Seminar Talk: “Like by Hand: Improving Visual Exploration of Control Flow...” (Speaker: Dr. Kate Isaacs)

Matthew Whitesides

Abstract

In today’s presentation, Dr. Kate Isaacs takes us into the world of code visualization. Dr. Isaacs and her team have made various novel advancements in control flow graph generation and code visualization over the past years, particularly related to language-agnostic generate, readability, and the complexity of the generated graphs, keeping the detail from the source code in the generated graphs.

I. Introduction

CODE visualization is the process of taking project code from a given programming language and generating various human-readable direct graphs depicting things like object relationships, program execution flows, memory states, etc. Generating these visualization graphs is a deceptively complex topic. Essentially you need to build a full language compiler that understands relationships between objects beyond a typical compiler, on top of a programmatic visualization layer that can be readable under any permutation of graphs. Generating these graphs is particularly helpful in software engineering. They enable a designer to view how the program architecture executes, spot unreachable code areas, reduce complexity, debug business logic errors, and more. In this presentation, Dr. Issacs shows how visualization methods have been improved, and new development methods represent complex computing processes for exploration and analysis. Using these techniques, we can enhance advanced technologies such as human cyber-physical systems, high-performance and distributed computing algorithm development, and environmental optimization planning.

II. Background

So what is a control flow graph (CFG)? Simply it is a graph describing how statements in a program are executed given specific inputs. It shows statements executing sequentially and describes the logic gates in the program, such as if statements and loops. They provide a human-readable abstract of code execution. The nodes in the graph are blocks of code, while the edges represent places where we can jump or branch to the next instruction in the sequence. The importance of control flow graphs is shown in each area of the program lifecycle. Once generated, they can show program compilation dependencies, help optimize code, and are vital to developing security code models, all of which rely on graph abstractions of commands executed in code. These models all fall in the uncertainty principle that no program is guaranteed to terminate, so control flow graphs greatly aid in mapping command and object outcomes that your particular program could generate. CFGs are particularly useful when tracing lower level code such as assembly dumps which is a vary complex and laborious process if done manually.

III. Research Contributions

A. Existing Tools

There exist a few main CFG generating tools for different applications. The LLVM compiler has a CFG generating feature built-in. However, the scope of the generated graphs is limited, and LLVM is specific to the C language compiler. On a broad security side, IDA Pro is a disassembler capable of creating maps of their execution to show the executed binary instructions. This is a language-agnostic disassembler/debugger that has CFG capabilities. However, it is costly and has limited CFG capabilities as well. Another issue with these existing tools is their focus. These tools focus mainly on program debugging and security for limited-scope programs. Dr. Isaacs and the team have focused on using CFGs on large-scale data projects optimization, something nearly impossible on these existing debugging tools.

B. New Developments

CFG is used to improve the performance of programs, from the smallest application to those used in the most powerful infrastructures in the world. Specifically, we’re focusing on high-performance machine learning and AI programs that simulate weather patterns or biological simulations. Even a tiny improvement in the performance or accuracy of these applications could save countless hours, dollars, or even lives.

Generally, examples of CFGs are shown using small programs, and even a moderate-sized application would generate a CFG so large it is generally not very useful to a human reading it. On top of that, mapping the actual code or assembly to each node in the CFG would exponentially complicate the visualization. This is where Dr. Isaacs comes in. She and her team have improved techniques for generating a CFG that the actual developers would find helpful and integrated methods of doing the performance analysis tasks, so performing these complex operations is more accessible. The focus is on automating what generally was done by hand after the CFG is generated, such as mapping lines of code to the CFG nodes or understanding where locks and issues may occur.

The first collaboration was a language-agnostic optimization from dynamic instruction traces. Language independence came from the idea that the most common data science programming language is Python. This high-level language typically does not have the lower-level analysis tools that languages such as C have. Iscaas and team developed a language-independent interpreter that utilizes an LLVM backend to produce an optimized executable. Optimization is achieved using program instruction traces, not the compiled language itself, to produce the optimizations. Using this, a developer can generate an instruction trace and control flow graph to begin their analysis. However, this proved to be a challenge because the low-level execution interpreting and developing these optimizations took many hours per program and rarely used the final output.

IV. Results

CcNav was developed to support tasks requiring assembly linkage that supports linking code, disassembly, CFG, and call graphs. They discovered that most researchers are merely looking for anything evident to optimize in a CFG, such as extensive loops or comparing compiler optimizations. However, not all users need this much detail and only get in the way of discovering the program flow, so CFGExplorer was made just for CFG visualization. To further support specific customer needs, CFGConf was designed to give precise control over the analysis and visualizations generated by the CFG system. This takes in a JSON config file that lets you limit what is produced, such as if you do not want to see nodes in the graph related to external libraries or execution not part of your specific analysis. Configuration cuts down the size and complexity of graphs and enables useful visualizations depending on the user's needs.

These projects show how a key focus of program optimization and visualization is about understanding and considering how these tools will be used in a development workflow. From the initial survey of researchers, Dr. Isaacs discovered that most utilization of these tools was very time-consuming and produced little useful information or CFGs too complex to decipher much actionable information. Therefore tools such as CFGExpolorer, CcNav, and CFGConf were developed with these needs in mind to enable a user to quickly see the useful visualizations information and configure the outputs for their specific needs.

V. Lessons Learned

Visualization is an exciting topic, one that, as a developer, you can take for granted. The first thing that strikes me is how complex the task is to develop these graphs and optimizations. Essentially you need to build a full compiler and then build on top of that an engine to understand and map execution relationships to each other, all while building a somehow readable control flow graph after the fact. It essentially takes complete mastery of all aspects of computer science to pull off often for little appreciation. In my experience, using code visualization tools takes time and provides little practical information, so the efforts are unnoticed. However, with advancements such as the ones proposed by Dr. Isaacs and her team are shown, I believe these tools could be more widely adopted and, more importantly, can be used to address security vulnerabilities and optimizations to large real-world programs typically too complex for general visualizations.

VI. Conclusion

Overall, Dr. Kate Isaacs presented an exciting exploration in developing code optimization and visualization tools, the existing state of CFG generation, to advancements that support users' needs to improve code quality and debugging. This presentation shows a unique area of computer science research that I believe is overlooked in modern developments. Improvements in debugging, analysis, and code generation are fundamental to software engineering and will help facilitate future development for all software applications.

Acknowledgment

The author would like to thank Professor Sajal Das with the Department of Computer Science, Missouri University of Science and Technology and Dr. Kate Isaacs with the University of Arizona.

Matthew Whitesides Master's Student at Missouri University of Science and Technology.