**Category: Computer Science / IT, Engineering**

CST: A Tool for Optimizing the Efficiency and Effectiveness of Static-Code Analysis Tools

Madeleine Weber1, Cecil H. O’Dell2

*1West Virginia University, Statler College of Engineering and Mineral Resources, 306 Evansdale Dr, Morgantown, WV 26506, United States*

*2Katherine Johnson IV&V Facility, 5000 NASA Blvd., Fairmont, WV 26554, Suite 2200, B2, Office C347*

**Abstract:** Static Code Analysis (SCA) is a vital component of NASA IV&V’s mission assurance for safety-critical software as it reduces the likelihood of software-induced hazards impacting mission success. Static Code Analysis achieves this by identifying hazards that may not have been otherwise detectable by typical code reviews or other testing. Using SCA tools, however, can be intimidating due to steep learning curves, especially considering tool performance and defect coverage varies greatly. Because of this variation amongst SCA tools, understanding which tools support certain defects and which do not, as well as understanding how to run an analysis based on steps that are unique to each tool, can be difficult to both new and experienced analysts alike. To mitigate this, the SCAWG or the IV&V Static Code Analysis Working Group, created the SCA Checker Taxonomy and Starting Point Profiles. The Checker Selection Tool (CST) incorporates these two SCAWG products into an interactive tool which allows the user to: select organized categories of defects they would like the SCA tools to discover, select default checkers depending on their mission type (e.g. flight), and configure multiple SCA tools at once. C/C++, Java, and Python defect checkers from four common SCA tools were utilized in this iteration of the CST. This iteration also includes the addition of training, SCA tool specific help, and taxonomy guide links, into its design to help users new to Static Code Analysis learn how to perform SCA more efficiently. The CST has been subject to beta testing by experienced static code analysts from the SCAWG to ensure a usable and accurate final product. The implications of the CST in the mission assurance of NASA safety-critical software are profound, as the CST can help identify and reduce false positives and false negatives, fundamentally improving overall SCA efficiency and accuracy.

**Introduction:** The majority of published static code analysis research focuses on the tools themselves, their accuracy, use cases, etc., assuming the user is already experienced in Static Code Analysis (SCA). The implications of the challenges that can arise from using SCA Tools for the first time deserve to be explored and mitigated. The Checker Selection Tool presented in this paper attempts to improve the accessibility of Static Code Analysis while fulfilling an existing need within SCA to organize the large volume of checkers that exists across many different SCA tools. The Checker Selection Tool imports checkers from several SCA tools, allows users to enable or disable these checkers via an easily consumable user interface and then outputs the checkers in the form of configuration files that can be uploaded to a SCA tool for fast and simple customization.

**Methods:** The Checker Selection Tool was implemented using the tkinter package,

Python’s standard interface to the Tk GUI toolkit and openpyxl, a Python library to read/write Excel files. Using tkinter and openpyxl, the tool takes the checkers organized in categories from the SCA Checker Taxonomy (Figure 1) and adds them to the interface.

**Text

Description automatically generated**

*Figure 1: SCA Checker Taxonomy with the lowest level category designation in blue and the top-level category name in green.*

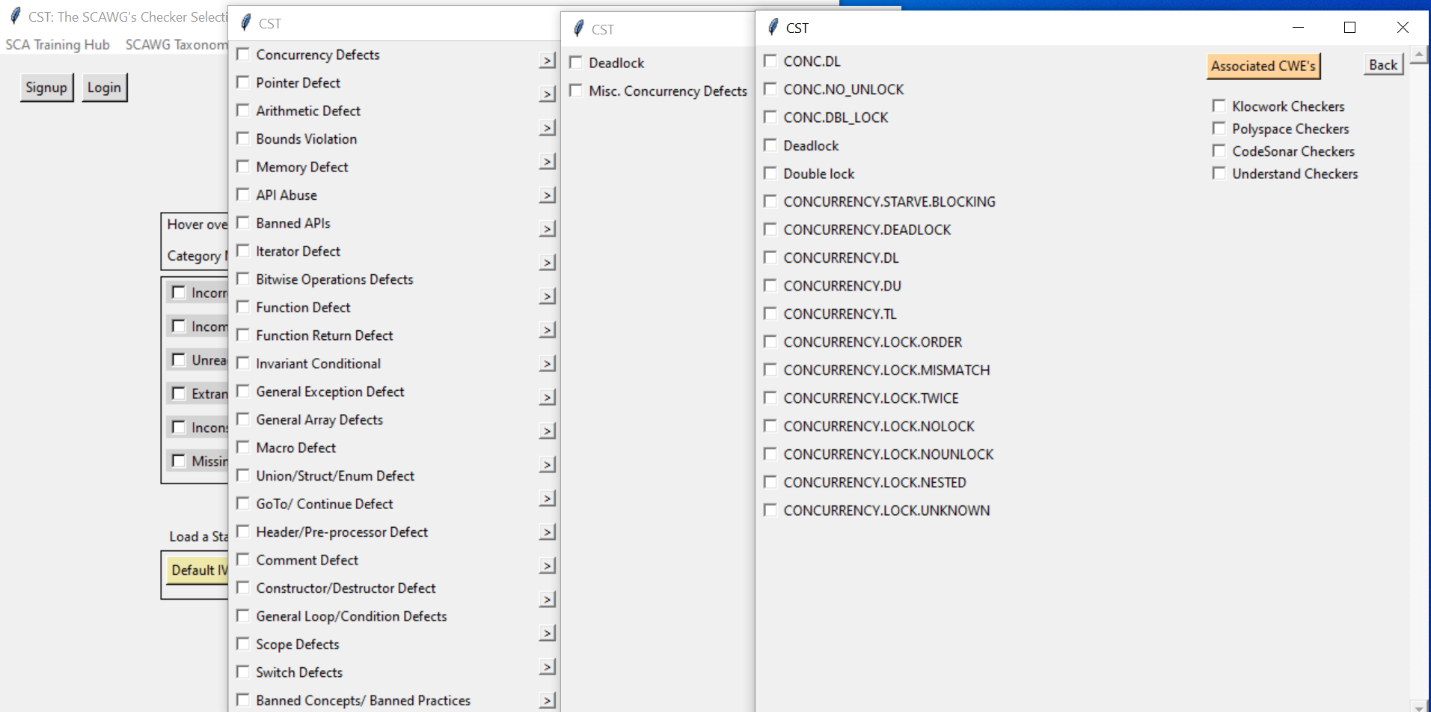
In coordination with SCAWG, the SCA Checker Taxonomy was created by examining each checker in each Static Code Analysis software documentation, examining each checker from that software, and grouping those checkers into a larger category, creating cascading levels of categories beginning with the broadest grouping, and ending with the most specific. The Starting Point Profiles (SPP) were implemented in a similar way, however, each SPP consists of a list of checkers to be selected for each mission type. The CST then goes through this list enabling the identified checkers. Login and Signup buttons allow users to create and access an account where they can save configurations created in the tool for later use without having to export to a configuration file. To maximize the Checker Selection Tool’s applicability, a “Switch to Java Version” button on the home page of the tool (Figure 2), enables users to switch to a version of the CST that supports Java defect checkers and that outputs to SCA tools with Java support.

**Graphical user interface, text, application

Description automatically generated**

*Figure 2: Checker Selection Tool (CST) home page.*

In an effort to integrate “standard” taxonomies into the tool, each child checker category was mapped to its corresponding weakness(es) found in the Common Weakness Enumeration (CWE) taxonomy. This mapping is shown under the “Associated CWE’s” button on each child category’s checker page shown in Figure 3.

*Figure 3: All the checker categories in the CST with the broadest on the left, ending with the individual checkers on the right.*

At the top of the Checker Selection Tool, links to training and documentation were added. The training links were created in conjunction with SCAWG and the documentation links direct the user to specific SCA tool information. The Training Hub consists of tool instructions, videos, and short quizzes. The SCAWG Taxonomy Guide is shown in Figure 4.

*Graphical user interface, application, Teams

Description automatically generated*

*Figure 4: Training link and corresponding training page.*

This link leads to a page explaining the categorical breakdown of the taxonomy as well as the Starting Point Profiles and CWE assignments. To further improve accessibility, a SCA tool quiz (Figure 5) was created and added to the Training Hub to help users choose the optimal Static Code Analysis Tool for their project based on rigor, programming language, desired CWE coverage, and buildability of the project code.

*Graphical user interface, application

Description automatically generated*

*Figure 5: Static Code Analysis Tool quiz.*

**Discussion and Results:** A literature review suggests that the Checker Selection Tool is a unique attempt to reduce the difficulties that can arise from using Static Code Analysis Tools for the first time. These difficulties include the lack of standardization among tools regarding how to perform analyses, checker/defect coverage differences, and variations in tool performance. They are often exacerbated by a lack of documentation and training resources. The paper “Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter?” gets close to examining these difficulties, but instead focuses on comparing software developers prior experience and determining if more experience correlates to less false positives in post-run analyses. Not only does this tool attempt to improve an area of Static Code Analysis that can intimidate potential users, but it also brings attention to an area not often explored in Static Code Analysis research. “A Survey of Static Code Analysis Methods for Security Vulnerabilities Detection” explores Static Code Analysis applications and algorithms, but notably, when discussing future improvements, mentions, “A step forward in improving user experience is a well-structured and intuitive UI that groups related defects (Kulenovic & Donko, 2014)”. The Checker Selection Tool is this step forward. The Checker Selection Tool helps to reduce the number of SCA results to a manageable number by allowing the user to enable the most important checkers individually, or by allowing the user to select a pre-programmed profile (SPP) that enables the necessary checkers automatically. The CST also consolidates available trainings and documentation into a user-friendly interface, making Static Code Analysis Tools less intimidating and easier to use. Testing of the Checker Selection Tool was performed by members of the IV&V Static Code Analysis Working Group (SCAWG) and the Software Assurance Tools Team (SWAT). Feedback was gathered from email responses to tool trials as well as from verbal responses to tool demonstrations. Testers found that the tool and its checker selection layout and categorization made checker configuration easier, more efficient, and more effective. They also noted that identifying differences in tool coverage was straightforward and, as a result, deciding which Static Code Analysis tool to best serve their project was a lot simpler. The use of multiple SCA tools for one project was made easier as well. Because users can import configuration files with the option to enable all checkers in the categories of the enabled checkers from the imported file, users can easily create similar configuration files for multiple tools at once, standardizing analyses across tools. This standardization, along with use of the Code Analysis Pipeline (CAP), made comparing analysis results easier as well. Along with the overwhelmingly positive feedback received from testers, additions to the tool were also suggested. These suggestions included adding Python supported tool and checker functionality.

**Use Case:** Mars Endeavor is an unmanned rover mission to explore Mars. Due to the nature of this mission, the rover requires some level of autonomy and needs to be working at optimal performance. To ensure that the rover’s software is functioning properly and can handle any number of different scenarios, the code is analyzed using Static Code Analysis. If the code was run through a Static Code Analysis tool with its default checkers selected, the resulting number of errors raised is likely to be so large that resolving each issue is unrealistic and it is unlikely every issue would be reached, leaving errors un-fixed in a variety of categories. However, if the Checker Selection Tool is used and the Flight Starting Point Profile is selected (Default Flight), the most critical issues in the rover’s code will be found succinctly because the Flight SPP enables only the checkers that are crucial to the success of this mission type. After the Checker Selection Tool creates the proper configuration file with the checkers selected from the Flight Starting Point Profile, the config file can be uploaded and used for testing.

**CAP Use Case:** The Mars Perseverance rover is equipped with the first robotic helicopter to operate on Mars, Ingenuity. Due to the 5-to-20-minute radio signal lag between Earth and Mars, Ingenuity must perform its maneuvers autonomously. Thus, it is critical that the helicopter’s software works as expected. To identify possible software hazards, the code is analyzed via Static Code Analysis. If the code was run through a Static Code Analysis tool with its default checkers selected, the resulting number of errors raised is likely to be so large that resolving each issue is unrealistic and it is unlikely every issue would be reached, leaving errors un-fixed in a variety of categories. However, if the Checker Selection Tool is used and the Flight Starting Point Profile is selected (Default Flight), the most critical issues in the rover’s code will be found succinctly because the Flight SPP enables only the checkers that are crucial to the success of this mission type. The Checker Selection Tool can then output the required configuration file for several different Static Code Analysis tools. To compare and perform SCA utilizing multiple tools, the Code Analysis Pipeline (CAP) is used with the CST-generated configuration files to maximize the number of critical hazards identified. CAP is designed to allow the user to run and compare multiple SCA tools in one environment. Using multiple Static Code Analysis Tools is crucial to generating a complete and accurate picture of the potential defects in a piece of software by increasing Recall. Recall is the number of flaws the tool reported divided by the number of actual flaws in the software. Theoretically, Recall can be increased by using multiple SCA tools to find defects undetected by other tools.

**Conclusion:** Static Code Analysis is crucial to NASA IV&V’s software assurance of safety-critical software and thus mission success, adding to the importance of an intermediary tool like the Checker Selection Tool that makes Static Code Analysis more approachable and therefore, more widely implemented. Integration of the CST into NASA’s SCA training will ensure that the Checker Selection Tool is utilized by both new and existing Static Code Analysts. Future work on the Checker Selection Tool is set to include adding a Python version of the CST with Python checker and SCA tool support, as well as ensuring consistent updating of the taxonomy/tool to keep up with regular checker updates in Static Code Analysis tools and to ensure that the tool does not fall into obsolescence.

**References**

D. Baca, K. Petersen, B. Carlsson and L. Lundberg, "Static Code Analysis to Detect Software Security Vulnerabilities - Does Experience Matter?," 2009 International Conference on Availability, Reliability and Security, 2009, pp. 804-810, doi: 10.1109/ARES.2009.163.

M. Kulenovic and D. Donko, "A survey of static code analysis methods for security vulnerabilities detection," *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, Opatija, Croatia, 2014, pp. 1381-1386, doi: 10.1109/MIPRO.2014.6859783.