Software Engineering Institute

Evaluation of CERT Secure Coding Rules through Integration with Source Code Analysis Tools

Stephen Dewhurst
Chad Dougherty
Yurie Ito
David Keaton
Dan Saks
Robert C. Seacord
David Svoboda
Chris Taschner
Kazuya Togashi

June 2008

TECHNICAL REPORT
CMU/SEI-2008-TR-014
ESC-TR-2008-014

CERT Program

Carnegie Mellon

This report was prepared for the

SEI Administrative Agent
ESC/XPK
5 Eglin Street
Hanscom AFB, MA 01731-2100

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

Table of Contents

List of Figures

List of Tables

Acknowledgments

Thanks to our sponsor, JPCERT/CC and to the JPCERT/CC employees who assisted in this study: Takayoshi Shiigi and Masaki Kubo. Thanks to the staff members at Software Research Associates (SRA) who volunteered their time to participate in this study. Thanks to our collaboration partners at Fortify Software and Fortify Software Japan, including Brian Chess, Jacob West, Geoff Morrison, Erik Klein, John Forsythe, Kannan Goundan, Ning Wang, and Gensei Endo. Thanks to Daniel Quinlan at Lawrence Livermore National Laboratory for all his help getting us up and running with Compass/ROSE. Thanks to our SEI editor Pamela Curtis, and to Bob Rosenstein, Jason Rafail, and Jeff Carpenter for making this work possible.

Executive Summary

This report describes the results of a study to evaluate the effectiveness of secure coding practices, including the use of static analysis tools coupled with secure coding rule sets such as the CERT C Programming Language Secure Coding Standard [CERT 07a] and the CERT C++ Programming Language Secure Coding Standard [CERT 07b].

This study represents a joint effort between the CERT Secure Coding Initiative and JPCERT/CC.

The CERT Secure Coding Initiative was established to work with software developers and software development organizations to eliminate vulnerabilities resulting from coding errors before they are deployed. The goal of this effort is to reduce the number of vulnerabilities to a level where they can be handled by existing vulnerability analysis teams around the world and decrease remediation costs by eliminating vulnerabilities *before* software is deployed.

JPCERT/CC is the first CSIRT (computer security incident response team) established in Japan. The organization coordinates with network service providers, security vendors, government agencies, and industry associations. By so doing, it acts as a “CSIRT of the CSIRTs” for Japan. In the Asia Pacific region, JPCERT/CC helped form the Asia Pacific Computer Emergency Response Team (APCERT) and provides a secretariat function for APCERT. Globally, as a member of the Forum of Incident Response and Security Teams (FIRST), JPCERT/CC coordinates its activities with the trusted CSIRTs worldwide.

The objectives of the study were to evaluate the efficacy of the CERT Secure Coding Standards and source code analysis tools in improving the quality and security of commercial software projects. Two static analysis tools, Fortify Source Code Analysis (SCA) from Fortify Software and Compass/ROSE from Lawrence Livermore National Laboratory were selected for their extensibility as well as overall effectiveness. Checkers were then developed for each of these tools to check code for violations of the CERT C and C++ Secure Coding Standards. These tools were then provided to Software Research Associates, Inc. (SRA), a well-established Japanese software development firm. SRA evaluated the extended versions of Fortify SCA and Compass/ROSE on two existing projects: a toll collection system-related GUI application written in C++ and an Video Service Communication Protocol written in the C programming language.

The project successfully extended source code analysis tools to discover a number of software defects in both projects evaluated, demonstrating the effectiveness of both the CERT Secure Coding Standards and the static analysis tools evaluated in improving software quality. The project was also successful in identifying ways in which both the CERT Secure Coding Standards and the static analysis tools could be further improved.

Abstract

This report describes a study conducted by the CERT Secure Coding Initiative and JPCERT to evaluate the efficacy of the CERT Secure Coding Standards and source code analysis tools in improving the quality and security of commercial software projects. In addition to assessing the ability of existing tools to detect violations of the standard, the ability to extend and improve the tools is surveyed. Finally, the use of a selected tool to improve the quality of code in the real-world case of a Japanese software vendor's product is described.

1 Overview

This report describes the results of a study to determine the effectiveness of the secure coding rules and recommendations from the CERT C Programming Language Secure Coding Standard [CERT 07a] and the CERT C++ Programming Language Secure Coding Standard [CERT 07b].

1.1 SECURE CODING STANDARDS

Society's increased dependency on networked software systems has been matched by an increase in the number of attacks aimed at these systems. These attacks—directed at governments, corporations, educational institutions, and individuals—have resulted in the loss and compromise of sensitive data, system damage, lost productivity, and financial loss [Seacord 05].

Software vulnerability reports continue to grow at an alarming rate [CERT 07c] and a significant number of them result in technical alerts [US-CERT 08]. To address this growing threat, the introduction of software vulnerabilities during software development and ongoing maintenance must be significantly curtailed.

An essential element of secure software development is well-documented and enforceable coding standards. Coding standards encourage programmers to follow a uniform set of rules and guidelines determined by the requirements of the project and organization, rather than by the programmer's familiarity or preference. Once established, these standards can be used as a metric to evaluate source code (using manual or automated processes) to determine compliance with the standard.

The secure coding standards proposed by CERT are based on documented standard language versions as defined by official or de facto standards organizations. For example, secure coding standards are planned for the following languages:

- C programming language (ISO/IEC 9899:1999) [ISO/IEC 9899-1999]
- C++ programming language (ISO/IEC 14882:2003) [ISO/IEC 14882-2003]

Applicable technical corrigenda and documented language extensions such as the ISO/IEC TR 24731 extensions to the C library [ISO/IEC TR 24731-1-2007] will also be considered.

The scope allows specific guidance to be provided to broad classes of users. Programming language standards, like those created by ISO/IEC, are primarily intended for compiler implementers. Secure coding standards are ancillary documents that provide rules and guidance directly to developers who program in languages defined by these standards.

The goal of each coding standard is to define a set of rules that is necessary (but not sufficient) to ensure the security of software systems developed in the respective programming languages.

The CERT secure coding standard consists of *rules* and *recommendations*. Coding practices are defined to be rules when all of the following conditions are met:

1. Violation of the coding practice will result in a security flaw that may result in an exploitable vulnerability.
2. There is a denumerable set of conditions for which violating the coding practice is necessary to ensure correct behavior.
3. Conformance to the coding practice can be determined through automated analysis, formal methods, or manual inspection techniques.

Rules must be followed to claim compliance with a standard unless an exceptional condition exists. If an exceptional condition is claimed, the exception must correspond to a predefined exceptional condition and the application of this exception must be documented in the source code.

Recommendations are guidelines or suggestions. Coding practices are defined to be recommendations when all of the following conditions are met

1. Application of the coding practice is likely to improve system security.
2. One or more of the requirements necessary for a coding practice to be considered a rule cannot be met.

Compliance with recommendations is not necessary to claim compliance with a coding standard. It is possible, however, to claim compliance with one or more verifiable guidelines. The set of recommendations that a particular development effort adopts depends on the security requirements of the final software product. Projects with high-security requirements can dedicate more resources to security, and are consequently likely to adopt a larger set of recommendations.

1.2 THE STUDY

Figure 1 shows an overview of the study described in this report. The project ran from August 2007 through March 2008 and consisted of four major tasks.

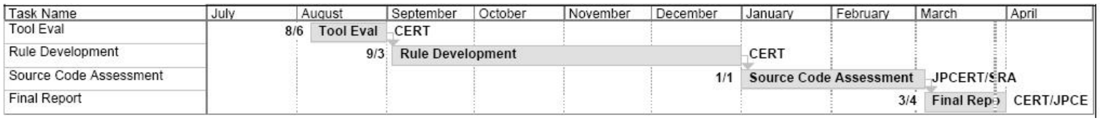


Figure 1. Software verification and validation project.

In the first phase of the study, the CERT Secure Coding Initiative (SCI) evaluated static analysis tools for use in the study. In the second phase of the study, CERT SCI developed checkers for the static analysis tools selected in the first phase. These first two phases of the study are described in detail in Section 2 of this report. Once the checkers were developed, they were provided to JPCERT and SRA for use in evaluating source code. Finally, a month was scheduled to analyze the results and prepare the final report.

2 Evaluation and Rule Development

The first phase of the project involved evaluating existing commercial and non-commercial source code analysis tools for suitability for the project. The evaluation process followed was a first-fit approach adapted from earlier work at the Software Engineering Institute [Wallnau 01]. The evaluation criteria included, but were not limited to

- **Effectiveness.** To have an impact on industrial practice, it is necessary to start with a tool that already represents the best of existing practices. By enhancing this tool further, it was felt that we could advance the state of the practice.
- **Extensibility.** Extensibility is a critical criterion, because it is necessary to extend the existing set of rules supported by the static analysis tool to incorporate support for the CERT secure coding rules being developed.
- **Suitability.** Because this tool is being used by a particular software developer, in a particular context, it must be suitable to the needs of this developer and the selected projects. Among other properties, suitability includes availability on the platforms and compilers being used and the preferences of the software developer.

The following tools were evaluated for this study:

- Fortify SCA version 4.5 (used to create the rules that were then run using version 5.0)
- Lawrence Livermore National Laboratory (LLNL) Compass/ROSE

2.1 FORTIFY SCA

The Fortify Source Code Analyzer Engine (SCA) renders a variety of programming languages (Java, C, and C++, for example) into an intermediate form that is then processed with a set of algorithms used to identify and flag dangerous coding constructs [Chess 02]. The Fortify Source Code Analyzer (SCA) consists of five analysis engines. The data flow analyzer, the control flow analyzer, the semantic analyzer, the structural analyzer, and the configuration analyzer combine to identify vulnerabilities in source code. Each of these analysis engines uses a different approach to vulnerability detection.

The data flow analyzer attempts to track user-supplied data from input into a program as it propagates through functions. The purpose of this analyzer is to identify vulnerabilities resulting from unsafe use of user-supplied data. This analyzer uses global interprocedural taint propagation to detect vulnerabilities in function calls or operations.

The control flow analyzer looks for vulnerabilities in sequences of operations in a single function or method. This analyzer applies state machines characterizing unsafe behavior to the supplied source code. The control flow analyzer determines whether operations are executed in a certain order by analyzing control flow paths in a program. Using these methods, the analyzer can detect vulnerabilities in sequences of operations resulting in the absence of function calls in addition to identifying dangerous sequences of function calls.

The semantic analyzer attempts to use signatures to find unsafe function calls. This analyzer scrutinizes uses of functions and application programming interfaces (APIs) at an intraprocedural level to detect vulnerabilities. This analysis engine includes specialized methods of buffer overflow, format string, and execution path vulnerability detection in addition to detections of other types of more general issues. Using these techniques the semantic analyzer can flag any potentially dangerous function call.

The structural analyzer looks for vulnerabilities in code constructs. This analysis engine is the most flexible and potentially powerful of those offered in Fortify SCA. The relationships between a line of code and the block that contains the line are scrutinized for possible vulnerabilities by this analysis engine. The structural analyzer identifies violations of secure programming practices and techniques that are often difficult to detect through inspection because they encompass a wide scope involving both the declaration and use of variables and functions.

The configuration analyzer detects vulnerabilities in text properties files and XML configuration files. This analyzer attempts to identify dangerous key value pairs and unsafe XML elements and attribute definitions.

Projects are scanned with the Fortify SCA using the `sourceanalyzer` program provided in the Fortify program pack. The following is an example of the sequence of commands:

```
sourceanalyzer -Xmx400M -b sci-a gcc -g3 -o a a.c


sourceanalyzer -Xmx400M -b sci-a -scan -rules sci-rules.xml -rules
sci-structure-rules.xml -level broad -logfile filename.log
```

The Fortify source code analysis process consists of three phases. These phases are translation, analysis, and verification.

During the translation phase of the analysis process, Fortify gathers the source code via a series of commands. The source code is then translated into an intermediate format that is associated with a user-specified build ID. The build ID should be unique for the project being scanned by Fortify.

The analysis phase follows the translation phase. During this phase, source files identified in the translation phase are scanned and Fortify SCA creates an analysis results file.

The translation and analysis phase is then followed up by verification. The user inspects the analysis file for any significant errors reported by Fortify SCA. Errors are in the following form:

```
[<rule ID number> : <severity> : <rule title> : <analysis engine>]

      <source file name>(<line number>) : <details>
```

Each of the analysis engines included in Fortify SCA is extendable. Custom rules can be written for each of these analysis engines. These custom rules are written in XML and can be used on their own or in concert with the rule sets built into Fortify SCA.

Investigation was done on 113 of 180 Secure Coding Standard C rules and 63 of 99 Secure Coding Standard C++ rules. The example code from these rules was run through Fortify SCA. Of these rules, Fortify SCA flagged 16 C rules and 6 C++ rules without further extension of the analysis engines.

For Fortify SCA to detect violations of the Secure Coding Standards, custom rules were created to extend the analysis engines. These custom rules extend the control flow analyzer, the semantic analyzer, and the structural analyzer. Eighteen rules were created to extend the control flow analyzer, 9 rules were created to extend the semantic analyzer, and 31 rules were created to extend the structural analyzer.[1]

After these custom rules were added, Fortify SCA was able to catch all aspects of 27 C rules and17 C++ rules. Fortify SCA was able to detect only some aspects of an additional 14 C rules and 5 C++ rules. Overall, Fortify SCA was able to at least partially catch 85 of 176 of the Secure Coding Standard rules investigated.

	C	**C++**	**Total**
Total number of Secure Coding Standard Rules	180	99	279
Rules investigated for the study	113	63	176
Rules Fortify SCA implements by default	16	6	22
Rules implemented for Fortify SCA	27	17	44
Rules partially implemented for Fortify SCA	14	5	19
Total number of rules at least partially implemented	57	28	85

Table 1. Summary of Rules Implemented for Fortify SCA.

Some limitations were observed while extending the Fortify analysis engines to detect violation of CERT secure coding rules. These limitations hampered the quality and quantity of the analysis rules that could be created. For each secure coding rule, attempts were first made to create an analysis rule to flag on all noncompliant code. Analysis rules were created to flag some or most noncompliant code when all could not be detected.

The timing of Fortify SCA analysis, and the consequent nature of the code analyzed, limits many of the analyzer rules written for this project. Fortify SCA does not examine the source code before compile time. Rather, Fortify SCA examines the results of the translation phase described above. Because analysis takes place after translation, none of the analysis engines can match all syntactical patterns in the original code.

[1] There are some rules that appear in both the C and the C++ standards. Duplicates have been removed from this count, which is by analysis engine.

Post-translation analysis also hinders the ability of Fortify SCA to detect many aspects of the Secure Coding Standard rules. Fortify SCA could not detect any of the rules in either the “Preprocessor” or the “Signals” categories. All `for` loops are converted to `while` loops before Fortify SCA analyzes the code. The `sizeof` operator, the `const` qualifier, and the `enum` specifier are all removed during the translation phase. Multiple identifier definitions are automatically resolved during the translation phase.

Another limitation of Fortify SCA is that its analysis is confined by function boundaries. Fortify SCA tracks taint, constants, and program state across these boundaries, but does not perform global typestate analysis. This affects all of the rules in the Secure Coding Standard designed to address issues that develop because of the way multiple functions interact with one another. This includes issues associated with referencing objects outside of their lifetime and reopening file streams, among others.

Fortify SCA’s array-handling limitations also hampered the ability to extend the analysis engines to flag on the Secure Coding Standard rules. Arrays were transformed into pointers during Fortify SCA’s translation phase. This conversion causes rules specific to arrays to flag on all pointers, which results in far too many false positives.

Conditional expressions also posed a problem for Fortify SCA. Fortify SCA ignores conditional expressions within the Structural analysis engine. This means that rules designed to flag conditional expressions would have to be written in one of the other analysis engines, which limited the cases of Secure Coding Standards violations that could be flagged.

Tracking variables in the Control Flow analysis engine was also an issue. Fortify SCA tracks data, not variables, therefore when 0 is assigned to a variable, the state machine stops tracking that variable. This causes false negatives within rules that attempt to track variables.

The work done to create Fortify rules to extend the Control Flow analyzer, the Semantic analyzer, and the Structural analyzer was done in continued cooperation with the technical staff at Fortify.

Appendix A provides a detailed description about the implementation of checkers for specific CERT secure coding rules.

2.2 COMPASS / ROSE

ROSE is a source-to-source framework for source code transformations. Compass will be included in the next release of ROSE, but can work with the current release.

Compass is an open source tool designed to easily evaluate arbitrary code using existing rule sets. Users can also easily configure Compass to use their own domain-specific rules sets. Compass is simple to extend: Users can add rules by completing the following steps:

1. Add a name for the checker to the script used to generate code template.
2. Fill in the checker code template with approximately a dozen lines of code to define the code pattern (all in C++ and using the high-level AST IR nodes from ROSE). The amount of code depends on the rule to specify.

3. Use the Latex page (generated in the aforementioned directory) to document the checker.
4. Provide test code that demonstrates the problem (which will be used in the documentation and in testing).
5. Tar up the directory and run the script to submit the checker (the script copies the checker tarball to a common directory where all the contributed checkers are stored).

Separately (at the other end of the submission process), all the submitted checkers can be automatically assembled into Compass:

1. Run the script that automatically assembles all the checker tarballs into an existing version of Compass.
2. Run "make" to complete the process, run "make docs" to build the documentation, run "make test" to test the whole process.

The process is designed to support the contribution of checkers by many users. Compass can be run in either text or GUI mode. The GUI mode was created in collaboration with Imperial College London, where a former student developed a Qt-based, ROSE-specific GUI builder.

For example, Compass/ROSE can be used to check for compliance with SIG30-C (“Only call async-safe functions within signal handlers”) using the following algorithm:

1. Assume an initial list of async-safe functions. This list would be specific to each OS, although POSIX does require a set of functions to be async-safe.
2. Add all application-defined functions that satisfy the async-safe property to the async-safe function list. Functions satisfy the async-safe property if they (a) only call functions in the list of async-safe functions, and (b) do not reference or modify external variables except to assign a value to a volatile static variable of `sig_atomic_t` type, which can be written uninterruptedly. This handles the interprocedural case of calling a function in a signal handler that is itself an async-safe function.
3. Traverse the abstract syntax tree (AST) to identify function calls to the signal function `signal(int, void (*f)(int))`.
4. At each function call to `signal(int, void (*f)(int))`, get the second argument from the argument list. To make sure that this is not an overloaded function, the function type signature is evaluated and/or the location of the declaration of the function is verified to be from the correct file (because this is not a link-time analysis it is not possible to test the library implementation). Any definition for `signal()` in the application is suspicious, because it should be in a library.
5. Perform a nested query on the registered signal handler to get the list of functions that are called. Verify that each function being called is in the list of async-safe functions. To avoid repeatedly reviewing each function, the result of the first test of the function should be stored.
6. Report any violations detected.

For performance reasons, Code patterns in Compass/ROSE are specified directly on the AST. One can also specify patterns on the control flow or any of the other program analysis graphs in ROSE: call graph, SDG, etc. This solution is not ideal, because it is ROSE specific, but it permits the level of detail required to specify complex code patterns (mostly because ROSE does not normalize the AST, so all source-level details are provided).

In the design of the attribute-grammar-based AST traversals, apparently separate traversals of the AST can be alternatively executed within a single traversal with a performance improvement of about 110X. Also, current work has demonstrated that this can be additionally parallelized with good efficiency (multi-core optimization).

ROSE has an easy-to-use and intuitive architecture, which makes it possible to quickly implement straightforward rules. ROSE also retains considerable information about the compiled code, to the extent that it is even possible to write rules to check items such as indentation and commenting.

ROSE exhibited the following shortcomings, but these are not considered severe:

1. The documentation is well-structured but incomplete. Consequently, one must often read the code to determine what something is or does.
2. The structure of the type system model is at a low-level and somewhat opaque. This is mostly due to the OO nature of the type classes. To address this issue, we wrote a thin layer over the type system that makes it easier to use and insulates checkers from changes in the implementation. This approach can be extended to write generalized utility functions to provide more high-level information about the AST.
3. The ROSE data structures are not easily discoverable. ROSE alleviates this with utilities to display an AST as a graph or node list with attributes, but the data structures are too complex and fluctuating for these utilities to display everything.
4. There are some minor, but distracting, issues regarding use of const member functions in the implementation.
5. ROSE's ability to recognize the use of macros remains largely experimental because macro processing takes place in an earlier stage than parsing.
6. A few bugs remain.
7. Because some portions of the implementation remain in flux, the currently recommended usage is not always apparent.

Despite these shortcomings, the overall quality of the tool is good, and as we become more familiar with it, we should be able to write checkers fairly quickly. The learning curve is surmountable, but it is steep at the beginning.

One repair strategy for issue (2) is to write a thin layer over the type system that makes it easier to handle and insulates checkers from changes in the implementation. This can be extended to writing generalized utility functions to provide more high-level information about the AST.

3 Project Analysis

Software Research Associates, Inc. (SRA) was selected as the commercial software developer for the study. Founded in 1967, SRA is one of Japan’s oldest and largest independent software firms. SRA has a strong presence in the software product market, distributing and supporting many communications, database and Internet/World Wide Web applications.

SRA evaluated the use of supplemented versions of the Fortify SCA and Compass/ROSE on the two projects described in Table 2.

Project	**Language**	**OS**	**Complier**	**Size (KLOC)**
Toll Collection system related GUI application	C++ (Qt base GUI)	Miracle Linux	GCC 3.2.3	264
Video Service Communication Protocol	C	Cent OS (Linux)	GCC 3.4.6	30

Table 2: Project selection.

These two software development projects are aimed to develop particular software products for the private sector, and these products are intended to be used within production environments.

3.1 MEASURING AND ANALYSIS

Each of these two projects was analyzed using versions of the Fortify SCA and Compass/ROSE supplied with checkers for the CERT Secure Coding rules. Each tool identified a number of defects, and these defects were categorized as *positives*. To the extent of their available time and abilities, developers familiar with each project analyzed each positive to determine if they were *true positives*, *false positives*, or *false negatives*. Because the tools do not make any positive assertions about code correctness, the notion of *true negatives* was considered an anomalous case and only applied to code not identified by either tool as defective.

True positives are defects correctly identified by the tool as such; false positives are incorrectly identified defects. False negatives are defects not detected by a particular tool and must be discovered by other means, such as the use of additional tools, or by manual inspection and testing. False negatives of particular interest in this project are defects that were true positives from one tool but undetected by the other. In all cases, the developers performing the analysis recorded the manner in which the defect was detected. True negatives are only useful for comparison with other tools that incorrectly identify false positives to indicate that these tools were not fooled.

For the purpose of this study, all of the true positive results are, by definition, *software defects*. Due to the nature of the defects these tools are configured to detect, we presume that they at least potentially constitute *security flaws*. A determination of whether the security flaws discovered as a result of this analysis actually constitute practical *security vulnerabilities* is outside the scope of this study [Seacord 05]. Because access to the product source code was not available to the authors, determinations about the accuracy of the results were made solely by the staff at SRA.

Because of time constraints, checkers were not implemented for each secure coding rule. Consequently, the rules for which checkers were implemented had to be selected.

The rules in the C coding standard have been assigned high, medium, and low priorities. A high priority rule is one whose violation is likely to result in exploitable code, the exploit is likely to have severe consequences (for example, allowing an attacker to run arbitrary code), or one that is difficult or expensive to fix manually. The rules in the C++ secure coding standard are currently being prioritized, but many rules have not yet been assigned a priority.

The Fortify SCA analysis engines were extended to flag all possible rules in the C coding standard. Every C coding standard rule was examined with Fortify SCA. All those already flagged by Fortify SCA were identified. Each of the remaining C coding standard rules were evaluated to determine if a rule could be written to extend Fortify SCA.

The Fortify SCA analysis engines were also extended to flag all possible rules in the C++ coding standard. Fortify SCA with the extended rule set created to flag the C++ standard rules was used to examine every C++ coding standard rule. C++ coding standard rules not flagged were evaluated to determine if a rule could be written to extend Fortify SCA.

The C and C++ coding standard recommendations were evaluated separately after the C and C++ coding standard rules were evaluated. Attempts were made to evaluate every C and C++ coding standard recommendation in the order they appear in the secure coding standards, starting with the C coding standard. Efforts concluded, due to time constraints, with the C coding standard recommendations in the "INT" category. Any of the implemented C coding standard recommendations that applied to C++ coding standard was noted as well.

ROSE-checker development for the C coding standard focused solely on high-priority rules. Due to schedule constraints, approximately half of the high-priority rules were implemented. Rules that seemed difficult or unenforceable were ignored. No other criteria were used in the selection of which high-priority C rules to implement.

The rules in the C++ secure coding standard are currently being prioritized, but many rules have not yet been assigned a priority. Consequently, prioritization could not be used to determine which C++ rules to implement. The C++ rules that were implemented consisted of

- Rules that were clearly easy to implement (for example, *EXP03-A: Do not overload the & operator*).

- Rules that relied on type information and class hierarchy analysis (for example, *OBJ03-A: Do not overload virtual functions*). These all relied on a small body of infrastructure that, once implemented, made the rules as a group easy to implement.
- A few random rules that were singularly challenging but implementable (for example *ERR01-A: Prefer special types for exceptions*).

4 Results

This section contains the results of the analysis of the Toll Collection system GUI application and Video Service Communication Protocol projects using both the Fortify SCA and ROSE static analysis tools. The study focused on CERT rules checkers implemented as part of this study. More checkers were implemented for Fortify than ROSE, primarily because this effort was started while ROSE was still being evaluated for suitability in the study.

Project	Lang	Tool	Rule Group	Checkers		
				Implemented	Detected	Analyzed
Toll Collection	C++	Fortify	CERT	23	8	8
			default	?[2]	21	0
		ROSE	CERT	15	6	3
			default	?[2]	14	0
Video Service	C	Fortify	CERT	38	20	7
			default	?[2]	23	0
		ROSE	CERT	12	3	3
			default	?[2]	6	0

Table 3. Summary of rules analyzed.

4.1 Fortify Results

Table 4 shows the total number of reports generated by Fortify SCA for the Toll Collection and Video Service Communication projects. The results are separated by project and rule set. The “default” rule set consists of checkers shipped by default with Fortify SCA. The CERT rule set consists of checkers for CERT secure coding rules developed as part of this study.

Project	Language	Rule Set	Total
Toll Collection	C++	CERT	186
Toll Collection	C++	default	572
Video Service	C	CERT	408
Video Service	C	default	396

Table 4. Total Reports.

[2] It is difficult to determine precisely how many rules ship with Fortify SCA because of the way they are listed and grouped. We eventually omitted this value because it was not essential to the study. The number of default ROSE rules was also omitted, for the same reasons.

For these two projects, the CERT rule set generated a higher number of reports for the C language project than the Fortify default rule set (408 to 396). Conversely, 572 reports were generated by the Fortify default rule set for C++ compared to 186 reports generated by the CERT C++ rule set. These results may be explained by the limited number of CERT C++ checkers implemented for Fortify.

Table 5 illustrates the results for both the Toll Collection and Video Service Communication projects when analyzed with the Fortify SCA tool for compliance with the CERT C and C++ Secure Coding rules.

Project	Language	Total	True +	False +	Unknown	True+ Rate
Toll Collection	C++	186	125	61	0	67%
Video Service	C	408	90	100	218	47%

Table 5. CERT rules checked with Fortify SCA.

Checkers for C++ rules had a true positive rate of 67 percent, while the C rules had a true positive rage of 47 percent. All of the C++ reports in this study were validated, while only 190 out of 408 of the C reports were validated because of time constraints. The 47 percent figure is based on the 190 validated reports.

4.1.1 CERT C++ Secure Coding Standard

Violations of 8 of the 23 CERT C++ rules (37.5 percent) implemented for Fortify were detected.

The true positive rate for these checkers was mixed, as shown in Table 6.

Rule	Total	True +	False +	Unkown	True + Rate
INT35-C	4	4	0	0	100%
RES32-C	8	8	0	0	100%
INT06-A	88	88	0	0	100%
INT31-C	12	7	5	0	58%
INT32-C	38	15	23	0	39%
DAN34-C	33	3	30	0	9%
INT33-C	1	0	1	0	0%
DCL30-C	2	0	2	0	0%

Table 6. Analysis of Fortify CERT C++ secure coding rules.

A large number of violations of CERT C++ rule INT06-A ("Use `strtol()` or a related function to convert a string token to an integer") were detected with a true positive rate of 100 percent suggesting that the checker is accurate in detecting violations of this rule and that the developers may not have been previously aware of this recommended coding practice. Violations of some of these rules, such as DAN34-C ("Do not dereference invalid pointers") can have rather severe consequences as shown in Table 7.

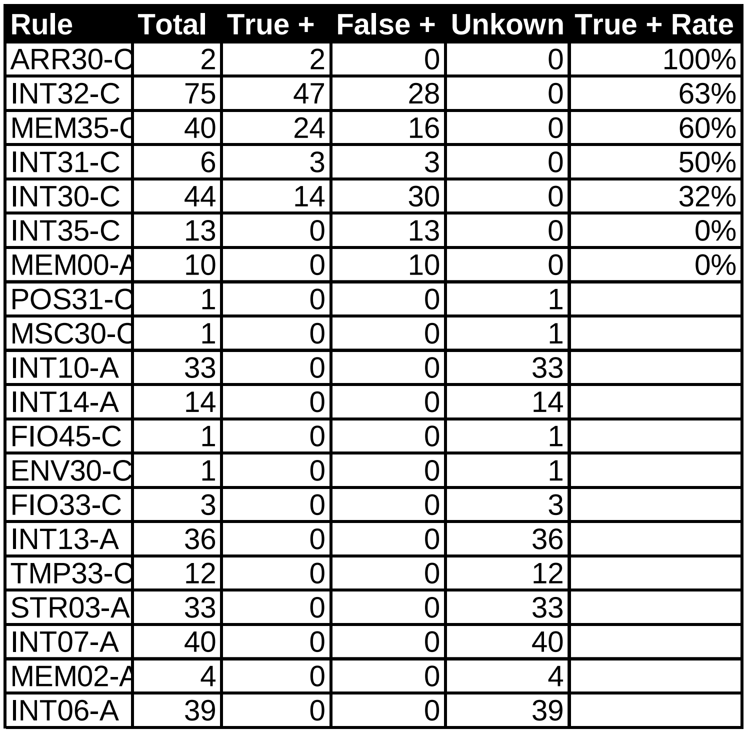
Rule	Severity	Likelihood	Remediation Cost	Priority	Level
DAN34-C	3 (high)	3 (likely)	1 (high)	P9	L2

Table 7. Risk analysis for DAN34-C.

Unfortunately, the checker for DAN34-C had a low true positive rate of 9 percent. The cause of this low positive rate is the use of the exception-handling code block to catch failing new operations and Fortify's inability to detect situations in which memory allocation errors are caught by the exception-handling code block. This accounted for all 30 false positives of DAN34-C analysis, indicating that the false positive rate of DAN34-C could be significantly reduced if the condition is properly handled.

4.1.2 CERT C Secure Coding Standard

Table 8 shows the results of evaluating the Video Service Communication project using Fortify SCA for compliance with the CERT C Secure Coding standard.



Rule	Total	True +	False +	Unkown	True + Rate
ARR30-C	2	2	0	0	100%
INT32-C	75	47	28	0	63%
MEM35-C	40	24	16	0	60%
INT31-C	6	3	3	0	50%
INT30-C	44	14	30	0	32%
INT35-C	13	0	13	0	0%
MEM00-A	10	0	10	0	0%
POS31-C	1	0	0	1	
MSC30-C	1	0	0	1	
INT10-A	33	0	0	33	
INT14-A	14	0	0	14	
FIO45-C	1	0	0	1	
ENV30-C	1	0	0	1	
FIO33-C	3	0	0	3	
INT13-A	36	0	0	36	
TMP33-C	12	0	0	12	
STR03-A	33	0	0	33	
INT07-A	40	0	0	40	
MEM02-A	4	0	0	4	
INT06-A	39	0	0	39	

Table 8. Analysis of Fortify CERT C secure coding rules.

Many of these results were not fully analyzed because of time constraints. Of the rules that were analyzed, both MEM00-A, INT30-C, and INT35-C had noticeably poor results. False positives for MEM00-A stem from a limitation in Fortify. If `malloc()` is called but `free()` is not called within the same function, Fortify cannot determine if `free()` is called.

The low true positive rates for INT30-C, and INT35-C are still being analyzed.

4.2 Rose Results

Table 9 shows the total number of reports generated by ROSE on the two projects. The results are separated by project and rule set. The default rule set consists of checkers that are shipped by default with ROSE. The CERT rule set consists of checkers for CERT secure coding rules developed as part of this study.

Project	Language	Rule Set	Total
Toll Collection	C++	CERT	200
Toll Collection	C++	default	1476
Video Service	C	CERT	7
Video Service	C	default	70

Table 9. ROSE total reports.

The CERT checkers generated far fewer reports than the default ROSE checkers. Also, the CERT C rule set generated far fewer reports for the Video Service Communication project than the C++ rule set generated for the Toll Collection project. This is because the ROSE C++ checkers generated many reports on external header files, while the C checkers did not. Many of the header file reports were duplicates, because header files are often imported into multiple program files. Consequently, a report in one header file would appear once for every program file that included it.

Table 10 illustrates the results broken down for both the Toll Collection and Video Service Communication projects when analyzed with the ROSE tool for compliance with the CERT C and C++ Secure Coding rules.

Project	Language	Total	True +	False +	Unknown	True+ Rate
Toll Collection	C++	200	19	51	130	27%
Video Service	C	7	5	2	0	71%

Table 10. CERT rules checked with ROSE.

Table 11 provides more detail, by listing the ROSE rules that flagged violations and their rates of success. ROSE reported violations of 6 of the 15 CERT C++ rules implemented, and 3 of the 12 CERT C rules implemented. The 6 C++ rules were ERR01-A, ERR02-A, OBJ32-C OBJ00-A, OBJ03-A, and RES35-C, and the C rules were MSC33-C, ARR00-A, and STR31-C.

Rule	Total	True +	False +	Unkown	True + Rate
OBJ32-C	23	9	0	14	100%
ERR02-A	18	10	8	0	56%
ERR01-A	43	0	43	0	0%
OBJ00-A	7	0	0	7	
OBJ03-A	72	0	0	72	
RES35-C	37	0	0	37	
ARR00-A	3	3	0	0	100%
MSC33-C	1	1	0	0	100%
STR31-C	3	1	2	0	33%

Table 11. CERT rules checked with ROSE.

The C results are encouraging, as only one rule had any false positives. Two rules yielded no false positives: ARR00-A ("Be careful using the `sizeof` operator to determine the size of a type") and MSC33-C ("Do not use the `rand()` function for generating pseudorandom numbers"). Both indicate fairly clear patterns; ARR00-A catches use of a specific coding idiom in improper places, and MSC33-C catches any usage of the `rand()` function. In addition, ARR00-A can have severe consequences, as shown in Table 12:

Rule	Severity	Likelihood	Remediation Cost	Priority	Level
ARR00-A	3 (high)	2 (medium)	3 (low)	P9	L2

Table 12. Risk analysis for ARR00-A.

ROSE also reported violations in many external header files. These files were part of the operating system, or ROSE itself, or external software packages used by Toll Collection. Due to time constraints, we did not analyze these violations and report them as "Unknown." It might be prudent to filter out ROSE errors on external header files. However, header files that are part of the project being analyzed should definitely not be excluded, because they will usually contain the class definitions, template definitions, and namespaces, which are a rich source of coding rule violations. See the next section for a cursory analysis of header-file reports.

The reports on C and C++ files are fairly encouraging. Three of our six rules reported no false positives and several true positives. One rule (ERR01-A) reported 43 false positives, out of a total of 53 false positives. Because ERR01-A reported no true positives, removing the rule would have improved the overall success rate significantly without any cost. The results were kept, however, to preserve the integrity of the study. Fortunately, every other rule that had false positives also had true positives.

Two rules (ERR02-A and STR31-C) had both false positives and true positives. Fortunately, they both had fairly few invocations and a promising success rate (56 percent and 33 percent, respectively). Inspecting each report of these rules was consequently a manageable task. Checkers for C++ rules had a true-positive rate of 27 percent, while the C rules had a true-positive rate of 71 percent. All 7 of the C reports in this study were analyzed, while, because of time constraints, only 70 out of 400 of the C++ reports were analyzed. The low true positive rate of 27 percent was mainly due to the rule ERR01-A, which alone provided 43 of the 51 false positives.

Upon further analysis, we discovered the cause of the false positives in ERR01-A and ERR02-A. These rules state that one should neither throw, nor catch, data objects except those that inherit from `std::exception`. The false positives were indeed throwing and catching standard exceptions provided in the standard C++ include files, but our diagnostic code failed to recognize the exceptions as valid. We will fix the code for future analysis, so these false positives should not recur.

There was exactly one secure coding rule that both Fortify SCA and ROSE reported violations on. Table 13 presents both ROSE and Fortify's analyses of this rule:

Project	Tool	Rule	Filename	Line Numbe	Fortify CERT	ROSE CERT
IPTV	FORTIFY	MSC30-C	File00039.c	44	?+	T+
IPTV	ROSE	MSC33-C	File00039.c	44	?+	T+

Table 13. Rules reported by both Fortify and ROSE.

The slightly different names (MSC30-C and MSC33-C) actually refer to the same rule, which is correctly labeled MSC30-C. Both Fortify and ROSE identify a single violation of this rule; both identify the same file. The ROSE report was considered a true positive, while the Fortify report was not analyzed. It is reasonable to assume Fortify and ROSE are reporting the same violation.

The rule in question (MSC30-C or MSC33-C) cautions against using the `rand()` function as a pseudo-random number generator, as its results are not sufficiently random. This is an easy rule to enforce, as all one needs to do is search the source code for instances of a `rand()` function call.

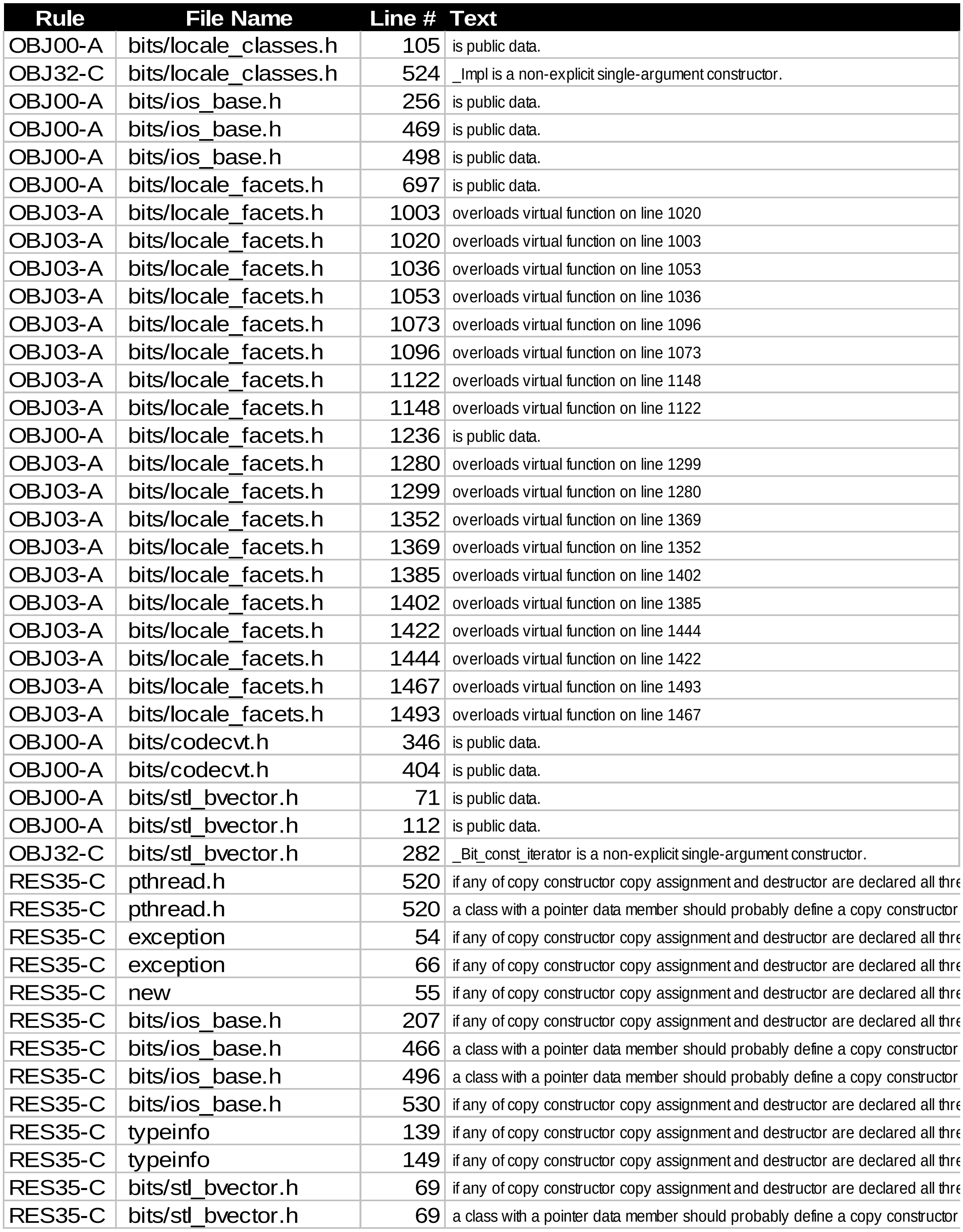
4.2.1 External Header Rose Analysis

Because ROSE issued many reports on C++ header files, we performed a small analysis of ROSE on these files. We did this by running a ROSE diagnostic tool on the tiny program outlined in Figure 2. This program was run on Linux (Ubuntu 7.10), and the header files are provided by Rose (version 0.9.1a), so no compiler was actually involved in the study. The results are shown in Table 14.

```
#include <locale>
#include <vector>

int main() {
  return 0;
}
```

Figure 2. Simple header diagnosis program.



Rule	File Name	Line #	Text
OBJ00-A	bits/locale_classes.h	105	is public data.
OBJ32-C	bits/locale_classes.h	524	_Impl is a non-explicit single-argument constructor.
OBJ00-A	bits/ios_base.h	256	is public data.
OBJ00-A	bits/ios_base.h	469	is public data.
OBJ00-A	bits/ios_base.h	498	is public data.
OBJ00-A	bits/locale_facets.h	697	is public data.
OBJ03-A	bits/locale_facets.h	1003	overloads virtual function on line 1020
OBJ03-A	bits/locale_facets.h	1020	overloads virtual function on line 1003
OBJ03-A	bits/locale_facets.h	1036	overloads virtual function on line 1053
OBJ03-A	bits/locale_facets.h	1053	overloads virtual function on line 1036
OBJ03-A	bits/locale_facets.h	1073	overloads virtual function on line 1096
OBJ03-A	bits/locale_facets.h	1096	overloads virtual function on line 1073
OBJ03-A	bits/locale_facets.h	1122	overloads virtual function on line 1148
OBJ03-A	bits/locale_facets.h	1148	overloads virtual function on line 1122
OBJ00-A	bits/locale_facets.h	1236	is public data.
OBJ03-A	bits/locale_facets.h	1280	overloads virtual function on line 1299
OBJ03-A	bits/locale_facets.h	1299	overloads virtual function on line 1280
OBJ03-A	bits/locale_facets.h	1352	overloads virtual function on line 1369
OBJ03-A	bits/locale_facets.h	1369	overloads virtual function on line 1352
OBJ03-A	bits/locale_facets.h	1385	overloads virtual function on line 1402
OBJ03-A	bits/locale_facets.h	1402	overloads virtual function on line 1385
OBJ03-A	bits/locale_facets.h	1422	overloads virtual function on line 1444
OBJ03-A	bits/locale_facets.h	1444	overloads virtual function on line 1422
OBJ03-A	bits/locale_facets.h	1467	overloads virtual function on line 1493
OBJ03-A	bits/locale_facets.h	1493	overloads virtual function on line 1467
OBJ00-A	bits/codecvt.h	346	is public data.
OBJ00-A	bits/codecvt.h	404	is public data.
OBJ00-A	bits/stl_bvector.h	71	is public data.
OBJ00-A	bits/stl_bvector.h	112	is public data.
OBJ32-C	bits/stl_bvector.h	282	_Bit_const_iterator is a non-explicit single-argument constructor.
RES35-C	pthread.h	520	if any of copy constructor copy assignment and destructor are declared all thre
RES35-C	pthread.h	520	a class with a pointer data member should probably define a copy constructor
RES35-C	exception	54	if any of copy constructor copy assignment and destructor are declared all thre
RES35-C	exception	66	if any of copy constructor copy assignment and destructor are declared all thre
RES35-C	new	55	if any of copy constructor copy assignment and destructor are declared all thre
RES35-C	bits/ios_base.h	207	if any of copy constructor copy assignment and destructor are declared all thre
RES35-C	bits/ios_base.h	466	a class with a pointer data member should probably define a copy constructor
RES35-C	bits/ios_base.h	496	a class with a pointer data member should probably define a copy constructor
RES35-C	bits/ios_base.h	530	if any of copy constructor copy assignment and destructor are declared all thre
RES35-C	typeinfo	139	if any of copy constructor copy assignment and destructor are declared all thre
RES35-C	typeinfo	149	if any of copy constructor copy assignment and destructor are declared all thre
RES35-C	bits/stl_bvector.h	69	if any of copy constructor copy assignment and destructor are declared all thre
RES35-C	bits/stl_bvector.h	69	a class with a pointer data member should probably define a copy constructor

Table 14. CERT C++ rules checked with ROSE in standard header file.

The OBJ00-A rule forbids class-member data being declared public. In C++ a `struct` is merely a class with all members public. Accordingly, the OBJ00-A reports are all of `struct` member declarations.

OBJ03-A stipulates that virtual functions should not be overloaded, lest a derived class override only some of the overloaded virtual function, which “hides” the others. The `locale_facets.h` defines classes and templates for character types, and several classes provide `do_widen()`, `do_narrow()`, `do_toupper()`, and `do_tolower()` methods to operate on their respective character types. Each method is virtual and overloaded; each class provides one method to act on a single character, and one method to act on a range of characters. In this case, the header is clearly designed in violation of this rule. Thus, the rule may be wrong, or the header may be wrong, but the checker is correctly enforcing the rule as specified.

OBJ32-C stipulates that all single-argument constructors should be declared `explicit`. This prevents unexpected type conversions by forcing the developer to convert types by using explicitly-specified constructors or by type-casting. The OBJ32-C reports are non-explicit, single-argument constructor declarations internal to vectors. This suggests that OBJ32-C should be reclassified as a recommendation and not a rule.

Finally, RES35-C consists of two components. First, if a class declares a copy constructor, assignment operator, or destructor, it should declare all three methods. Second, a class should also declare all three methods if it contains a pointer. This requires an object to assume some responsibility for any other object it may point to.

The RES35-C reports within `<pthread>` all complain about a class that contains pointer members. The other RES35-C reports all complain about classes that have virtual member functions and consequently virtual destructors, but lack assignment operators or copy constructors.

In all these cases, the checkers faithfully enforced the rules as they are written. But the rules simply do not include certain exceptional cases (for example, virtual destructors), or are interpreted overly strictly (for example, disallowing `structs`), or can be judiciously broken under special circumstances without loss of security (for example, non-explicit 1-arg constructors). Finally, all the rules are considered to be of low severity because violations of these rules tend not to have severe consequences and are unlikely to result in an exploitable vulnerability. They all conform to common C++ style guidelines.

In conclusion, the checkers are indeed performing as advertised, and are indeed signaling rule violations in the header files. However, these violations are not necessarily security vulnerabilities waiting to be exploited but are more indicative of valid exceptions to the rules. Consequently, there is a need to improve the definitions of these rules, adding exceptions where appropriate, and modify the checkers to handle these exceptions accordingly.

5 Summary

Static analysis tools can be effectively used to identify and eradicate software defects that, in many cases, contribute to vulnerabilities in software products. Out-of-the-box versions of Fortify and ROSE are capable of finding software defects in commercially developed code that, left unmitigated, could result in software vulnerabilities. However, this study demonstrated that both of these tools benefited from the development of additional checkers to validate compliance with the CERT C and C++ Secure Coding Standards.

Both Fortify and ROSE can be extended to evaluate source code for violations of CERT secure coding rules and recommendations, because of their ability to perform syntactic (and some semantic) analysis. ROSE has a slight advantage in syntactic analysis, because it does not normalize the syntactic parse tree. Consequently, it can discover violations in areas, such as signal handling, that are not possible to check using Fortify. On the other hand, Fortify has the advantage of shipping with a larger number of default rules and, as a commercial tool, is more polished than ROSE.

Both ROSE and Fortify perform static analysis only. Because of this limitation, it is possible to create code that Fortify and ROSE cannot verify. In fact, it is easy to create code that cannot be easily be verified without actually running the code. This undermines any advantage static analysis might have over dynamic analysis. Because it is not possible to fully validate compliance with the CERT secure coding standards, the study focused on discovering violations of rules that are susceptible to detection using static analysis.

Neither ROSE nor Fortify are able to validate compliance with secure coding rules dealing with preprocessor directives. This is not surprising, because most C compilers are also unable to detect these problems because these directives are handled by a separate preprocessor. There is currently a research project to endow ROSE with macro awareness, but it is not ready for widespread use. Fortify has additional limitations on how it simplifies the parse tree. It eliminates some typecasts, and `sizeof()` operators and signal handling. Consequently, Fortify cannot be used to discover violations or rules and recommendations concerning the use of the `sizeof()` and signals.

The exercise of writing and testing the checkers helps refine the rules themselves, requiring the reexamination of corner-cases and exceptions to the rules. In particular, some of the C++ recommendations may be violated under special circumstances (such as non-explicit single-argument constructors), when the benefits of violating the rule are significant, and judged to outweigh the costs. These cases need to be enumerated as allowable exceptions in the CERT C++ Secure Coding Standard.

Finally, many secure coding rules and recommendations have not yet been implemented in either Fortify or ROSE. Furthermore, refinement of the C and C++ secure coding rules will require us to make corresponding refinements to the checkers.

Overall, the project was successful because the extended source code analysis tools successfully discovered a number of software defects in both projects evaluated, demonstrating the effectiveness of both the CERT Secure Coding Standards and the static analysis tools evaluated in improving software quality. The project was also successful in identifying ways in which both the CERT Secure Coding Standards and the static analysis tools evaluated could be further improved.

Appendix A Fortify C Rules

The information included in this appendix is an artifact of the analysis process and has been included to provide some of the details behind the analysis presented in the main body of this report. This information is neither complete nor definitive and should be used with caution.

Rule	Severity	Progress	Description	Notes
PRE00-A	low	UNABLE	Prefer inline functions to macros.	Fortify analyzes code after preprocessing is done.
PRE01-A	low	UNABLE	Use parentheses within macros around variable names.	Fortify analyzes code after preprocessing is done.
PRE02-A	low	UNABLE	Macro expansion should always be parenthesized for function-like macros	Fortify analyzes code after preprocessing is done.
PRE03-A	low	UNABLE	Avoid invoking a macro when trying to invoke a function.	Fortify analyzes code after preprocessing is done.
PRE05-A	low	UNABLE	Use parenthesis around any macro definition containing operators.	Fortify cannot detect this, code is analyzed after preprocessing.
PRE30-C	low	UNABLE	Do not create a universal character name through concatenation.	Fortify analyzes code after preprocessing is done.
PRE31-C	low	UNABLE	Never invoke an unsafe macro with arguments containing assignment, increment, decrement, or function call.	Fortify analyzes code after preprocessing is done.
DCL01-A	low	UNABLE	Do not reuse variable names in sub-scopes.	Fortify cannot compare contents of two different scopes.
DCL02-A	low	UNABLE	Use visually distinct identifiers.	This can be done with a simple `grep`, Fortify doesn't seem to have anything built in to do this.
DCL03-A	low	UNABLE	Place const as the rightmost declaration specifier.	This can be done with a simple `grep`, Fortify doesn't seem to have anything built in to do this.
DCL04-A	low	UNABLE	Take care when declaring more than one variable per declaration.	This can be done with a simple `grep`, Fortify doesn't seem to have anything built in to do this.

DCL05-A	low	UNABLE	Use type definitions to improve code readability.	This can be done with a simple `grep`, Fortify doesn't seem to have anything built in to do this.
DCL06-A	low	UNABLE	Use meaningful symbolic constants to represent literal values in program logic.	This can be done with a simple `grep`, Fortify doesn't seem to have anything built in to do this.
DCL07-A	low		Include type information in function declarators.	The evaluator could not get any of these code examples to compile. These recommendations may have been addressed by the compiler.
DCL08-A	medium	UNABLE	Declare function pointers using compatible types.	This can be done with a simple `grep`, Fortify doesn't seem to have anything built in to do this.
DCL09-A	low	UNABLE	Declare functions that return an `errno` with a return type of `errno_t` .	Fortify cannot detect the return value of a function.
DCL10-A	medium	UNABLE	Take care when using variadic functions.	This is not possible, as `insert (loc++, ..)` is translated into two statements: `loc_0 = loc++` and `insert(loc_0, ...)`.
DCL30-C	high	FORTIFY PARTIAL	Declare objects with appropriate storage durations.	Fortify catches the case of declaring an array and then returning a pointer to that array within a function. This is no longer possible with Fortify.
DCL32-C	low	UNABLE	Guarantee identifiers are unique.	Fortify does not have a mechanism to deal with this issue properly.
DCL33-C	medium	UNABLE	Ensure that source and destination pointers in function arguments do not point to overlapping objects if they are restrict qualified.	
DCL34-C	medium	UNABLE	Use volatile for data that cannot be cached.	The evaluators do not believe Fortify can do this.
DCL36-C	low	UNABLE	Do not use identifiers with different linked classifications.	This can’t be found with the structural analyzer. Currently, the Fortify front end tries to resolve multiple definitions as well as it can and then presents a consistent view to the analyzers, so a structural rule doesn't even see the multiple definitions.

EXP00-A	low	UNABLE	Use parentheses for precedence of operation.	This can be done with a simple `grep`, Fortify doesn't seem to have anything built in to do this.
EXP01-A	high	UNABLE	Do not take the `sizeof` a pointer to determine the size of a type.	`sizeof` is preprocessed out before Fortify can analyze the code.

EXP02-A	low	UNABLE	The second operands of the logical AND and OR operators should not contain side effects.	Cannot flag on "&&".
				Fortify front-end translates the
				`if ( a && b) ... to if (a)`
				`{ if (b) ... }`
				and '(i++) == max' is translated to 'tmp = i; i = i + 1; tmp == max'
				To handle this case, Fortify needs to reverse the translation which it cannot now do.

EXP03-A	medium	UNABLE	Do not assume the size of a structure is the sum of the sizes of its members.	`sizeof` is preprocessed out before Fortify can analyze the code.
EXP04-A	medium	UNABLE	Do not perform byte-by-byte comparisons between structures.	Can create a semantic rule to flag on `memcmp()`, but can't flag structures being passed that fn.
EXP05-A	low	UNABLE	Do not cast away a const qualification.	Cannot flag on const.
EXP06-A	low	UNABLE	Operands to the `sizeof` operator should not contain side effects.	`sizeof` is preprocessed out before Fortify can analyze the code.
EXP08-A	high	UNABLE	Ensure pointer arithmetic is used correctly.	`sizeof` is preprocessed out before Fortify can analyze the code.
EXP09-A	high	UNABLE	Use sizeof to determine the size of a type or variable.	`sizeof` is preprocessed out before Fortify can analyze the code.
EXP30-C	medium	PARTIAL	Do not depend on order of evaluation between sequence points.	
EXP31-C	low	UNABLE	Do not modify constant values.	

EXP32-C	low	UNABLE	Do not access a volatile object through a non-volatile reference.	
EXP33-C	high	FORTIFY PARTIAL	Do not reference uninitialized variables.	This catches the example code, but doesn't always recognize initialization. If initialization is done in another function, it is not recognized. Unexpected behavior occurs when pointers are used. There are many false positives. Fortify catches as "low : Uninitialized Variable : controlflow."
EXP33-C	high	FORTIFY PARTIAL	Do not reference uninitialized variables.	This catches the example code, but doesn't always recognize initialization. If initialization is done in another function, it is not recognized. Unexpected behavior occurs when pointers are used. There are many false positives. Fortify catches as "low : Uninitialized Variable : controlflow."
EXP34-C	high	FORTIFY	Ensure a pointer is valid before dereferencing it.	
EXP35-C	low	UNABLE	Do not access or modify the result of a function call after a subsequent sequence point.	
EXP36-C	low	UNABLE	Do not cast between pointers to objects or types with differing alignments.	
EXP37-C	low	UNABLE	Call functions with the correct arguments.	The Structural Rule language does not support the predicate required to detect the function call violation because it requires the universal quantifier "forall", which we don't support at this moment.
INT01-A	medium	PARTIAL	Use `size_t` for all integer values representing the size of an object.	This is partially covered by the rule for INT32-C - but Fortify can't flag on type `size_t`, Fortify sees `size_t` as unsigned long.
INT05-A	medium	DONE	Do not use functions that input character data and convert the data if these functions cannot handle all possible inputs.	This is covered by FIO33-C.

INT06-A	medium	DONE	Use strtol() to convert a string token to an integer.	Created a structural rule to flag atoi, atol, and atoll when they're passed strings.
INT07-A	medium	DONE	Explicitly specify signed or unsigned for character types.	Created structural rule to check if int is assigned a char (not unsigned or `signed char`) or if an operation is done with a `char`.
INT09-A	low	UNABLE	Ensure enumeration constants map to unique values.	Can't flag on enum.
INT10-A	low	DONE	Do not make assumptions about the sign of the resulting value from the remainder % operator.	Created a structural rule that flags on percent
INT13-A	high	DONE	Do not assume that a right-shift operation is implemented as a logical or an arithmetic shift.	Able to create a structural rule to flag when a right-shift operation is performed.
INT14-A	medium	PARTIAL	Distinguish bitmaps from numeric types.	Able to create a structural rule to match when an arithmetic operation is preformed in the same line as a bit manipulation operation. Not able to distinguish between operations on positive numbers and operations on negative numbers.
INT30-C	low	PARTIAL	Do not perform certain operations on questionably signed results.	Currently, there's no way to determine the size of a type, but in SCA 5.0 the structural analyzer's "Type" objects will have a "storageSize" property that gives the number of bytes the type occupies. Currently, if you have the Type object for "MyStruct*", you can determine the size of "MyStruct*", but you can't just "remove the pointer" and obtain the size of "MyStruct". To do that, enhancements would be required to make the "Type" object to be a tree structure.
INT31-C	high	DONE	Ensure that integer conversions do not result in lost or misinterpreted data.	Able to create a structural rule that looks for type conversion without checking the variable on the left hand side of the assignment.
INT32-C	high	DONE	Ensure that integer operations do not result in an overflow.	Able to create a structural rule that tests to see if the affected operations are being preformed and there is no "if"

				statement. Further testing revealed many false positives, most of which related to for loops. Added a check to flag only if it's not part of a for loop statement. Not sure whether this is the correct way to address these false positives.
INT33-C	low	DONE	Ensure that division and modulo operations do not result in divide-by-zero errors.	Created rule similar to INT32-C.
INT35-C	high	DONE	Upcast integers before comparing or assigning to a larger integer size.	
INT36-C	high	DONE	Do not shift a negative number of bits or more bits than exist in the operand.	Covered by rule for INT32-C.
INT37-C	low	DONE	Arguments to character handling functions must be representable as an `unsigned char.`	Wrote a structural rule to flag when a `char` handling fn from ctype.h is passed anything but an `unsigned char`.
FLP30-C	low	DONE	Take granularity into account when comparing floating point values.	Created a structural rule to flag when an if statement consists of two floats being compared in the form f == g.
FLP31-C	low	UNABLE	Do not call functions expecting real values with complex values.	Fortify won't flag any of these functions.
FLP32-C	medium	DONE	Prevent domain errors in math functions.	Created a control flow rule to handle this.
FLP33-C	low	PARTIAL	Convert integers to floating point for floating point operations.	Created a structural rule to flag when an assignment statement for a variable of type double or float contains an operation that does not result in either a double or a float. Fortify incorrectly flags the first compliant code solution with the rule written.
FLP34-C	low	DONE	Ensure that demoted floating point values are within range.	Created a structural rule to flag when a double or long double is demoted to float or when a long double is demoted to float and there is no if block to check to make sure the larger variable doesn't contain a value that the smaller can't contain.

ARR00-A	high	UNABLE	Be careful using the `sizeof` operator to determine the size of an array.	`sizeof` is preprocessed out before Fortify can analyze the code.
ARR30-C	high	PARTIAL	Guarantee that array indices are within the valid range.	
ARR31-C	high	UNABLE	Use consistent array notation across all source files.	This can’t be found with the structural analyzer. Currently, the Fortify front end tries to resolve multiple definitions and then presents a consistent view to the analyzers, So, a structural rule doesn't even see the multiple definitions. Can't do this for the same reason we can't do DCL36-C.
ARR32-C	high	UNABLE	Ensure size arguments for variable-length arrays are in a valid range.	Created a structural rule to flag when an array is dynamically allocated and the value is not properly checked. This will flag on the example compliant code because Fortify can't see outside of a single function's scope. This rule generates too many false positives. Fortify can't explicitly detect arrays (verses references to other types of pointers), and this rule ends up flagging on many things that don't fall into the rule.
ARR33-C	high	FORTIFY	Guarantee that copies are made into storage of sufficient size.	Catches the code in the NCCE.
ARR34-C	high	UNABLE	Ensure that array types in expressions are compatible.	Fortify won’t flag on the array access/assignments nor identify the differences in the types in the example code.
STR02-A	medium	FORTIFY	Sanitize data passed to complex subsystems.	Fortify flags the example code as "\[1E605754626A177B9721905D023B495E : medium : Command Injection : semantic \]".
STR03-A	low	DONE	Do not inadvertently truncate a null-terminated byte string.	Created a control flow rule to flag when `strncpy`, `strncat`, `fgets`, or `snprintf` are called and the result is used without a test.

STR05-A	low	UNABLE	Prefer making string literals const-qualified.	Fortify cannot distinguish between `char` and `char const`.
STR06-A	low	FORTIFY	Don't assume that `strtok()` leaves its string argument unchanged.	Fortify catches this with \[CDFD0C4C211178014C47940B7C19EA30 : medium : Missing Check against Null : controlflow \] and \[2F6C99890155DBEB91367CA22A5D7E74 : low : Obsolete : semantic \].
STR07-A	low	PARTIAL	Use plain `char` for character data.	The “” tag was added to rule. Created a structural rule to flag when an `unsigned char` is assigned a String Literal. Only a partial fix because Fortify can't differentiate between `char` and `signed char`.
STR30-C	low	UNABLE	Do not attempt to modify string literals.	Not currently possible in Fortify.
STR31-C	high	FORTIFY	Guarantee that storage for strings has sufficient space for character data and the null terminator.	Fortify flags the `strcpy` and the `getenv` example code with " high : Buffer Overflow : dataflow ", the first bit of example code Fortify can't deal with since it uses `sizeof`.
STR32-C	high	UNABLE	Guarantee that all byte strings are null terminated.	
STR33-C	high	PARTIAL	Size wide character strings correctly.	
STR34-C	medium	DONE	STR34-C. Cast characters to unsigned types before converting to larger integer sizes.	Created a structural rule to flag when an int or long is assigned a `char`.
MEM00-A	high	DONE	Allocate and free memory in the same module, at the same level of abstraction.	Created a control flow rule to catch fns that have just malloc or just free.
MEM02-A	low	DONE	Do not cast the return value from `malloc().`	Created structural rule to flag a variable assignment when the return value of `malloc()` type doesn't match the type being assigned.
MEM30-C	high	FORTIFY	Do not access freed memory.	
MEM31-C	high	FORTIFY	Free dynamically allocated memory exactly once.	

MEM32-C	low	FORTIFY	Detect and handle critical memory allocation errors.	
MEM33-C	low	UNABLE	Use flexible array members for dynamically sized structures.	Fortify can't flag on array declarations inside a struct.
MEM34-C	low	PARTIAL	Only free memory allocated dynamically.	
MEM35-C	high	PARTIAL	Allocate sufficient memory for an object.	Created a structural rule to flag when `malloc()`, `calloc()`, or `realloc()` are called and `multsize_t` is not or when `memcpy()` is called with a length value other than `size_t`. Fortify can't flag on `sizeof`, so won't catch the third noncompliant example.
FIO30-C	high	FORTIFY	Exclude user input from format strings.	
FIO31-C	medium	UNABLE	Do not simultaneously open the same file multiple times.	Fortify cannot detect this.
FIO32-C	medium	FORTIFY	Detect and handle file operation errors.	
FIO33-C	low	DONE	Detect and handle input output errors resulting in undefined behavior.	
FIO34-C	high	DONE	Use int to capture the return value of character IO functions.	
FIO35-C	high	DONE	Use `feof()` and `ferror()` to detect end-of-file and file errors.	
FIO36-C	high	UNABLE	Do not assume a newline character is read when using `fgets()`.	
FIO37-C	high	FORTIFY	Don't assume character data has been read.	
FIO38-C	low	PARTIAL	Do not use a copy of a FILE object for input and output.	Pointer issues leads to false negatives.
FIO39-C	medium	DONE	Do not read in from a stream directly following output to that stream.	
FIO40-C	low	PARTIAL	Reset strings on `fgets()` failure.	
FIO41-C	medium	UNABLE	Do not call `getc()` or `putc()` with parameters that have side effects.	

FIO42-C	medium	DONE	Ensure files are properly closed when they are no longer needed.	
FIO43-C	high	FORTIFY PARTIAL	Do not copy data from an unbounded source to a fixed-length array.	Fortify catches the `gets()` and `scanf()`, but not the `getchar()` example. It remains uncertain how to address the `getchar()` example.
FIO44-C	medium	DONE	Only use values for `fsetpos()` that are returned from `fgetpos()`.	
FIO45-C	medium	UNABLE	Do not reopen a file stream.	Attempted a control flow rule, but this can't be done because it requires a larger scope than Fortify can deal with.
TMP30-C	high	UNABLE	Temporary files must created with unique and unpredictable file names.	
TMP32-C	high	FORTIFY	Temporary files must be opened with exclusive access.	Fortify catches this with "Insecure Temporary File", a semantic rule of low severity. However, it also catches our advised compliant solution (`mkstemp()`) with "Insecure Temporary File".
TMP33-C	medium	DONE	Temporary files must be removed before the program exits.	Created a control flow rule to flag when `tmpfile()`, `fopen()`, `mktemp()`, etc. are called rather than `tmpfile_s()` or `mkstemp()`.
ENV30-C	low	DONE	Do not modify the string returned by `getenv()`.	Check for completion.
ENV31-C	low	UNABLE	Do not rely on an environment pointer following an operation that may invalidate it.	
ENV32-C	low	UNABLE	Do not call the `exit()` function more than once.	
ENV33-C	low	UNABLE	Do not call the `longjmp()` function to terminate a call to a function registered by `atexit()`.	Fortify can't flag this because it cannot compare the contents of two different scopes.
SIG30-C	high	UNABLE	Only call asynchronous-safe functions within signal handlers.	Can't detect signal handlers in Fortify.

SIG31-C	high	UNABLE	Do not access or modify shared objects in signal handlers.	Can't detect signal handlers in Fortify.
SIG32-C	high	UNABLE	Do not call `longjmp()` from inside a signal handler.	Can't detect signal handlers in Fortify.
SIG33-C	low	UNABLE	Do not recursively invoke the `raise()` function.	Can't detect signal handlers in Fortify.
MSC30-C	low	DONE	Do not use the `rand` function.	
MSC31-C	low	UNABLE	Ensure that return values are compared against the proper type.	Fortify can't distinguish between `time_t` and long or `size_t` and unsigned long.
POS30-C	low	UNABLE	Use the `readlink()` function properly.	`sizeof` is preprocessed out before Fortify can analyze the code.
POS31-C	medium	DONE	Do not unlock or destroy another thread's mutex.	Created a control flow rule to flag when a function destroys a lock before acquiring that lock.
POS32-C	medium	UNABLE	Include a mutex when using bit-fields in a multi-threaded environment.	Fortify can't flag on this. There's a scope issue and a preprocessing issue here.
POS33-C	low	FORTIFY	Do not use `vfork()`.	
POS34-C	high	DONE	Do not call `putenv()` with an automatic variable as the argument.	Created a Fortify structural rule to flag when `putenv()` is called with a variable that is not static.

Appendix B Fortify C++ Rules

The information included in this appendix is an artifact of the analysis process and has been included to provide some of the details behind the analysis presented in the main body of this report. This information is neither complete nor definitive and should be used with caution.

Rule	Severity	Progress	Description	Notes
PRE31-C	low	UNABLE	Prefer inline functions to macros.	Fortify analyzes code after preprocessing is done.
PRE32-C	low	UNABLE	Use parentheses within macros around variable names.	Fortify analyzes code after preprocessing is done.
PRE33-C	low	UNABLE	Macro expansion must always be parenthesized.	Fortify analyzes code after preprocessing is done.
DCL01-A	low	UNABLE	Do not reuse variable names in sub-scopes.	Fortify is not able to address scope issues.
DCL02-A		UNABLE	Use visually distinct identifiers.	This can be done with a simple `grep`. Fortify doesn't seem to have anything built in to do this.
DCL03-A		UNABLE	Place const as the rightmost declaration specifier.	This can be done with a simple `grep`. Fortify doesn't seem to have anything built in to do this.
DCL04-A	low	UNABLE	Declare no more than one variable per declaration.	This can be done with a simple `grep`. Fortify doesn't seem to have anything built in to do this.
DCL30-C	low	DONE	Do not use names reserved for the implementation.	Created a structural rule to catch variables named with two underscores or those that begin with an underscore followed by a capital letter.
DCL31-C	low	FORTIFY	Avoid self initialization.	Fortify flags this as "Poor Style : Redundant Initialization : structural".
EXP06-A		UNABLE	Use parentheses for precedence of operation.	This can be done with a simple `grep`. Fortify doesn't seem to have anything built in to do this.
EXP07-A	low	UNABLE	Operands to the `sizeof` operator should not contain side effects.	`sizeof` is preprocessed out before Fortify can analyze the code.

EXP30-C	low	UNABLE	Do not cast away `const`.	Fortify cannot distinguish weather a variable is `const` or not.
EXP31-C	low	UNABLE	Do not modify constant values.	Same as C rule.
EXP32-C	low	UNABLE	Do not access a volatile object through a non-volatile reference.	Same as C rule.
EXP33-C	low	FORTIFY PARTIAL	Do not reference uninitialized variables.	Same as C rule. This catches the example code, but doesn't always recognize initialization. If initialization is done in another function, it is not recognized. Unexpected behavior occurs when pointers are used. Many false positives. Fortify catches as "low : Uninitialized Variable : controlflow".
EXP34-C	medium	UNABLE	Do not depend on order of evaluation between sequence points.	Same as EXP30-C C rule. Need to identify sequence point. Probably overly difficult to implement in Fortify.
EXP35-C	high	DONE	Ensure that the right hand operand of a shift operation is within range.	Not sure whether this is possible in a control flow rule. This is caught with the structural rule for INT31-C - it looks to me like these rules are related enough to allow one rule to catch them both, so I'm going to leave INT31-C to catch it and mark it "DONE"
EXP36-C	medium	PARTIAL	Do not cast or delete pointers to incomplete classes.	Wrote a structural rule that detects when a variable of type \[UnknownType\] is assigned something that is not \[UnknownType\]. There is no way to detect casts in Fortify. It won't detect inside of a class declaration.
EXP37-C	low	PARTIAL	Avoid side effects in assertions.	The evaluator was able to create a structural rule that catches `assert( index+\+ == 0 );` or `assert( index == c.size() );` in a function other than "main", but was not able to ensure that "index" or "c" (in these examples) are parameters of the enclosing function.

INT01-A		PARTIAL	Use `size_t` for all integer values representing the size of an object.	This is partially covered by the rule for INT32-C, but Fortify can't flag on type `size_t`. Fortify sees `size_t` as unsigned long.
INT05-A		DONE	Do not input integer values using formatted input.	This is covered by FIO33-C.
INT06-A		DONE	Use `strtol()` to convert a string token to an integer.	Created a structural rule to flag `atoi`, `atol()`, `atoi()`, `atoll()`, and `atoll()` when they're passed strings.
INT31-C	high	DONE	Ensure that integer conversions do not result in lost or misinterpreted data.	Same as C rule. Able to create a structural rule that looks for type conversion without checking the variable on the left hand side of the assignment.
INT32-C	high	DONE	Ensure that integer operations do not result in an overflow.	Same as C rule. Able to create a structural rule that tests to see if the affected operations are being preformed and there is no if statement.
INT33-C	low	DONE	Ensure that division and modulo operations do not result in divide-by-zero errors.	Same as C rule. Created structural rule similar to INT32-C.
INT35-C	medium	DONE	Do not truncate the return value from a character input function.	Created a structural rule to flag on a function that returns an int to a `char` variable.
FLP30-C		DONE	Take granularity into account when comparing floating point values.	Same as C rule.
FLP31-C	low	DONE	Do not use floating point variables as loop counters.	Created a rule that flags conditional loops that test a float that is changed in the loop before breaking out of the loop.
FLP32-C	low	DONE	Prevent domain errors in math functions.	Same as C rule.
ARR00-A		UNABLE	Avoid using the `sizeof` operator to determine the size of an array	`sizeof` is preprocessed out before Fortify can analyze the code.
ARR30-C	high	PARTIAL	Guarantee that array indices are within the valid range.	Same as C rule.

ARR31-C	high	UNABLE	Use consistent array notation across all source files.	Same as C rule.
DAN30-C	high	UNABLE	Do not refer to an object outside of its lifetime.	Fortify can’t address scope issues.
DAN31-C	high	FORTIFY	Do not access deleted objects.	Caught by \[445B3F3C1AB46D8CC28EA535D6436803 : medium : Use After Free : controlflow \] .
DAN32-C	high	DONE	Do not delete this.	Structural rule to catch function delete when called with variable named this~.
DAN33-C	high	UNABLE	Do not use invalid iterators.	This is the same as STL30-C.
DAN34-C	high	UNABLE	Do not dereference invalid pointers.	Could not get Fortify to flag on `new`.
ERR31-C	low	UNABLE	Destructors must be exception-safe.	Could not get Fortify to flag on `throw`.
RES30-C	low	UNABLE	Never allocate more than one resource in a single statement.	The front-end will introduce temporary variables and convert the non-compliant one to compliant one (the order is implementation-defined). So the structural analyzer cannot match all syntactical patterns in the original code. There is a plan to solve the problems due to introduced temporary variables, but it's not the highest priority in the coming release. The more general rule is "not allow more than one side-effect in the call parameters". This will require more work. The principle behind structural analysis and structural rules is: structural rules are supposed to match exact syntactical patterns. So, it's not possible to match all semantically equivalent code patterns by one structural rule in most cases. To achieve the goal of one rule matching all semantically equivalent code patterns, we need introduce more sophisticated analysis will need to be introduced, which may cost too much overhead.

RES31-C	low	FORTIFY	Perform every resource allocation in its own statement that immediately assigns the resource to an owning object.	The front-end translates
				`int i = xxxx;`
				to
				`int i;`
				`i = xxxx;`
				The dataflow analyzer will mark all used variables that are not initialized in any execution path.
				Fortify catches the NCCE with \[9C8847DF979C3B2462D6E0C7C30BACB2 : low : Uninitialized Variable : controlflow \].

RES32-C	high	DONE	Use new and delete rather than raw memory allocation and deallocation.	
RES33-C	low	Fortify	Object and array delete must be properly paired with the corresponding new.	Caught by \[B0B21546D73D736EF3111D2D80AAA168 : medium : Memory Leak : controlflow \] .
RES34-C	low	UNABLE	Encapsulate resources that require paired acquire and release in objects.	Not sure how to do this; might be possible with a control flow rule.
RES35-C	low	UNABLE	Declare a copy constructor, a copy assignment operator, and a destructor in a class that manages resources.	Fortify cannot detect violations of this rule. Fortify can't do this.
RES36-C	low	UNABLE	Ensure that copy assignment operators do not damage an object that is copied to itself.	Can't get Fortify to flag when a member function is deleting a member variable and then attempting to use the contents of that variable.
RES37-C	low	UNABLE	Release resources that require paired acquire and release in the object's destructor.	Fortify can't flag on the existence (or lack) of an explicit destructor.
RES38-C	low	UNABLE	Do not leak resources when throwing exceptions.	Can't get Fortify to flag on `throw` statement.
RES39-C	low	DONE	Do not use `longjmp().`	Created semantic rule to flag all calls to `longjmp()`.

OBJ30-C	high	UNABLE	Do not use pointer arithmetic polymorphically.	Unclear how to implement.
OBJ31-C	high	UNABLE	Do not treat arrays polymorphically.	See OBJ30-C.

OBJ32-C	high	UNABLE	Ensure that single-argument constructors are marked "explicit".	Attempted a structural rule, but could not tell the difference between a constructor and an explicit constructor. Cannot because we ignore conditional expressions. If it's in assignment statements, for example:
				`Widget * wt;`
				`w1 = 2;`
				Then we can write a structural rule and be written to match this case. To match conditional expressions, a new label "ConditionalExpression" must be defined/implemented.

OBJ33-C	low	PARTIAL	Do not slice polymorphic objects.	Won't flag on a member of class that extends the class of the member to which it is being set equal belongs. One idea is to check the types of the lhs and rhs of an assignment statement. If the types are not primitive, then the assignment might cause object slicing. But the SCA cannot distinguish the assignments in initializations from others in code, so it flags more assignments than necessary.
OBJ34-C	medium	UNABLE	Ensure the proper destructor is called for polymorphic objects.	Fortify can't tell the difference between a derived class and a non-derived class.
BSC30-C	low	DONE	Use the `c_str()` member to retrieve a const pointer to a null-terminated byte string.	Created semantic rule to catch all uses of class `basic_string` member function "data".
BSC31-C	low	DONE	Do not modify the null-terminated byte string returned by the `c_str()` member.	Created control flow rule to flag when the string returned by `c_str()` is altered with `str*cat()` or `str*cpp().`

BSC32-C		DONE	Do not use the pointer value returned by `c_str()` after any subsequent call to a non-const member function.	Created a structural rule to flag when a non-const member function of the basic_string class is called after a pointer value is returned by c_str. `c_str()`.
BSC34-C	high	UNABLE	Range check element access.	The C rule ARR30-C appears unable to test array access in Fortify.
STR30-C	low	UNABLE	Do not attempt to modify string literals.	Same as C rule.
STR32-C	high	FORTIFY	Allocated adequate space when copying bounded strings.	Example code caught by Fortify rule \[577ED976ECB85D475F17575778932434 : high : Buffer Overflow : dataflow \]. This may be a result of the overall sample code.
STR35-C	high	UNABLE	Limit input when reading into a fixed length array.	Attempted to write a control flow rule. Could not flag on `cin` or `operator>>` or >>.
STL30-C	low	UNABLE	Use Valid iterators.	Fortify doesn't seem to distinguish between different types of unary operators. This results from the front end introducing temporary variables. `d.insert (pos++, data[i]+41)` is converted to `t0 = pos++; d.insert(t0, data[i]+41)`. We will be able to match on `t0 = pos++` in the next release, but this is still an internal feature.
STL31-C	high	UNABLE	Use Valid iterator ranges.	Iterators seem to be processed out before Fortify gets to them.
STL32-C	low	UNABLE	Use a Valid Ordering Rule.	Fortify can't flag on this.
MSC31-C	high	UNABLE	Obey the One Definition Rule.	Fortify can't flag on this.

Appendix C C Rules Implemented in Compass Rose

The information included in this appendix is an artifact of the analysis process and has been included to provide some of the details behind the analysis presented in the main body of this report. This information is neither complete nor definitive and should be used with caution.

Rule	Severity	Progress	Description	Notes
DCL30-C	high	PARTIAL	Declare objects with appropriate storage durations.	Rose automatically complains about returning pointer to local variable. Rose could also catch other specific examples, such as assigning an automatic variable to a static pointer.
EXP01-A	high	DONE	Do not take the sizeof a pointer to determine the size of a type.	Rose flags template code: * `T1* x = malloc(sizeof(T2) * y)` `(T1*) malloc(sizeof(T2) * y)` where T1 != T2
EXP08-A	high	NO	Ensure pointer arithmetic is used correctly.	
EXP34-C	high	DONE	Ensure a pointer is valid before dereferencing it.	Rose now ensures that, after receiving a `malloc()` result, a pointer is next used in `==` or `!=` operation (e.g., *`if (ptr == NULL)`*`...`), or a cast-bool operation (e.g., *`if (ptr)`*`...`) Rose doesn't handle cases where ptr is assigned to something besides a simple variable (e.g. struct member, array member, dereference ptr, etc.
INT13-A	high	NO	Do not assume that a right-shift operation is implemented as a logical or an arithmetic shift.	Able to create a structural rule to flag when a right shift operation is performed. Could do the same in Rose, but a >> op per se is not bad. It’s not understood how to check for assumptions about a >> ops' results.

INT31-C	high	PARTIAL	Ensure that integer conversions do not result in lost or misinterpreted data.	Able to create a structural rule that looks for type conversion without checking the variable on the left hand side of the assignment. Rose already throws warnings about sign conversion and integer types. Don't entirely trust it, because these warnings appear whenever `limits.h` is included.
INT32-C	high	NO	Ensure that integer operations do not result in an overflow.	Able to create a structural rule that tests to see if the affected operations are being preformed and there is no "if" statement. Probably doable in Rose, but there will be many uncheckable instances where addition cannot result in overflow. How to limit check to “reasonable” usage?
INT35-C	high	NO	Upcast integers before comparing or assigning to a larger integer size.	AFAICT ROSE does not distinguish between explicit typecasts and implicit typecasts (e.g., promotions done by the compiler). Still, it is possible this rule can be enforced. By limiting scope to equations of the form <exp> <op> (<exp> <op> <exp>), where the outer operation is assignment or comparison, and the inner operator(s) isn't. Overflow checking on the inner operator should normally be mitigated by judicious typecasting on the inner expressions.
INT36-C	high	NO	Do not shift a negative number of bits or more bits than exist in the operand.	We could ensure that any variables used for a << or >> operator previously appear in comparison expressions. This is one rule where dynamic analysis will always fare better than static.
ARR00-A	high	PARTIAL	Be careful using the `sizeof` operator to determine the size of an array.	Rose distinguishes between complete array declarations and incomplete array declarations, but it does not distinguish between incomplete array declarations and pointer declarations. So, we check both that a `sizeof` operand

				type is a pointer (or incomplete array), and that a `sizeof` is the divisor in a divides expression. We don't flag `sizeof(p)` if `p` is an incomplete array or pointer that doesn't live in a divided-by expression It is unclear whether there is a way to do this.
ARR30-C	high	NO	Guarantee that array indices are within the valid range.	The general problem of array bounds checking lends itself to dynamic analysis much better than static analysis.
ARR31-C	high	NO	Use consistent array notation across all source files.	ROSE might be able to do this, assuming it distinguishes between pointer types and incomplete array types. (ROSE's behavior on this changed recently.) Building this for one file is easy, but we will need to apply this to whole projects to catch interfile inconsistencies.
ARR32-C	high	NO	Ensure size arguments for variable length arrays are in a valid range.	Created a structural rule to flag when an array is dynamically allocated and the value is not properly checked. This will flag on the example compliant code as Fortify can't see outside of a single function's scope. We can ensure that a variable used in an array ref was last used in a comparison operator, but that might not be very comprehensive. Also we would need to view multiple files, as a variable could be modified in one file, and then sent to another file's function to declare the array.
ARR33-C	high	NO	Guarantee that copies are made into storage of sufficient size.	Rose catches the code in the NCCE. Another case of ensuring a variable has a reasonable size. This is probably better done dynamically. In this case, we want to ensure the size of memory allotted to arg1 of `memcpy` is large enough to accommodate the size of data (specified in arg3 of `memcpy).` This might be doable statically, but we need an infrastructure to determine if one value is

				greater than other at compile time, which we currently lack.
ARR34-C	high	PARTIAL	Ensure that array types in expressions are compatible.	Rose's default rules already flag incompatible array copies. So does gcc. Flagging variable-length arrays is not done; it probably could be, but not easily.
STR00-A	high	N/A	Use TR 24731 for remediation of existing string manipulation code.	
STR01-A	high	N/A	Use managed strings for development of new string manipulation code.	
STR31-C	high	PARTIAL	Guarantee that storage for strings has sufficient space for character data and the null terminator.	Rose flags if the first arg in `strcpy` is declared a fixed-length array. Doesn't currently support `strcpy_s`. Unclear how to identify the first example (which does a manual `strcpy`).
STR32-C	high	NO	Guarantee that all byte strings are null-terminated.	Complex but doable. Search for those functions that may remove null-termination status from a string (e.g., `strcpy`, `strncpy`, `realloc`, `memcpy`, others?). For any such function, make sure that it is either inside an if statement based on string length (it has `strlen( string)`). Or make sure that the next usage of our string serves to add a null-termination character. That should catch all example code.
STR33-C	high	DONE	Size wide character strings correctly.	This rule is covered by EXP09-A, and by ROSE itself.
STR34-C	medium	NO	Cast characters to unsigned types before converting to larger integer sizes.	We can look for `<int> = <char>` and diagnose if `<char>` is not first typecast to `<unsigned char>` .
STR35-C	high	NO	Do not copy data from an unbounded source to a fixed-length array.	Report on any usage of `gets()`. Also catch any "%s" in `scanf()`. No clue how to catch the `getc()` example.

MEM00-A	high	NO	Allocate and free memory in the same module, at the same level of abstraction.	Easy to create a rule to catch fns that have just `malloc` or just `free`. But, that will catch many false positives. What we need is a 'free-containing' function associated with each 'malloc-containing' function. (This is one area where C++ and RAII wins over C.)
MEM01-A	high	NO	Set pointers to dynamically allocated memory to NULL after they are released.	Add rule to ROSE to ensure that the usage of any pointer after 'free' is on the left-hand-side of an assignment operator (e.g., it is being set to another value, or NULL).
MEM02-A	low	N/A	Do not cast the return value from `malloc()`.	This rule has been changed.
MEM04-A	high		Do not make assumptions about the result of allocating 0 bytes.	Would have to ensure that the value in `malloc arg`, or `realloc arg` is nonzero, which would (probably) require some variable value assertions (see ARR33-C for a similar problem).
MEM07-A	high	NO	Ensure that size arguments to `calloc()` do not result in an integer overflow.	Identifying a potential integer overflow in the `calloc` arguments is moderately difficult. But, even more difficult would be recognizing leading code that would prevent such overflow.
MEM30-C	high	PARTIAL	Do not access freed memory.	Rose now flags any variable used in any function (other than an assignment) after being freed. Does not catch first example; that requires more sophisticated analysis; not sure it's worthwhile; dynamic analysis is necessary for more comprehensive coverage.
MEM31-C	high	NO	Free dynamically allocated memory exactly once.	The ROSE code for MEM30-C may flag double frees, but it should be a simple matter to copy that rule to specifically identify double frees.
MEM35-C	high	NO	Allocate sufficient memory for an object.	Would need to create some tricky math rules to ensure that a multiplication inside a `malloc arg` does not result in

				overflow. More difficult rules in place to recognize preceding code that prevents overflow.
FIO07-A	low	DONE	Prefer `fseek()` to `rewind()`**.**	
FIO12-A	low	DONE	Prefer `setvbuf()` to `setbuf()` .	
FIO30-C	high	NO	Exclude user input from format strings.	It's probably too difficult to ascertain the origin the contents of a variable used as a format string. Probably a sufficient strategy is to flag on usage of a variable format string in `printf` (and other format functions), *unless* that variable's previous usage is to be initialized to a constant string. This hampers i18n, but i18n is pretty vulnerable to format string insecurities already.
FIO34-C	high	NO	Use int to capture the return value of character IO functions.	Going by the examples, the proper rule here would be to flag any implicit typecast of EOF to an unsigned int. Or, ignoring typecasts, any comparison of EOF to an unsigned int. or `char` should be flagged.
FIO35-C	high	NO	Use `feof()` and `ferror()` to detect end-of-file and file errors.	Here, we should flag any comparison of EOF with the result of `getchar()` (or a similar function). Or with a variable that was last assigned the result of `getchar()` (or a similar function). Very similar, but not quite the same rule as FIO34-C.
FIO36-C	high	NO	Do not assume a newline character is read when using `fgets()`.	Don't know any way to know if code is assuming the last character of an `fgets()` invocation is a newline.
FIO37-C	high	NO	Don't assume character data has been read.	Like FIO36-C, a Rose rule would need to know what implicit assumptions the programmer made.

FIO43-C	high	NO	Do not copy data from an unbounded source to a fixed-length array.	Looks like a duplicate of STR35-C.
TMP00-A	high	NO	Do not create temporary files in shared directories.	Lots of easy cases to flag; we can look for `mktemp()` and related functions. We can also look for `const` strings with "/tmp" or "c:/TMP", or other similar patterns. But what would be a compliant code example for this rule? It appears there is currently no right way to do this.
TMP30-C	high	NO	Temporary files must be created with unique and unpredictable file names.	The rule should be fairly easy to enforce, but I'm not confident about the compliant solution. This rule also overlaps a great deal with TMP00-A.
TMP32-C	high	DONE	_Temporary files must be opened with exclusive access.	Rose now flags every instance of `tmpfile()`. It also flags any usage of `tmpnam()` after `fopen()`, `tmpnam_s()` after `fopen_s()`, and `mktemp()` after `open()` (assuming they use the same variable).
ENV01-A	high	NO	Do not make assumptions about the size of an environment variable.	A solution here would be to scan all usage for a string (`char[]` or `char*`) that receives a result of `getenv()`. Do not allow this string to be strcpy'd into another string, unless the 2nd string was allocated with a `malloc` that involved `strlen(string1).`
ENV04-A	high	DONE	Do not call system() if you do not need a command interpreter.	Rose flags all calls to `system()`; can't tell if user needs a command interpreter
MSC30-C	low	DONE	Do not use the rand function.	
POS33-C	low	DONE	Do not use `vfork()`.	
POS34-C	high	DONE	Do not call `putenv()` with an automatic variable as the argument.	Rose flags `putenv()` with array arg. (`ptr arg` is not flagged.) Incorrectly flags static arrays. Incorrectly misses improper ptr usage (but MEM00-a should catch those).

Appendix D ROSE C++ Rules

The information included in this appendix is an artifact of the analysis process and has been included to provide some of the details behind the analysis presented in the main body of this report. This information is neither complete nor definitive and should be used with caution.

Rule	Severity	Progress	Description	Notes
DCL30-C	low	PARTIAL	Do not use names reserved for the implementation.	Currently disabled due to difficulty of configuring rule for each platform. Without configuration, many false positives.
DCL32-C		PARTIAL	Avoid runtime static initialization of objects with external linkage.	Currently disabled due to false positives on extern declarations that are not definitions.
EXP00-A		PARTIAL	Do not use C-style casts.	Currently disabled due to false positives.
EXP02-A		DONE	Do not overload the logical AND and OR operators.	
EXP03-A		DONE	Do not overload the & operator.	
EXP04-A		DONE	Do not overload the comma operator.	
EXP08-A		DONE	A switch statement should have a default clause unless every enumeration value is tested.	
EXP09-A		DONE	Treat relational and equality operators as if they were non-associative non-associative.	
EXP10-A		PARTIAL	Prefer the prefix forms of ++ and --.	Disabled, buggy.
EXP36-C	medium	PARTIAL	Do not cast or delete pointers to incomplete classes.	Disabled, buggy.
EXP38-C		DONE	Avoid calling your own virtual functions in constructor and destructors.	
EXP39-C		PARTIAL	Don't bitwise copy class objects.	Disabled, buggy.

ERR01-A		DONE	Prefer special-purpose types for exceptions.	
ERR02-A		DONE	Throw anonymous temporaries and catch by reference.	
RES35-C	low	DONE	Declare a copy constructor, a copy assignment operator, and a destructor in a class that manages resources.	
OBJ00-A		DONE	Declare data members private.	
OBJ01-A		DONE	Be careful with the definition of conversion operators.	
OBJ02-A		DONE	Do not hide inherited non-virtual member functions.	
OBJ03-A		DONE	Prefer not to overload virtual functions.	
OBJ04-A		DONE	Prefer not to give virtual functions default argument initializer.	
OBJ32-C	high	DONE	Ensure that single-argument constructors are marked "explicit"	

References/Bibliography

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

[Almossawi 06] Almossawi, A.; Lim, K; & Sinha, T. "Analysis Tool Evaluation: Coverity Prevent." Pittsburgh, PA: Carnegie Mellon University, 2006. http://www.cs.cmu.edu/~aldrich/courses/654/tools/cure-coverity-06.pdf.

[CERT 07a] CERT. "CERT C Programming Language Secure Coding Standard." Pittsburgh, PA: Software Engineering Institute, CERT, 2008. https://www.securecoding.cert.org/confluence/x/HQE.

[CERT 07b] CERT. "CERT C++ Programming Language Secure Coding Standard." Pittsburgh, PA: Software Engineering Institute, CERT, 2008. https://www.securecoding.cert.org/confluence/x/fQI.

[CERT 07c] CERT. "CERT Statistics." Pittsburgh, PA: Software Engineering Institute, CERT, 2008. http://www.cert.org/stats/cert_stats.html.

[Chess 02] Chess, B. "Improving Computer Security using Extended Static Checking," *Proceedings of the 2002 IEEE Symposium on Security and Privacy*. Los Alamitos, CA : IEEE CS Press, 2002.

[ISO/IEC 9899-1999] ISO/IEC. *Programming Languages — C, Second Edition* (ISO/IEC 9899-1999). Geneva, Switzerland: International Organization for Standardization, 1999.

[ISO/IEC 14882-2003] ISO/IEC. *Programming Languages — C++, Second Edition* (ISO/IEC 14882-2003). Geneva, Switzerland: International Organization for Standardization, 2003.

[ISO/IEC TR 24731-1-2007] ISO/IEC TR 24731. *Extensions to the C Library —Part I: Bounds-checking interfaces*. Geneva, Switzerland: International Organization for Standardization, April 2006.

[Larochelle 01] Larochelle, D. & Evans, D. "Statically Detecting Likely Buffer Overflow Vulnerabilities," *Proceedings of the 10th Usenix Security Symposium* (USENIX'01). Berkeley, CA: Usenix Association, 2001.

[Seacord 05] Seacord, R. *Secure Coding in C and C++*. New York, NY: Addison-Wesley, 2005.

[US-CERT 08] US-CERT. "US-CERT Technical Cyber Security Alerts." Washington, DC, 2008. http://www.us-cert.gov/cas/techalerts/index.html.

[Wallnau 01] Wallnau, K.; Hissam, S.; & Seacord, R. *Building Systems from Commercial Components*. New York, NY: Addison-Wesley, 2001.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. agency use only (Leave Blank)	2. report date June 2008	3. report type and dates covered Final

4. title and subtitle Evaluation of CERT Secure Coding Rules through Integration with Source Code Analysis Tools	5. funding numbers FA8721-05-C-0003

6. author(s)

Stephen Dewhurst, Chad Dougherty, Yurie Ito, David Keaton, Dan Saks, Robert C. Seacord, David Svoboda, Chris Taschner, Kazuya Togashi

7. performing organization name(s) and address(es) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	8. performing organization report number CMU/SEI-2008-TR-014
9. sponsoring/monitoring agency name(s) and address(es) HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	10. sponsoring/monitoring agency report number ESC-TR-2008-014

11. supplementary notes

12a distribution/availability statement Unclassified/Unlimited, DTIC, NTIS	12b distribution code

13. abstract (maximum 200 words)

This report describes a study conducted by the CERT Secure Coding Initiative and JPCERT to evaluate the efficacy of the CERT Secure Coding Standards and source code analysis tools in improving the quality and security of commercial software projects. In addition to assessing the ability of existing tools to detect violations of the standard, the ability to extend and improve the tools is surveyed. Finally, the use of a selected tool to improve the quality of code in the real-world case of a Japanese software vendor's product is described.

14. subject terms secure coding rules, C++, C, source code analysis tool, CERT	15. number of pages 69

16. price code

17. security classification of report Unclassified	18. security classification of this page Unclassified	19. security classification of abstract Unclassified	20. limitation of abstract UL