Software Engineering Institute

Improving the Automated Detection and Analysis of Secure Coding Violations

Daniel Plakosh
Robert Seacord
Robert Stoddard
David Svoboda
David Zubrow

June 2014

TECHNICAL NOTE
CMU/SEI-2014-TN-008

CERT® Division

http://www.sei.cmu.edu

Carnegie Mellon University

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

This report was prepared for the
SEI Administrative Agent
AFLCMC/PZM
20 Schilling Circle, Bldg 1305, 3rd floor
Hanscom AFB, MA 01731-2125

DM-0001389

Table of Contents

List of Figures

List of Tables

Abstract

Coding errors cause the majority of software vulnerabilities. For example, 64% of the nearly 2,500 vulnerabilities in the National Vulnerability Database in 2004 were caused by programming errors. The CERT Division's Source Code Analysis Laboratory (SCALe) offers conformance testing of C language software systems against the CERT C Secure Coding Standard and the CERT Oracle Secure Coding Standard for Java, using various analysis tools available from commercial software vendors. Unfortunately, the current SCALe analysis process and tools do not collect any statistics about the accuracy of the code analysis tools or about the coding violations they flag, such as frequency of occurrence. This paper describes the approach used to add the ability to collect and statistically analyze data regarding coding violations and tool characteristics along with the initial results. The collected data will be used over time to improve the effectiveness of the SCALe analysis.

1 Background

The Source Code Analysis Laboratory (SCALe) is a proof-of-concept demonstration that software systems can be conformance tested against secure coding standards. It provides a consistent measure that can be used to assess the security of deployed software systems, specifically by determining if they are free of coding errors that lead to known vulnerabilities. Error-free code in turn reduces the risk to these systems presented by increasingly sophisticated hacker tools. However, this type of conformance testing has the following problems that impact its widespread use:

1. The effectiveness of analysis tools and coding rules are not well understood.
2. Code analysis adds a large amount of time to the development schedule.
3. Code analysis is error prone because a human must validate tool results.

It is anticipated that gaining a better understanding of the effectiveness of the analysis tools will reduce the amount of time a human must spend validating results and therefore will reduce the amount of time conformance testing contributes to the development schedule.

1.1 Software Security

Software vulnerability and exploitation reports continue to grow at an alarming rate, and a significant number of these reports result in technical security alerts. To address this growing threat to the government, corporations, educational institutions, and individuals, systems must be developed that are free of software vulnerabilities.

The CERT® Division of Carnegie Mellon University's Software Engineering Institute takes a comprehensive approach to identifying and eliminating software vulnerabilities and other flaws. The CERT Division produces books and courses that foster a security mindset in developers, and it develops secure coding standards and automated analysis tools to help developers code securely. Secure coding standards provide a detailed enumeration of coding errors that have caused vulnerabilities, along with their mitigations for the most commonly used software development languages. The CERT Division also works with vendors and researchers to develop analyzers that can detect violations of the secure coding standards.

Improving software security by implementing code that conforms to the CERT secure coding standards can be a significant investment for a software developer, particularly when refactoring or otherwise modernizing existing software systems [Seacord 2008]. However, a software developer does not always benefit from this investment because it is not easy to market code quality.

To address these problems, the CERT Division has created SCALe, which offers testing to determine conformance of software systems to CERT secure coding standards.

® CERT is a registered mark owned by Carnegie Mellon University.

1.2 SCALe

SCALe evaluates client source code using multiple static analysis (SA) tools. The CERT Division reports any deviations from secure coding standards to the client. The client may then repair and resubmit the software for reevaluation. Once the reevaluation process is completed, the CERT Division provides the client a report detailing the software's conformance or nonconformance to each secure coding rule. The SCALe process consists of the following sequence of steps:

1. **Clients contact CERT Division.** The process is initiated when a client contacts the CERT Division with a request to evaluate a software system.
2. **CERT Division communicates the requirements.** The CERT Division communicates requirements to the client including (1) selection of secure coding standard(s) to be used, (2) a buildable version of the software to be evaluated, and (3) a build engineer.
3. **Client provides buildable software.** The client selects standards(s), provides a buildable version of the software to be evaluated, and notifies the build engineer, who is available to respond to build questions about the system.
4. **CERT Division selects tool set.** The CERT Division chooses and documents the tool set to be used and procedures for using that tool set in the evaluation of the system.
5. **CERT Division analyzes source code and generates a conformance test report.** The CERT Division evaluates the system against specified standard(s) and provides the conformance test results to the client. If the system if found to be conforming, the CERT Division issues a certificate and terminates the conformance testing process.
6. **Client repairs software.** The client has the opportunity to repair nonconforming code. The client sends the system back to the CERT Division for final evaluation.
7. **CERT Division issues conformance test results and certificate.** The CERT Division reevaluates the system using the tools and procedures used in the initial assessment. The CERT Division provides conformance test results to the client and, if the system is found to be conforming, a certificate.

Unfortunately, the current SCALe analysis process and tools do not collect any statistics about the accuracy of the code analysis tools or about the coding violations they flag, such as frequency of occurrence. The ability to collect and statistically analyze data regarding coding violations and tool characteristics should improve the effectiveness of SCALe analysis.

This paper describes how the CERT Division instrumented the SCALe process to provide insight into (1) the effectiveness of SA as a means of exposing vulnerabilities and (2) the performance of the SCALe process's analyzers. The paper also presents our initial results.

2 Approach

Our approach was as follows:

1. Modify SCALe workflow, infrastructure, and database(s) to support data collection and instrumentation, such as the measurement of true/false positives, the ability to detect false negatives through use of multiple analyzers, and measurement of violation occurrences based on programming language.
2. Collect data from the re-analysis of source code previously analyzed in SCALe and from ongoing source code evaluations.
3. Develop sampling space/characterization of codebase by identifying salient dimensions of the sampling space and then defining measures representing the dimensions of the sampling space.
4. Conduct analyses to address the research questions (described in Section 2.1).
5. Use results collected over time to improve the SCALe infrastructure, process, and tools.

2.1 Research Questions

1. Which analyzers are best at detecting specific coding rule violations?
2. How common are violations of secure coding rules in systems developed under various code cultures? (*Example code cultures: device driver, desktop application, weapons system, smartphone app*)
3. How effective is SA as a means of exposing vulnerabilities?
4. Can the pattern of results across rules tell us what is likely to be a true or false positive?

2.2 Methodology

2.2.1 Use of the SCALe Process

The process for analyzing any codebase using SCALe is fairly simple, as outlined in Figure 1. First, each SA tool (appropriate for the language) is run on the codebase, producing a set of diagnostics (also known as *flagged nonconformities*) on the codebase. These diagnostics may indicate true violations of the CERT secure coding rules, or they may be false positives. Each tool's diagnostics are then merged into one complete list of diagnostics.

The next task for an auditor is to determine which diagnostics in the list are true positives and which are false positives. This task can be daunting. Because some tools emphasize eliminating false negatives and so produce a large number of false positives, a codebase can sometimes have more than 10,000 associated diagnostics.

To mitigate this challenge, we classify all the diagnostics into *buckets*, where each bucket represents all the diagnostics associated with a particular CERT secure coding rule. Thus, there will be exactly as many buckets as there are CERT rules being violated (according to the SA tools). The auditor executes the following procedure for each bucket:

1. Identify a random sample of diagnostics in the bucket, determining the sample size according to the scale shown in Table 1. If there are fewer than 50 diagnostics, the sample trivially includes all of the diagnostics in the bucket.
2. Analyze each diagnostic in the sample until a true positive has been found or the sample is exhausted.
3. Assign each false diagnostic a verdict of *False*.
4. If a true positive is found, assign it a verdict of *True*, and assign all remaining unmarked diagnostics in the bucket a verdict of *Suspicious*. If no true positives are found, meaning every diagnostic in the sample is a false positive, assign all unmarked diagnostics for that bucket a verdict of *Ignored*.

Table 1: Sample Sizes

Bucket Size	Sample
0–50	All
51–90	50
91–150	80
151–280	95
281–500	105
501–1,200	125
1201–10,000	200

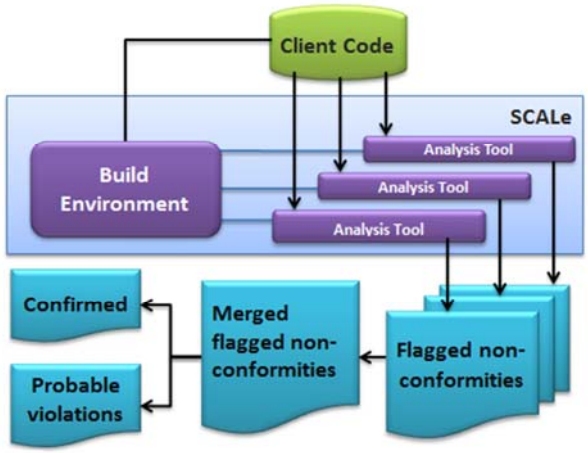

Figure 1: SCALe Diagnostics Process

This system works well, but it can still be suboptimal if any buckets have a high false-positive rate. Any bucket with a high false-positive rate requires a manual audit of up to 200 diagnostics before it can be discarded. Furthermore, if, over multiple codebase analyses, buckets for violations of a particular CERT rule have chronically high false-positive rates, every future bucket for that rule must be manually audited.

Because these diagnostics can be identified by different tools or by different algorithms in the same tool, we partition the diagnostics produced by a tool into *checkers*. Each checker represents a single algorithm in a single tool whose diagnostics are all associated with a single secure coding rule. Each checker also produces a unique form of diagnostic message. Distinct messages pro-

duced by a checker frequently vary in the specification of a variable or type identifier. Most checkers can be identified by analyzing the diagnostic messages using regular expressions. Furthermore, many tools categorize their error messages and produce distinct identification tags for each type of error. These tags can serve to identify the checker for these tools, bypassing regular expression analysis completely.

We have added a procedure to the SCALe process to identify the true-positive rate for any checker. We perform this procedure only on diagnostics that appear to have a low true-positive rate. If the procedure indicates that the true-positive rate is sufficiently low, we ignore the checker's diagnostics for the codebase under analysis and for any future version of it.

This procedure consists of an Excel spreadsheet that takes three variables: the total number of diagnostics in the checker, the number of diagnostics analyzed so far, and the number of diagnostics that have been found to be true. The spreadsheet then computes the 95% confidence interval of the actual false-positive rate based on the sample false-positive rate using the formula for computing a confidence interval for a proportion [Crossley 2000]. If the minimum bound of this interval is above 80%, we can be 95% confident that the true-positive rate is less than 20%, so we drop the checker from the current audit and any future audits of this codebase unless improvements are made to the checker.

Additionally, for three of the rules, we developed binary logistic regression models that may be used to assess the probability that a given diagnostic is a true versus a false positive. Although this approach was attempted for a set of 34 secure coding rules, as seen in Appendix A, only seven rules (ARR30-C, ARR36-C, DCL31-C, DCL32-C, EXP33-C, EXP34-C, and INT31-C) possessed sufficient data to enable an attempt at binary logistic regression. Of those seven rules, only three rules (ARR30-C, DCL31-C, and INT31-C) resulted in statistically significant models, which produced output grid tables depicting probability expectations associated with each scenario of which tools flagged versus did not flag a given diagnostic. We believe development of such binary logistic models using more data across multiple codebases will eventually produce a lookup table, by secure coding rule, of the probability of a true versus a false positive for remaining diagnostics that are not evaluated. In this fashion, management may choose to prioritize efforts on investigating suspicious diagnostics based on their likelihood of being a true versus a false positive.

2.2.2 Data for Research

Table 2 and Table 3 list the basic statistics of the codebases analyzed.

Table 2: Java Codebase Metrics

Codebase	Files	LOC	Size
java 1	28	4,161	131,477
java 2	33	5,453	201,215

Table 3: C Codebase Metrics

Codebase	Files	LOC	Size
C1	155	50,223	1,770,822
C2	124	118,425	4,511,085

Each SA tool produces output in its own idiosyncratic format. All tools can be adapted to produce a file containing the contents of the diagnostics they produced. SCALe provides scripts that convert such an output file into a list of diagnostics. Each tool is assumed to contain a set of checkers. Many tools produce diagnostics with specific error IDs that indicate the category of error they catch; for these tools, the error ID serves as the checker string. Tools that do not provide error IDs still provide strings that describe the error in some brief text message. These messages can quickly be categorized using regular expressions, and for these tools, the best matching regular expression indicates the checker for any diagnostic the tool may produce.

For each tool, SCALe provides a mapping between checker identifiers and CERT secure coding rules. We continue to maintain this map as the set of checker identifiers grows and changes with newer versions of the tool. This map is used to associate each diagnostic with a secure coding rule.

Many checkers identify problems that are not related to security. SCALe's maps show that these checker identifiers map to no secure coding rule, allowing analysts to ignore such diagnostics for the remainder of the audit. Occasionally, the checker ID may map to multiple secure coding rules, and the error message may be useful in choosing the most appropriate rule for mapping. In these cases, regular expression analysis on the diagnostic message indicates the proper mapping of diagnostic to CERT rule.

For Java codebases, we have used the SA tools listed in Table 4: FindBugs, Fortify, and Coverity, described shortly. We also used the warnings produced by the Eclipse compiler and thus include Eclipse as an SA tool. These tools check about 43% of the rules in the CERT Oracle Secure Coding Standard for Java.

Table 4: Java Rule Coverage

Tool	Rules
Coverity	35
FindBugs	44
Fortify	26
Eclipse	10
All Checked Rules	67
All Rules	156

For C and C++ codebases, we have used Fortify, Coverity, Rosecheckers, and PC-Lint. We have also used the warning messages produced by the GCC compiler, so we included GCC as an SA tool. These tools check about 63% of the rules in the CERT C Secure Coding Standard, as shown in Table 5.

Table 5: C Rule Coverage

Tool	Rules
Coverity	32
Fortify	9
LDRA	19
MSVC	17
PCLint	27
GCC	19
Rosecheckers	51
All Checked Rules except Rosecheckers	66
All Checked Rules	81
All Rules	129

FindBugs

FindBugs is an open source program that looks for bugs in Java code.[1] It uses static analysis to identify hundreds of different potential types of errors in Java programs. FindBugs is written in "pure" Java and is therefore platform-independent. It provides a simple command-line interface, as well as a graphical interface, and provides a plug-in enabling it to be integrated into Eclipse.

FindBugs operates on Java bytecode, so it is technically a binary-code analyzer rather than a source-code analyzer. However, it produces useful diagnostics and is free, so we included it in our study.

Fortify 360 SCA

Fortify 360 is a commercial product developed by Fortify Software, now owned by Hewlett-Packard. The product provides an extensive suite of tools for software security assurance. We focused on the *source code analysis* (SCA) tool. It can be used to analyze software written in Java, C, C++, .NET, ASP.NET, ColdFusion, "Classic" ASP, PHP, VB6, VBScript, JavaScript, PL/SQL, T-SQL, and COBOL.

Coverity Prevent

Coverity Prevent is a commercial product developed by Coverity, Inc.[2] The product provides an extensive suite of tools for software security assurance. We focused on the Coverity Static Analysis tool, which can be used to analyze software written in C, C++, Java, and C#. We also used the Coverity Integrity Manager, a web-based framework for viewing the results of Coverity Static Analysis. It provides a rich detail of each diagnostic found, including multiple locations in the source code that serve to create the diagnostic.

---

1 http://findbugs.sourceforge.net

2 http://www.coverity.com

MSVC /analyze

Several editions of Microsoft Visual C++ provide a built-in SA tool.[3] This includes MSVC 2008 Team Edition and several editions of MSVC 2010. It is named *analyze mode* because of the `/analyze` option that is fed to the Microsoft C++ compiler command. This tool can be enabled by turning on a switch called *Enable Code Analysis*. Consequently, any C/C++ program compiled by Visual Studio can be examined by the SA tool.

PC-Lint

PC-Lint is a commercial SA tool produced by Gimpel Software for the C and C++ programming languages.[4] First released in 1985, it is supported on all versions of Windows as well as MS-DOS and OS/2. It provides a command-line interface but can also be integrated as an external tool into many integrated development environments (IDEs), including Microsoft Visual Studio. It provides references to several coding guidelines, such as MISRA-C (both 2004 and 2008 editions).

Rosecheckers

The Rosecheckers project was internally developed at the CERT Division to provide an SA tool for analyzing C and C++ code. The project was designed to enforce the rules in the CERT C Secure Coding Standard and the CERT C++ Secure Coding Standard. Each rule in the standard that can be statically analyzed has one or more code checkers as part of the Rosecheckers project. The source for the Rosecheckers project is freely downloadable at the Rosecheckers website,[5] and the website also provides a virtual machine containing a complete build of the Rosecheckers project on Linux. The Rosecheckers project leverages the Compass/ROSE[6] project developed at Lawrence Livermore National Laboratory. This project provides a high-level API for accessing the abstract syntax tree (AST) of a C or C++ source code file.

2.2.3 Define and Describe the Various Performance Measures Statistics

The following measures are used to assess the performance of the various tools. They are based on the template table shown in Table 6.

A similar table was constructed for each rule and tool combination for each of the codebases analyzed. The table shows the number of diagnostics that have been confirmed as true positives (TP) or false positives (FP) as well as the count for the number of false negatives (FN). We did not measure the number of true negatives in this work. Although the number of true and false positives was determined by experts evaluating the diagnostics reported by the tools, the false negatives were computed as follows: for any given confirmed true diagnostic (a diagnostic being defined as a violation of a coding rule at a specific location), any tool that checks that rule but does not detect the violation is assigned a false negative.

---

[3] http://msdn.microsoft.com/en-us/library/vstudio/ms173498.aspx

[4] http://www.gimpel.com

[5] http://rosecheckers.sourceforge.net

[6] http://rosecompiler.org

Table 6: Performance Measures

		Actual TRUE	FALSE	
Predicted	TRUE	# of TP	# of FP	
	FALSE	# of FN	# of TN	← Not measured

These measures provide the counts needed for three of the four quadrants in the table. Conceivably, the number of true negatives could be computed based on the lines of code, but this was not done for two reasons. First, the three other values are based on confirmed results and are sufficient to compute standard performance measures. Second, because there are many "suspicious" diagnostics that are left uninvestigated, the count of true negatives would be an estimate with a potentially wide standard error.

From the three values, the following performance measures were computed:

- Predicted positive rate (TP/[TP + FP]). This is the proportion of true positives across all diagnostics reported by a tool. It was computed by checker and by rule and aggregated for a tool.
- Sensitivity rate (TP/TP + FN). This is the proportion of true positives detected among all of the true positives that the tool should have discovered. It was computed by rule and aggregated for a tool. Note that this value could not be computed by checker because the mapping from tool to tool was only done at the rule level.
- F-score (2 * [PPV * Sensitivity]/[PPV + Sensitivity]). This is used as the overall performance index because it is a function of performance in both of the previous dimensions. Note that it is scaled so that 1 is a perfect score.
- Rule set coverage (number of rules checked/total number of rules checked by all tools used in the analysis). This was included to provide a sense of coverage as some tools may perform well but only across a limited set of rules while others may cover more of the rule set.

We additionally developed a prediction model across the set of static code analyzer outputs to determine the likelihood of a given diagnostic being a true or false positive. By using binary logistic regression, we can capitalize on the knowledge from the set of tools related to a given diagnostic. This approach allows us to benefit further from the joint track records of the tools for a given secure coding rule. As shown in Appendix A, 34 secure coding rules were analyzed in a given set of diagnostics. Two coding rules (INT31-C and EXP40-C) were involved in 1,466 of a total of 1,749 diagnostics, or about 84% of the diagnostics. Also evident is that 20 rules were involved in single-digit amounts of diagnostics, ruling out any attempt at binary logistic regression due to a lack of sufficient data. Of the rules with sufficient data volumes, another set of rules was involved: those with only true verdicts or those with only false verdicts. As a result, they were not candidates for attempts at binary logistic regression.

3 Results

The research conducted during this project focused primarily on the first research question: What analyzers are best at detecting specific coding rule violations? The results produced to answer this question do not provide adequate evidence for a thorough analysis in relation to the other questions, though they are relevant. The main reason is the limited number of codebases that were available to analyze. Nonetheless, the research has produced a method and initial results for addressing the questions.

Each of the following analyses was conducted on the four codebases described earlier. The results are summarized here, and more detailed results are contained in Appendix A. Due to potential vendor issues, the names of the vendors' tools have been anonymized.

3.1 What Analyzers Are Best at Detecting Specific Coding Rule Violations?

As noted in Section 2.2.3, several measures of performance are used to characterize the performance of each tool. Table 7 shows the average results for the tools used on the two Java codebases. All of these measures range from 0 to 1, with a higher value indicating better performance.

The F-score offers an overall performance measure based on the sensitivity and predicted positive rate. As can be seen, tool ja2 has the best overall performance because of its superior performance in terms of sensitivity and relatively good predicted positive rate. Ja1 ranks second because of its high predicted positive rate and relatively good sensitivity. The java compiler was used on only one codebase and, as can be seen, its overall performance is much lower than that of any of the analyzers. When looking at the coverage of the rule set, ja3 covers the largest proportion of rules while the other tools check less than a third of the rules. This suggests that the use of multiple tools is justified.

Table 7: Results for Java Codebases

Tool	Sensitivity	Predicted Positive Rate	F-Score	Coverage
jc1	0.38	0.08	0.14	25%
ja1	0.62	0.76	0.62	20%
ja2	0.91	0.65	0.75	32%
ja3	0.31	0.65	0.40	55%

The averaged results for the two C codebases shown in Table 8 show somewhat different outcomes. The major difference to be noted is the generally poor results of all the tools. The maximum F-score is associated with a C compiler, but it is only 0.353. Although other tools have better performance on the two constituent performance measures, they tend to be good on one measure and poor on another. For instance, ca1 has relatively good scores for sensitivity and predicted positive rate values on average. However, the detailed scores comprising the average values are reflected in the low average F-score. What drives this is the alternating high and low

scores for the two different C codebases. For one codebase, ca1 has a relatively high sensitivity and low predicted positive rate yielding a low F-score. On the other codebase, the results are the reverse but yield a similar F-score.

Finally, note that the average coverage by any given tool across the union of the set of rules checked by the tools is relatively modest at best. Ja3 is the only tool that checked more than half of the rules.

Table 8: Results for C Codebases (averaged across both C codebases)

Tool	Sensitivity	Predicted Positive Rate	F-Score	Coverage
ca1	0.338	47.5%	0.112	15%
ca2	0.143	22.2%	0.174	6%
ca3	0.024	5.7%	0.033	6%
ca6	0.590	4.4%	0.081	25%
ca7	0.128	2.2%	0.037	17%
cc1	0.050	25.0%	0.083	11%
cc2	0.293	44.6%	0.353	13%

As the tables show, the overall pattern of scores for the C codebases is lower than that for the Java codebases. Some of the tools, such as ca3, have very poor results, suggesting they are of limited value. Ca1 looks to have a low average F-score while having relatively good sensitivity and predicted positive rate values in comparison to cc2. The explanation for the ca1 performance numbers is the lack of consistency in its results across the two codebases. In neither case did ca1 have both high sensitivity and a high predicted positive rate. While the average result may seem inconsistent with the average F-score, they are accurate.

Ca6 illustrates a common problem with static analyzers. There is often a tradeoff made between identifying the true violation at the expense of having a high false-positive rate. Ca6 has relatively good sensitivity. That is, among all of the true violations, on average it identified 59% of them. However, the price for this is a low predicted positive rate, or analogously, a high false-positive rate. Note that for ca6, of the violations it detects, only 4.4% of them are confirmed to be true violations. Interestingly, cc2, a compiler, had the best overall performance.

However, it is important to note the limited rule coverage by the tools in general. For the C tools, ca6 had the highest average coverage, which was only 25% of the rules. See Figure 2 for codebase C1 and Figure 3 for codebase C2.

With this more detailed look at the results, we can observe the very different performance of the tools on the different codebases. Cc2 is a case in point. While it had the best performance and coverage on the first codebase, it had the worst performance and was tied for the least amount of rule coverage on the second.

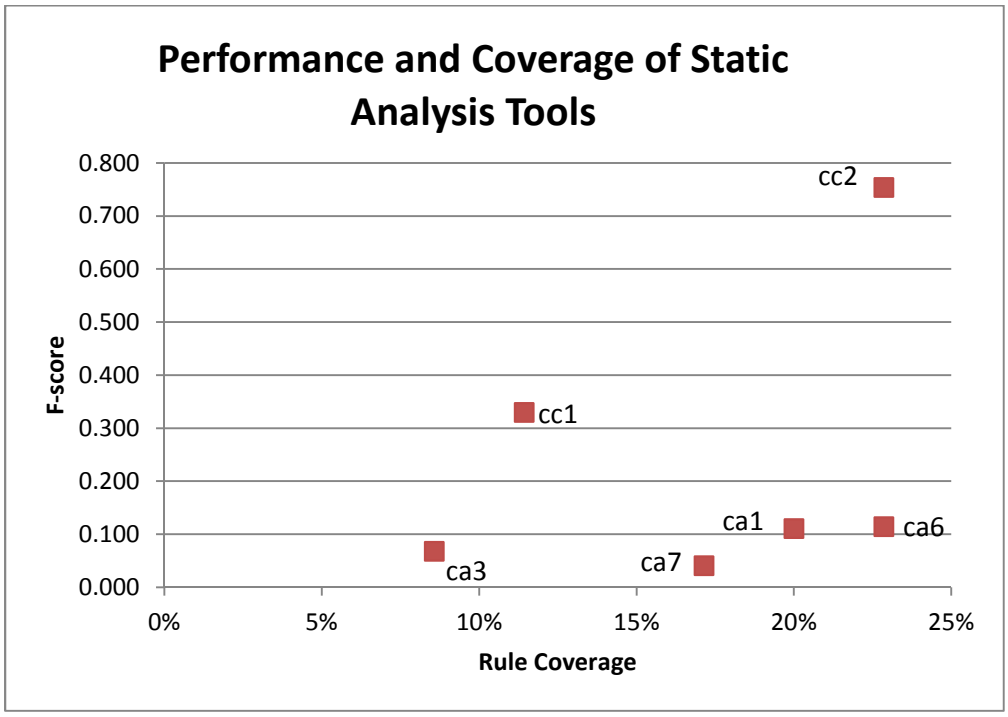

Figure 2: Performance and Rule Coverage for the C1 Codebase[*]

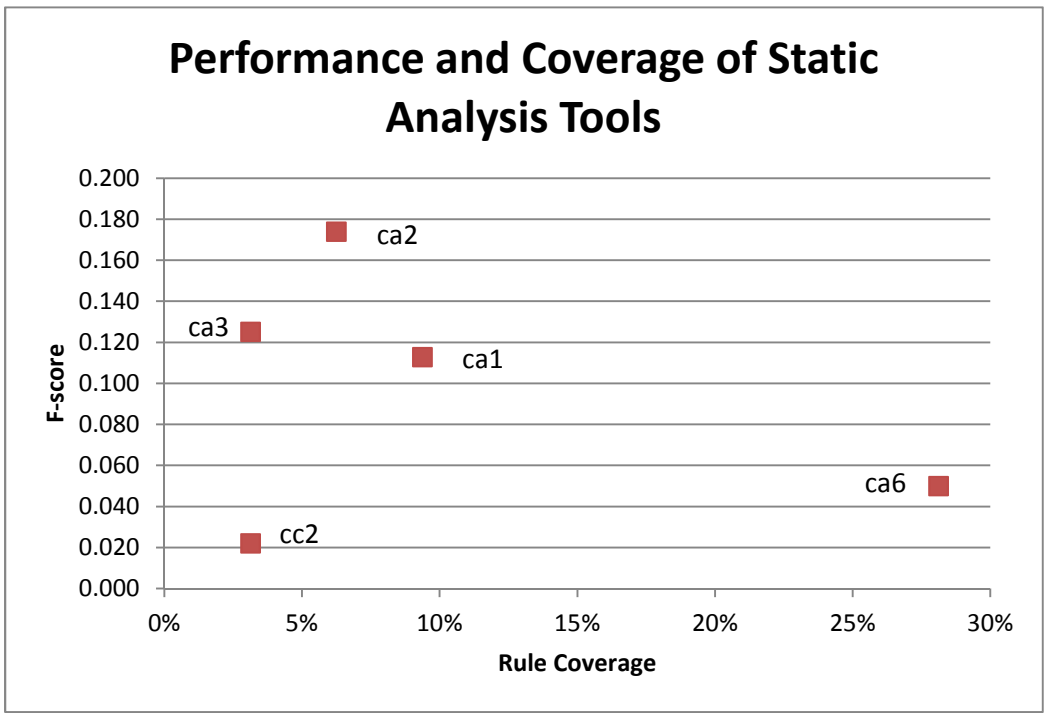

*Figure 3: Performance and Rule Coverage for the C2 Codebase**

[*] Note that the y-axes in Figures 2 and 3 are on different scales.

3.2 How Common Are Violations of Secure Coding Rules in Systems Developed under Various Code Cultures?

Table 9 shows a mapping in terms of detected rule violations for the two Java codebases in the study. Overall, 27 rules had flagged an anomaly in the codebase. Of these, 7, or 26%, were common across the two codebases. This suggests that for these two codebases there is little in common in terms of the pattern of violations. Whether this is due to variability across code cultures or inherent variability in codebases caused by other factors (for example, maturity of the code), we cannot say at this time.

Table 9: Mapping of Rules Violated in Two Java Codebases

Rule	Java 1	Java 2
DCL00-J	X	
DCL01-J	X	
ENV05-J		X
ERR00-J		X
ERR01-J		X
ERR07-J	X	X
ERR08-J	X	X
ERR09-J		X
EXP01-J	X	X
EXP04-J	X	
FIO02-J		X
FIO04-J	X	X
FIO13-J		X
IDS00-J		X
IDS02-J		X
IDS03-J		X
IDS04-J	X	X
IDS09-J	X	X
IDS13-J		X
MET02-J	X	
MSC03-J	X	
OBJ01-J	X	
OBJ05-J		X
OBJ06-J	X	X
OBJ09-J	X	
OBJ10-J	X	
VNA00-J	X	

3.3 How Effective Is Static Analysis as a Means of Exposing Vulnerabilities?

Assessing the effectiveness of static analysis would be best addressed by having results from subsequent testing that could detect the violations that escaped from the static analysis. This was not done. As a second-best alternative, we can review the results shown in Table 7 and Table 8 and focus on the sensitivity scores. Recall that sensitivity is the percentage of actual violations that were accurately detected by the tool. For the Java tools, we can see quite a range of performance, with ja2 having a very high detection rate and ja3 detecting slightly less than one-third of actual coding violations. A more detailed look at ja2 shows its sensitivity ranging from 0.56 to 1.0 for one codebase and 0.33 to 1.0 for the second codebase. From this we learn that sensitivity to the various secure coding rules can vary greatly within a tool, even one that performs well.

Taking a similar look at the C tools, ca6, with a value of 0.59, has the highest sensitivity while ca3 and cc1 both detect 5% or less of the actual violations in the codebases. As with the Java tools, sensitivity across the rule set varies widely. For ca6, the sensitivity values range from 0.0 to 1.0 for both codebases.

3.4 Can the Pattern of Results across Rules Tell Us What Is Likely to Return a True or False Positive?

Table 10 displays a subset of the 34 secure coding rules associated with varying numbers of diagnostics in the analyzed SCALe results. These diagnostics, with confirmed false and true positives, were analyzed in an attempt to apply logistic regression in predicting the probability of a given diagnostic being a false positive given knowledge of which static code analyzers flagged versus did not flag the diagnostic.

Table 10: Subset of Rules with Varying Numbers of Diagnostics

Rule	N Rows	N(False)	N(True)
INT31-C	1,347	1,298	76
EXP40-C	92	92	0
DCL36-C	34	0	34
EXP34-C	32	18	14
ARR30-C	30	27	3
INT34-C	25	25	0
EXP36-C	19	18	1
DCL31-C	18	15	3
EXP30-C	18	18	0
INT33-C	18	17	1
EXP33-C	17	14	3
INT32-C	15	15	0
DCL30-C	10	10	0
DCL32-C	10	9	1
ARR36-C	9	8	1

Appendix A includes example logistic regression analysis [Ryan 2012] associated with the three secure coding rules that possessed sufficient data to enable logistic regression models at this time. Table 11, Table 12, and Table 13 depict an output grid table for each rule showing the different scenarios in which tools flagged (1) or did not flag (0) a given diagnostic, along with the updated probability of a false versus true verdict. For example, for a diagnostic involving the ARR30-C rule, in which the only tool to flag an example diagnostic is cc2, we are almost certain this diagnostic is a true positive. On the other hand, if a diagnostic is flagged by multiple tools (cc1, ca7, and cc2), we are almost certain this is a false positive. Finally, if the diagnostic is flagged only by the cc1 and cc2 tools, we have 87% confidence that the diagnostic is a false positive.

Consequently, these three models enhance our outlook of the probability of a diagnostic being a false versus true positive based on which tools did or did not flag the diagnostic. Note too that the probabilities are not a function of a simple majority vote.

Table 11: ARR30-C Rule

Ca1	Ca2	Cc1	Ca7	Cc2	Ca6	Ca3	Probability (Verdict = False)	Probability (Verdict = True)
0	0	0	0	1	0	0	1.2433e-8	0.999999
0	0	0	1	0	0	0	0.999999	1.2433e-8
0	0	0	1	1	0	0	0.999999	1.2433e-8
0	0	1	0	0	0	0	0.866666	0.133333
0	0	1	0	1	0	0	0.866666	0.133333
0	0	1	1	0	0	0	1	0
0	0	1	1	1	0	0	1	0

Table 12: DCL31-C Rule

Ca1	Ca2	Cc1	Ca7	Cc2	Ca6	Ca3	Probability (Verdict = False)	Probability (Verdict = True)
0	0	0	0	0	1	0	0.9375	0.0625
0	0	0	1	0	0	0	9.1869e-8	0.999999
0	0	0	1	0	1	0	9.1869e-8	0.999999

Table 13: INT31-C Rule

Ca1	Ca2	Cc1	Ca7	Cc2	Ca6	Ca3	Probability (Verdict = False)	Probability (Verdict = True)
0	0	0	0	0	1	0	0.971145	0.028854
0	0	0	0	1	0	0	0.280701	0.719298
0	0	0	0	1	1	0	0.280701	0.719298
0	0	0	1	0	0	0	0.999999	1.243320
0	0	0	1	0	1	0	0.999999	1.2433e-8
0	0	0	1	1	0	0	0.999998	1.0723e-6
0	0	0	1	1	1	0	0.999998	1.0723e-6

We believe the ability to create three significant binary logistic regression models on this initial set of diagnostics highlights the practicality and promise of continuing to assemble data related to diagnostics investigated with verdicts of true and false across additional codebases. Armed with a growing inventory of confirmed verdicts, the SCALe team will be better positioned to create

models for each secure coding rule, thereby enabling lookup tables of the different scenarios of static code analyzers involved, so that real-time automation of the updated probabilities of true versus false positives can be performed. This feedback could prove invaluable in the prioritization and scheduling of work to investigate the high volume of suspicious flags from the static code analyzers.

4 Summary and Future Work

To summarize, a modified version of the SCALe process was used to generate the data for the analysis of various static analyzers and compilers in terms of their performance detecting violations of secure coding rules for Java and C. Two codebases for Java and C were analyzed. Performance measures used included the number of true positives, false positives, and false negatives. From these values, the predicted positive rate, sensitivity, and F-scores were computed for each rule and tool combination for each codebase. The results generally show that the tools perform unevenly in terms of sensitivity and predicted positive rate, as illustrated by ca6, or just have generally low results. Ja2 and ja1 stand out among all of the tools in terms of their better performance, as reflected in their relatively high F-scores.

None of the tools covered all of the secure coding rules. Indeed, the results ranged from 13% to 72% for the Java tools, depending on the codebase, and from 3% to 28% for the C tools, depending on the codebase. Hence, even with good performance, use of just one tool is likely to leave vulnerabilities in the codebase.

Regarding the identification of code cultures, this study did not have sufficient data to address the question. Table 9 provided an initial observation that showed the limited overlap in the rules detected in the two Java codebases. This is weak evidence with respect to code cultures and simply demonstrates that different codebases can have different profiles in terms of their rule violations.

Finally, in terms of the effectiveness of static analysis, we found that the tools' performance varies greatly. Additionally, each tool's detection of individual rules varied over multiple analyses. This raises a concern about reliance on a single tool for detecting secure coding violations and exposes the need for additional screenings of diagnostics, using models that account for how different analyzers have treated a given diagnostic.

We would like to continue to leverage the SCALe process and obtain additional codebases to analyze. Additional results will allow us to address some of the unanswered questions that we began with and to further refine our knowledge of the effectiveness of SA tools. Additional studies would include

- profiling of code cultures
- improving the estimates of the false-positive and false-negative rates
- experiments with the sequencing of tools and the time to perform a SCALe evaluation
- experiments to test the prediction of the remaining true violations

Appendix A

Nominal Logistic Fit for Verdict ID=ARR30-C

Converged in Gradient, 17 iterations

Whole Model Test

Model	-LogLikelihood	DF	ChiSquare	Prob[7] > ChiSq
Difference	3.8623720	2	7.724744	0.0210*
Full	5.8901172			
Reduced	9.7524892			

RSquare (U)[8]	0.3960
AICc	21.3802
BIC	25.385
Observations (or Sum Wgts)	30

Measure	Training	Definition
Entropy RSquare	0.3960	1-Loglike(model)/Loglike(0)
Generalized RSquare	0.4749	(1-(L(0)/L(model))^(2/n))/(1-L(0)^(2/n))
Mean -Log p	0.1963	∑ -Log(ρ[j])/n
RMSE	0.2404	√ ∑(y[j]-ρ[j])²/n
Mean Abs Dev	0.1156	∑ |y[j]-ρ[j]|/n
Misclassification Rate	0.0667	∑ (ρ[j]≠ρMax)/n
N	30	n

Parameter Estimates

Term		Estimate	Std Error	ChiSquare	Prob > ChiSq
Intercept	Biased	10.0373485	1198.4347	0.00	0.9933
cc1[0]	Biased	−10.037348	4484.1319	0.00	0.9982
ca7[0]	Biased	−18.202895	4641.5175	0.00	0.9969
cc2[0]	Zeroed	0	0	.	.

[7] P value < .05 indicates a statistically significant model.

[8] RSquare (U), which provides a measure of how well the model explains the outcome, may be considered good in the 40%–50%+ range, as it does not behave similarly to traditional RSquared measures from linear regression.

Nominal Logistic Fit for Verdict ID=DCL31-C

Converged in Gradient, 15 iterations

Whole Model Test

Model	-LogLikelihood	DF	ChiSquare	Prob > ChiSq
Difference	4.3694350	1	8.73887	0.0031*
Full	3.7406667			
Reduced	8.1101018			

RSquare (U)	0.5388
AICc	15.1956
BIC	16.1524
Observations (or Sum Wgts)	18

Measure	Training	Definition
Entropy RSquare	0.5388	1-Loglike(model)/Loglike(0)
Generalized RSquare	0.6476	(1-(L(0)/L(model))^(2/n))/(1-L(0)^(2/n))
Mean -Log p	0.2078	∑ -Log(ρ[j])/n
RMSE	0.2282	√ ∑(y[j]-ρ[j])²/n
Mean Abs Dev	0.1042	∑ |y[j]-ρ[j]|/n
Misclassification Rate	0.0556	∑ (ρ[j]≠ρMax)/n
N	18	n

Parameter Estimates

Term		Estimate	Std Error	ChiSquare	Prob > ChiSq
Intercept	Unstable	−6.7474222	1166.4576	0.00	0.9954
ca7[0]	Biased	9.45547241	1166.4576	0.00	0.9935
ca6[0]	Zeroed	0	0	.	.

Nominal Logistic Fit for Verdict ID=INT31-C

Converged in Gradient, 17 iterations

Whole Model Test

Model	-LogLikelihood	DF	ChiSquare	Prob>ChiSq
Difference	101.44029	3	202.8806	<.0001*
Full	192.41881			
Reduced	293.85910			

RSquare (U)	0.3452
AICc	392.867
BIC	413.74
Observations (or Sum Wgts)	1374

Measure	Training	Definition
Entropy RSquare	0.3452	1-Loglike(model)/Loglike(0)
Generalized RSquare	0.3944	(1-(L(0)/L(model))^(2/n))/(1-L(0)^(2/n))
Mean -Log p	0.1400	∑ -Log(ρ[j])/n
RMSE	0.1820	√ ∑(y[j]-ρ[j])²/n
Mean Abs Dev	0.0662	∑ |y[j]-ρ[j]|/n
Misclassification Rate	0.0371	∑ (ρ[j]≠ρMax)/n
N	1374	n

Parameter Estimates

Term		Estimate	Std Error	ChiSquare	Prob>ChiSq
Intercept	Unstable	8.63095571	1087.5618	0.00	0.9937
ca7[0]	Unstable	-7.3433347	1087.5617	0.00	0.9946
cc2[0]		2.22860432	1189.0797	0.00	0.9985
ca6[0]		-4.5287e-9	1189.0797	0.00	1.0000

Bibliography

URLs are valid as of the publication date of this document.

[Bass 2012]
Bass, L.; Clements, P.; & Kazman, R. *Software Architecture in Practice*, 3rd ed. Reading, MA: Addison-Wesley, 2012.

[Crossley 2000]
Crossley, Mark. *The Desk Reference of Statistical Quality Methods*. ASQ Quality Press, 2000, pp. 81–89.

[Heffley 2004]
Heffley, J. & Meunier, P. "Can Source Code Auditing Software Identify Common Vulnerabilities and Be Used to Evaluate Software Security?" *Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS–04), Track 9, Volume 9*. Big Island, HI, January 2004. IEEE Computer Society, January 2004.

[Long 2011]
Long, Fred; Mohindra, Dhruv; Seacord, Robert C.; Sutherland, Dean F.; & Svoboda, David. *The CERT Oracle Secure Coding Standard for Java* (SEI Series in Software Engineering). Addison-Wesley Professional, 2011.

[Long 2013]
Long, Fred; Mohindra, Dhruv; Seacord, Robert C.; Sutherland, Dean F.; & Svoboda, David. *Java Coding Guidelines: 75 Recommendations for Reliable and Secure Programs* (SEI Series in Software Engineering). Addison-Wesley Professional, 2013.

[Ryan 2012]
Ryan, Barbara F.; Joiner, Brian L.; & Cryer, Jonathan D. *Minitab Handbook, 6th Edition: Update for Release 16*. Cengage Learning, 2012.
http://store.minitab.com/781/catalog/category.10753/language.en/currency.USD/?id=wXvyNH6kHn

[Seacord 2008]
Seacord, Robert C. *The CERT C Secure Coding Standard.* Addison-Wesley Professional, 2008.

[Seacord 2010]
Seacord, Robert; Dormann, Will; McCurley, James; Miller, Philip; Stoddard, Robert; Svoboda, David; & Welch, Jefferson. *Source Code Analysis Laboratory (SCALe) for Energy Delivery Systems* (CMU/SEI-2010-TR-021). Software Engineering Institute, Carnegie Mellon University, 2010.
http://resources.sei.cmu.edu/library/asset-view.cfm?assetID=9561

[SEI 2013]
Software Engineering Institute. *CERT C++ Secure Coding Standard.*
https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637 (2012).

[Svoboda 2009]
Svoboda, D. *CERT Rose Checkers*. http://rosecheckers.sourceforge.net (2009).

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	June 2014	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Improving the Automated Detection and Analysis of Secure Coding Violations	FA8721-05-C-0003

6. AUTHOR(S)

Daniel Plakosh, Robert Seacord, Robert Stoddard, David Svoboda, David Zubrow

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2014-TN-008

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116	n/a

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Coding errors cause the majority of software vulnerabilities. For example, 64% of the nearly 2,500 vulnerabilities in the National Vulnerability Database in 2004 were caused by programming errors. The CERT Division's Source Code Analysis Laboratory (SCALe) offers conformance testing of C language software systems against the CERT C Secure Coding Standard and the CERT Oracle Secure Coding Standard for Java, using various analysis tools available from commercial software vendors. Unfortunately, the current SCALe analysis process and tools do not collect any statistics about the accuracy of the code analysis tools or about the coding violations they flag, such as frequency of occurrence. This paper describes the approach used to add the ability to collect and statistically analyze data regarding coding violations and tool characteristics along with the initial results. The collected data will be used over time to improve the effectiveness of the SCALe analysis.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Secure coding, static analysis, coding flaws, secure coding rules, SCALe	33

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL