Research Statement

Baris Kasikci

Research Assistant, Ph.D. Candidate

Ecole Polytechnique Fédérale de Lausanne (EPFL)

It is hard to write reliable, secure, and efficient programs. Software bugs cause massive losses, security vulnerabilties, and performance problems. My main research objective is to develop techniques that help developers build more reliable, secure, and efficient software. In that regard, I develop techniques that help developers better reason about their programs, detect bugs, classify them, and diagnose their root cause.

Dissertation Research

My professional background and interactions drive my research interests. Prior to my PhD, I worked four years as a software engineer designing and implementing safety-critical real-time systems. During that time, I realized how arduous it is to write correct, efficient and secure code. My professional experience and my one-on-one interactions with developers at Siemens, VMware, and Microsoft motivated me to work on techniques that solve hard challenges faced by developers. In my dissertation, I built techniques to help developers write reliable, secure and efficient programs with a primary emphasis on concurrent programs.

In particular, I developed techniques for the automated detection, classification and root cause diagnosis of bugs. I designed and implemented: **RaceMob** [4, 6], the first high-accuracy data race detector that can be used always-on and in production; **Portend** [5, 7], the first high-accuracy data race classifier that can predict the consequences of data races under various memory models; **Gist** [2, 3, 8], the first technique to accurately determine root causes of in-production failures without relying on custom hardware or runtime checkpointing. Many of the techniques I developed require efficient runtime instrumentation and sampling. In that regard, I also developed a technique, **Bias-Free Sampling** [1] that allows sampling rarely executed code—where bugs mainly reside—without over-sampling frequently executed code. In the next four sections, I briefly describe each project in more detail. I then talk about my general problem solving approach.

RaceMob: Data Race Detection

Data race detection that is both accurate and efficient has long been an open problem mainly because of the tension between accuracy and efficiency: accurate data race detection requires gathering runtime execution information, which hurts efficiency. In particular,purely *static* data race detectors work offline without access to the runtime execution information (e.g., variable addresses). Therefore, static detectors are efficient (i.e., don't incur runtime performance overhead) but have low accuracy. On the other hand,purely *dynamic* data race detectors monitor execution information, therefore they can accurately detect data races, but they are not efficient. It is already hard to efficiently and accurately detect data races during testing, therefore prior work did not attempt to detect data races in-production, where the challenges of data race detection are only exacerbated.

In order to solve the in-production data race detection problem, I developed RaceMob [4, 6], a technique that combines in-house static program analysis with adaptive dynamic analysis. RaceMob first statically detects *potential* data races, and then dynamically validates whether these are true data races using its dynamic data race detection technique. RaceMob's static data race detection is complete: it does not have false negatives (i.e., it does not miss true data races), however it has many false positives (i.e., reports that do not correspond to real data races). RaceMob uses real-user crowdsourcing to dynamically determine whether the potential data races are real data races. RaceMob's mixed static-dynamic data race detection achieves low overhead and a high degree of accuracy.

RaceMob's dynamic data race detection approach deviates from the conventional wisdom that data race detectors need to always keep track of synchronization operations in order to accurately detect data races. In particular, data race detectors keep track of synchronization operations to build a happens-before graph that helps determine the ordering of memory accesses. If synchronization operations enforce a happens-before relationship between two accesses, then they do not race with each other. I showed that we can relax the requirement of always tracking happens-before edges by carefully determining points in the execution where tracking should start and end, meanwhile guaranteeing efficient and scalable data race detection.

Using RaceMob's mixed static-dynamic approach and its novel dynamic data race detection scheme, I showed that we can perform data race detection that is more accurate than state-of-the art data race detectors (e.g., Google ThreadSanitizer) with negligible overheads of around 2%. Such low overheads make in-production data race detection feasible, thereby paving the way to more reliable and secure concurrent software.

Portend: Data Race Classification

Another standing problem in concurrency has been the accurate identification of the consequences of data races in order to prioritize their fixing. Modern software has a lot of data races, therefore developers need to prioritize the fixing of data races to efficiently use their time[1].

I showed that the heuristics used in prior work for data race classification were prone to false positives, and the abstraction level of the classification criteria that prior work employed harms classification accuracy. I then developed Portend [5], the first technique to accurately classify data races based on their consequences. Portend is a data race classifier that explores multiple paths and schedules of a program to accurately identify the consequences of data races. Unlike what prior work suggested, I showed that low-level effects of data races on programs' memory state is not an appropriate criterion for classification, and looking at higher-level effects of data races such as externalized program state yields up to 89% higher accuracy.

I also explored the effects of the memory model (such as instruction reordering) on data races. These effects can be caused by multiple layers in the system stack such as the hardware (store buffer reordering) or the compilers (instruction reordering). I developed symbolic memory consistency modeling [7], a technique that can be used to model various memory models (e.g., weak memory) in a principled way in order to perform data race classification under those memory models.

Gist: Root Cause Diagnosis of Bugs

Detecting and classifying concurrency bugs allows developers to discover the bugs that exist in their programs, and understand which of these bugs have a high priority to be fixed. However, in order to fix a bug, developers need an explanation of how the bug manifests, that is how their program reaches the failure (e.g., what are the failure inducing inputs, thread schedules, etc.). This explanation is useful, given that developers traditionally seek such an explanation when debugging a program by reproducing failures to determine their root cause. Determining the root cause becomes even harder if failures recur only rarely in production.

I developed a technique called Gist that produces a high-level execution trace called the *failure sketch*. A failure sketch includes statements that lead to a failure and the differences between the properties of failing and successful program executions. I showed through a series of experiments on real-world software (e.g., the Apache web server) that these differences point developers to the root causes of failures, and that Gist can identify these differences accurately and efficiently: Gist built failures sketches with 96% accuracy with an average overhead of below 4%. Although I mainly focused on concurrency bugs in my evaluation (because finding the root causes of such bugs is very hard), Gist is applicable to non-concurrency bugs too.

I transferred some of the technology I built for Gist to an Intel product during my 2015 internship at Intel. In particular, I built a tool that uses static analysis to determine which statements operate on variables of given data type during a failing run. I showed that this tool reduces the number of statements a developer needs to look at while debugging by an order of magnitude. This tool is now being maintained and used at Intel.

I took a collaborative and interdisciplinary approach when building Gist. In particular, I designed Gist in collaboration with researchers from Intel and Microsoft Research. Moreover, Gist uses mixed static-dynamic program analysis: the static analysis happens offline and helps ensuing dynamic analysis be more efficient. Gist's dynamic analysis relies on hardware and operating system support.

[1]Some language semantics (e.g., C++) consider all data races as bugs, whereas others don't (e.g., Java). Regardless, modern concurrent software is ridden with data races, and developers need to better understand the consequences of data races.

Bias-Free Sampling

Many of the techniques I developed rely on gleaning execution information from in-production runs, which is challenging to do without undue runtime performance overhead. Most of the time, execution information of interest (e.g., buggy statements) resides in cold code, that is rarely executed code, and therefore monitoring and sampling cold code is particularly useful. Alas, cold code is not known a priori, therefore it is challenging to sample cold code without over-sampling hot code.

To overcome the challenge of sampling cold code efficiently, I developed a technique called Bias-Free Sampling (BfS) together with researchers from Microsoft Research. BfS allows sampling the machine instructions of a dynamic execution independently of their execution frequency by using breakpoints. The BfS overhead is therefore independent of a program's runtime behavior and is fully predictable: it is merely a function of program size. BfS forms the foundation of dynamic monitoring in my root cause diagnosis work, which uses hardware breakpoints for a component of its dynamic analysis.

BfS is very efficient: it incurs between 1—6% overhead for all the Windows System Binaries. My work on BfS had a product impact in Microsoft. In particular, using BfS, we built a coverage measurement tool for both native and managed (i.e., C#) code, which is internally used at Microsoft.

Approach to Research

Looking back at the techniques I have developed, there are three key principles that shaped my approach to problem solving:

First, I have a systems-oriented approach to problem solving. This means that I take a holistic view of a given problem, identifying the constituents of the problem at multiple layers of the system stack. In my research, this approach helped me develop techniques to deal with concurrency issues, which are ubiquitous in many layers of the system stack from parallel hardware to language memory models to operating system schedulers and to multithreaded and multiprocess applications.

Second, despite my systems-oriented approach, I frequently employ techniques from programming languages, and in particular I rely on static and dynamic program analysis. I primarily focus on finding a sweet spot in the balance between static vs. dynamic program analysis to develop efficient and accurate techniques. Static analysis tends to be efficient but inaccurate, whereas dynamic analysis tends to be inefficient but accurate, and a carefully designed mix is accurate and efficient. In my research, this mixed approach helped me strike a balance between in-house static analysis and in-production dynamic analysis.

Third, I don't shy away from unconventional approaches. In my research, adopting an unconventional approach at times allowed me to solve some key open problems, such as low overhead data race detection and accurate data race classification.

Future Research Plans

In the future, I plan to continue working on helping developers build more reliable, secure and efficient programs. In particular, I will focus on system security, privacy, and performance, primarily in the context of large-scale distributed systems. Below, I describe my short term and long term research goals by explaining how, broadening the scope of my interdisciplinary approach could benefit my goals.

Today, one of the most challenging problems in computer systems is building secure software. I am currently extending my work on root cause diagnosis to encompass security vulnerabilities. In the short term, I plan to look into using hardware support for sampling execution path profiles from user executions to detect control flow hijack attacks. Taking this idea further, in the long term, I would like to answer more general questions such as: Can we identify *good* execution paths versus *bad* execution paths (e.g., ones that lead to security vulnerabilities and failures) using techniques from machine learning? Can we automatically infer properties (e.g., performance behavior) about paths relying on statistical techniques and help developers better structure their code based on such properties? What are the meaningful boundaries of programs to monitor when gathering path information? Can we have intelligent strategies to sample paths of programs (e.g., strategies that do better than random sampling)? I believe that there are a lot of similar hard and interesting questions to answer in order to enable developers build secure software.

Some of the techniques I developed rely on gathering execution information from users, and therefore have privacy implications. To improve the privacy of the users, I initially intend to work on techniques to quantify and limit the amount of execution information extracted from user endpoints. In the long term, I would also like to work on techniques with strong privacy guarantees (e.g., computing hash-based signatures) to anonymize the execution information. For example, one can compute an anonymous *signature* describing the control flow of an execution to provide better privacy. However, it remains to be seen what are the right boundaries (i.e., within the code) for computing such signatures. Effective computation of signatures in the presence of concurrency and non-determinism is also an open question.

In the long run, I plan to extend my work on root cause diagnosis of bugs to build techniques for diagnosing performance bugs. Performance bugs could arise because efficient designs and implementations are not always obvious to the developer. I plan to build techniques to monitor real-user executions to identify performance issues in production software. I am specifically interested in identifying relations between performance bugs and developers' assumptions in their code. I would like to also study the relation between performance bugs and correctness bugs. It is not uncommon for developers to introduce correctness bugs while trying to squeeze more performance out of their programs (e.g., by allowing a seemingly benign data race to remain in their program). I would like to help developers to determine whether the optimizations they use in their code are really useful (i.e., they indeed improve performance), or they are premature and potentially leading to correctness bugs.

Many of the problems I attacked in my dissertation have incarnations in large-scale distributed systems (e.g., internet-scale systems). For instance, data races and atomicity violations manifest as process-level races that cause correctness and performance problems as well as resource leaks in distributed systems. I would like to adapt my techniques for the detection and root cause diagnosis of concurrency bugs in the context of large-scale distributed systems.

References

[1] B. Kasikci, T. Ball, G. Candea, J. Erickson, and M. Musuvathi. Efficient tracing of cold code via bias-free sampling. In *USENIX Annual Technical Conf. (USENIX ATC)*, Philadelphia, PA, June 2014.

[2] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, and G. Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Symp. on Operating Systems Principles (SOSP)*, Monterey, CA, October 2015.

[3] B. Kasikci, B. Schubert, C. Pereira, G. Pokam, M. Musuvathi, and G. Candea. Failure sketches: A better way to debug. In *Workshop on Hot Topics in Operating Systems (HotOS)*, Kartause Ittingen, Switzerland, May 2015.

[4] B. Kasikci, C. Zamfir, and G. Candea. Cord: A collaborative framework for distributed data race detection. In *Workshop on Hot Topics in Dependable Systems (HotDep)*, Hollywood, CA, October 2012.

[5] B. Kasikci, C. Zamfir, and G. Candea. Data races vs. data race bugs: Telling the difference with portend. In *Intl. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, London, UK, March 2012.

[6] B. Kasikci, C. Zamfir, and G. Candea. RaceMob: Crowdsourced data race detection. In *Symp. on Operating Systems Principles (SOSP)*, Farmington, PA, November 2013.

[7] B. Kasikci, C. Zamfir, and G. Candea. Automated classification of data races under both strong and weak memory models. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, May 2015.

[8] C. Zamfir, B. Kasikci, J. Kinder, E. Bugnion, and G. Candea. Automated debugging for arbitrarily long executions. In *Workshop on Hot Topics in Operating Systems (HotOS)*, Santa Ana Pueblo, NM, May 2013.