Research Statement

David Devecsery

Vision

Software system errors and misbehaviors present a major problem in writing, maintaining, and using significant software bases. Security vulnerabilities can lead to catastrophic data and privacy losses and great fiscal cost. Software downtime due to bugs is highly costly for companies, even when software doesn’t crash the cost of developer effort to eliminate bugs is far too high. My research seeks to greatly reduce the burden of software errors. In an ideal world, once an individual learns of some misbehavior, such as a security vulnerability, information leak, misconfiguration, erroneous data, or unexpected crash; he or she should be able to quickly identify and understand the origin of the error, what state it has affected, and the system should either identify a solution to the problem, or self correct.

My research focuses on creating systems and tools that enable efficient monitoring and understanding of complex system behavior. I combine ideas from program analysis and system design to create systems which are more robust to security vulnerabilities, enable faster and more complete understanding of prior execution, and are more stable.

I am drawn to this area because of its broad scale, and potentially lasting impact. As we’ve come to depend on software systems more, our demand for complexity in our software systems has grown. Developers must now develop applications for a variety of platforms, or distribute computation over a vast network of machines. Given the complexity and heterogeneity of today’s systems more than ever things are bound to fail, and given our increasing reliance on software systems the impact of these failures will only grow.

Research Contributions

My research is rooted in the areas of software systems and program analysis, with the goal of my research focusing on enabling quick and efficient detection and recovery of software system mis-behaviors. In an ideal world programmers would find and remove all bugs before production; however, due to the complexity of today’s software systems, this has proven intractable. Ultimately, we encounter production code with deep, complex errors, many of which are neither detected or logged by today’s runtime systems.

To address this, I have worked on three sub-problems in this space: enabling complete analysis of prior computation, reducing work done by program analysis, and reducing the latency of program analysis through parallelization. In working with these problems I found a need to use a wide array of techniques from different communities, including: dynamic and static program analysis, deterministic record and replay, distributed computing techniques [3], parallelization methodologies [4], speculation, and dynamic causality tracking.

I will now further discuss some of my contributions and insights in these areas.

Optimistic Hybrid Analysis [1]

I developed a new way to dramatically accelerate dynamic analysis without loss of soundness or precision by introducing a new type of unsound static analysis. Dynamic program analysis is foundational to our ability to detect, understand, and correct unwanted program behavior. Dynamic analysis tools, such as those that detect data-races, verify memory safety, and identify information flow, have become a vital part of testing and debugging complex software systems. However, their substantial runtime overhead currently limits their effectiveness. Researchers have explored many techniques to reduce this overhead, such as hybrid analysis. Hybrid analysis uses static analysis to prove certain properties of a program, then applies those properties towards
optimizing a dynamic analysis. Unfortunately, to remain sound, static analysis must make many conservative assumptions, limiting its accuracy and reducing its effectiveness in a hybrid analysis.

Due to the conservative nature of sound static analysis, hybrid analyses are unable to adequately optimize many dynamic analyses. It is common that only a subset of possible execution paths in a program will ever be run (imagine a web-server run with a single configuration, or a crypto library with many cipher suites that only uses one), consequently, to adequately optimize a dynamic analysis, I observe that we should target static analyses towards only the executions that actually run.

Optimistic hybrid analysis (OHA) does just this, by optimizing the analysis with an unsound predicated static analysis, and then speculatively executing the final dynamic analysis. OHA first profiles executions to learn the commonly executed states of the program, summarizing them as likely invariants. These likely-invariants are then assumed as predicates by a predicated static analysis, a highly-accurate, conditionally sound static analysis. This conditionally sound static analysis is then used to optimize dynamic checks. Finally, the dynamic analysis is run. It executes only a subset of the original checks due to its more accurate static analysis, and also verifies the likely invariants hold at runtime to ensure the predicated static analysis remains sound for the given execution. This dynamic analysis is speculative; if the invariants do not hold at runtime, the analysis must recover from potentially missing analysis state to retain soundness.

I demonstrate OHA’s ability to dramatically accelerate heavyweight program analysis by applying it to two analyses: data-race detection for Java, and dynamic backward slicing for C. I show this methodology dramatically accelerates both race-detection and backward slicing, with dynamic run-times on average 1.6 and 7.7 times faster respectively than a traditional hybrid analysis and no loss in analysis precision. The dramatic reduction in overhead from OHA greatly enables the use of many powerful, but traditionally heavy-weight dynamic analyses, such as always-on data-race detection and taint tracking, or forensic All-to-all DIFT queries.

Eidetic Systems [2]

The vast majority of state produced by a computer system is generated, consumed, then lost forever. With this lost state is lost value: users cannot recover information about past computation that would be useful for auditing, forensics, debugging, error tracking, and many other purposes. Today’s systems employ ad-hoc methods to preserve a limited amount of state, such as logging. I propose instead systems should by default remember all state. This work proposes the first practical eidetic system, Arnold. Arnold can record years of computation on a commodity hard drive and has under 8% runtime overhead for most benchmarks evaluated.

Arnold uses many new techniques to achieve both efficient recording and recall at modest costs. At its heart is a novel deterministic record/replay architecture which structures recordings into groups of communicating processes. These process groups allow Arnold to limit the amount of inter-process communication it records, without significant sacrifices on replay time. Arnold additionally uses a novel inter-process group dependency tracking mechanism, model-based log compression, and replay file-caching, allowing it to record/replay vast amounts of data efficiently.

I show Arnold’s benefit by looking at two forensic queries. One where a user error causes erroneous data to propagate through a system, and another where a Heartbleed vulnerable web server leaks private user data. In both instances Arnold is able to recreate the entire history of the data, allowing lineage tools to trace precisely where the error occurred, and what data was affected by it. With these results Arnold shows an exciting direction in forensic analyses, in which always-on recording and analysis is both practical and powerful, users never have to wonder “what happened,” and data lost in major data breaches can be quickly identified.
Future Directions

Errors are present in complex computer systems and take an enormous effort by users, programmers, and system administrators to resolve. I envision a world in which programmers, users, and system administrators have a large, powerful set of tools at their disposal to quickly and efficiently reason about, understand, or even prevent these errors. In my prior work I show that by combining expertise from both systems and program analysis I can create powerful systems and analysis methodologies which greatly enable these robust tools. In the future I will continue to apply the techniques I’ve discovered, as well as insights from my experience in this area to further develop these powerful techniques to aid in mitigating, understanding, and resolving runtime errors.

Strong Guarantees in Runtime Environments

Consistently writing reliable programs requires runtime systems that support and define manageable and reliable semantics for the programs run on them. Unfortunately, many of the guarantees provided by today’s runtime systems are fragile, often providing undefined behavior for a single programmer error. My experience with hybrid static-dynamic program analyses can greatly aid in creating more stable, usable runtime environments. I will start by exploring parallel program memory models.

Today’s memory models typically adopt weak DRF0-like semantics. These models allow programs to express strong semantics, such as region serializability (RS) or sequential consistency (SC), only if they are data-race free. In the instance of a race, they leave the execution semantics entirely unspecified. Since current tools are far from efficiently and soundly detecting all data-races, programmers are unsure if the code they have written even has defined semantics.

I observe that since the original DRF0 systems were designed in the early 90s, new analysis techniques and hardware, such as my work on optimistic hybrid analysis and Intel’s Hardware Transactional Memory, have reduced the overhead of providing stronger runtime guarantees. I will explore using these new features to either enforce a known strong memory model, such as RS, or explore a new memory model that allows reasoning about programs with races, presents strong semantics, and still has reasonable runtime overheads.

Beyond memory models, I believe similar techniques can help to create reliable powerful abstractions which aid in other aspects of system creation, such as memory safety, file-system consistency, dead-lock avoidance, and atomicity violation avoidance.

Creating Better Analyses

Program analyses, both static and dynamic, are foundational to better understanding and stronger runtime guarantees of today’s critical applications. Analyses such as memory safety, taint tracking, race-detection, and more help detect errors, and guarantee safety in modern programs. Unfortunately, these analyses are often not adopted in production today. To achieve my goal of having practical, powerful tools at all users, administrators, and developer’s fingertips, the latency, overhead, and usability of these systems all need to be improved.

In my prior works I have introduced and discussed the novel analysis methodologies, such as OHA and analysis parallelization, aimed at significantly improving the performance and latencies of dynamic analysis without sacrificing soundness or precision. However, my dynamic static hybrid approach to analysis has only scratched the surface on the contributions I can make to program analyses, including OHA. I plan on focusing on program analysis in several ways.

First, I will create more efficient analyses. My work on OHA was a good start to reducing the burden to running traditionally heavy-weight analyses in production systems, however OHA still has many issues, such as rollback and latency/overhead predictability which make it impractical
for production use. I hope to work on overcoming these issues, while reducing the overhead of what are traditionally heavy-weight analyses.

Second, I will explore creating more powerful analyses. There are many similarities between dynamic and static data-flow analysis, however, many of the optimizations or techniques used in one family of analyses have not been used in its counterpart. I believe that many fundamental data-flow analyses used today are still sub-optimal, and could be greatly improved by applying known techniques, from a different area.

Finally, the tools to create static/dynamic hybrid data-flow analyses today are lacking. There have been many observations in both the static and dynamic analysis communities about how to build and optimize analyses, general and specific, yet no good method of easily creating a provably correct hybrid static/dynamic analysis exists today. I believe there are many challenges to overcome in creating such a framework, but it is an essential step to making hybrid analyses a practical tool for system designers and developers.

Debugging Tools for Production Systems

In an ideal world, developers, users, and system administrators would always monitor their program and have all information needed to detect and understand an unexpected behavior or a system crash. However, in reality, limited resources and limited foresight into system misbehaviors result in many systems only having partial information, such as program outputs, basic logging, or possibly stack dumps available when a crash occurs.

It is important to have tools to investigate system misbehaviors that happen even when the users of those systems are not expecting it, and I will investigate making these tools in the future. This is a challenging problem, as the partial information provided by the system’s logging functions is likely insufficient to conclude either the existence or absence of some bug within the system. However, automated tools can still greatly aid in the debugging process, either identifying and ranking potential bugs, or creating useful analyses to help programmers search the information flow of the program.

I believe my prior experiences translate well to solving this form of problem. My prior work with hybrid analyses has shown me how analyses can incorporate both sound and unsound assumptions to create much more meaningful conclusions about programs. Additionally, my work with record-and-replay systems has analyzed how differing inputs and different program states can lead to different program behaviors and outputs. I believe that techniques I’ve developed in these areas could greatly aid in reducing programmer’s burdens when debugging production crashes in large complex systems.

References