ABSTRACT
As cyber-attacks have become more prevalent in the recent decade, companies and governments have learnt the significant importance of enforcing robust programming practices to ensure software security and reliability during code generation. Various tools have been developed for the purpose of assisting programmers in secure coding, and the initial version of our tool called “Secure Coding Assistant” is one of such development efforts. Designed to support CERT rule violation detection, the tool is featured by “providing a mechanism to detect rule violations early” and by “filling the void of open source tools”. The tool is promising in secure programming education compared to other commercial products, however, the initial version does not provide assistance in error correction, nor does it take into account the potentials of employing contract programming enforcement to assist users in improving program reliability. To achieve error correction and defect localization for both software security and reliability in Java programs, this paper presents our efforts for the implementations of assisting error correction and enforcing contract programming. Our tool is maintained on GitHub at http://benw408701.github.io/SecureCodingAssistant/.

Keywords
Robust Programming, Software Reliability, Software Security, Eclipse, Java

1. INTRODUCTION
In the recent decade, cyber-attack has become more prevalent. Based on the statistic published on Statista.com, between 2005 and 2014 millions of data records have been breached in the United Stated [1]. For example, a data breach to Heartland Payment Systems in 2008 and 2009 resulted in 130 million records comprised [2]. JP Morgan Chase, the largest bank in the United States, was the victim of a security breach in 2014 impacting over 76 million household accounts and seven million small businesses [3]. In 2015, Anthem Blue Cross, one of the largest health insurance companies in the US, was attacked resulting in about 78.8 million people’s personal information being stolen [4]. Had the security vulnerabilities been detected at the software development stage, the likelihood of those incidents would have been greatly reduced.

Despite of the security vulnerabilities which are exploited by hackers to compromise the system, low software reliability is another contributor to poor quality of software and often leads to security loopholes. The well-known incident of Ariane 5 in 1996 was the result of poor software reliability, which led to the fatal crash of the rocket shortly after its launch. Investigation conducted by the French Space Agency and European Space Agency pointed to an overflow error introduced in the guidance software converting a 64-bit floating-point horizontal velocity to a 16-bit signed integer. The error led to the shutdown of the guidance system and eventually caused the rocket to veer off course. The catastrophic event of Ariane 5 cost European Space Agency 10 years of R&D effort and 7 billion dollars, making the bug the most costly error in history [5]. The disaster could have been avoided if contract programming methodology had been applied to the guidance software development.

The above incidences call for “robust programming practices” to detect and correct any defects in the code, for the purposes of ensuring both software security (in case of any threats coming from hackers) and reliability (in case of any risks from non-malicious activities)1. The tool named “Secure Coding Assistant” is our response to the community’s demands and efforts. The initial version [6] mainly targets at solving the insecure coding practices, by providing a mechanism based on CERT rules [7] to detect any rule violations at the stage as early as writing code. To fill the void of open source products, the tool is developed as a non-commercial contribution and released through GitHub to promote secure programming education.

However, with the focus on early defect detection, the initial version of Secure Coding Assistant didn’t contain facilities to provide assistance in error correction. It also didn’t provide support to improve code correctness for the sake of risk mitigation in terms of software reliability, where the scenarios may not have any attackers (i.e. the vulnerability is trigger by other factors). Of course, the design rationales at that time didn’t include the potentials of contract programming, which can be integrated with rule violation in one coherent ecosystem to enforce programming practices to ensure software reliability as well as security. This paper presents our efforts in the current version to fill the afore-mentioned gaps, achieving the enforcement of code robustness, based on assisting error correction and contract programming.

This paper is structured as follows: Section 2 reviews related work, where Section 2.2 introduces more details about contract programming. Section 3 outlines the design, Section 4 gives the implementation details, and Section 5 makes the conclusion.

1 Following paper [23], this paper uses the terms “secure programming” and “robust programming” synonymously, meaning actually “secure and robust programming”.

Enhancing Secure Coding Assistant
With Error Correction and Contract Programming
Chen Li, Benjamin White, Jun Dai, Cui Zhang
Department of Computer Science
California State University, Sacramento
6000 J Street, Sacramento, CA 95819, USA
{li3, benwhite, jun.dai, zhangc}@csus.edu

This publication incorporates portions of the “SEI CERT Oracle Coding Standard for Java” (c) 2017 Carnegie Mellon University, with special permission from its Software Engineering Institute”. See Section 6 for more acknowledgements and restrictions.
2. RELATED WORK

2.1 Existing Tools for Enforcing Secure Programming Practices

Fifteen existing static analysis tools, which were developed to support the enforcement of secure coding practices, have been reviewed by our previous work [6]. As shown in Table 1, although nine of them support early detection, others are late in defect localization, and most of them don’t provide much detail regarding vulnerabilities and the mechanisms to find them. The only three open source tools before our initial release are FindBugs [8], ASIDE [9] and PMD [24], with FindBugs lacking support in early detection of rule violation, ASIDE limited in only providing assistance in web application development, and PMD focusing on detection of inefficient code.

Amongst all the tools, only ASIDE supports assistance in automatic fix [9]. However, it follows the OWASP rules and primarily focuses on detecting and fixing vulnerabilities in web application development. In contrast, the Secure Coding Assistant is more generic, following CERT rules [7] and focuses on application development. In contrast, the Secure Coding Assistant primarily focuses on detecting and fixing vulnerabilities in web automatic fix [9]. However, it follows the OWASP rules and among them, Jass and iContract are comment-based tools, jContractor uses additional methods to define contract and performs contract checking using name conversion, while Modern Jass, Contract4J and Cofoja specify the contract via annotation.

Table 1 is the expansion of our previous review of static analysis tools [6]. The highlighted columns compare the newly added features between those tools and our new version.

Table 1. Review of static analysis tools for security vulnerabilities

<table>
<thead>
<tr>
<th>Tool</th>
<th>Early/Late Detection</th>
<th>Open/Closed</th>
<th>Error Correction</th>
<th>Contract Programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>White Box Testing/Binary Static Analysis</td>
<td>Late</td>
<td>Closed</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Fortify Static Code Analyzer</td>
<td>Late</td>
<td>Closed</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Sentinel Source Insight</td>
<td>Late</td>
<td>Closed</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Klockwork</td>
<td>Early</td>
<td>Closed</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SecureAssist</td>
<td>Early</td>
<td>Closed</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Early Security Vulnerability Detector (ESVD)</td>
<td>Early</td>
<td>Closed</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Static Security Vulnerability Analyzer</td>
<td>Early</td>
<td>Closed</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Contrast for Eclipse</td>
<td>Late</td>
<td>Closed</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>SonarLint</td>
<td>Early</td>
<td>Closed</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>CxSuite</td>
<td>Late</td>
<td>Closed</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Goanna Studio</td>
<td>Early</td>
<td>Closed</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>FindBugs</td>
<td>Late</td>
<td>Open</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Coverity Prevent</td>
<td>Early</td>
<td>Closed</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>ASIDE</td>
<td>Early</td>
<td>Open</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>PMD</td>
<td>Early</td>
<td>Open</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Secure Coding Assistant (Current Version)</td>
<td>Early</td>
<td>Open</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

2.2 Existing Tools for Contract Programming in Java

Contract programming is also called Design by Contract [10]. It is an approach which helps producing correct and reliable software by specifying the contracts in terms of pre-conditions, post-conditions for methods, and invariants for class, and by checking the contracts automatically at run time. Eiffle [11] is the first programming language which implemented Design by Contract. Most other languages do not provide this built-in mechanism, however, much effort has been made to provide tools for supporting Design by Contract in other programming languages such as Java [12-17], C# [18], PHP [19].

Specifically, Table 2 shows several Design by Contract tools supported for Java [12-17]. Among them, Jass and iContract are comment-based tools, jContractor uses additional methods to define contract and performs contract checking using name conversion, while Modern Jass, Contract4J and Cofoja specify the contract via annotation.

After evaluating those tools, Cofoja [16] was selected for integration with the new version of Secure Coding Assistant. This tool uses annotations like @Invariant, @Requires and @Ensures to specify the invariant, pre-condition, and post-condition contracts, respectively. It provides run-time checking by utilizing annotation processing and bytecode instrumentation [16]. The rationales for this selection are listed below:

- Cofoja is featured with syntactic checking, which provides instant feedback to developers upon the detection of syntactic error in the contracts;
- The enforcement of Design by Contract methodology requires Secure Coding Assistant to be able to analyze the contract. Annotation-based tools have such advantages over comment-based tools in locating and extracting contracts;
- Defining contract via annotations is more elegant, and brings the benefit of code readability and ease of testing;
- Unlike Modern Jass, which doesn’t provide Window version of the Eclipse plug-in jar file [20], Cofoja is compatible with any system installed with JDK 6 or higher.

Table 2. Existing tools for Design by Contract in Java

<table>
<thead>
<tr>
<th>Tool</th>
<th>Implementation</th>
<th>Support Contract Syntactic Checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jass</td>
<td>comment</td>
<td>No</td>
</tr>
<tr>
<td>Modern Jass</td>
<td>annotation</td>
<td>Yes</td>
</tr>
<tr>
<td>iContract</td>
<td>comment</td>
<td>No</td>
</tr>
<tr>
<td>jContractor</td>
<td>contract method</td>
<td>No</td>
</tr>
<tr>
<td>Contract4J</td>
<td>annotation</td>
<td>No</td>
</tr>
<tr>
<td>Cofoja</td>
<td>annotation</td>
<td>Yes</td>
</tr>
</tbody>
</table>

3. Design

3.1 Goal

The goal of the project is to help developers learn and adopt robust programming practices by providing them a mechanism embedded into the Eclipse code development environment. The mechanism is supposed to support assistance to users for both defect localization and error correction, either through rule violation checking or contract programming enforcement.

3.2 Architecture

To build an early detection tool which provides instant feedback to the developers, the new version of Secure Coding Assistant inherits its previous design to run background checking to
monitor code changes and to detect rule violations [6]. For code analysis, the abstract syntax tree (AST) which represents the structure of source code is traversed. When a rule violation is detected, a marker is created at the place where a violation occurs. Any subsequent code changes will clear all the markers created and will trigger a new round of AST node traversal.

In the new version of the Secure Coding Assistant, we find that the detection of the absence of “Design by Contract” can be achieved based on the same design, i.e. AST node traversal aforementioned to provide corrective solutions to developers. Based on this observation, the enforcement of contract programming is also done in terms of rule violation detection. Specifically, rules used for checking the existence of pre-condition, post-condition, and invariant are added to the RuleFactory along with CERT rules. Upon detection of a rule violation, the newly added method getSolutions() will be called and the solutions will be rendered to developers along with the problem description. The workflow of the new version of Secure Coding Assistant is shown in Figure 1. This Figure is the expansion of our previous design [6]. The highlighted blocks show the expansion of our new version.

![Figure 1 Workflow of Secure Coding Assistant.](image)

4. Implementation

4.1 Implementation of “Quick Fix” Feature

Plugin Development Environment (PDE) provided by Eclipse allows developers to extend and customize the development environment [21]. In the initial version, the extension point org.eclipse.jdt.core.compilationParticipant was extended, which enabled the tool to detect the occurrence of an event, such as build action, clean action or reconcile operation, etc., and such events further initiated code analysis for rule violation detection. org.eclipse.core.resources.problemmarker was the other point extended, which allowed the tool to customize a problem marker upon the detection of a CERT rule violation [6].

In the new version of Secure Coding Assistant, an extra extension point org.eclipse.ui.ide.markerResolution is extended to generate solutions to the problem indicated by rule violations. To bind a resolution generator with the problem marker, the ID of org.eclipse.core.resources.problemmarker is assigned to the markerType of org.eclipse.ui.ide.markerResolution. In addition, two new attributes, ruleID and hashCode are added to org.eclipse.core.resources.problemmarker to serve as a solution map key. Once a problem marker is created, the getSolutions() method is invoked with a problem marker object. Then, by appending the hash code value to the rule ID, a key can be easily formed and the ASTRewrite object used to generate the quick fixes can be retrieved.

As an infrastructure that describes changes made to AST nodes, the AST rewriter can translate those modifications into text edits. By clicking on the label that is associated with the text edits, the modification will be applied to the source code [22]. To fit into user needs in different scenarios, solutions are offered as two groups of options as shown by Figure 2: to fix the issue or to skip rule check. By choosing the former one, the original source code will be modified upon the solution selected (Figure 3-top), whereas by choosing the latter one, a @SkipRuleCheck annotation containing the rule name is added before the method (Figure 3-bottom). As a result, all the violations of the specific rule in the method will be ignored and the corresponding problem marker will be removed. If two or more rules are ignored within the same method, the rule names will be listed together in the annotation.

![Figure 2 Secure Coding Assistant quick fix feature](image)

![Figure 3 Secure random generator quick fix results: use SecureRandom (top) or skip rule check (bottom)](image)

4.2 Integration of Design by Contract Methodology

In the new version of Secure Coding Assistant, the enforcement of “Design by Contract” is accomplished by the mandatory usage of Cofoja, which means the tool itself only checks for the presence of the contract annotations against Cofoja’s library, while leaving the syntactic and semantic checking to Cofoja. The same logic used for CERT rule violation detection is applied for checking the presence of the annotations. Three classes implementing the IRule interface are added to the RuleFactory, and each method or type declaration node will be evaluated by calling the violated() method in all the classes to check for inclusion of the annotations of @Requires, @Ensures or
@Invariant based on the node’s type. As shown by Figure 4, a problem marker will be created if a contract annotation is found missing in the node.

Similar to the CERT rule violation, the checking of the contracts can be skipped by adding the variable name or condition to the value array of @SkipInvariantCheck or @SkipConditionCheck, placing the annotation before the class or method declaration. As a result, the specified invariant or contract checking will be waived and the problem marker will be removed accordingly, as shown by Figure 5.

![Figure 4 Class and method without contract definition](image)

@Invariant(("bookRepository != null", "count > 0"))

![Figure 5 Method skip precondition check](image)

In addition, to facilitate the documentation of contracts, the contract annotation and the method signature can be exported by clicking on the “Export Contract Annotation” button under “Secure Coding Assistant” menu, which, as shown by Figure 7, yields a text file with the same name of the class.

![Figure 7 Exporting the contracts and method signatures](image)

5. Limitations, Conclusion and Future Work

This paper presents a coding assistance tool, which supports the detection of CERT rule violations, the enforcement of contract programming, and a one-click feature to help users quickly correct the detected code defects. As an open source development, the tool can serve as a practical and efficient application in educating developers on robust programming practices. However, for proof of concept, current implementation priorities are given to only sample rules in a small subset (about one quarter) of CERT library. In addition, the capability of the quick fix feature has not been evaluated for all cases. The future work will focus on expansion of the rule set, thorough tests of the tool features, as well as education evaluations with the tool applied in possible computer science classes with Java programming.

6. Acknowledgements

Acknowledgements and attributions are given to Carnegie Mellon University and its Software Engineering Institute, as this publication incorporates portions of the “SEI CERT Oracle Coding Standard for Java” (c) 2017 Carnegie Mellon University, with special permission from its Software Engineering Institute”. Any material of Carnegie Mellon University and/or its software engineering institute contained herein is furnished on an “as-is” basis. Carnegie Mellon University makes no warranties of any kind, either expressed or implied, as to any matter including, but not limited to, warranty of fitness for purpose or merchantability, exclusivity, or results obtained from use of the material, Carnegie Mellon University does not make any warranty of any kind with respect to freedom from patent, trademark, or copyright infringement. This publication has not been reviewed nor is it endorsed by Carnegie Mellon University or its Software Engineering Institute. CERT and CERT Coordination Center are registered trademarks of Carnegie Mellon University. Java is a registered trademark of Oracle Inc.

7. References


[12] Bartetzko, D., Fischer, C., Möller, M., and Wehrheim, H., 2001. Jass — Java with Assertions1 1This work was partially funded by the German Research Council (DFG) under grant OL 98/3-1. Electronic Notes in Theoretical Computer Science 55, 2 (2001/10/01), 103-117. DOI= http://dx.doi.org/http://dx.doi.org/10.1016/S1571-0661(04)00247-6.


