GSTF Journal on Computing (JoC) Vol.2 No.1, April 2012

Performance Evaluation of Automated Static Analysis Tools

Cathleen L. Blackmon, Daisy F. Sang, and Chang-Shyh Peng

Abstract—Automated static analysis tools can perform efficient thorough checking of important properties of, and extract and summarize critical information about, a source program. This paper evaluates three open-source static analysis tools; Flawfinder, Cppcheck and Yasca. Each tool is analyzed with regards to usability, IDE integration, performance, and accuracy. Special emphasis is placed on the integration of these tools into the development environment to enable analysis during all phases of development as well as to enable extension of rules and other improvements within the tools. It is shown that Flawfinder be the easiest to modify and extend, Cppcheck be inviting to novices, and Yasca be the most accurate and versatile.

Index Terms*—*static, code, analysis, automation.

I. INTRODUCTION

THE demand for reliable, quality software has grown in all areas, from consumer and business applications to mission-critical commercial, industrial and governmental applications. It is however not feasible to exhaustively test all possible executions and to remove all potential risks from large complex software product. However, the use of automated software analysis tools has enabled organizations to produce products that are as defect free as practically possible. Automated analysis, while not a replacement for human effort, is a substantial aid to developers particularly where software quality is of primary importance [13]. Automated software analysis tools can perform efficient and thorough checking of various properties, and can extract and summarize critical information of the source program.

There are two main categories of automated software analysis tools: dynamic and static. Dynamic analysis is performed at runtime on the executable images of the software. Tests are conducted on specific behaviors, such as memory corruption, memory leaks and race conditions, of software during execution. Since defects will not be discovered until late in the software development lifecycle, dynamic analysis can be costly. On the other hand, static analysis tools perform analysis on source code or byte code modules. It does not require any instrumentation or development of test cases, and can be utilized upon the availability of the code. Static analysis tools can go through all paths of the code and uncover significantly more and wider range of defects, including detect logic errors, dead code, security vulnerabilities, and so on.

Static analysis techniques range widely. Simple style checkers identify poorly written code that may violate coding standards or consistency rules. Bug pattern checkers search for common error patterns not caught by the compiler, such as memory leaks and out of bounds errors. Dataflow and control flow analysis techniques, which can apply intra-procedurally or inter-procedurally, use annotations to reduce the occurrence of false positives. Model checkers test whether the software meets specification. Formal methods apply mathematical techniques to perform in-depth analysis for more accurate results. Other techniques, such as data mining, have also been successful in implementing static analysis.

Static analysis tools each deploys selected technique and exhibits unique features. FindBugs [8], [9], [10] and XFindBugs [16] implement bug pattern matching. They perform effective analysis and keep the false positive rate low. They offer an intuitive user interface and friendly reporting mechanism. However, FindBugs and XFindBugs can detect only limited types of software defects and has to make trade-offs in order to achieve low false positive or false negative rates. ESC/Java [6] uses modular checking with the help of annotated code, and provides more formal theorem-proving techniques. While it aims to reach some middle ground of cost vs. usability, ESC/Java requires developers to set up annotations for each routine. DSD-Crasher [5] adopts a dynamic-static-dynamic hybrid approach. In the first step, dynamic inference, DSD-Crasher captures the execution behavior and detects program invariants dynamically. Secondly, a static analysis is performed to exhaustively analyze the program paths within the restricted input domain. Lastly,

---

Manuscript received February 29, 2012.

Cathleen F. Blackmon was with Cal Poly Pomona, Pomona, CA 91768 USA,

Daisy F. Sang is with Cal Poly Pomona, Pomona, CA 91768 USA, (e-mail:fcsang@csupomona.edu).

Chang-Shyh Peng is with California Lutheran University, Thousand Oaks, CA, 91360, USA (phone: 805-493-3819; fax: 440-296-8002; e-mail:peng@callutheran.edu).

DOI: 10.5176_2010-2283_2.1.153

in dynamic verification, test cases are automatically generated to test the results of the static analysis. DSD-Crasher inherits the limitation of dynamic analysis, and is limited by the paths designated in the applied test cases. CP-Miner [12] implements data mining to find replicated code in large software suites. Due to its significant overhead in the implementation of data mining techniques, CP-Miner is more suitable for large developments. Abstract interpretation [2], [3] is a mathematics-based formal method. It is commonly used for mission-critical embedded systems in avionics, aerospace, railway and automotive industries. Abstract interpretation techniques have been applied to memory usage analysis, timing analysis, bug finding, inter- and intra-procedural analyses including control flow and data flow analysis, stack analysis, and more [2], [3], [4], [7], [11], [14], [17], [18]. Earlier research showed evaluations for avionics industry [1] and for detecting buffer overflow vulnerabilities [19].

In this paper, three relatively new and open source static analysis tools are studies. Section 2 introduces these three tools and test platforms. In section 3, test cases and results are presented. Section 4 concludes with future work.

II. STATIC ANALYSIS TOOLS AND PLATFORM

Among all available open source static analysis tools for the C/C++ language, three were identified based upon their ease of use, report method, result interpretation, installation, user interface, extensibility, and IDE integration. They are Flawfinder, Cppcheck, and Yasca.

Flawfinder [25] is an open-source static analysis tool. It was developed in Python, and designed to detect security vulnerabilities in C and C++ source code. Flawfinder was written primarily in Python, requires the installation of both Python and Cygwin, and can run in Linux/Unix and Windows. Flawfinder is a command line tool that is simple and intuitive to use. Flawfinder rates potential security flaws, called *hits*, from level 0 (very little risk) to level 5 (high risk). By default the program reports only flaws with a minimum risk level of 1, but the user has the option of selecting the minimum level. There are various filtering options.

The output of the analysis is limited to HTML or plain text file formats. Reports provide mostly standard information, including filename, line numbers and types of the flaws, and remediation advice.

Cppcheck [23] is a standalone, and open source, static analyzer. Cppcheck was written in C++ and can analyze C/C++ code for common defects such as memory leaks. Cppcheck is designed to promise a low false positives rates. Its setup in Windows is straightforward. Cppcheck has four levels of severity: error, possible error, style, possible style. By default, only errors are reported. Cppcheck provides both the command line and GUI usages. The command line usage can specify warning levels, output format/template, etc. The GUI usage supports seven languages. While the GUI usage is friendlier, the command line usage is more versatile.

Yasca is a command line open source static analysis tool designed to assist in quality assurance testing and vulnerability scanning. It was developed with Java and PHP. This tool is an aggregation of the Yasca core software and various open source tools embedded in Yasca. It includes plug-ins for Antic, ClamAV, Grep, Jlint, Javascriptlint, Fxcop, Findbugs, Findbugs-plugin, Grep, Rats, PMD, Pixy, Phplint, Cppcheck, Clamav. Since this paper focuses on C/C++ source code, only the Antic, ClamAV, Grep, Rats and Cppcheck plug-ins were enabled. The Yasca core itself is not meant to be modified except as an official release; however the plug-ins can be modified as needed. Yasca is fairly intuitive to use. Yasca does not offer as many options as Flawfinder or Cppcheck, but does provide flexible output formats including that of MySQL.

These three tools are to be evaluated on a set of carefully selected test cases, which are embedded with various classes of flaws. The primary IDE in this project is Eclipse [24]. It is chosen for its versatility, ease of use, and wide acceptance in industry and education environments. Eclipse has the native support of Java, and can support other languages with corresponding plug-ins. Other supporting utilities include version control tool Subversion [20], [21], [26], [27] and Visual C++ IDE.

III. TEST CASES AND EVALUATIONS

Source code analyzed in this project is freely available online from the National Institute of Standards and Technology (NIST) Software Assurance Metrics And Tool Evaluation (SAMATE) Project [15]. This paper used SAMATE Test Suites 9, 45, 46, 47, 57, 58, and 59, with a total of 225 C/C++ source code files (test cases). Many of these test cases include both a *bad* version (with flaws or weaknesses) and a *good* version (with flaws or weaknesses removed). These 225 cases represent twenty-three Common Weakness Enumeration (CWE) flaw classes [22].

Within this set of test cases, virtually all could be classified as potential security vulnerabilities under various circumstances. 137 cases represent high or very high security risks. There are 32 code injection vulnerabilities (command, XSS, SQL, and resource injection). Injection vulnerabilities can allow attackers to compromise the system. There are 20 buffer overflow (heap and stack) vulnerabilities. Buffer overflows

TABLE I
TEST CASES AND RISKS

Weakness	Test Cases	Bad	Good	Description	Risks
CWE-078	18	10	8	Command Injection (OS)	High security risk. Malicious attack - read/modify data, execute commands.
CWE-079	16	8	8	Cross-site Scripting (XSS) Injection (Web-based)	Very high security risk. Malicious attack -inject malicious script execution.
CWE-089	13	6	7	SQL Injection (DB Server)	Very high security risk. Malicious attack - read/modify/delete sensitive data including username/password
CWE-099	16	8	8	Resource Injection (System)	High security risk. Malicious attack - modify/access protected system resources.
CWE-121	21	11	10	Buffer Overflow (Stack)	High security risk. Malicious attack - execute code/subvert security, System can crash or hang, Data corruption.
CWE-122	19	9	10	Buffer Overflow (Heap)	High security risk. Malicious attack - execute code/subvert security, System can crash or hang, Data corruption.
CWE-134	10	5	5	Uncontrolled Format String	Very high security risk. Malicious attack - execute arbitrary code/access confidential information, Buffer overflow risk, System can crash or hang. Incorrect data representation error.
CWE-170	10	5	5	Improper Null Termination	Security risk. Malicious attack - Disclosure of sensitive information/execute arbitrary code, Overflow risk due to off-by-one errors, Write-what-where condition, System crash, Segmentation fault crash, Corrupted data.
CWE-244	1	1	0	Heap Inspection	Security risk. Malicious attack - Sensitive information not removed from heap could be read by attacker.
CWE-251	10	5	5	Misused String Manipulation	Potential buffer overflow condition leading to security risks, system crashes, data corruption, etc.
CWE-259	19	10	9	Use of Hard-coded Password	High security risk. Malicous attack - Attacker given access to account.
CWE-362	4	2	2	Race Condition	Possible security risk - if in security-critical mode. System crash or hang.
CWE-367	4	4	0	Time-of-check Time-of-use (TOCTOU) Race Condition	Possible security risk - if in security-critical mode. System crash or hang.
CWE-391	4	2	2	Unchecked Error Condition	Security risk - Unexplained behavior hard to attribute to an attack, Hard to diagnose unexpected program behavior.
CWE-401	11	4	7	Memory Leak	Possible security risk if attacker triggers a memory leak causing denial-of-service attack, Memory not released after last use - program can crash or hang when memory is too low, Data corruption or loss of data.
CWE-411	2	1	1	Resource Locking	Possible security risk, Inability to control access to resources.
CWE-412	2	2	0	Unrestricted Externally Accessible Lock	Possible security risk if attacker gains control of lock, Denial-of-service.
CWE-415	10	6	4	Double Free	Security risk. Malicious attack - execute arbitrary code, Write-what-where condition, Corrupted data, data loss.
CWE-416	10	6	4	Use After Free	Security risk. Malicious attack - execute arbitrary code. Write-what-where condition. Invalid or corrupt data.
CWE-457	5	3	2	Use of Uninitialized Variable	High security risk - can contain previously-used memory. Unpredictable or unintended system behavior, Possible data loss.
CWE-468	4	2	2	Incorrect Pointer Scaling	Security risk. Potential for buffer overflow, Corrupt data or data loss, System may crash or hang.
CWE-476	15	7	8	NULL Pointer Dereference	Medium security risk - if combined with other flaws. Failure of software - crash or exit, Invalid data, possible data loss.
CWE-489	2	1	1	Leftover Debug Code	Security risk - sensitive information may be accessed by attacker.

represent a serious threat to system stability and security, and have been the target of a multitude of attacks in recent years. There are 5 cases of uncontrolled format strings. Uncontrolled format strings can cause buffer overflow in some instances. There are 11 test cases that can cause memory leaks. 125 cases can lead to invalid or corrupted data, or data loss (without a malicious attack present). Nearly all weaknesses can cause the system to crash or hang. Quality issues like data errors and memory leaks can lead to system freeze up or crash. Also of great concerns are errors that cause the program to display erroneous information. In life-safety situations, such as medical, aviation, and automotive fields, these errors can

TABLE II
DETECTION RATE

Tool	True Positives	False Negatives	False Positives	True Negatives
CPPCHECK	32%	68%	16%	84%
FLAWFINDER	68% (54%)	32% (46%)	60% (46%)	40% (54%)
YASCA	73%	27%	67%	33%

TABLE III
DETECTION ACCURACY

SAMATE ERROR CODE	FLAW	CPPCHECK	FLAWFINDER	YASCA
CWE-078	OS Command Injection	Medium	High	High
CWE-079	Cross-site scripting XSS	Low	Medium	Medium
CWE-089	SQL Injection	None	Medium	Medium
CWE-099	Resource Injection	None	High	High
CWE-121	Buffer Overflow (Stack)	Low	High	High
CWE-122	Buffer Overflow (Heap)	Medium	Medium	High
CWE-134	Uncontrolled Format String	None	High	High
CWE-170	Improper Null Termination	None	High	High
CWE-244	Heap Inspection	None	None	High
CWE-251	String Management	None	High	High
CWE-259	Hard-Coded Password	None	None	Low
CWE-362	Race Condition	None	Medium	Medium
CWE-367	TOUTOU Race Condition	None	High	Medium
CWE-391	Unchecked Error Condition	None	Medium	High
CWE-401	Memory Leak	Low	Low	Medium
CWE-411	Resource Locking	None	High	None
CWE-412	Unrestricted Lock on Critical Resource	None	High	None
CWE-415	Double Free	Low	Low	Low
CWE-416	Use After Free	None	Medium	High
CWE-457	Use of Uninitialized Variable	None	None	Low
CWE-468	Incorrect Pointer Scaling	None	None	None
CWE-476	Null Pointer Dereference	Low	Low	Low
CWE-489	Leftover Debug Code	None	None	High

have serious consequences. Table I summarizes the test cases and corresponding risks.

Flawfinder, Cppcheck, and Yasca are evaluated on detection rate and detection accuracy, and are benchmarked by the SAMATE Flaw Classification Schema. Detection rate measures the ability to accurately identify the weaknesses in the source code. There are 4 categories: true positive, false positive, true negative, and false negative. True positive is when true errors/flaws were detected and reported correctly. False positive is when errors were reported when there were actually none. True negative is when no errors were found, because the source was in fact error/flaw-free. False negative is when existing flaws were not detected. Detection accuracy indicates the ability to detect the error correctly according to the following criteria: high, medium, low, and none. High accuracy is when a tool could detect flaws correctly in more than 70% of the cases. Medium accuracy is when a tool could detect flaws correctly in 40% to 69% of the cases. Low accuracy is when a tool could detect flaws correctly in less than 40% of the cases. None accuracy is when a tool was not able to detect any existing flaws at all. Results are as follows.

Flawfinder reported a total of 156 flaws in 79 of the 117 bad source files, but was unable to detect flaws in the remaining 38 bad files. Flawfinder correctly detected 68% of files with true errors, but also had a high false positive rate of 60%. The tool detected a total of 92 (with 42 of level 2 and higher) flaws in 50 of the 108 good files. At the default setting of security flaw level 2 Flawfinder was able to detect flaws in all but five flaw classes; including memory leaks and buffer overflows. When the security flaw level was set to 1, Flawfinder could detect all flaw classes except CWE-457 and CWE-468.

Due to its design to reduce the false positive rate, Cppchecker missed many true flaws in the test cases. Cppcheck failed to detect any flaws in 16 of the 23 flaw classes. Among the ones being detected, Cppcheck reported flaws in 32% of the bad source files with a false positive rate of 16%. Cppcheck detected 25% of test cases with memory leak flaws. And Cppcheck caught 44% of the heap overflow cases, 18% of the stack overflow cases, 40% of the OS command injection cases, and 29% of the null pointer dereference cases.

Yasca reported a total of 270 flaws in 117 bad test files and another 212 flaws in the 108 good test files. Yasca reported 73% of all known flaws; the highest of the three tools. The false positive rate was high as well at 67%. Of the true flaws that were detected, the RATS plug-in detected 113 flaws in 70 test cases. The Antic and Yasca plug-in detected a dozen flaws between them. The GREP plug-in detected 98 flaws in 51 test cases. These plug-ins worked nicely together and accomplished more detections than they would have separately. Yasca accurately detected 100% of the known flaws in nine of the flaw classes, and was very effective in the recognition of buffer overflows, string errors, and command injection vulnerabilities.

IV. SUMMARY

Table II summarizes the detection rates among Flawfinder, Cppcheck, and Yasca. And Table III presents the detection accuracy among these three tools. Flawfinder was shown to be the easiest to modify and extend. Cppcheck provided an easy-to-use GUI interface that may be attractive to novices. Yasca provided the most accurate results, and its hyperlinked HTML report was the most useful and versatile. Open source tools are experimental in nature. If used early in the software development process, these tools can catch common errors and offer suggestions for improvement. For small and/or educational developments, these tools can be particularly valuable.

V. REFERENCES

[1] Bartholomew, R., Evaluation of Static Source Code Analyzers for Avionics Software Development. *Ada Letters*. Volume XXVIII, Issue 1. pp 83-87. April 2008.

[2] Cousot, P., Cousot, R. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. *POPL '77: Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of ProgrammingLanguages*. Los Angeles, CA, January 1977. pp. 238-252. ACM Press.

[3] Cousot, P., Cousot, R. Systematic design of program analysis frameworks. *POPL '79: Proceedings of the 6th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. San Antonio, TX, January 1979. pp. 269-282. ACM Press.

[4] Cousot, P., Cousot, R., Feret, J., Mine, A., Mauborgne, L., Monniaux, D., Rival, X. Varieties of Static Analyzers: A Comparison with ASTREE. *TASE '07: First Joint IEEE/IFIP Symposium on Theoretical Aspects of Software Engineering*. 2007. IEEE.

[5] Csallner , C., Smaragdakis, Y. DSD-Crasher: a hybrid analysis tool for bug finding. *Proceedings of the 2006 international symposium on Software testing and analysis*. July 17-20, 2006. Portland, Maine, USA.

[6] Flanagan, C., Rustan, K., Leino, M., Lillibridge, M., Nelson, G., Saxe, J., Stata, R. Extended static checking for Java, *Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, June 17-19, 2002, Berlin, Germany.

[7] Heckmann, R., Ferdinand, C. Verifying Safety-Critical Timing and Memory-Usage Properties of Embedded Software by Abstract Interpretation. *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe - Volume 1*. IEEE Computer Society. pp 618-619. March 2005

[8] Hovemeyer, D., Pugh, W. Finding bugs is easy. *ACM SIGPLAN Notices. Volume 39 Issue 12*. pp 92-106. December 2004.

[9] Hovemeyer,D., Pugh, W. Finding more null pointer bugs, but not too many. ACM PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering. June 2007.

[10] Hovemeyer, D., Spacco, J., Pugh, W. Evaluating and tuning a static analysis to find null pointer bugs. *PASTE '05: Proceedings of the 6th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. January 2006.

[11] Kaestner, D. Safe worst-case execution time analysis by abstract interpretation of executable code. *LCTES '07: Proceedings of the 2007 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*. p 135. San Diego, CA USA. July 2007.

[12] Li, Z., Lu,S., Myagmar, S., Zhou, Y., CP-Miner: Finding Copy-Paste and Related Bugs in Large-Scale Software Code. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 32, NO.* Louridas, P., Static Code Analysis. IEEE Software, pages 58-61. July/August 2006. IEEE Computer Society.

[13] Pezze, M., Young, M., *Software Testing and Analysis: Process, Principles, and Techniques*. John Wiley & Sons, Inc. 2008. ISBN-13: 978-0-471-45593-6

[14] Regehr, J., Reid, A., Webb, K. Eliminating stack overflow by abstract interpretation. *ACM Transactions on Embedded Computing Systems (TECS)*. Volume 4 Issue 4. pp 751-778. November 2005.

[15] SAMATE Reference Dataset, *National Institute of Standards and Technology*, http://samate.nist.gov/SRD/

[16] Shen, J., Zhang, S., Zhao, J., Fang, J., Yao, S. XFindBugs: eXtended FindBugs for AspectJ. *PASTE '08: Proceedings of the 9th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. November 9-10. Atlanta, Georgia USA. 2008.

[17] Thesing, S. Modeling a system controller for timing analysis. *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*. pp 292-300. October 2006.

[18] Wei-Ngan Chin, Huu Hai Nguyen, Corneliu Popeea, Shengchao Qin. Analysing memory resource bounds for low-level programs. *ISMM '08: Proceedings of the 7th international symposium on Memory management.* pp 151-160. June 2008.
[19] Zitser, M., Lippmann, R., Leek, T., Testing Static Analysis Tools Using Exploitable Buffer Overflows from Open Source Code. *SIGSOFT'04/FSE12, Oct. 31–Nov. 6, 2004,* Newport Beach, CA, USA. 2004. ACM 1581138555/04/0010.
[20] AnkhSVN. http://ankhsvn.open.collab.net/
[21] Apache Subversion. http://subversion.apache.org/
[22] Common Weakness Enumeration (CWE). http://cwe.mitre.org/index.html
[23] Cppcheck Project. http://sourceforge.net/projects/cppcheck/
[24] Eclipse IDE. http://www.eclipse.org/downloads/
[25] Flawfinder Project. http://sourceforge.net/projects/flawfinder/
[26] Subclipse. http://subclipse.tigris.org/
[27] VisualSVN Server Standard. http://www.visualsvn.com/server/download/

Cathleen L. Blackmon is a Software Engineer at Rockwell Collins in Cedar Rapids, Iowa. She received B.S. and M.S. degrees in Computer Science from California State Polytechnic University

Dr. Daisy F. Sang is a professor in the Department of Computer Science at California State Polytechnic University. She received both M.S. and Ph.D. degrees from the University of Texas at Dallas. Her research interests include software testing, web services discovery and composition, graph algorithms, and fault tolerant distributed computing. She has supervised many graduate and undergraduate research projects, funded by NASA Partnership Awards for Integration Research, NSF Advance, Women's Educational Equity Act, Lockheed Martin-CSU Partnership program, and local industry such as Amptron Inc, GTE, and Southern California Edison.

Dr. Chang-Shyh Peng is the Professor and Chair of the Computer Science Department at California Lutheran University. He received B.S. degree from National Taiwan University, and the M.S. and Ph.D. degrees from the University of Texas at Dallas. His research interests include data communication and networking, parallel processing, client/server development, multimedia, and software development. He has published various articles in network simulations and modeling, network applications, and design and analysis of algorithms.