2006.

[17] S. Drossopoulou, A. Francalanza, P. Müller, and A. J. Summers. A unified framework for verification techniques for object invariants. In *ECOOP*, 2008.

[18] M. Fähndrich and R. DeLine. Adoption and focus: Practical linear types for imperative programming. In *PLDI*, 2002.

[19] C. Flanagan. Hybrid type checking. In *POPL*, 2006.

[20] T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, 1991.

[21] M. Furr, J.-h. D. An, and J. S. Foster. Profile-guided static typing for dynamic scripting languages. In *OOPSLA*, 2009.

[22] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *PLDI*, 2005.

[23] B. S. Gulavani, S. Chakraborty, G. Ramalingam, and A. V. Nori. Bottom-up shape analysis. In *SAS*, 2009.

[24] W. R. Harris, S. Sankaranarayanan, F. Ivancic, and A. Gupta. Program analysis via satisfiability modulo path programs. In *POPL*, 2010.

[25] Y. P. Khoo, B.-Y. E. Chang, and J. S. Foster. Mixing type checking and symbolic execution. In *PLDI*, 2010.

[26] V. Laviron, B.-Y. E. Chang, and X. Rival. Separating shape graphs. In *ESOP*, 2010.

[27] B. Livshits, J. Whaley, and M. S. Lam. Reflection analysis for Java. In *APLAS*, 2005.

[28] M. J. Parkinson. *Local Reasoning for Java*. PhD thesis, University of Cambridge, Computer Laboratory, 2005.

[29] J. G. Politz, A. Guha, and S. Krishnamurthi. Semantics and types for objects with first-class member names. In *FOOL*, 2012.

[30] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[31] P. M. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.

[32] P. M. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, 2010.

[33] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1), 1998.

[34] R. Tate, J. Chen, and C. Hawblitzel. Inferable object-oriented typed assembly language. In *PLDI*, 2010.

[35] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *POPL*, 2008.

[36] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman. TAJ: Effective taint analysis of web applications. In *PLDI*, 2009.

[37] H. Xi. Imperative programming with dependent types. In *LICS*, 2000.

[38] H. Xi and F. Pfenning. Dependent types in practical programming. In *POPL*, 1999.

Thresher: Precise Refutations for Heap Reachability

Sam Blackshear
University of Colorado Boulder
samuel.blackshear@colorado.edu

Bor-Yuh Evan Chang
University of Colorado Boulder
evan.chang@colorado.edu

Manu Sridharan
IBM T.J. Watson Research Center
msridhar@us.ibm.com

Abstract

We present a precise, path-sensitive static analysis for reasoning about *heap reachability*; that is, whether an object can be reached from another variable or object via pointer dereferences. Precise reachability information is useful for a number of clients, including static detection of a class of Android memory leaks. For this client, we found that the heap reachability information computed by a state-of-the-art points-to analysis was too imprecise, leading to numerous false-positive leak reports. Our analysis combines a symbolic execution capable of path-sensitivity and strong updates with abstract heap information computed by an initial flow-insensitive points-to analysis. This novel *mixed* representation allows us to achieve both precision and scalability by leveraging the pre-computed points-to facts to guide execution and prune infeasible paths. We have evaluated our techniques in the THRESHER tool, which we used to find several developer-confirmed leaks in Android applications.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification

Keywords heap reachability; path-sensitive analysis; symbolic execution

1. Introduction

Static reasoning about nearly any non-trivial property of modern programs requires effective analysis of heap properties. In particular, a heap analysis can be used to reason about *heap reachability*—whether one heap object is reachable from another via pointer dereferences at some program point. Precise heap reachability information improves heap-intensive static analyses, such as escape analysis, taint analysis, and cast checking. A heap reachability checker would also enable a developer to write statically checkable assertions about, for example, object lifetimes, encapsulation of fields, or immutability of objects.

Our interest in heap-reachability analysis arose while developing a tool for detecting an important class of memory leaks in Android applications. Briefly, such a leak occurs when an object of type Activity remains reachable from a static variable after the end of its life cycle and thus cannot be freed by the garbage collector (explained further in Sections 2 and 4). For this client, we found that highly precise reasoning about heap reachability, including flow-, context-, *and* path-sensitivity *with* materialization [43], was required to avoid emitting too many spurious warnings. We are unaware of an existing analysis that can provide such precision for heap-reachability queries while scaling to our target applications (40K SLOC with up to 1.1M SLOC of libraries). While approaches based on predicate abstraction or symbolic execution [2, 4, 11, 31] could provide the necessary precision in principle, to our best knowledge such approaches have not been shown to handle heap-reachability queries for real-world object-oriented programs (further discussion in Section 5).

We present an analysis for precise reasoning about heap reachability via on-demand refinement of a flow-insensitive points-to analysis. When the points-to analysis cannot refute a heap-reachability query Q, our technique performs a backwards search for a path program [7] witnessing Q; a failed search means Q is refuted. Our analysis is flow-, context-, and path-sensitive with location materialization, yielding the precision required by clients like the Android memory leak detector in an on-demand fashion.

In contrast to previous approaches to refinement-based or staged heap analysis [25, 26, 28, 36, 44], our approach refines points-to facts directly and is capable of path-sensitive reasoning. Also, unlike some prior approaches [28, 36, 44], our analysis does *not* attempt to refine the heap abstraction of the initial points-to analysis. Instead, we focus on the orthogonal problem of on-demand flow- and path-sensitive reasoning given a points-to analysis with a sufficiently precise heap abstraction.

Our analysis achieves increased scalability through two novel uses of the initial points-to analysis result. First, we introduce a *mixed symbolic-explicit* representation of heap reachability queries and transfer functions that simultaneously enables strong updates during symbolic execution, exploits the initial points-to analysis result, and mitigates the case split explosion seen with a fully explicit representation. Crucially, our representation allows the analysis to avoid case-splitting over entire points-to sets when handling heap writes, key to scaling beyond small programs. We also maintain *instance constraints* for memory locations during analysis, based on points-to facts, and use these constraints during query simplification to obtain contradictions earlier, improving performance.

Second, our analysis utilizes points-to facts to handle loops effectively, a well-known source of problems for symbolic analyses. Traditionally, such analyses either require loop invariant annotations or only analyze loops to some fixed bound. We give a sound algorithm for *inferring loop invariants over points-to constraints on path programs* during backwards symbolic execution. Our algorithm takes advantage of the heap abstraction computed by the up-front points-to analysis to effectively over-approximate the effects of the loop during the backwards refinement analysis (Section 3.3). Our technique maintains constraints from path conditions separately from the heap reachability constraints, enabling heuristic "dropping" of path constraints at loops to avoid divergence without significant precision loss in most cases.

PLDI'13, June 16–19, 2013, Seattle, WA, USA.

Contributions. This paper makes the following contributions:

- We describe a technique for refining a points-to analysis with strong updates, path sensitivity, and context sensitivity to improve precision for heap-reachability queries. Our technique centers on enriching a backwards symbolic execution with the over-approximate information from the up-front points-to analysis to guide the execution and prune infeasible paths.
- We introduce a *mixed symbolic-explicit* representation of heap reachability queries and corresponding transfer functions that enable strong updates and precise disaliasing information during symbolic execution while reducing case splitting compared to a fully explicit approach. This representation is important both for finding contradictions early and for collapsing paths effectively.
- We present a method for *inferring loop invariants for heap-reachability queries* during symbolic execution based on over-approximating the heap effects of loops and dropping path constraints that may lead to divergence.
- We demonstrate the usefulness of our analysis by applying it to the problem of finding memory leaks in Android applications. Our tool refuted many of the false alarms that a flow-insensitive analysis reported for real-world Android applications and finished in less than twenty minutes on most programs we tried. We used our tool to successfully identify real (and developer-confirmed) memory leaks in several applications.

2. Overview

Here we present a detailed example to motivate our precise heap-reachability analysis and illustrate our technique. The example is based on real code from Android applications and libraries, and verifying the desired heap-reachability property requires strong updates, context sensitivity, *and* path sensitivity, which our technique provides in an on-demand fashion.

Our techniques were motivated by the need to detect leaks of Activity objects in Android applications. Every Android application has least one associated Activity object to control the user interface. Android development guidelines state that application code should not maintain long-lived pointers to Activity objects, as such pointers prevent the objects from being garbage collected at the end of their lifetimes, causing significant memory leaks (we discuss this issue further in Section 4). To detect such leaks in practice, it is sufficient to verify that Activity objects are *never* reachable from a static field via object pointers.

Figure 1 is a simple application that illustrates the difficulties of precisely checking this heap reachability property (ignore the boxed assertions for now). The `Main` class initializes and starts the application's Activity, the `Act` class. The `Vec` class captures the essence of a list data structure, as implemented in Android. This example is free of the leak described above, as the `Act` object allocated on line 1 is never made reachable from a static field.

For this example program, flow-insensitive points-to analysis techniques cannot prove the desired heap (un)reachability property due to the manner in which the `Vec` class is implemented. In the inset, we show a heap graph obtained by applying Andersen's analysis [1] with one level of object sensitivity [40] to the example. Graph nodes represent classes or abstract locations (whose names are shown at the corresponding allocation site in Figure 1), and edges represent possible values of field pointers. Object-sensitive abstract locations are named appropriately, for example, $\mathsf{vec}_0.\mathsf{arr}_1$ for arr_1 instances allocated when `Vec.push(−)` is invoked on instances of vec_0. Each edge indicates a may points-to relationship, written as $a_1 \cdot f \Rightarrow a_2$, meaning there may be an execution where field f of abstract location a_1 contains the address of location a_2. The graph imprecisely shows that

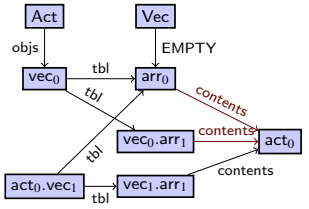


Figure 2.

```
   public class Main {
     public static void main(String[] args) {
1      Act a = new_act0 Act(); a.onCreate();
2    }
   }
   public class Act extends Activity {
     private static final Vec objs = new_vec0 Vec();
     public void onCreate() {
3      Vec acts = new_vec1 Vec();
4                  [ v̂sz^int < v̂cap^int, acts·sz ↦ v̂sz^int, acts·cap ↦ v̂cap^int,
                     this ↦ this, acts ↦ acts, acts·tbl ↦ arr0, this ↦ act0 ]
       acts.push(this
5                  [ v̂sz^int < v̂cap^int, this·sz ↦ v̂sz^int, this·cap ↦ v̂cap^int,
                     this ↦ vec1, vec1·tbl ↦ arr0, val ↦ act0 ]
       );
6      ...
7      objs.push("hello"
8                  [ v̂sz^int < v̂cap^int, this·sz ↦ v̂sz^int, this·cap ↦ v̂cap^int,
                     this ↦ vec0, vec0·tbl ↦ arr0, val ↦ act0 ] †
       );
9    }
   }
   public class Vec {
     private static final Object[] EMPTY = new_arr0 Object[1];
     private int sz; private int cap; private Object[] tbl;
     public Vec() {
10     this.sz = 0; this.cap = -1; this.tbl = EMPTY;
11                 [ v̂sz^int < v̂cap^int, this·sz ↦ v̂sz^int, this·cap ↦ v̂cap^int,
                     this ↦ this, this·tbl ↦ arr0 ] †
     }
     public void push(Object val) {
12     Object[] oldtbl = this.tbl;
13                 [ v̂sz^int < v̂cap^int, this·sz ↦ v̂sz^int, this·cap ↦ v̂cap^int,
                     this ↦ this, this·tbl ↦ arr0, val ↦ act0 ]
       if (this.sz >= this.cap) {
14       this.cap = this.tbl.length * 2;
15       this.tbl = new_arr1 Object[this.cap];
16                 [ this ↦ this, this·tbl ↦ arr0, val ↦ act0 ] †
         for (int i = 0; i < this.sz; i++) {
17         this.tbl[i] = oldtbl[i]; // copy from oldtbl
18       }
19     }
20                 [ this ↦ this, this·tbl ↦ arr0, val ↦ act0 ]
       this.tbl[this.sz] = val;
21                 [ arr0·contents ↦ act0 ]
       this.sz = this.sz + 1;
22   }
   }
```

Figure 1. Refuting a false alarm with context-sensitive, path-sensitive witness search. We show witness queries in the boxes. The † indicate refuted branches of the witness search.

Activity object act_0 is reachable from both static fields Act·objs and Vec·EMPTY, hence falsely indicating that a leak is possible.

The root cause of the imprecision in Figure 2 is the edge $\mathsf{arr}_0 \cdot \mathsf{contents} \mapsto \mathsf{act}_0$, which indicates that the array assigned to Vec·EMPTY may contain the Activity object. `Vec` is implemented using the *null object pattern* [45]: rather than allocating a separate internal array for each empty `Vec`, all `Vec` objects initially use Vec·EMPTY as their internal array. The code in `Vec` is carefully written to avoid ever adding objects to Vec·EMPTY while also avoiding additional branches to check for the empty case. But, a flow-insensitive points-to analysis is incapable of reasoning precisely about this code, and hence it models the statement **this**`.tbl[`**this**`.sz] = val` on line 20 as possibly writing to Vec·EMPTY, polluting the points-to graph. Real Android collections are implemented in this way.[1] Note that a more precise heap abstraction would not help in this case—because the (concrete) null object is shared among *all* instances the `Vec` class, *no refinement on the heap abstraction* alone would be sufficient to rule out this false alarm.

More precise analysis of this example requires reasoning about the relationship between the `sz` and `cap` fields of each `Vec`. This relationship is established in the `Vec`'s constructor and must be preserved until its `push` method is called. Though there is a large body of work focused on the important problem of refining heap abstractions (e.g., [36, 44]), this example shows that doing so alone is sometimes not sufficient for precise results. An analysis that lacks path sensitivity and strong updates will be unable to prove that `Vec`'s never write into the shared array and must therefore conflate the contents of all `Vec` objects. The *witness-refutation* technique that we detail in this paper enables after-the-fact, on-demand refinement to address this class of *control-precision* issues.

Refinement by Witness Refutation. We refine the results of the flow-insensitive points-to analysis by attempting to refute all executions that could possibly witness an edge involved in a leak alarm. The term "witness" is highly overloaded in the program analysis literature, so we begin by carefully defining its use in our context. We first define a *path* as a sequence of program transitions. A *path witness* for a *query* Q is a path that ends in a state that satisfies Q. Such a path witness may be concrete/under-approximate/must in that it describes a sequence of program transitions according to a concrete semantics that results in a state where Q holds (e.g., a test case execution). Analogously, an abstract/over-approximate/may path witness is such a sequence over an abstract semantics (e.g., a trace in an abstract interpreter). Building on the definition of a path program [7], we define a *path program witness* for a query Q as a path program that ends in a state satisfying Q. A *path program* is a program projected onto the transitions of a given execution trace (essentially, paths augmented with loops). Note that a path program witness may be under- or over-approximate in its handling of loops. In this paper, we use the term "witness" to refer to over-approximate path program witnesses unless otherwise stated, as our focus is on sound refutation of queries.

Our analysis performs a goal-directed, backwards search for a path program witness ending in a state that satisfies a query Q. We *witness* a query by giving a witness that produces it, or we *refute* a query by proving that no such witness can exist. Our technique proceeds in three phases.

Obtain a Conservative Analysis Result. First, we perform a standard points-to analysis to compute an over-approximation of the set of reachable heaps, such as the points-to graph in Figure 2.

[1] In fact, we discovered buggy logic in the actual Android libraries that allowed writing to a null object, thereby polluting all empty containers! The bug was acknowledged and fixed by Google (`https://code.google.com/p/android/issues/detail?id=48055`).

Formulate Queries. Second, we formulate queries to refute alarms generated using the points-to analysis result. For the Activity leak detection client, an alarm is a points-to path between a **static** field and an Activity object. For example, the following points-to path from the graph in Figure 2 is a (false) leak alarm:

$$\mathsf{Act}\cdot\mathsf{objs} \Mapsto \mathsf{vec}_0, \mathsf{vec}_0\cdot\mathsf{tbl} \Mapsto \mathsf{arr}_0, \mathsf{arr}_0\cdot\mathsf{contents} \Mapsto \mathsf{act}_0$$

To refute an alarm, we attempt to refute each individual edge in the corresponding points-to path. If we witness all edges in the path, we report a leak alarm. If we refute some edge e in the path, we delete e from the points-to graph and attempt to find another path between the source node and the sink node. If we find such a path, we restart the process with the new path. If we refute enough edges to disconnect the source and sink in the points-to graph, we have shown that the alarm raised by the flow-insensitive points-to analysis is false.

For our client, we wish to show the flow-insensitive property that a particular points-to constraint cannot hold at any program point. Thus, for each points-to edge e to witness, we consider a query for e at every program statement that could produce e. This information can be obtained by simple post-processing or instrumentation of the up-front points-to analysis [8].

Search for Witnesses. Finally, given a query Q at a particular program point, we search for path program witnesses on demand. In Figure 1, we illustrate a witness search that produces a refutation for the points-to constraint $\mathsf{arr}_0\cdot\mathsf{contents} \mapsto \mathsf{act}_0$ holding at program point 21. That is, we prove that the points-to constraint is unrealizable at that program point. By starting a witness search from each statement that potentially produces the edge, we will see that $\mathsf{arr}_0\cdot\mathsf{contents} \mapsto \mathsf{act}_0$ is in fact unrealizable at any program point.

Notationally, we use a single arrow $\mapsto$ to denote an *exact points-to constraint*, whose source and sink are symbolic values typically denoting addresses of memory cells, and a double arrow $\Mapsto$ to denote a may points-to edge between abstract locations (cf., Section 3.1). For example, the exact points-to constraint $\mathsf{arr}_0\cdot\mathsf{contents} \mapsto \mathsf{act}_0$ describes a single memory cell whose address is some instance in the concretization of arr_0 and contents is some instance in the concretization of act_0. This distinction is critical for enabling strong updates in a backwards analysis.

2.1 Mixed Symbolic-Explicit Queries

We illustrate the witness search by showing the sub-queries (boxed) that arise as the search progresses. Moving backwards from our starting point at line 21, the sub-query at program point 20 says that we need the following heap state at that point:

$$\mathsf{this} \mapsto \widehat{this}, \widehat{this}\cdot\mathsf{tbl} \mapsto \mathsf{arr}_0, \mathsf{val} \mapsto \mathsf{act}_0 \qquad (\dagger)$$

where $\widehat{this}$ is a *symbolic variable* that represents the receiver of the method. A symbolic variable (written as a hatted letter $\widehat{v}$) is an existential standing for an arbitrary instance drawn from some definite set of abstract locations. Here, *this* represents some instance drawn from the points-to set of local variable this, which is $\{\mathsf{vec}_0, \mathsf{vec}_1\}$. We represent this fact with an *instance constraint*:

$$\widehat{this}\ \mathsf{from}\ \{\mathsf{vec}_0, \mathsf{vec}_1\} \qquad (\ddagger)$$

that we track as part of the query at program point 20. For the moment, we elide such 'from' constraints and discuss them further in Section 2.2 and Section 3. This sub-query conjoining the heap state from (†) and the instance constraint from (‡) is an example of a *mixed* symbolic-explicit state because we introduce a fresh symbolic variable for the contents of **this**, but also have named abstract locations arr_0 and act_0. We say that a query is fully *explicit* if all of its points-to constraints are between named abstract locations from the points-to abstraction. A named abstract location can be seen as a symbolic variable that is constrained to be from a singleton

abstract location set. This connection to the points-to abstraction in an explicit query enables our witness search to prune paths that are inconsistent with the up-front points-to analysis result, as we demonstrate in Section 2.2.

Backwards Path-By-Path Analysis. Returning to the example, the path splits into two prior to program point 20, one path entering the **if** control-flow branch at point 19, the other bypassing the branch to point 13. We consider both possibilities and indicate the fork in Figure 2 by indenting from the right margin. For the path into the branch, the loop between program points 16 and 19 has no effect on the query in question from point 20, so it simply continues backward to program point 16. Observe that because we are only interested in answering a specific query, this irrelevant loop poses no difficulty. At program point 16, we encounter a refutation for this path: the preceding assignment statement writes an instance of arr_1 to the *this*·tbl field, which contradicts the requirement that *this*·tbl hold an instance of arr_0 (underlined). Thus, we have discovered that no concrete program execution can assign a newly allocated array to **this**.tbl at line 15, that is, an instance of arr_1 and then place an Activity object in the EMPTY array at line 20 because **this**.tbl will point to that newly allocated array by then.

Resuming the path that bypasses the **if** branch, the analysis at program point 13 determines that entering the **if** branch changed the query and thus adds a *control-path constraint* to the abstract state indicating that the value of the **this**.sz field (i.e., $\widehat{v}^{\mathsf{int}}_{\mathsf{sz}}$) must be less than the value of the **this**.cap field (i.e., $\widehat{v}^{\mathsf{int}}_{\mathsf{cap}}$). As we will see, tracking the path constraint above is critical to obtaining a refutation for the example. From here, this path reaches the method boundary, leading the analysis to process the possible call sites at program points 8 and 5. The path at program point 8 can be refuted at this point, as the query requires that the val parameter be bound to an instance of act_0 (underlined), but the actual argument is the string `"hello"`. Thus, we have identified that this call to `push` cannot be the reason that an Activity object is placed into the EMPTY array because it is pushing a string, not an Activity.

The other path from the `acts.push` call site (i.e., program point 5) can continue. The query at program point 4 before the call simply changes the program variables of the callee to those of the caller. Continuing this path, we enter the constructor of `Vec` at program point 11. Here, we discover that the values of the `sz` and `cap` fields as initialized in the constructor contradict the control constraint $\widehat{v}^{\mathsf{int}}_{\mathsf{sz}} < \widehat{v}^{\mathsf{int}}_{\mathsf{cap}}$. Intuitively, the witness search has observed the invariant that a `Vec`'s `tbl` field cannot point to the EMPTY array after a call to its `push` method. We have refuted the last path standing, and so we have shown that the statement at line 21 cannot produced the edge $\mathsf{arr}_0{\cdot}\mathsf{contents} \mapsto \mathsf{act}_0$.

In the above, we have been rather informal in describing why certain points-to facts can be propagated back unaffected and why producing certain facts can be done with a strong update-style transfer function. Furthermore, in the example program from Figure 1, there is one (and only one) more statement that could produce the constraint $\mathsf{arr}_0{\cdot}\mathsf{contents} \mapsto \mathsf{act}_0$, which is the assignment at line 17 inside the copy loop. A witness search for this query starting at line 17 leads to a refutation similar to the one described above, but to discover it we must first infer a non-trivial loop invariant. Because we are interested in an over-approximate path program witness-refutation search, we have to obtain loop invariants. We consider these issues further in Section 3.

2.2 Taming Path Explosion From Aliasing

In the previous subsection, we focused on how refutations can occur. For example, backwards paths were pruned at line 15 and at program point 8 because we reach an allocation site that conflicts with our instance constraints in the query (e.g., we need an arr_0 not an arr_1 at line 15). In this section, we present a simple example to explain how our mixed symbolic-explicit representation enables our analysis to derive such contradictions earlier and thus mitigates the *aliasing path explosion problem*.

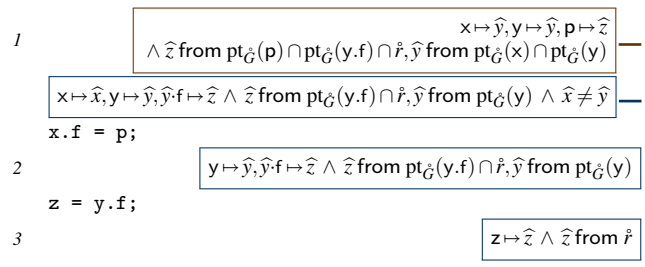


Figure 3. An example illustrating how 'from' constraints help tame path explosion from aliasing. Note that the set of abstract locations to which symbolic variable $\widehat{z}$ might belong is restricted each time we observe $\widehat{z}$ flow through a variable or field.

An over-approximate backwards symbolic executor that lacks information about aliasing will be forced to fork a number of cases to account for aliasing at every field write, quickly causing a case explosion that is worst-case exponential in the number of field writes. This case explosion is independent of (but compounded by) the well-known scalability problems caused by conditional branching in a path-sensitive analysis.

To address this aliasing path explosion problem, the key observation that we make is that contradictions from instance constraints can be derived *before* the allocation site by exploiting information from the up-front points-to analysis. In particular, the set of possible allocation sites for any instance can be restricted as we reason about how they flow into and out of program variables and heap locations. This observation motivates our mixed symbolic-explicit representation, which we demonstrate with a simple example shown in Figure 3. Our initial query at point 3 asks if program variable `z` can point to an instance $\widehat{z}$ from some set of abstract locations $\mathring{r}$—we call this a *points-to region*. Moving backwards across the statement `z = y.f`, we derive a pre-query at point 2 that says our original query can be witnessed if `y` points to some instance $\widehat{y}$ and that instance points to $\widehat{z}$ through its `f` field (i.e., $\mathsf{y} \mapsto \widehat{y}, \widehat{y}{\cdot}\mathsf{f} \mapsto \widehat{z}$). Additionally, we now know that the instance $\widehat{z}$ must be drawn from the *intersection* of $\mathring{r}$ and the abstract locations in the points-to set of y.f, which we write as $\mathrm{pt}_{\mathring{G}}(\mathsf{y.f})$. If we can use the points-to graph $\mathring{G}$ to determine that no such abstract location exists (e.g., if we had $\mathring{r} = \{a_0, a_2\}$ and $\mathrm{pt}_{\mathring{G}}(\widehat{y}.\mathsf{f}) = \{a_1\}$), then we have refuted this query and can prune this path immediately.

Assuming $\mathring{r}$ and $\mathrm{pt}_{\mathring{G}}(\mathsf{y.f})$ are not disjoint (i.e., $\mathring{r} \cap \mathrm{pt}_{\mathring{G}}(\mathsf{y.f}) \neq \emptyset$), we proceed with our backwards analysis to the field write `x.f = p`. We must consider two cases at program point 1: one where `x` and `y` are aliased (the top query) and one where they are not (the bottom query). In the aliased case, we can further constrain the instance $\widehat{u}$ to be from $\mathrm{pt}_{\mathring{G}}(\mathsf{x}) \cap \mathrm{pt}_{\mathring{G}}(\mathsf{y})$. Some previous tools have used an up-front, over-approximate points-to analysis as an aliasing oracle to rule out aliased cases like this one (e.g., PSE [39])—if `x` and `y` cannot possibly point to the same abstract location (i.e., $\mathrm{pt}_{\mathring{G}}(\mathsf{x}) \cap \mathrm{pt}_{\mathring{G}}(\mathsf{y}) = \emptyset$), this aliased case is ruled out. Our approach generalizes this kind of aliasing check by explicitly introducing 'from' constraints that are incrementally restricted. For example, we also constrain $\widehat{z}$ to be from $\mathrm{pt}_{\mathring{G}}(\mathsf{p}) \cap \mathrm{pt}_{\mathring{G}}(\mathsf{y.f}) \cap \mathring{r}$, where the additional restriction is $\mathrm{pt}_{\mathring{G}}(\mathsf{p})$. This constraint says that the field write `x.f = p` produced the query in question only if the instance $\widehat{z}$ is drawn from some abstract location shared by these three sets.

Finally, we consider the case where x and y are not aliased (i.e., $\widehat{x} \neq \widehat{y}$). Here, the only change to the query is the addition of the constraints $\mathsf{x} \mapsto \widehat{x}$ and $\widehat{x} \neq \widehat{y}$. This disequality further constrains the query so that if we later discover that x and y are in fact aliased, we can refute this query. Accumulation of this kind of disaliasing constraint is common (e.g., [11]), but expensive (cf., Section 3.3).

We remark that the instance 'from' constraints can be viewed as a generalization of a fully explicit representation. To represent 'from' constraints explicitly, instead of a symbolic points-to constraint $\mathsf{x} \mapsto \widehat{x}$, we disjunctively consider all cases where we replace the symbolic variable $\widehat{x}$ with an abstract location from $\mathrm{pt}^{\mathring{}}_{\mathring{G}}(\mathsf{x})$ (the points-to set of x) For example, suppose $\mathrm{pt}^{\mathring{}}_{\mathring{G}}(\mathsf{x}) = \{a_1, a_2\}$; then we case split and consider two heap states: (1) $\mathsf{x} \mapsto a_1$ and (2) $\mathsf{x} \mapsto a_2$. This representation corresponds roughly to a backwards extension of *lazy initialization* [33] over abstract locations instead of types. Note that while PSE-style path pruning only applies to ruling out the aliased case in field writes, the explicit representation of 'from' constraints permits the same kind of flow-based restriction shown in Figure 3. However, the cost is case splitting a separate query for each possible abstract location from which each symbolic variable is drawn (e.g., $|\mathrm{pt}^{\mathring{}}_{\mathring{G}}(\mathsf{y.f}) \cap \mathring{r}| \cdot |\mathrm{pt}^{\mathring{}}_{\mathring{G}}(\mathsf{y})|$ queries at program point 2).

3. Refuting Path Program Witnesses

We formalize our witness-refutation analysis and argue that our technique is *refutation sound*—that we only declare an edge refuted when no concrete path producing that query can exist. The language providing the basis for our formalization is defined as follows: This language is a standard imperative programming language with object fields and dynamic memory allocation.

statements	$s ::= c \mid \mathtt{skip} \mid s_1 \,;\, s_2 \mid s_1 \,[]\, s_2 \mid \mathtt{loop}\, s$
commands	$c ::= x := y \mid x := y.f \mid x.f := y \mid x := \mathtt{new}_a\, \tau() \mid \mathtt{assume}\, e$
expressions	$e ::= x \mid \cdots$
types	$\tau ::= \{f_1, \ldots, f_n\} \mid \cdots$
program variables x, y	object fields f abstract locations a

Atomic commands c include assignment, field read, field write, object allocation, and a guard. For the purposes of our discussion, it is sufficient if an object type is just a list of field names. We leave unspecified a sub-language of pure expressions, except that it allows reading of program variables. The label a on allocation names the allocation site so that we can tie it to the points-to analysis. Compound statements include a do-nothing statement, sequencing, non-deterministic branching, and looping. Standard `if` and `while` statements can be defined in the usual way (i.e., $\mathtt{if}\,(e)\, s_1\, \mathtt{else}\, s_2 \stackrel{\text{def}}{=} (\mathtt{assume}\, e \,;\, s_1) \,[]\, (\mathtt{assume}\, !e \,;\, s_2)$ and $\mathtt{while}\,(e)\, s \stackrel{\text{def}}{=} \mathtt{loop}\,(\mathtt{assume}\, e \,;\, s) \,;\, \mathtt{assume}\, !e$).

For ease of presentation, our formal language is intraprocedural. However, our implementation is fully interprocedural. We handle procedure calls by modeling parameter binding using assignment and keeping an explicit abstraction of the call stack. The call stack is initially empty representing an arbitrary calling context but grows as we encounter call instructions during symbolic execution. If we reach the entry block of a function with an empty call stack, we propagate symbolic state backwards to all possible callers of the current function. We determine the set of possible callers using the call graph constructed alongside the points-to analysis.

3.1 Formulating a Witness Query over Heap Locations

Given a program, we first do a standard points-to analysis to obtain a points-to graph $\mathring{G}$: $\langle \mathring{V}, \mathring{E} \rangle$ consisting of a set of vertices $\mathring{V}$ and a set of edges $\mathring{E}$ (e.g., Figure 2). A vertex represents a set of memory addresses, which include program variables $x \in \mathbf{Var}$ and abstract locations $a \in \mathbf{AbsLoc}$ (i.e., $\mathring{V} \supseteq \mathbf{Var} \cup \mathbf{AbsLoc}$). An abstract location a abstracts non–program-variable locations (e.g., from dynamic memory allocation). We do not fix the heap abstraction, such as the level of context sensitivity, but we do assume that we are given the abstract location to which any `new` allocation belongs (via the subscript annotation). A points-to edge from $\mathring{E}$ is either of the form $x \Mapsto a$ or $a_0 \cdot f \Mapsto a_1$. The form $x \Mapsto a$ means a concrete memory address represented by the program variable x may contain a value represented by abstract location a. We write $a_0 \cdot f \Mapsto a_1$ to denote that f is the label for the edge between nodes a_0 and a_1. This edge form means that $a_0 \cdot f$ is a field of an object in the concretization of a_0 that may contain a value represented by abstract location a_1. A **static** field in Java can be modeled as a global program variable.

Our analysis permits formulating a query Q over a finite number of heap locations along with constraints over data fields:

queries	$Q ::= M \wedge P \mid \mathsf{false}$
memories	$M ::= \mathsf{any} \mid x \mapsto \widehat{v} \mid \widehat{v}.f \mapsto \widehat{u} \mid M_1 * M_2$
pure formulæ	$P ::= \mathsf{true} \mid P_1 \wedge P_2 \mid \widehat{v}\ \mathsf{from}\ \mathring{r} \mid \cdots$
points-to regions	$\mathring{r}, \mathring{s} ::= a \mid \mathsf{data} \mid \mathring{r}_1 \Cup \mathring{r}_2$
instances	$\widehat{v}, \widehat{u}$
refutation states	$R ::= Q \mid R_1 \vee R_2 \mid \exists \widehat{v}. R$

We give a heap location via an *exact points-to edge constraint* having one of two forms: $x \mapsto \widehat{v}$ or $\widehat{v}.f \mapsto \widehat{u}$. Recall that in contrast to points-to edges that summarize a *set* of concrete memory cells, an exact points-to constraint expresses a single memory cell. The first form $x \mapsto \widehat{v}$ means a memory address represented by the program variable x contains a value represented by a symbolic variable $\widehat{v}$ (and similarly for the second form for a field). Since we are mostly concerned with memory addresses for concrete object instances, we often refer to symbolic variables as *instances*. The memory any stands for an arbitrary memory.

We introduce one non-standard pure constraint form: the *instance constraint* $\widehat{v}\ \mathsf{from}\ \mathring{r}$ says the symbolic variable $\widehat{v}$ is an instance of a points-to region $\mathring{r}$ (i.e., is in the set of values described by region $\mathring{r}$). A *points-to region* is a set of abstract locations a or the special region data. For uniformity, the region data is used to represent the set of values that are not memory addresses, such as integer values. As we have seen in Section 2.2, instance constraints enable us to use information from the up-front points-to analysis in our witness-refutation analysis. As an example, the informal query $\mathsf{arr}_0 \cdot \mathsf{contents} \mapsto \mathsf{act}_0$ from Section 2 is expressed as follows:

$$\widehat{v}_1 \cdot \mathsf{contents}_{[\widehat{v}_3]} \mapsto \widehat{v}_2 \wedge \widehat{v}_1\ \mathsf{from}\ \{\mathsf{arr}_0\} \wedge \widehat{v}_2\ \mathsf{from}\ \{\mathsf{act}_0\} \wedge \widehat{v}_3\ \mathsf{from}\ \mathsf{data}$$

where $\widehat{v}_3$ stands for the index of the array. This query considers existentially an instance of each abstract location arr_0 and act_0.

When writing down queries, we assume the usual commutativity, associativity, and unit laws from separation logic [41]. Since we are interested in witnessing or refuting a subset of edges corresponding to part of the memory, we interpret any memory M as $M * \mathsf{any}$ (or intuitionistically instead of classically [32, 41]).

3.2 Witness-Refutation Search with Instance Constraints

As described in Section 2, we perform a path-program–by–path-program, backwards symbolic analysis to find a witness for a given query Q. A refutation state R is simply a disjunction of queries, which we often view as a set of candidate witnesses. We include an existential binding of instances $\exists \widehat{v}. R$ to make explicit when we introduce fresh instances, but we implicitly view instances as renamed so that they are all bound at the top-level (i.e., all formulæ are in prenex normal form). Informally, a path program witness is a query Q_{wit} bundled with a sub-program examined so far s_{wit} and a sub-program left to be explored s_{pre}.

Definition 1 (Path Program Witness)**.** A *path program witness* for an input statement-query pair $\langle s, Q \rangle$ is a triple $\langle s_{\text{pre}}, Q_{\text{wit}}, s_{\text{wit}} \rangle$ where (1) $s \equiv s_{\text{pre}} \,;\, s_{\text{post}}$, and (2) s_{wit} is a sub-statement of s_{post} such that (a) if an execution of s_{post} leads to a store σ_{post} satisfying the input query Q, then it must be from a store σ_{wit} satisfying Q_{wit} and (b) executing s_{wit} from σ_{wit} also leads to σ_{post}.

$$\boxed{\vdash \{R\}\, s\, \{Q\} \qquad \vdash \{R'\}\, s\, \{R\}}$$

$$\textsc{WitRefuted}\ \frac{}{\vdash \{\mathsf{false}\}\, s\, \{\mathsf{false}\}} \qquad \textsc{WitCases}\ \frac{\vdash \{R_1'\}\, s\, \{R_1\} \qquad \vdash \{R_2'\}\, s\, \{R_2\}}{\vdash \{R_1' \vee R_2'\}\, s\, \{R_1 \vee R_2\}}$$

$$\textsc{WitSkip}\ \frac{}{\vdash \{\mathsf{any} \wedge \mathsf{true}\}\, s\, \{\mathsf{any} \wedge \mathsf{true}\}} \qquad \textsc{WitFrame}\ \frac{\vdash \{\bigvee_i M_i' \wedge P_i'\}\, s\, \{M \wedge P\} \quad s \text{ must not modify } M_{\text{fr}}}{\vdash \{\bigvee_i (M_{\text{fr}} * M_i') \wedge P_i'\}\, s\, \{(M_{\text{fr}} * M) \wedge P\}}$$

$$\boxed{\vdash \{R\}\, c\, \{Q\}}$$

$$\textsc{WitNew}\ \frac{}{\vdash \{\mathsf{any} \wedge \ \widehat{v}\ \mathsf{from}\ a \cap \mathring{r}\ \wedge P\}\ x := \mathtt{new}_a\, \tau()\ \{x \mapsto \widehat{v} \wedge \widehat{v}\ \mathsf{from}\ \mathring{r} \wedge P\}}$$

$$\textsc{WitAssign}\ \frac{}{\vdash \{y \mapsto \widehat{v} \wedge\ \widehat{v}\ \mathsf{from}\ \mathrm{pt}_{\mathring{G}}(y) \cap \mathring{r}\ \wedge P\}\ x := y\ \{x \mapsto \widehat{v} \wedge \widehat{v}\ \mathsf{from}\ \mathring{r} \wedge P\}}$$

$$\textsc{WitRead}\ \frac{P' = \widehat{u}\ \mathsf{from}\ \mathrm{pt}_{\mathring{G}}(y) \wedge\ \widehat{v}\ \mathsf{from}\ \mathrm{pt}_{\mathring{G}}(y.f) \cap \mathring{r}\ \wedge P}{\vdash \{\exists \widehat{u}.\, y \mapsto \widehat{u} * \widehat{u}.f \mapsto \widehat{v} \wedge P'\}\ x := y.f\ \{x \mapsto \widehat{v} \wedge \widehat{v}\ \mathsf{from}\ \mathring{r} \wedge P\}}$$

$$\textsc{WitWrite}\ \frac{\begin{array}{c} M_i = x \mapsto \widehat{v}_i * y \mapsto \widehat{u}_i * (\mathop{*}_{j \neq i} \widehat{v}_j.f \mapsto \widehat{u}_j \wedge \widehat{v}_j\ \mathsf{from}\ \mathring{r}_j \wedge \widehat{u}_j\ \mathsf{from}\ \mathring{s}_j) \\ Q_i = M_i \wedge\ \widehat{v}_i\ \mathsf{from}\ \mathrm{pt}_{\mathring{G}}(x) \cap \mathring{r}_i\ \wedge\ \widehat{u}_i\ \mathsf{from}\ \mathrm{pt}_{\mathring{G}}(y) \cap \mathring{s}_i\ \wedge (\bigwedge_{j \neq i} \widehat{v}_j \neq \widehat{v}_i) \wedge P \\ Q = \exists \widehat{x}.\, x \mapsto \widehat{x} * (\mathop{*}_i \widehat{v}_i.f \mapsto \widehat{u}_i \wedge \widehat{v}_i\ \mathsf{from}\ \mathring{r}_i \wedge \widehat{u}_i\ \mathsf{from}\ \mathring{s}_i \wedge \widehat{v}_i \neq \widehat{x}) \wedge \widehat{x}\ \mathsf{from}\ \mathrm{pt}_{\mathring{G}}(x) \wedge P \end{array}}{\vdash \{Q \vee \bigvee_i Q_i\}\ x.f := y\ \{(\mathop{*}_i \widehat{v}_i.f \mapsto \widehat{u}_i \wedge \widehat{v}_i\ \mathsf{from}\ \mathring{r}_i \wedge \widehat{u}_i\ \mathsf{from}\ \mathring{s}_i) \wedge P\}}$$

Figure 4. Witness-refutation search is a path-program–by–path-program backwards analysis. Boxed terms emphasize opportunities for refuting paths using instance constraints.

Any intermediate state in our backwards analysis is such a path program witness. Intuitively, the statement s_{wit} captures the path sub-program identified by the backwards analysis that is relevant to producing the input query Q (so far). A refutation occurs when Q_{wit} is false, that is, we have discovered that it is not possible to end up in a state satisfying Q. A "full" witness is when Q_{wit} is any; that is, we can no longer find a refutation. A "partial" witness is a witness where Q_{wit} is a query other than any or false.

We formalize a backwards path program enumeration transforming queries into sub-queries to eventually produce an any or a false refutation state. To describe the analysis, we define the judgment form $\vdash \{R'\}\, s\, \{R\}$ in Figure 4. This judgment form is a standard Hoare triple, but because our analysis is backwards, we read this judgment form from right-to-left. It says, "Given a post-formula R, we find a pre-formula R' such that executing statement s from a state satisfying R' yields a state satisfying R (up to termination)." Conceptually, the post-formula R is a (disjunctive) set of queries, and the pre-formula R' is the set of sub-queries. The path program s_{wit} can be obtained by a simple instrumentation of the rules similar to our prior work [8].

Deriving Refutations. At any point, we can extend the witness-refutation search for some disjunct. Here, we express this step with WitCases, which says a disjunctive refutation state $R_1 \vee R_2$ can be derived by finding a witness for R_1 and R_2. We make this system algorithmic for an implementation by representing cases as a disjunctive set of pending queries $\cdots \vee Q_i \vee \cdots$ that we extend individually. Rule WitRefuted simply says if we have derived false in the post-state for a statement, then we have false in the pre-state as well. To scale beyond the tiniest of programs, we need to be able to refute queries quickly so that the number of queries to consider, that is, the number of symbolic execution paths to explore, remains small. We have three tools for refuting queries: (1) separation (i.e., a query where a single memory cell would need to point to two locations simultaneously), (2) instance constraints (i.e., a query where an instance cannot be in any points-to region), and (3) other pure constraints (i.e., a query with pure constraints that are unsatisfiable, such as, from detecting an infeasible control-flow path). In our inference rules, we assume a refutation state R is always normalized to false if the formula is unsatisfiable. Since we are interested in sound refutations and over-approximate witnesses, for scaling, we can also weaken queries at the cost of potentially losing precision. We revisit this notion in Section 3.3.

Instance constraints are pure constraints that tie the exact points-to constraints in the query to the accumulated information from the up-front points-to analysis and the flow of the symbolic variables as discussed in Section 2.2. They can axiomatized as follows:

$$\widehat{v}\ \mathsf{from}\ \emptyset \iff \mathsf{false} \tag{1}$$

$$\widehat{v}\ \mathsf{from}\ \mathring{r}_1 \wedge \widehat{v}\ \mathsf{from}\ \mathring{r}_2 \iff \widehat{v}\ \mathsf{from}\ \mathring{r}_1 \cap \mathring{r}_2 \tag{2}$$

$$\mathsf{true} \iff \widehat{v}\ \mathsf{from}\ \mathbf{AbsLoc} \cup \{\mathsf{data}\} \tag{3}$$

In particular, we derive a contradiction when we discover an instance that can be drawn from any abstract location (axiom 1). Though our formalism groups instance constraints and other pure constraints together, our implementation keeps them separate for simplicity in checking. Instance constraints are checked using basic set operations and other pure constraints are checked with an off-the-shelf SMT solver, though it should be possible to encode instance constraints into the solver using this axiomatization.

We include a frame rule on M, WitFrame, to simplify our presentation, which together with WitSkip allow us to isolate the parts of the query that a block of code may affect and to ignore irrelevant statements. In other words, any statements that cannot affect the memory state in the query can be skipped. We can thus focus our discussion on an auxiliary judgment form $\vdash \{R\}\, c\, \{Q\}$ that describes how the assignment commands affect a query and its point-to and instance constraints. Informally, it says, "We can witness a query Q after assignment command c by executing c if we can also witness one of the sub-queries R before c." From an algorithmic perspective, we can view $\vdash \{R\}\, c\, \{Q\}$ as generating sub-queries that when combined with a frame may yield additional contradictions. We assume this judgment implicitly has access to the points-to graph $\mathring{G}$: $\langle \mathring{V}, \mathring{E} \rangle$ computed by the up-front analysis. We use the $\mathrm{pt}_{\mathring{G}}^{\mathring{}}(\cdot)$ function to get the points-to set of a program variable via $\mathrm{pt}_{\mathring{G}}^{\mathring{}}(x) \stackrel{\text{def}}{=} \{a \mid (x \mathrel{\dot{\mapsto}} a) \in \mathring{E}\}$ or a field of a program variable via $\mathrm{pt}_{\mathring{G}}^{\mathring{}}(y.f) \stackrel{\text{def}}{=} \{a_j \mid a_i \in \mathrm{pt}_{\mathring{G}}^{\mathring{}}(y) \text{ and } (a_i.\mathsf{f} \mathrel{\dot{\mapsto}} a_j) \in \mathring{E}\}$.

Backwards Transfer Functions and Instance Constraints. Rule WitNew says that the exact points-to constraint $x \mapsto \widehat{v}$ can be witnessed, or *produced*, by the allocation command $x := \mathtt{new}_a\, \tau()$ if instance $\widehat{v}$ may have been created at allocation site a. Following our axiomatization, the instance constraint $\widehat{v}\ \mathsf{from}\ a \cap \mathring{r}$ may immediately reduce to false if $a \notin \mathring{r}$. Or, we can drop it (without loss of precision) because this instance cannot exist before its allocation at this statement. Such a contradiction is precisely the reason for refuting the path at the new arr_1 allocation (program point 16) in our motivating example (Figure 1).

Now, consider rule WitAssign: it says that the exact points-to constraint $x \mapsto \widehat{v}$ can be produced by the assignment command $x := y$ if $y \mapsto \widehat{v}$ can be witnessed before this assignment *and* the instance $\widehat{v}$ can come from a region common to the points-to set of y and the region $\mathring{r}$. If the points-to set of y and the region $\mathring{r}$ are disjoint, we can derive a contradiction because the instance $\widehat{v}$ cannot come from any allocation site. Observe that WitAssign leverages the 'from' constraint and the up-front points-to analysis result to eagerly discover that *no* allocation site satisfies the conditions

required for a witness (rather than observing that a *particular* allocation site does not satisfy the conditions required for a witness, as in WitNew). To get a sense for why these eager refutations are critical for scaling, consider the path refutation due to the binding of `val` to `"hello"` at the `objs.push` call site (program point 8). In our example, this refutation is via WitNew because `"hello"` is a `String` allocation, but we can easily imagine a variant where `objs.push` is called with a program variable y that can conservatively point-to a large set of non-Activity objects. For such a program, WitAssign would allow us to discover a path refutation at the assignment corresponding to the binding rather than requiring us to continue exploration across the potentially exponential number of paths to the allocation sites that flow to y.

The WitRead rule is quite similar to WitAssign except that we existentially quantify over the instance to which y points (i.e., $\widehat{u}$). We set up an initial points-to region constraint for the fresh symbolic variable $\widehat{u}$ based on the points-to set of y and narrow the points-to region of $\widehat{v}$ using the points-to set of $y.f$. As in WitAssign, we can derive a contradiction based on this narrowing if the region $\mathring{r}$ and the points-to set of $y.f$ are disjoint. Here, we have taken some liberty with notation placing, for example, 'from' constraints under the iterated separating conjunctions; we recall that $*$ collapses to $\wedge$ for pure constraints.

In the WitWrite rule, the post-formula consists of two cases for each edge $\widehat{v}_i.f \mapsto \widehat{u}_i$ in the pre-formula: (1) the field write $x.f := y$ did not produce the edge $\widehat{v}_i.f \mapsto \widehat{u}_i$ (the first disjunct Q), or (2) the field write did produce the edge (the second set of disjuncts over all Q_i). If the write $x.f := y$ did produce the points-to edge $\widehat{v}_i.f \mapsto \widehat{u}_i$, then the points-to regions of $\widehat{v}_i$ and $\widehat{u}_i$ are restricted based on the points-to sets of x and y, respectively. The "not produced" case represents the possibility that this write updates an instance other than a $\widehat{v}_i$ (as reflected by the $x \mapsto \widehat{x}$ and $\widehat{v}_i \neq \widehat{x}$ conditions).

While WitWrite can theoretically generate a huge case split, we have observed that the combination of instance constraints and separation typically allow us to find refutations quickly in practice (see Section 4). In particular, the "not written" case can often be immediately refuted by separation. For example, we end up with a contradictory query where a local variable `x` has to point to two different instances simultaneously (i.e., $\mathtt{x} \mapsto \widehat{v} * \mathtt{x} \mapsto \widehat{u} \wedge \widehat{v} \neq \widehat{u}$).

Guards and Control Flow. Except for loops (see Section 3.3), the remaining rules mostly relate to control flow and are quite standard (shown inset). To discover a contradiction on pure constraints, we state that the guard condition of an `assume` must hold in the pre-query. We write $e[M]$ for interpreting the program expression e in the memory state M.

The WitChoice rule analyzes each branch independently. Our implementation avoids path explosion due to irrelevant path sensitivity by adding pure constraints from `if`-guards only when the queries on each side of the branch are different (as in previous work [18, 39]), though our rules do not express this.

WitAssume
$$\frac{}{\vdash \{M \wedge P \wedge e[M]\}\ \mathtt{assume}\, e\ \{M \wedge P\}}$$

WitSeq
$$\frac{\vdash \{R''\}\ s_1\ \{R'\} \quad \vdash \{R'\}\ s_2\ \{R\}}{\vdash \{R''\}\ s_1\,;\,s_2\ \{R\}}$$

WitChoice
$$\frac{\vdash \{R_1\}\ s_1\ \{R\} \quad \vdash \{R_2\}\ s_2\ \{R\}}{\vdash \{R_1 \vee R_2\}\ s_1 \,[]\, s_2\ \{R\}}$$

3.3 Loop Invariant Inference and Query Simplification

In this section, we finish our description of witness-refutation search by discussing our loop invariant inference scheme.

Roughly speaking, we infer loop invariants by repeatedly performing backwards symbolic execution over the loop body until we reach a fixed point over the domain of points-to constraints. To ensure termination, we drop all pure constraints affected by the loop body and fix a static bound on the number of instances of each abstract location to materialize. In our experiments, a static bound of one has been sufficient for precise results. The WitLoop rule (shown inset) simply states that if the loop body has no effect on the query, the loop has no effect on it.

WitLoop
$$\frac{\vdash \{R\}\ s\ \{R\}}{\vdash \{R\}\ \mathtt{loop}s\ \{R\}}$$

WitAbstraction
$$\frac{R_2' \models R_1' \quad \vdash \{R_2'\}\ s\ \{R_2\} \quad R_1 \models R_2}{\vdash \{R_1'\}\ s\ \{R_1\}}$$

By itself, this rule only handles the degenerate case where a loop can be treated as `skip` with respect to the query. For this case, the disjunctive set of queries R is trivially a loop invariant. For more interesting cases, we use WitCases to consider each query in the refutation state individually so that we can infer an over-approximate loop invariant for each one. Thus, we infer a loop invariant on-the-fly for each *path program* rather than joining all queries at the loop exit and then inferring an invariant for all backwards paths into the loop (similar to [35] but with heap constraints).

To preserve refutation soundness, we want to ensure that a contradiction false is only derived when there does not exist a concrete path witnessing the given query and thus must over-approximate loops. A sound, backwards over-approximation can be obtained by weakening the post-loop query Q_i. Since an individual query is purely conjunctive, we can weaken it quite easily by "dropping" constraints (i.e., removing conjuncts). Intuitively, dropping constraints is refutation-sound because it can only make it more difficult to derive a contradiction. This over-approximation is captured by the WitAbstraction rule. The rule says that at any program point, we can drop constraints, and doing so preserves refutation soundness (Theorem 1).

We write $R_1 \models R_2$ for the semantic entailment and correspondingly rely on a sound decision procedure in our implementation (used in WitAbstraction). Entailment between a finite separating conjunction exact points-to constraints can be resolved in a standard way by subtraction [5]. Without inductive predicates, the procedure is a straightforward matching. Entailment between the 'from' instance constraints can be defined as follows:

$$(\widehat{v}_1\ \mathsf{from}\ \mathring{r}_1) \models (\widehat{v}_2\ \mathsf{from}\ \mathring{r}_2) \quad \text{iff} \quad \widehat{v}_1 = \widehat{v}_2 \text{ and } \mathring{r}_1 \subseteq \mathring{r}_2 \qquad (\S)$$

As previously mentioned, 'from' constraints are represented as sets associated with a symbolic variable and solved with ordinary set operations. We discharge other pure constraints using an off-the-shelf SMT solver, so precision of reasoning about those constraints is with respect to the capabilities of the solver.

With these tools, our loop invariant inference is a rather straightforward fixed-point computation. For a loop statement `loop`s and a post-loop query Q, we iteratively apply the backwards predicate transformer for the loop body s to saturate a set of sub-queries at the loop head. Let R_0 be some refutation state such that $\vdash \{R_0\}\ s\ \{Q\}$, and let $R_{i+1} = R_i \vee R'$ where $\vdash \{R'\}\ s\ \{R_i\}$. We ensure that the chain of $R_0 \models R_1 \models \cdots$ converges by bounding the number of instances or materializations from the abstract locations. Since there are a finite number of abstract locations, the number of points-to constraints in any particular query is bounded by the number of edges in the points-to graph (i.e., $|\mathring{E}|$). For the base domain of pure constraints, widening [14] can be used to ensure convergence. Our implementation uses a trivial widening that drops pure constraints that may be modified by the loop.

Query Simplification with Disaliasing. The WitAbstraction rule captures backwards over-approximation by saying that at any point, we can weaken a refutation state without losing refutation soundness. Conceptually, we can weaken by replacing any symbolic join $\vee$ with an over-approximate join $\sqcup$. We perform one such join by replacing the refutation state $Q_1 \vee Q_2$ with Q_2 if $Q_1 \models Q_2$. Note that this join does not lose precision. Intuitively, for a refutation state R, we are interested in witnessing *any* query in R or refuting *all* queries in R. Here, a refutation of query Q_2 implies a refutation of Q_1, so we only need to consider Q_2.

To enable this join to apply often, we enforce a normal form for our queries by dropping certain kinds of constraints. As formalized in Figure 4, the backwards transfer functions for assignment commands c are as precise as possible, including the generation of disequality constraints in WITWRITE. These disequality constraints are needed locally to check for refutations due to separation, as detailed in Section 3.2. However, if this check passes, we drop them before proceeding and instead keep only the disaliasing information implied by separation and the instance from constraints. While this weakening could lose precision (e.g., if the backwards analysis would later encounter an `if`-guard for the aliasing condition), we hypothesize that this situation is rare and that the most useful disaliasing information is captured by separation and instance constraints.

In our implementation, we are much closer to a path-by-path analysis than our formalization would indicate. Refutation states are represented as a worklist of pending (non-disjunctive) queries to explore. To apply the simplification described above, we must keep a history of queries seen at a given program point: if we have previously seen a weaker query at this program point, then we can drop the current query. We keep a query history only at procedure boundaries and loop heads. This simplification has been especially critical for procedures.

Soundness. We define a concrete store σ be a finite mapping from variables or address-field pairs to concrete values (i.e., $\sigma : \mathbf{Var} \uplus (\mathbf{Addr} \times \mathbf{Field}) \rightharpoonup_{\text{fin}} \mathbf{Val}$) and give a standard big-step operational semantics to our basic imperative language The judgment form $\sigma \vdash s \downarrow \sigma'$ says, "In store σ, statement s evaluates to store σ'." Furthermore, we write $\sigma \models R$ to say that the store σ is in the concretization of the refutation state R. The definition of $\sigma \models R$ is as would be expected in comparison to separation logic. We need to utilize two other concrete semantic domains: a valuation η that maps instances to values (i.e., $\eta : \mathbf{Instance} \rightarrow \mathbf{Val}$) and a *regionalization* ρ that maps abstract locations to sets of concrete addresses (i.e., $\rho : \mathbf{AbsLoc} \rightarrow \wp(\mathbf{Addr})$). The regionalization gives meaning to the 'from' instance constraint. With these definitions, we precisely state the soundness theorem.

Theorem 1 (Refutation Soundness)**.** *If* $\vdash \{R_{\text{pre}}\}\ s\ \{R_{\text{post}}\}$ *and* $\sigma_{\text{pre}} \vdash s \downarrow \sigma_{\text{post}}$ *such that* $\sigma_{\text{post}} \models R_{\text{post}}$*, then* $\sigma_{\text{pre}} \models R_{\text{pre}}$*. As a corollary, refutations (i.e., when* R_{pre} *is* false*) are sound.*

Interestingly, the standard consequence rule from Hoare logic states the opposite in comparison to WITABSTRACTION by permitting the strengthening of queries. Doing so would instead preserve witness precision; that is, any path program witness exhibits some witness path (up to termination).

4. Case Study: Activity Leaks in Android

We evaluated our witness-refutation analysis by using it to find Activity leaks, a common class of memory leaks in open-source Android applications. We explain this client in more detail below. Our experiments were designed to test two hypotheses. The first and most important concerns the *precision* of our approach: we hypothesized that witness-refutation analysis reports many fewer false alarms than a flow-insensitive points-to analysis. We tried using a flow-insensitive analysis to find leaks, but found that the number of alarms reported was too large to examine manually. To be useful, our technique needs to prune this number enough for a user to effectively triage the results and identify real leaks. Our second hypothesis concerns the *utility* of our techniques: we posited that (1) our mixed symbolic-explicit is an improvement over both a fully explicit and a fully symbolic representation, (2) our query simplification significantly speeds up analysis, and (3) our on-the-fly loop invariant inference is needed to preserve precision in the presence of loops.

Client. Activity leaks occur when a pointer to an Activity object can outlive the Activity. The operating system frequently destroys Activity's when configuration changes occur (e.g., rotating the phone). Once an Activity is destroyed, it can never be displayed to the user again and thus represents unused memory that should be reclaimed by the garbage collector. However, if an application developer maintains a pointer to an Activity after it is destroyed, the garbage collector will be unable to reclaim it. In our experiments, we check if *any* Activity instance is ever reachable from a static field, a flow-insensitive property. Though a developer could safely keep a reference to an Activity object via a static field that is cleared each time the Activity is destroyed, this is recognized as bad practice.

Activity leaking is a serious problem. It is well-documented that keeping persistent references to Activity's is bad practice; we refer the reader to an article[2] in the Android Developers Blog as evidence. The true problem is that it is quite easy for developers to inadvertently keep persistent references to an Activity. Sub-components of Activity's (such as Adapter's, Cursor's, and View's) typically keep pointers to their parent Activity, meaning that any persistent reference to an element in the Activity's hierarchy can potentially create a leak.

Precision of Our Techniques: Threshing Alarms. We implemented our witness-refutation analysis in the THRESHER tool, which is publicly available.[3] Additional details on our implementation are included at the end of this section. All of our experiments were performed on a machine running Ubuntu 12.04.2 with a 2.93 GHz Intel Xeon processor and 32GB of memory. Though our analysis is quite amenable to parallelization in theory, our current implementation is purely sequential.

To evaluate the precision of our approach, we computed a flow-insensitive points-to graph for each application and the Android library (version 2.3.3) using WALA's 0-1-Container-CFA pointer analysis (a variation of Andersen's analysis with unlimited context sensitivity for container classes). For each heap path from a static field f to an Activity instance A in the points-to graph, we asked THRESHER to witness or refute each edge in the path from source to sink. If we refuted an edge in the heap path, we searched for a new path. We repeated this until THRESHER either witnessed each edge in the heap path (i.e., confirmed the flow-insensitive alarm) or refuted enough edges to prove that no heap path from f to A can exist (i.e., filtered out the leak report). We allowed an exploration budget of 10,000 path programs for each edge; if the tool exceeded the budget, we declared a timeout for that edge and considered it to be not refuted. On paths with call stacks of depth greater than three, we soundly skipped callees by dropping constraints that executing the call might produce (according to a mod/ref analysis computed alongside the points-to analysis). We limited the size of the path constraint set to at most two. Allowing larger path constraint sets slowed down the symbolic executor without increasing precision. We ran in two configurations: one with the Android library as-is (**Ann?**=N), and one where we added a single annotation to the HashMap class to indicate that the shared `EMPTY_TABLE` field can never point to anything (**Ann?**=Y). We did this because we observed that the use of the null object pattern in the HashMap class was a major source of imprecision for the flow-insensitive analysis (cf. Figure 1), but we wanted to make sure that it was not the only one our tool was able to handle.

[2] `http://developer.android.com/resources/articles/avoiding-memory-leaks.html`

[3] `https://github.com/cuplv/thresher`

	Benchmark *Size*			*Filtering* Effectiveness						Computational *Effort*			
Benchmark	**SLOC**	**CGB**	**Ann?**	**Alrms**	**RefA(%)**	**TruA(%)**	**FalA(%)**	**Flds**	**RefFlds**	**RefEdg**	**WitEdg**	**TO**	**T (s)**
PulsePoint♠	no src	198K	N	24	16 (67)	8 (33)	0 (0)	3	2	47	40	1	750
			Y	16	8 (50)	8 (50)	0 (0)	2	1	40	31	0	95
StandupTimer♣	2K	240K	N	25	15 (60)	0 (0)	10 (40)	5	3	18	26	0	1199
			Y	25	15 (60)	0 (0)	10 (40)	5	3	18	26	0	1068
DroidLife♠	3K	132K	N	3	0 (0)	3 (100)	0 (0)	1	0	0	4	0	1
			Y	3	0 (0)	3 (100)	0 (0)	1	0	0	4	0	1
OpenSudoku	6K	229K	N	7	1 (14)	0 (0)	6 (86)	1	0	2	21	1	1596
			Y	0	0 (0)	0 (0)	0 (0)	0	0	0	0	0	0
SMSPopUp♠	7K	232K	N	5	1 (20)	4 (80)	0 (0)	1	0	10	24	0	49
			Y	5	1 (20)	4 (80)	0 (0)	1	0	10	24	0	46
aMetro♠	20K	326K	N	144	18 (12)	36 (25)	90 (63)	8	1	62	66	3	4226
			Y	54	18 (33)	36 (67)	0 (0)	3	1	55	24	0	18
K9Mail♠	40K	394K	N	364	78 (21)	64 (18)	222 (61)	14	3	141	106	1	1130
			Y	208	130 (63)	64 (49)	14 (7)	8	5	124	80	0	374
Total	78K	1751K	N	572	129 (22)	115 (20)	332 (58)	33	9	280	287	6	8991
			Y	311	172 (55)	115 (37)	24 (8)	20	10	247	189	0	1602

Table 1. This table characterizes the size of our benchmarks, highlights our success in distinguishing false alarms from real leaks, and quantifies the effort required to find refutations. ♠'s indicate a benchmark in which we found an observable leak, and ♣ indicates a latent leak. The *Size* column grouping gives the number of source lines of code **SLOC** and the number of bytecodes in the call graph **CGB** for each app as well as the annotation configuration **Ann?**=Y/N. The *Filtering* column grouping characterizes the effectiveness of our approach for filtering false alarms. The first four columns list the number of (static field, Activity) alarm pairs reported by the points-to analysis **Alarms**, number of alarms refuted by our approach **RefA**, number of true alarms **TruA**, and the number of false alarms **FalA**. The final two columns of this group give the number of leaky fields reported by the points-to analysis **Fields** and the number of these fields **RefFlds** that we can refute (i.e., prove that the field in question cannot point to *any* Activity). The *Effort* columns describe the amount of work required by our filtering approach. We list the number of edges refuted **RefEdg**, edges witnessed **WitEdg**, edge timeouts **TO**, and the time **T (s)** taken by the symbolic execution phase in seconds. This number does not include points-to analysis time, which ranged from 8–46 seconds on all benchmarks.

Table 1 shows the results of this experiment. We first comment on the most interesting part of the experiment: the *filtering effectiveness* of our analysis. As we hoped, our analysis is able to refute many of the false alarms produced by the flow-insensitive points-to analysis. Overall, we refute 129/457 = 28% of these false alarms in the un-annotated configuration and 172/196 = 87% of these false alarms in the annotated configuration. Contrary to our expectations, we found many *more* refutations in the **Ann?**=Y configuration, confirming that our technique can indeed remedy imprecision other than the pollution caused by HashMaps.

Unfortunately, this also means that our analysis is not always able to remedy the imprecision caused by HashMaps. The major problem is that the **Ann?**=N configuration fails to refute many of the HashMap-related edges due to timeouts. In fact, most of the false alarms that are not common to both configurations stem from (soundly) not considering timed-out edges to be refuted. We observed that a timeout commonly corresponds to a refutation that the analysis was unable to find within the path program budget. This is not surprising; finding a witness for an edge only requires finding a single path program that produces the edge (which we can usually do quickly), but to find a refutation we must refute *all* path programs that might produce an edge (which is slow and sometimes times out, potentially causing precision loss).

For example, the single timeout in the **Ann?**=N run of K9Mail occurs on a HashMap-related edge that is refutable, but quite challenging to refute. As it turns out, refuting this edge is extremely important for precision—upon further investigation, we discovered that the analysis would have reduced the number of false alarms reported by over 100 if it had been able to refute it! In the **Ann?**=Y configuration, this edge disappears from the flow-insensitive points-to graph. We can see that this increases the number of alarms we are able refute from 78 to 130 even though the number of alarms reported by the flow-insensitive points-to analysis falls from 364 to 208.

```
public class EmailAddressAdapter extends ResourceCursorAdapter {
  private static EmailAddressAdapter sInstance;
  public static EmailAddressAdapter getInstance(Context context) {
    if (sInstance == null)
      sInstance = new_adr0 EmailAddressAdapter(context);
    return sInstance;
  }
  private EmailAddressAdapter(Context context) { super(context); }
}
```

Figure 5. A confirmed Activity leak discovered in K9Mail.

We now comment on the *computational effort* required to refute/witness edges. We first observe that the number of edges refuted is almost always greater then the number of alarms refuted, indicating that it is frequently necessary to refute several edges in order to refute a single (source, sink) alarm pair. For example, in the un-annotated run of aMetro, we refute 62 edges in order to refute 18 alarms. This demonstrates that the flow-insensitive points-to analysis is imprecise enough to find many different ways to produce the same false alarm.

We note that the running times are quite reasonable for an analysis at this level of precision, especially in the annotated configuration. No benchmark other than aMetro takes more than a half hour. Our tool would be fast enough to be used in a heavyweight cloud service or as part of an overnight build process.

Real* Activity *Leaks. As hypothesized, our tool's precision enabled us to ignore most false alarms and focus on likely leaks. We found genuine leaks in PulsePoint, DroidLife, SMSPopUp, aMetro, and K9Mail. Many of the leaks we found would only manifest under specialized and complex circumstances, but a few of the nastiest leaks we found would almost always manifest and are due to the same simple problem: an inappropriate use of the singleton pattern. We briefly explain one such leak from the K9Mail app.

In the code in Figure 5, the developer uses the singleton pattern to ensure that only one instance of EmailAddressAdapter is ever created. The leak arises when `getInstance()` is called with an Activity instance passed as the `context` parameter (which happens in several places in K9Mail). The Activity instance is passed backwards through the constructors of two superclasses via the `context` parameter until it is finally stored in the `mContext` instance field of the CursorAdapter superclass. For *every* Activity act_0 that calls `getInstance()`, the flow-insensitive points-to analysis reports a heap path $\mathsf{EmailAddressAdapter{\cdot}sInstance} \Mapsto \mathsf{adr}_0, \mathsf{adr}_0{\cdot}\mathsf{mContext} \Mapsto \mathsf{act}_0$. When the Activity instance is destroyed, the garbage collector will never be able to reclaim it because none of the pointers involved in the leak are ever cleared.

We found this leak in a version of K9Mail that was downloaded in September 2011 (all versions of the benchmarks we used are available in project's GitHub repository). We looked at the current version and noticed that the EmailAddressAdapter class had been refactored to remove the singleton pattern. We found the commit that performed this refactoring and asked the developers of K9Mail if the purpose of this commit was to address a leak issue; they confirmed that it was.[4]

We also discovered a very simple latent leak in StandupTimer that was also due to a bad use of the singleton pattern. We noticed that several of the path programs THRESHER produced for a field in this app would be a full witness for a leak if a single boolean flag `cacheDAOInstances` were enabled. Our tool correctly recognizes that this flag cannot ever be set and refutes the alarm report, but a modification to the program that enabled this flag would result in a leak. The path program witnesses our tool produces are always helpful in triaging reported leak alarms, but in this case even the *refuted* path program witness provided useful information that allowed us to identify an almost-leak. With a less constructive refutation technique, we might have missed this detail.

Utility of Our Techniques. To test our second set of hypotheses, we ran THRESHER on the benchmarks from Table 1 without using each of three key features of our analysis: mixed symbolic-explicit query representation, query simplification, and loop invariant inference. We hypothesized that: (1) using an alternative query representation would negatively affect scalability and/or performance, (2) not simplifying queries would negatively affect scalability and/or performance, and (3) the absence of loop invariant inference would negatively affect precision.

To test hypothesis (1), we implemented a fully symbolic query representation. In a fully symbolic representation, we do not track the set of allocation sites that a variable might belong to. We have up-front points-to information, but use it only to confirm that two symbolic variables are *not* equal (i.e., to prevent aliasing case splits in the style of [39]) and to confirm that a symbolic variable was allocated at a given site (as in WITNEW). This precludes both pruning paths based on the boxed 'from' constraints in Figure 4 and performing the entailment check between symbolic variables defined in Equation § in Section 3.3.

Using this fully symbolic representation, our analysis ran slower and timed out more often, but did not refute any fewer alarms than the run with the mixed representation. We observed several cases where a timeout caused the fully symbolic representation to miss refuting an edge that the mixed representation was able to refute, but in each case the edge turned out not to be important for precision (that is, it was one of many edges that needed to be refuted in order to refute an alarm, but both representations failed to refute all of these edges).

Benchmark	Ann?	T (slowdown)	TO (Δ)
PulsePoint	N	1237 (1.6X)	7 (+6)
	Y	220 (1.9X)	3 (+3)
StandupTimer	N	4946 (4.1X)	4 (+4)
	Y	4104 (3.8X)	4 (+4)
OpenSudoku	N	2984 (1.9X)	4 (+3)
	Y	-	-
SMSPopUp	N	95 (1.9X)	0 (+0)
	Y	76 (1.7X)	0 (+0)
aMetro	N	6863 (1.6X)	5 (+2)
	Y	18 (1X)	0 (+0)
K9Mail	N	990 (0.9X)	2 (+1)
	Y	454 (1.2X)	0 (+0)

Table 2. Performance of the fully symbolic representation as compared to the mixed symbolic-explicit representation.

The results of this experiment are shown in Table 2. We omit the results for DroidLife since they were unaffected by the choice of representation. For every other benchmark, we give the time taken with a fully symbolic representation, the number of times slower than the mixed representation this was (**T (slowdown)**), the number of edges that timed out, and how many timeouts were added over the mixed representation (**TO (Δ)**).

We can see that in both the annotated and un-annotated configurations, most benchmarks run at least 1.6X slower and time out on at least one more edge than they did with the mixed representation. The anomalous behavior of K9Mail in the un-annotated configuration occurs because the mixed representation is able to refute an edge that the fully symbolic representation times out on. Ultimately, this leads the mixed representation to make more progress towards (but ultimately fail in) refuting a particular alarm. The fully symbolic representation declares this particular alarm witnessed after the edge in question times out, which allows it to skip this effort and finish faster. Thus, hypothesis (1) seems to hold: using a fully symbolic representation negatively affected both performance and scalability as predicted, but choosing a fully symbolic representation did not ultimately affect the precision of the analysis in terms of alarms filtered.

To test hypothesis (2), we re-ran THRESHER on our benchmarks using the annotated Android library without performing any query simplification at all. This significantly hurt the performance of THRESHER on PulsePoint (102.4X slower), K9Mail (3.2X slower), and SMSPopUp (4.3X slower), but did not change the number of alarms refuted or witnessed for these benchmarks. On StandupTimer, not performing simplification caused the tool to run out of memory before completing the analysis, thus affecting both precision and performance. The performance of the tool on other apps was not significantly affected. Thus, hypothesis (2) seems to hold for the benchmarks that require significant computational effort.

Finally, to test hypothesis (3), we implemented a simple loop invariant inference that simply drops all possibly-affected constraints at any loop. With only this simple inference, the analysis was unable to refute some critical HashMap-related edges (using the unannotated library). This meant that the analysis could never distinguish the contents of different HashMap objects. This imprecision

[4] `https://groups.google.com/forum/?fromgroups=#!topic/k-9-mail/JhoXL2c4UfU`

prevented the analysis from refuting leak reports involving multiple HashMap's even on small, hand-written test cases. Our full loop invariant inference (Section 3.3) handled the hand-written cases precisely, but due to unrelated analysis limitations, it did not achieve any fewer overall refutations on our real benchmarks. Nevertheless, our testing confirmed hypothesis (3): our loop invariant inference was clearly necessary to properly handle Android HashMap's and similar data structures.

Implementation. THRESHER is built on top of the on the WALA program analysis framework for Java and uses the Z3 [19] SMT solver with JVM support via ScalaZ3 [34] to determine when path constraints are unsatisfiable. Like most static analysis tools that handle real-world programs, our tool has a few known sources of unsoundness. We do not reason about reflection or handle concurrency. We have source code for most (but not all) non-native Java library methods. In particular, the Android library custom implementations of core Java library classes (including collections) that we analyze. To focus our reasoning on Android library and application code, we exclude classes from Apache libraries, `java/nio/Charset`, and `java/util/concurrent` from the call graph. Though we track control flow due to thrown exceptions, we do not handle the **catch**`()` construct; instead, we assume that thrown exceptions are never caught.

Android apps are event-driven and (in general) Android event handlers can be called any number of times and in (almost) any order. We use a top-level harness that invokes every event handler defined for an application. Our harness allows event handlers to be invoked in any order, but insists that each handler is called only once in order to prevent termination issues. In our experiments, we did not observe any unsound refutations due to these limitations.

We do not do any modeling for special Android components such as Intent's and BroadcastReceiver's. Since most special components are used for communication between applications that run in separate memory spaces, we would not expect THRESHER to miss any memory leaks due to this modeling issue.

5. Related Work

Dillig et al. present precise heap analyses for programs manipulating arrays and containers [21, 22], with path and context sensitivity [20]. Our analysis introduces path and context sensitivity via on-demand refinement, in contrast to their exhaustive, summary-based approach. Our symbolic variables are similar to their index variables [21, 22] in that both symbolically represent concrete locations and enable lazy case splits. Unlike index variables, our symbolic variables do not distinguish specific array indices or loop iterations, since this was not required for our memory leak client. Also, our analysis does not require container specifications [22]; instead, we analyze container implementations directly. Hackett and Aiken [29] present a points-to analysis with intra-procedural path sensitivity, which is insufficient for our needs.

Several previous systems focused on performing effective backward symbolic analysis. The pioneering ESC/Java system [27] performed intra-procedural backward analysis, generating a polynomially-sized verification condition and checking its validity with a theorem prover. Snugglebug [11] performed inter-procedural backward symbolic analysis, employing directed call graph construction and custom simplifiers to improve scalability. Cousot et al. [17] present backward symbolic analysis as one of a suite of techniques for transforming intermittent assertions in a method into executable pre-condition checks. PSE [39] used backward symbolic analysis to help explain program failures, but for greater scalability, it did not represent full path conditions. Our work is distinguished from these previous systems by the integration of points-to analysis information, which enables key optimizations like mixed symbolic-explicit states and abstraction for loop handling.

Our analysis can be seen as refining the initial flow-insensitive abstraction of the points-to analysis based on a "counterexample" reachability query deemed feasible by that analysis. However, instead of gradually refining this abstraction as in, for example, counterexample-based abstraction refinement (CEGAR) [13] and related techniques [16] , our technique immediately employs concrete reasoning about the program, and then re-introduces abstraction as needed (e.g., to handle loops). In general, the above predicate-abstraction-based approaches have not been shown to work well for proving properties of object-oriented programs, which present additional challenges due to intensive heap usage, frequent virtual dispatch, etc. Architecturally, our system is more similar to recent staged analyses for typestate verification [25, 26], but our system employs greater path sensitivity and more deeply integrates points-to facts from the initial analysis stage. A path program [7] was originally defined in the context of improving CEGAR by pruning multiple counterexample traces through a loop at once. SMPP [30] performs SMT-based verification by exhaustively enumerating path programs in a forward-chaining manner (in contrast to our goal-directed search). The recent DASH system [4] refines its abstraction based on information from dynamic runs and employs dynamic information to reduce explosion due to aliasing.

Our witness-refutation search uses the "bounded" fragment of separation logic [41] and thus has a peripheral connection to recent separation-logic–based shape analyzers [6, 12]. In contrast to such analyzers, we do not use inductive summaries and instead use materializations from a static points-to analysis abstraction. Shape analysis using bi-abductive inference [10] enables a compositional analysis by deriving pre- and post-conditions for methods in a bottom-up manner and making a best effort to reach top-level entry points. The derivation of heap pre-conditions is somewhat similar to our witness-refutation search over points-to constraints, but our backwards analysis is applied on demand from a flow-insensitive query and is refined by incorporating information from an up-front, whole program points-to analysis. Recent work [24] has applied bi-abduction to detect real Java memory leaks in the sense of an object that is allocated but never used again. In contrast, our client is a flow-insensitive heap reachability property that over-approximates a leak that is not explicit in the code, but is realized in the Android run-time.

Similar to our path program witnesses, other techniques have aimed to either produce a concrete path witness for some program error or help the user to discover one. Bourdoncle [9] presents a system for "abstract debugging" of program assertions, in which the compiler aims to discover inputs leading to violations statically. Rival [42] presents a system based on combined forward and backward analysis for elucidating and validating error reports from the Astrée system [15]. Work by Ball et al. [3] observes that for showing the *existence* of program errors (as opposed to verifying their absence), a non-standard notion of abstraction suffices in which one only requires the existence of a concrete state satisfying any particular property of the corresponding abstract state (as opposed to all corresponding concrete states satisfying the property). We observe an analogous difference between refutation and witness discovery in Section 3. Similar notions underlie the "proof obligation queries" and "failure witness queries" in recent work on error diagnosis [23].

Previous points-to analyses have included refinement to improve precision. Guyer and Lin's client-driven pointer analysis [28] introduced context and flow sensitivity at possibly-polluting program points based on client needs. Sridharan and Bodik [44] presented an approach for adding field and context sensitivity to a Java points-to analysis via refinement. Recently, Liang et al. [36–38] have shown that highly-targeted refinements of a heap abstraction

can yield sufficient precision for certain clients. Unlike our work, none of the aforementioned techniques can introduce path sensitivity via refinement. A recent study on Andersen's analysis [8] used dependency rules akin to a fully-explicit analog of our mixed symbolic-explicit transfer functions in a flow-insensitive context.

6. Conclusion

We have presented THRESHER, a precise static analysis for reasoning about heap reachability with flow-, context-, and path-sensitivity and location materialization. THRESHER introduces such precision in an on-demand manner after running a flow-insensitive points-to analysis. By integrating flow-insensitive points-to facts directly into a mixed symbolic-explicit representation of the program state and computing sufficiently strong loop invariants automatically, our techniques scale well while maintaining good precision. In our evaluation, we applied THRESHER to the problem of detecting an important class of Android memory leaks and discovered real leaks while significantly improving precision over points-to analysis alone.

Acknowledgments. We thank Sriram Sankaranarayanan and the CUPLV group for insightful discussions, as well as the anonymous reviewers for their helpful comments. This research was supported in part by NSF under grant CCF-1055066.

References

[1] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, DIKU, 1994.

[2] T. Ball and S. K. Rajamani. Automatically validating temporal safety properties of interfaces. In *SPIN*, 2001.

[3] T. Ball, O. Kupferman, and G. Yorsh. Abstraction for falsification. In *CAV*, 2005.

[4] N. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons. Proofs from tests. In *ISSTA*, 2008.

[5] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.

[6] J. Berdine, C. Calcagno, B. Cook, D. Distefano, P. W. O'Hearn, T. Wies, and H. Yang. Shape analysis for composite data structures. In *CAV*, 2007.

[7] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path invariants. In *PLDI*, 2007.

[8] S. Blackshear, B.-Y. E. Chang, S. Sankaranarayanan, and M. Sridharan. The flow-insensitive precision of andersen's analysis in practice. In *SAS*, 2011.

[9] F. Bourdoncle. Abstract debugging of higher-order imperative languages. In *PLDI*, 1993.

[10] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.

[11] S. Chandra, S. J. Fink, and M. Sridharan. Snugglebug: a powerful approach to weakest preconditions. In *PLDI*, 2009.

[12] B.-Y. E. Chang and X. Rival. Relational inductive shape analysis. In *POPL*, 2008.

[13] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5), 2003.

[14] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[15] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The ASTREÉ analyzer. In *ESOP*, 2005.

[16] P. Cousot, P. Ganty, and J.-F. Raskin. Fixpoint-guided abstraction refinements. In *SAS*, 2007.

[17] P. Cousot, R. Cousot, and F. Logozzo. Precondition inference from intermittent assertions and application to contracts on collections. In *VMCAI*, 2011.

[18] M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI*, 2002.

[19] L. M. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.

[20] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI*, 2008.

[21] I. Dillig, T. Dillig, and A. Aiken. Fluid updates: Beyond strong vs. weak updates. In *ESOP*, 2010.

[22] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *POPL*, 2011.

[23] I. Dillig, T. Dillig, and A. Aiken. Automated error diagnosis using abductive inference. In *PLDI*, 2012.

[24] D. Distefano and I. Filipović. Memory leaks detection in Java by bi-abductive inference. In *FASE*, 2010.

[25] N. Dor, S. Adams, M. Das, and Z. Yang. Software validation via scalable path-sensitive value flow analysis. In *ISSTA*, 2004.

[26] S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Trans. Softw. Eng. Methodol.*, 17(2), 2008.

[27] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *PLDI*, 2002.

[28] S. Z. Guyer and C. Lin. Error checking with client-driven pointer analysis. *Sci. Comput. Program.*, 58(1-2), 2005.

[29] B. Hackett and A. Aiken. How is aliasing used in systems software? In *FSE*, 2006.

[30] W. R. Harris, S. Sankaranarayanan, F. Ivancic, and A. Gupta. Program analysis via satisfiability modulo path programs. In *POPL*, 2010.

[31] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *POPL*, 2002.

[32] S. S. Ishtiaq and P. W. O'Hearn. BI as an assertion language for mutable data structures. In *POPL*, 2001.

[33] S. Khurshid, C. S. Pasareanu, and W. Visser. Generalized symbolic execution for model checking and testing. In *TACAS*, 2003.

[34] A. S. Kksal, P. Suter, and V. Kuncak. Scala to the Power of Z3: Integrating SMT and Programming. In *CADE*, 2011.

[35] K. R. M. Leino and F. Logozzo. Loop invariants on demand. In *APLAS*, 2005.

[36] P. Liang and M. Naik. Scaling abstraction refinement via pruning. In *PLDI*, 2011.

[37] P. Liang, O. Tripp, M. Naik, and M. Sagiv. A dynamic evaluation of the precision of static heap abstractions. In *OOPSLA*, 2010.

[38] P. Liang, O. Tripp, and M. Naik. Learning minimal abstractions. In *POPL*, 2011.

[39] R. Manevich, M. Sridharan, S. Adams, M. Das, and Z. Yang. PSE: explaining program failures via postmortem static analysis. In *FSE*, 2004.

[40] A. Milanova, A. Rountev, and B. G. Ryder. Parameterized object sensitivity for points-to analysis for Java. *ACM Trans. Softw. Eng. Methodol.*, 14(1), 2005.

[41] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, 2002.

[42] X. Rival. Understanding the origin of alarms in Astrée. In *SAS*, 2005.

[43] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1), 1998.

[44] M. Sridharan and R. Bodík. Refinement-based context-sensitive points-to analysis for Java. In *PLDI*, 2006.

[45] B. Woolf. Null object. In *Pattern languages of program design 3*. Addison-Wesley Longman Publishing Co., Inc., 1997.