

Accelerating Verification and Software Standards Testing (AVASST) with Large Language Models (LLMs)

Ryan Karl*
Carnegie Mellon
University - Software
Engineering Institute
rmkarl@sei.cmu.edu

Yash Hindka*
Carnegie Mellon
University - Software
Engineering Institute
yhindka@sei.cmu.edu

Shen Zhang
Carnegie Mellon
University - Software
Engineering Institute
szhang@sei.cmu.edu

John Robert
Carnegie Mellon
University - Software
Engineering Institute
jer@sei.cmu.edu

*These authors contributed equally to this work

Abstract

The recent explosion in large language model (LLM) technology has highlighted the challenges of using public generative Artificial Intelligence (AI) tools in classified environments, especially for software analysis. Currently, software analysis falls on the shoulders of static analysis (SA) tools and manual code review, which tend to provide limited technical depth and are often time-consuming in practice. We show that LLMs can be used in unclassified environments to rapidly develop tools that accelerate software analysis in classified environments. Through LLM assistance, our work has produced several avenues for success, and preliminary experimentation has shown significant time savings (~40%) and improved accuracy (~10%) for certain software analysis tasks.

Keywords: large language models (LLMs), static analysis (SA), requirements verification, data flow analysis, control flow analysis

1. Introduction

Large language models (LLMs) are advanced artificial intelligence (AI) technologies that offer unprecedented capabilities in generating natural language text and code at scale. LLMs are significantly transforming the landscape of software engineering and have introduced new opportunities for automation. The application of LLMs to enhance and accelerate the software development lifecycle (SDLC)—which outlines processes for producing software in a time- and cost-efficient manner—is an active area for current research (Fan, et al., 2023; Hadi, et al., 2023). Recent work has defined a spectrum of AI augmentation from conventional systems built with conventional SDLC techniques to AI-augmented systems built with AI-augmented SDLC techniques (Ozkaya, Carleton, Robert, & Schmidt, 2023).

While LLMs are a relatively new technology, SA tools have long been a part of the SDLC (Thomson, 2021). SA tools facilitate software analysis without executing programs for quality assurance.

Despite their utility, out-of-the-box SA tools often fall short of providing the quality analysis required for complex systems. SA tools do not possess human-level comprehension of code semantics, leading to a gap in the depth of their analysis (Sass, 2015). On the other hand, manual code review, despite its effectiveness in identifying issues that automated tools might miss, is time consuming and scales poorly with the size of modern software projects (Ami, Moran, Poshyvanik, & Nadkarni, 2024). These shortcomings present a significant bottleneck in the SDLC, impacting both the speed and quality of software delivery.

2. Thesis Statement and Contributions

In an effort to resolve these SDLC bottlenecks, we demonstrate how LLMs can help develop tools that streamline traditionally manual aspects of software analysis. Unlike existing research on directly applying LLMs to software analysis tasks, our research suggests that leveraging public LLMs (i.e., an internet accessible, commercial/open-source model) for SA tool generation offers increased efficiency and a greater technical depth for software analysis tasks (Sherman, 2024). Our work can improve code verification in classified environments; for the purposes of this paper, classified environments refer to any area that handles sensitive information.

Our paper highlights the following contributions:

1. We provide a methodology for incorporating LLMs in SA tool generation for classified environments.

2. We use our methodology to demonstrate that LLMs can help develop tools that accelerate software analysis.
3. We present novel code visualization capabilities developed with the help of LLMs.
4. We describe our experimentation efforts to empirically verify that our tools and techniques improve efficiency and accuracy.

3. Background

3.1. Overview of Software Analysis and LLMs

Though SA tools began as simple linting tools, an increase in software systems' complexity over time led to a need for deeper analysis. This resulted in the development of modern SA frameworks that can detect defects and vulnerabilities (Thomson, 2021). While these advances have transformed how developers approach code quality, they have also required developers to obtain new skills to competently assess SA tool output (Chess & West, 2007).

However, modern SA tools can struggle to adapt to niche software environments, since their predefined rulesets may not be aligned with an organization's unique coding or security policies. Furthermore, software development practices and technologies evolve rapidly, and SA tools can become outdated if they do not adapt to new programming paradigms, vulnerability disclosures, or architectural styles (Brown, 2020). This can lead to a high rate of SA false positives and/or negatives, requiring analysts to use other approaches to prioritize the large volume of SA results (Flynn, et al., 2018; Johnson, Song, Murphy-Hill, & Bowdidge, 2013). As a result, advanced threats that exploit specific business logic may go undetected.

For example, the HeartBleed bug of 2014 (CVE-2014-0160) was a critical vulnerability in the OpenSSL cryptographic library, affecting over half a million web servers and various internet-connected devices. Many SA tools (e.g., Coverity) failed to detect the subtle memory handling issues that enabled HeartBleed (Synopsis Editorial Team, 2014). This was likely caused in part by market dynamics and the demand for faster development cycles, which force many SA tool providers to prioritize speed and reduced false positive rates over comprehensive defect detection (Sass, 2015).

These faster development cycles have also decreased the time allocated for manual code reviews. Manual code reviews, while crucial for ensuring code quality and security, can be time consuming and may

contribute to bottlenecks in development cycles and product release schedules. These bottlenecks can be further exacerbated if issues are found late in the development cycle, requiring significant software rewrites (Sass, 2015).

Innovative approaches to software analysis, such as the application of LLMs, are poised to help resolve these bottlenecks (White, Hays, Fu, Spencer-Smith, & Schmidt, 2023; Li, Hao, Yizhuo Zhai, & Zhiyun Qian, 2024). Research on the integration of LLMs with traditional software engineering environments is improving software developer productivity, and many researchers are active in this area (Zheng, et al., 2023; Ling, Kim, & Chen, 2024). New approaches, like rapid source code development through ChatGPT and prompt engineering, have been proposed to generate code efficiently, and evaluations of LLM-supported tasks highlight their potential in coding and debugging (Wang, Ning, Zhang, Liu, & Wang, 2024). Similarly, AI-assistant frameworks that use LLMs can facilitate rapid prototyping and iterative development, reducing time-to-market (Simaremare & Edison, 2023). The development of tools such as DevBench, which provides comprehensive benchmarks for evaluating LLM performance across different coding tasks and environments, ensures that these models meet professional standards (Li, et al., 2024).

3.2. Issues with LLMs in Software Analysis

LLM-generated code can be error-prone, which highlights the need for robust validation and testing frameworks (Liu, Tang, Luo, Zhou, & Zhang, 2024). LLM reliability issues are particularly concerning when LLMs are applied to critical tasks, such as vulnerability and quality analysis. These sophisticated tasks often require a deep contextual understanding of software's logic and execution flow. LLMs, constrained by their training data, often lack the subject matter expertise of trained software professionals and may fail to fully grasp the intricacies of complex systems. For example, the CERT Secure Coding team's experiments have demonstrated LLMs' limitations in identifying and correcting C code noncompliant with a coding standard (Sherman, 2024). Current studies acknowledge these limitations and emphasize the need for human oversight when using LLMs for software generation and evaluation (Alshahwan, et al., 2024; Borg, 2024). Rather than use LLMs in a vacuum, users should take care to rigorously validate LLM output before deploying it (although it is unclear if human quality assurance is always adequate) (Ami, Moran, Poshvanyk, & Nadkarni, 2024).

The inherent constraints of classified environments pose unique challenges to deploying LLMs. Many government organizations that steward classified data must ensure their data's integrity and confidentiality is preserved. Such organizations typically must keep all classified data air gapped or on premises. However, LLMs are often deployed via online web portals or APIs. Local offline support is often not available, and naïve data sanitization solutions (i.e., applying anonymization techniques before feeding obfuscated data into a web-based LLM) are not viable. Also, the resources needed to deploy LLMs on premises can be cost prohibitive, limiting their accessibility and utility in secure, resource-constrained environments.

We investigated deploying an open source LLM locally in a classified space. However, its natural language and code generation capabilities lagged significantly behind online LLM solutions, which required fewer iterations of prompting to achieve accurate and reliable outputs. Organizations that move forward with deploying local LLMs are unlikely to capitalize on frequent advancements in online LLM technology at the speed of relevance.

4. Software Engineering with LLMs

4.1. Methodology for Incorporating LLMs in SA Tool Development

Although there has been a great deal of work on the efficacy of LLMs for direct software analysis, the current limitations of LLMs prompted us to investigate their utility in developing *tools* for software analysis. This new research direction leverages natural language prompting to allow LLMs to assist with developing SA tools that can operate in classified environments. In addition, it allows end users with minimal software experience to customize these SA tools.

We developed a novel, hybrid methodology that allows public LLMs to develop tools to accelerate the analysis of classified code while still maintaining a safe distance between those LLMs and classified environments. In our methodology, we input public (i.e., unclassified) material related to software verification tasks into public LLMs, which assist with the design, implementation, and customization of SA tools that are later reviewed and migrated to a classified environment.

Software tools developed using our approach must be reviewed as described in software security verification standards before being deployed in classified

environments. The specific policies for reviewing software before deployment on a classified system follow National Institute of Standards and Technologies (NIST) guidelines and are generally proportional to the level of required security. Although these security reviews must still be performed with our approach, our use of LLMs to create SA tools provides significant control over the software deployed in a classified environment. Note that addressing internet confidentiality concerns (i.e. whether IP addresses reveal sensitive organization identity information) are beyond the scope of this paper.

Our methodology can be used to generate SA tools for unclassified environments as well. In such cases, the public LLM could have access to the codebase, test cases, and software output, potentially increasing its value in generating SA tools.

Our methodology can be split into two different workflows:

1. a tool developer workflow
2. a software analyst workflow

Figure 1 shows the workflow for a tool developer. In this workflow, the tool developer collaborates with software analysts to develop a set of software requirements (e.g., quality attributes, programming paradigms) that drive the prompts fed into a public LLM. The LLM is then used to help design architectures from which a tool developer can down-select based on the analysts' priorities (e.g., efficiency, maintainability). Once the final architecture is selected, the LLM can help tool developers implement various prototypes, from which the tool developer can further down-select based on quality attribute requirements (e.g., interoperability, scalability).

Upon completion, tool developers should have a tool that can help to verify some or all the requirements that encompass the verification task. To test the tool, developers can use a variety of approaches, including using an LLM to generate test code. After rigorously testing the generated SA tool, it can be safely deployed in a classified environment.

In a case study, we applied this workflow and leveraged an LLM to quickly iterate through various prototype architectures for a SA tool. After down-selecting to a plugin-based architecture, which could facilitate the seamless integration of future tool expansions, we leveraged LLMs to help us implement our SA tool. See Section 4.2 for more discussion of our case study.

Figure 2 shows the workflow for a software analyst. In this workflow, a software analyst leverages a public LLM to decompose the previously developed requirements into a suite of plugins (i.e., SA checkers). Once these plugins are migrated to a classified environment, they can integrate into our previously developed SA tool. Then, analysts can run the enhanced SA tool over classified code and use its output to help verify a codebase meets specified requirements. Because our methodology is iterative, analysts can continue to customize existing plugins to increase their efficacy and develop new plugins to fit the changing needs of their organization.

In Figure 1 and Figure 2, a red dotted boundary highlights the tools or activities that can be transitioned into classified environments. In the future, a local, on-premises LLM might be introduced into the classified environment to expand the classified boundary and potentially improve the efficacy of our workflows.

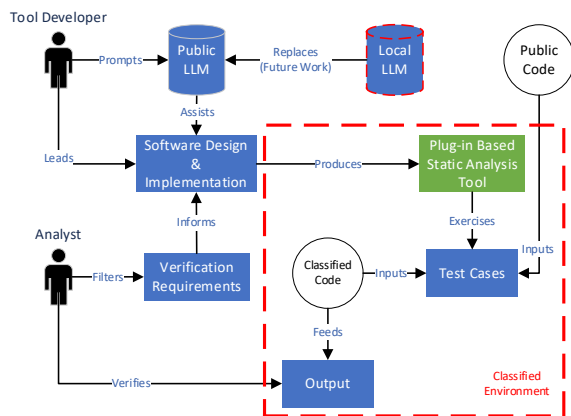


Figure 1. AVASST Tool Developer Workflow

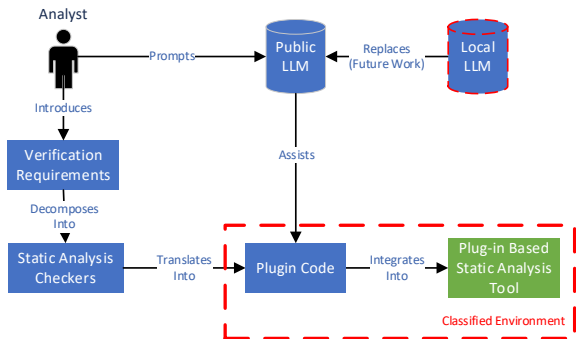


Figure 2. AVASST Analyst Workflow

In our experience, the proposed methodology can be applied to verification tasks that can leverage any of the following inputs:

1. a refined natural language prompt

2. an open-source government regulation (e.g., FIPS, NIST)
3. an open-source codebase
4. an open-source testing framework

For example, properly prompted inputs to LLMs can partially automate the generation of several artifacts:

1. SA checkers that conform to standard compliance rules (e.g., MISRA (The MISRA Consortium Limited, 2024))
2. code to perform control/data flow analysis
3. Graphical User Interface (GUI) code
4. scripts to generate visualization and reports

Often, the inherent limitations of LLMs must be considered when selecting tasks for our methodology. Because LLMs generate language and code based on statistical patterns rather than actual comprehension, tasks that require deterministic and accurate outputs must be verified by a human (note that there are known limitations to human verification of LLM output (Shankar, Zamfirescu-Pereira, Hartmann, Parameswaran, & Arawjo, 2024)). At the same time, effective prompt engineering techniques can help to mitigate some of these limitations.

4.1.1. Prompt Engineering

Prompt engineering, the process of structuring a prompt to be better interpreted by an LLM, significantly influences the utility of LLMs in software development (Desmond & Brachman, 2024). It is often required to generate high quality outputs and optimize LLM performance for specific tasking (Wang, Zhou, & Chen, 2024; Rosa, 2024). Current research highlights the importance of continuous refinement and adaptation of prompts to meet evolving project requirements (Sundberg, 2024; Ridnik, Kredo, & Friedman, 2024; Arawjo, Swoopes, Vaithilingam, Wattenberg, & Glassman, 2023; Rosa, 2024).

Building on existing research, which focuses on generalized best practices for prompt engineering in a software context, we developed our own set of prompt engineering guidelines tailored for SA tool code generation. We present our guidelines here:

1. When using vague terminology, provide a definition to allow the LLM to align itself with your intended use case. This synergizes with other research that recommends providing the LLM with important context (Rosa, 2024; Wang, Zhou, & Chen, 2024).
2. When generating code, instruct the LLM to do the following:

- a. Follow a specified design pattern (e.g., dependency injection, object-oriented).
 - b. Incorporate comments explaining how the code works.
 - c. Be as detailed as possible.
 - d. Generate code understandable by a professional software developer.
 - e. Incorporate error handling capabilities.
3. Review and test any code generated by an LLM for defects. Existing research also recommends testing the logic of LLM-generated code (Wang, Zhou, & Chen, 2024).

Integrating these guidelines with our methodology enabled us to use public LLMs to assist with parts of the tool-generation process. We speculate this significantly accelerated the development of our SA tools and decreased costs.

4.2. LLM-Assisted SA Tool Generation

To demonstrate the effectiveness of our hybrid methodology, we used it to develop a set of custom SA tools and collaborated with software analysts working in a classified environment to ensure the tools were relevant to their work. Iterating on the tools with these analysts allowed us to refine requirements and optimize use cases. From discussions with analysts, we found two primary pain points that were significantly impacting their productivity:

1. **Government software requirements validation:** This is the process of validating that software complies with government safety and/or security requirements.
2. **Critical signal analysis (CSA):** This is the process of verifying the integrity of safety critical data as it flows through the program.

We leveraged our methodology to develop two custom SA tools to assist analysts in these two areas:

1. **AVASST Plugin Suite:** This is a tool with a plugin-based architecture that enables analysts to develop custom checkers (i.e., plugins) using the Clang-Tidy framework to help verify government software requirements.
2. **FlowFusion:** This is a visualization tool that leverages CodeQL to perform control flow and taint analysis to help analysts perform CSA.

4.2.1. AVASST Plugin Suite

The AVASST Plugin Suite leverages Clang-Tidy to provide custom SA checkers for analysts. Clang-Tidy is a linter tool for C and C++ that is built on top of the Clang compiler frontend. It is used to perform SA on code to catch errors, enforce coding standards, and suggest improvements. It includes a library of standard checkers out-of-the-box and supports the creation of custom checkers that search for structural features of the compiled software's abstract syntax tree (AST) via the ASTMatcher library. We leveraged LLMs to assist with our understanding of the ASTMatcher library and generate skeleton code for using existing Clang-Tidy checkers. Then, as mentioned in Section 4.1, we used LLMs to iteratively develop a plugin architecture that supports the simple integration of future LLM- or human-generated custom checkers. In addition, we used LLMs to help evaluate the viability of existing Clang-Tidy checkers for validating government software requirements. Where we found gaps, we prompted LLMs to help us translate high-level software requirements into ASTMatcher queries. This helped us to develop an initial set of custom checkers for validating the integrity of message data. LLMs also helped us develop a Command Line Interface (CLI) and GUI to enhance usability and allow users to graphically combine checkers into groups to verify selected government software requirements.

4.2.2. FlowFusion

CodeQL is a tool developed by GitHub that uses a proprietary AST parser and other techniques to statically analyze a codebase (GitHub, 2024). It provides a query language to enable users to search for structural code features that manifest in an AST.

CodeQL can be used to perform taint analysis—the tracking of data that has been influenced (i.e., tainted) by an external, potentially untrusted source as it flows through a program. Taint analysis can be used to detect vulnerabilities such as injection attacks, where tainted data affects sensitive parts of a program lacking proper input validation. We leveraged LLMs to assist with the development of CodeQL taint analysis queries that track critical signals flowing through the software.

However, taint analysis alone only provided a narrow view of the critical signal pathway. Analysts found that there were additional function calls and conditions involving untainted data that were not fully captured but still important for a thorough CSA. This was not a flaw in the concept of taint flow, which is not designed to track untainted data. Nevertheless, we needed to

incorporate additional information into our tool’s output for it to be useful for analysts.

Thus, we proposed FlowFusion, the novel concept of overlaying a taint flow graph on top of a control flow graph. FlowFusion offered a more complete picture of the critical signal pathway, and enabled analysts to proceed with more confidence in their analysis. Figure 3 shows a FlowFusion visualization containing grey rectangle and orange oval nodes. The grey rectangles represent functions, and edges between rectangles represent control flow. In addition, the orange ovals represent tainted data, and edges between them represent taint flow. The various graph nodes are also interactive, allowing users to navigate to the exact code locations visualized in the graph by simply clicking on a node. By combining the control and taint flows into an interactive overlay visualization, analysts can have a more comprehensive view of the pathway critical data takes through the program and accelerate their verification of software requirements.

Figure 4 shows that CSA has seven steps, four of which are automatable (green) using FlowFusion and three of which require human intervention (red). New research in the field of human-AI teaming suggests we may be able to introduce LLMs to assist analysts with the red tasks (Vats, et al., 2024); we did not pursue this direction due to time constraints but plan to in future work.

4.2.3. Current Limitations

Although our work represents a step forward in improving software analysis, it does have some limitations. First, recall that even best-in-class, public SA tools often cannot fully verify requirements. Similarly, we should not expect our custom tools to provide a cure-all solution either. However, our tools are still worthwhile if they provide some acceleration

in requirements verification. Second, our tools require analysts to know the structural features they wish to search for in the codebase. However, analysts unfamiliar with a codebase may not know what to search for a priori, making it difficult to construct high-quality queries. Third, our approach requires a human-in-the-loop to verify that the tool outputs are accurate, since LLMs were used to create the tools and LLM-reliability issues (e.g., hallucinations) can sometimes lead to inaccurate output. The tools themselves can also output false positives or false negatives that require human adjudication, like most SA tools.

4.3. Assessing the Impact of the Tools

To evaluate the long-term usefulness of our tools, we delivered them to an in-house team of analysts for use during software analysis tasks. These tasks traditionally require extensive manual effort, which is both time-consuming and prone to human error. We hope that our LLM-augmented tools significantly speed up the analysis process and produce a higher degree of accuracy than traditional methods.

We expect our tools to enhance collaboration and consistency among team members by providing a common platform for sharing custom LLM-generated plugins. In addition, we project that efficiency gains achieved through our tools will extend across the entire SDLC. As analysts receive multiple versions of software over time, our tools should enable them to quickly replicate specific analysis checks and analyze changes between each version; a manual analysis of changes would likely prove more time consuming. While we await the long-term efficacy results, we have developed some initial confidence in our tools through preliminary experimentation and surveys.

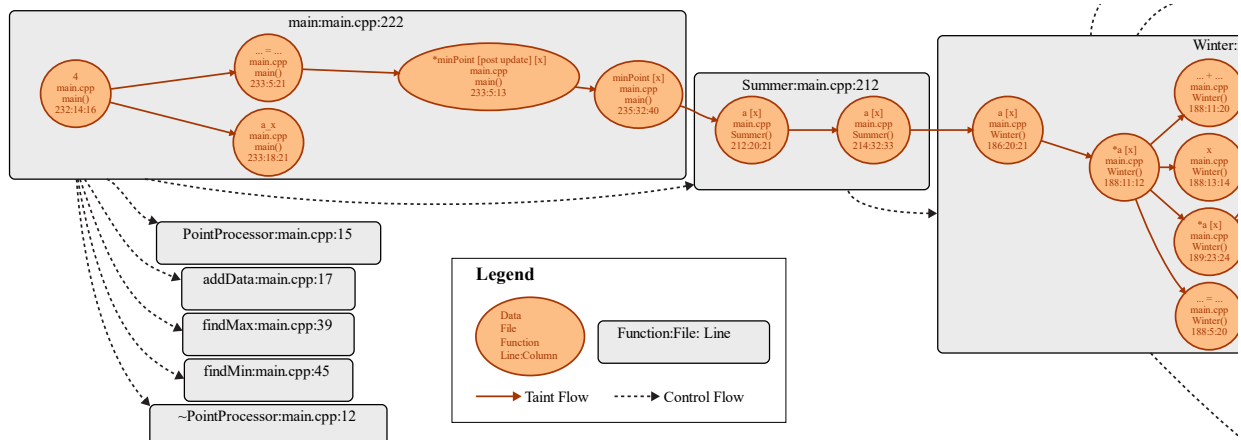


Figure 3. Example Visualization from FlowFusion

4.3.1. Quantitative User Testing and Results

We designed an experiment to quantitatively assess how FlowFusion enhances the efficiency and experience of our in-house team of analysts. Five analysts with software experience levels varying from entry level to senior participated in the experiment.

There were two stages to the experiment: control and experimental. In each stage, we asked participants to answer a set of nine identical questions related to various aspects of CSA, such as tracing critical signal data flow, identifying relevant control flow, and comprehending the overall code structure. In the control stage, participants used a nontrivial codebase to answer these questions using manual analysis. In the experimental stage, participants used a FlowFusion visualization that accompanied a second, similarly complex codebase to answer the same questions.

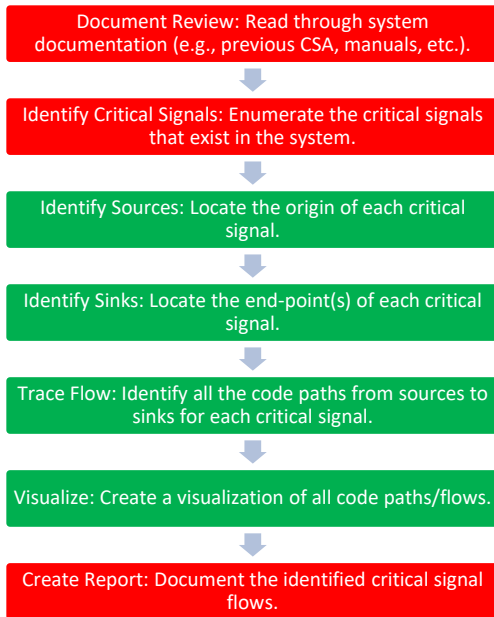


Figure 4. CSA Workflow

In Table 1, we present a mapping of each question pair (identical questions exercised with and without FlowFusion) versus CSA focal areas. The following focal areas are defined:

- 1. Taint Initiation and Termination Points (TITP):** These are locations in the code where taint flow begins and ends.
- 2. Control and Taint Flow Depth (CTFD):** This is the depth of relevant control flow and taint flow.
- 3. Identification of Tainted Variables (ITV):** This involves detecting a tainted variable and

tracking the propagation of its taint through a program.

- 4. Function Call Analysis (FCA):** This involves mapping what functions call a specific function and what functions are called by said function.
- 5. Order of Function Calls (OFC):** This involves tracking the topological ordering of function calls, especially those that propagate taint.

Table 1. Map of CSA Focal Areas to Question Pairs

Question	TITP	CTFD	ITV	FCA	OFC
Q1/Q10	X				
Q2/Q11		X			
Q3/Q12			X		
Q4/Q13				X	
Q5/Q14				X	
Q6/Q15					X
Q7/Q16				X	
Q8/Q17		X			
Q9/Q18					X

For our data analysis, we focused on comparing the time and accuracy metrics between the control and experimental stages. Overall, our findings indicated several increases in efficiency and accuracy during the experimental stage. The median percent of time saved per question across all analysts when using our visualization was 38.52%, while the median accuracy boost expressed as a percent was 10.00%. Although time savings were not uniform, there was time saved for at least one user for each question, suggesting that there was some acceleration across all questions. These time savings resulted in a cumulative savings of approximately 1.3 work hours during the experiment, which took a total of 6.75 work hours. A per-question breakdown of percentage time saved and change in percent accuracy is provided in the graphs below.

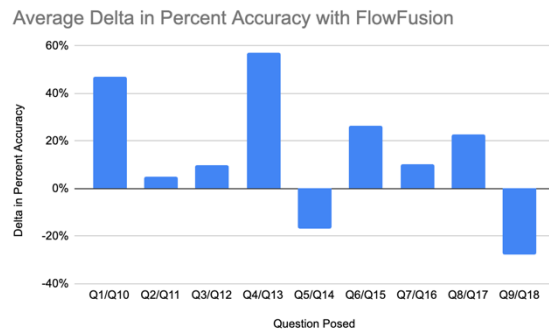


Figure 5. Avg. Δ in Percent Accuracy with FlowFusion

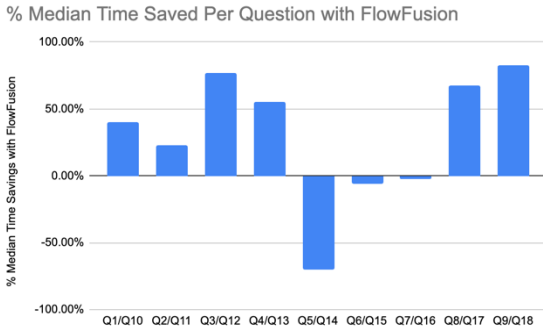


Figure 6. Percent Median Time Saved with FlowFusion

Figure 5 and Figure 6 show that there was significant variance among the results in accuracy and time saved across questions. While there was an overall benefit in both time and accuracy categories, FlowFusion is not entirely suitable for every question. On the questions (i.e., Q5/Q14 and Q9/Q18) that implicitly required analysts to use FlowFusion in conjunction with traditional tooling, rather than in isolation, we observed decreased accuracy or efficiency.

Our experiment demonstrated that FlowFusion can significantly improve the efficiency of CSA by clearly delineating critical signals within a codebase. These signals can be challenging to isolate manually, and by reducing the time required for analysis, FlowFusion can enable more rapid and accurate analyses of complex codebases.

4.3.2. Qualitative User Testing and Results

Post-experiment, we presented a survey to the same group of individuals to collect further assessments of FlowFusion. Overall, the survey results indicated a positive reception. Participants responded that they expected to use our tool for identifying defects, improving code quality, and ensuring compliance with standards. This distribution reflects the versatility of FlowFusion in addressing critical aspects of software analysis.

FlowFusion received high marks on usability, with all respondents rating its ease of use for analysis tasks as a 4 on a 1-5 scale. This suggests that FlowFusion has a user-friendly interface and intuitive features that streamline the software analysis process, allowing users to quickly integrate it into their workflows. Most participants also rated FlowFusion as moderately effective for identifying potential software defects. Additionally, participants provided ratings on accuracy in reporting issues, such as false positives and false negatives. There was no clear consensus on whether the tool would assist in more accurate reporting. We

believe this is due to the small sample size of analysts as well as the small scope of the experiment.

Regarding workflow efficiency, most participants expected the tools to improve their overall efficiency by up to 25%, with one engineer even claiming a 100% efficiency boost. Participants highlighted the time saved on routine tasks and the increased time available to focus on higher-level problem solving as key benefits. This speaks to FlowFusion’s ability to automate tedious aspects of the software analysis process.

Participants also provided constructive feedback on areas for improvement. They stated that enhancing the user interface and reducing occasional false positives would improve the tool’s adoption. Despite the requested improvements, participants unanimously recommended FlowFusion to other professionals.

Lastly, we decided to perform sentiment analysis to better understand participants’ survey responses. We leveraged ChatGPT, a well-regarded sentiment analyzer (Sudirjo, Diantoro, Ahmad, Azzaakiyyah, & Ausat, 2023), to analyze the results of three open-ended questions. Results were measured using sentiment polarity, which ranges from 1 to -1. The more positive the value of sentiment polarity, the more positive the opinion (and vice versa). The polarity value was determined based on the sentiment labeling of text from the model’s training corpus. We prompted the model multiple times to verify the same polarity values were generated each time. Per Table 2, there was consistently positive sentiment for questions 1 and 2 with mixed sentiment in question 3.

Table 2. Sentiment Analysis of Survey Results

Question	Mean Sentiment Polarity	Median Sentiment Polarity
What do you like most about the tools?	0.107	0.1
Would you recommend the tools to other professionals? Why or why not?	0.350	0.390
What are the key features that distinguish the tools from others you have used?	0.094	-0.063

4.3.3. Threats to Validity

Our experiment contains several limitations that impact the validity of its results. First, the small sample size for the experiment could impact the consistency of the findings. Second, analysts might have become more familiar with the tasks after the first stage, thus performing faster in the second stage regardless of the visualization's effectiveness. Third, the exit survey might suffer from biases, as participants' perceptions can be influenced by factors unrelated to the tool's actual performance. Despite these shortcomings, the experiment provides useful data by highlighting the efficiency and accuracy differences when using visualization tools versus manual analysis. In the future, we plan to conduct more rigorous studies to ensure our experimental results are more robust.

5. Future Work

Many opportunities exist to expand upon our research, and we highlight a few here:

- 1. Developing standards for evaluating SA and LLM-assisted software development:** To the best of our knowledge, there are no suitable standard metrics or test data sets for evaluating the efficiency or accuracy of our SA tools or methodology. Future work could explore developing a standard to compare our work to existing techniques.
- 2. Customizing LLM solutions for different programming paradigms:** Our current prompt engineering guidelines are tailored to specific code generation tasks (i.e., plugin-based architecture). Future work could focus on integrating continuous learning mechanisms or more flexible prompt engineering techniques to tune LLMs to generate code that follows other design patterns (e.g., microservices).
- 3. Integrating human-AI teaming:** New research in the field of human-AI teaming suggests we may be able to introduce LLMs into tasks that require human intervention to further accelerate analysis workflows. Future work can investigate best practices for incorporating human-AI teaming into workflows constrained by the unique requirements of classified environments.

6. Conclusion

Our work has developed the following contributions:

1. We provided a methodology for incorporating LLMs into SA tool generation.

2. We used our methodology to demonstrate that LLMs can assist with SA tool generation.
3. We generated novel code visualization capabilities using LLMs.
4. We conducted experiments to empirically verify that our tools and techniques improved efficiency and accuracy.

In particular, the AVASST Plugin Suite and FlowFusion tools we developed represent a step forward in the field of software analysis. Our case studies demonstrate how LLMs can be successfully integrated into software analysis processes to enhance productivity, especially in classified environments, which have unique connectivity and confidentiality constraints. Our work suggests a promising future in which tools developed with the aid of LLMs are integrated into existing software analysis workflows. Though the journey ahead is fraught with challenges and opportunities alike, the potential for transformative change is undeniable.

Acknowledgements

We would like to thank Alan Cohn, Michael Riley, and Daniel Plakosh for their support of this project.

Document Marking

Copyright 2024 Carnegie Mellon University. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. [DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. This work product was created in part using generative AI. DM24-0755

7. References

- Alshahwan, N., Harman, M., Harper, I., Marginean, A., Sengupta, S., & Wang, E. (2024). Assured LLM-Based Software Engineering. *arXiv preprint arXiv:2402.04380*.
- Ami, A. S., Moran, K., Poshyvanyk, D., & Nadkarni, A. (2024, June 18). "False negative - that one is going to kill you": Understanding Industry Perspectives of Static Analysis based Security Testing. *arXiv*.
- Arawjo, I., Swoopes, C., Vaithilingam, P., Wattenberg, M., & Glassman, E. (2023). *ChainForge: A Visual Toolkit for Prompt Engineering and LLM Hypothesis Testing*. Retrieved from arXiv: <https://arxiv.org/abs/2309.09128>

- Borg, M. (2024). Requirements Engineering and Large Language Models: Insights From a Panel. *IEEE Software* 41, no. 2.
- Brown, F. D. (2020). Sys: A Static/Symbolic Tool for Finding Good Bugs in Good (Browser) Code. *USENIX Security*.
- Chess, B., & West, J. (2007). *Secure Programming with Static Analysis*. Pearson Education.
- Desmond, M., & Brachman, M. (2024). *Exploring Prompt Engineering Practices in the Enterprise*. Retrieved from arxiv.org: <https://arxiv.org/html/2403.08950v1>
- Fan, A., Gokkaya, B., Harman, M., Lyubarskiy, M., Sengupta, S., Yoo, S., & Zhang, J. (2023). Large Language Models for Software Engineering: Survey and Open Problems. *IEEE/ACM ICSE-FOSE*, (pp. pp. 31-53). Melbourne, Australia.
- Flynn, L., Snavley, W., Svoboda, D., VanHoudnos, N., Qin, R., Burns, J., . . . Marce-Santurio, G. (2018). Prioritizing Alerts from Multiple Static Analysis Tools, Using Classification Models. *IEEE/ACM SQUADE*, (pp. 13-20).
- GitHub. (2024). *CodeQL*. Retrieved from GitHub: <https://github.com/github/codeql>
- Hadi, M. U., Al Tashi, Q., Qureshi, R., Shah, A., Muneer, A., Irfan, M., . . . Shah, M. (2023). Large Language Models: A Comprehensive Survey of its Applications, Challenges, Limitations, and Future Prospects. *Authorea Preprints*.
- Johnson, B., Song, Y., Murphy-Hill, E., & Bowdidge, R. (2013). Why don't software developers use static analysis tools to find bugs? *ICSE*. IEEE.
- Li, B., Wu, W., Tang, Z., Shi, L., Yang, J., Li, J., . . . Chen, K. (2024). DevBench: A Comprehensive Benchmark for Software Development. *arXiv preprint arXiv:2403.08604*. Retrieved from <https://arxiv.org/abs/2403.08604>
- Li, H., Hao, Y., Yizhuo Zhai, & Zhiyun Qian. (2024, April). Enhancing Static Analysis for Practical Bug Detection: An LLM-Integrated Approach. *ACM*.
- Ling, F., Kim, D. J., & Chen, T.-H. (2024). *When LLM-based Code Generation Meets the Software Development Process*. Retrieved from arXiv: <https://arxiv.org/abs/2403.15852>
- Liu, Z., Tang, Y., Luo, X., Zhou, Y., & Zhang, L. F. (2024). No Need to Lift a Finger Anymore? Assessing the Quality of Code Generation by ChatGPT. *IEEE TSE*.
- Ozkaya, I., Carleton, A., Robert, J., & Schmidt, D. (2023). *Application of Large Language Models (LLMs) in Software Engineering: Overblown Hype or Disruptive Change?* Retrieved from SEI Blog.
- Ridnik, T., Kreda, D., & Friedman, I. (2024). Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering. *arXiv*.
- Rosa, G. (2024). A Study on Prompt Engineering for Software Engineering Data: ChatGPT integration into Software Engineering Metric Generation Tool.
- Sass, J. (2015). *The Role of Static Analysis in Heartbleed*. Retrieved from <https://www.giac.org>: <https://www.giac.org/paper/gsec/36189/role-static-analysis-heartbleed/143117>
- Shankar, S., Zamfirescu-Pereira, J., Hartmann, B., Parameswaran, A. G., & Arawjo, I. (2024, April 18). *Who Validates the Validators? Aligning LLM-Assisted Evaluation of LLM Outputs with Human Preferences*. Retrieved from <https://arxiv.org/https://arxiv.org/pdf/2404.12272>
- Sherman, M. (2024). *Using ChatGPT to Analyze Your Code? Not so Fast*. Retrieved from SEI Blog: <https://insights.sei.cmu.edu/blog/using-chatgpt-to-analyze-your-code-not-so-fast/>
- Simaremare, M., & Edison, H. (2023). AI Assistant to Improve Experimentation in Software Startups Using Large Language Model and Prompt Engineering. *ICSOB-C*.
- Sudirjo, F., Diantoro, K., Ahmad, J. A., Azzaakiyyah, H. K., & Ausat, A. (2023). *Application of ChatGPT in Improving Customer Sentiment Analysis for Businesses*. Retrieved from <https://jurnal.unidha.ac.id/index.php/jteksis/article/view/871>
- Sundberg, L. (2024). Innovating by prompting: How to facilitate innovation in the age of generative AI. *ScienceDirect*.
- Synopsys Editorial Team. (2014, April 13). *On detecting Heartbleed with static analysis*. Retrieved from Synopsys.com: <https://www.synopsys.com/blogs/software-security/detecting-heartbleed-with-static-analysis.html>
- The MISRA Consortium Limited. (2024). *MISRA*. Retrieved from <https://misra.org.uk>
- Thomson, P. (2021, July - August). Static Analysis: An Introduction. *Queue Focus*.
- Vats, V., Nizam, M. B., Liu, M., Wang, Z., Ho, R., Prasad, M. S., . . . Gandamani, D. N. (2024). *A Survey on Human-AI Teaming with Large Pre-Trained Models*. Retrieved from arXiv.org: <https://arxiv.org/html/2403.04931v1>
- Wang, T., Zhou, N., & Chen, Z. (2024). *Enhancing Computer Programming Education with LLMs: A Study on Effective Prompt Engineering for Python Code Generation*. Retrieved from arXiv: <https://arxiv.org/abs/2407.05437>
- Wang, W., Ning, H., Zhang, G., Liu, L., & Wang, Y. (2024). Rocks Coding, Not Development--A Human-Centric, Experimental Evaluation of LLM-Supported SE Tasks. *arXiv preprint arXiv:2402.05650*.
- White, J., Hays, S., Fu, Q., Spencer-Smith, J., & Schmidt, D. C. (2023). *ChatGPT Prompt Patterns for Improving Code Quality, Refactoring, Requirements Elicitation, and Software Design*. Retrieved from <https://www.dre.vanderbilt.edu/~schmidt/PDF/prompt-patterns-book-chapter.pdf>
- Zheng, Z., Ning, K., Chen, J., Wang, Y., Chen, W., Guo, L., & Wang, W. (2023). Towards an Understanding of Large Language Models in Software Engineering Tasks. *arXiv:2308.11396*.