J.Natn.Sci.Foundation Sri Lanka 2023 51 (3): 437 - 450
DOI: http://dx.doi.org/10.4038/jnsfsr.v51i3.10201

RESEARCH ARTICLE

Software Engineering

Secure CodeCity: 3-dimensional visualization of software security facets

C Wijesiriwardana[1*], P Wimalaratne[2], T Abeysinghe[2], S Shalika[2], N Ahmed[2] and M Mufarrij[2]
[1] *Faculty of Information Technology, University of Moratuwa, Katubedda, Sri Lanka.*
[2] *University of Colombo School of Computing, Reid Avenue, Colombo 07, Sri Lanka.*

Submitted: 26 September 2022; Revised: 05 January 2023; Accepted: 27 January 2023

Abstract: Over the last few decades, the software industry investigated security best practices to guide software developers in producing less vulnerable software products. As a result, security engineering has emerged as an integral part of the software development lifecycle. With the increase in the number of security vulnerabilities discovered, the software industry encountered challenges finding software security experts. Despite the availability of static code analysis tools to detect security vulnerabilities, they are underused due to several reasons such as inadequate usability and the lack of integration support. For example, such tools are deficient in providing enough information, produce faulty warning messages, and miscommunicate with developers. As a solution, this work presents a conceptual framework and a proof-of-concept visualization tool, Secure CodeCity, as an extension to the CodeCity metaphor, to facilitate security analytics. Secure CodeCity extends the CodeCity metaphor into three different granularity levels in 3-dimensional space, facilitating the vulnerability analysis in different granularities. Thus, software practitioners can use Secure CodeCity to obtain useful security-related information such as *'What is the most vulnerable class/method in a particular software project?'*. A between-subjects design-based user study was conducted with 23 subjects using a set of security-related tasks with two benchmark open-source Apache projects. The evaluation results show that Secure CodeCity surpasses the state-of-the-art security analysis tools in terms of correctness, usability, and time efficiency.

Keyword : Metaphoric visualization, security vulnerabilities, software security, static code analysis.

INTRODUCTION

Over the last few decades, software security has been recognized as an integral part of software product quality. Thus it is identified as a major quality characteristic in ISO/IEC 25010 quality model. Consequently, the paradigm shift of Build Security In has emerged with the idea of incorporating security into each phase of the Software Development Lifecycle (SDLC) (Soman *et al*., 2021). As a result, a series of security best practices threat modelling and static code analysis have been identified concerning each phase of the SDLC.

Detecting software security defects early in the lifecycle is important because security vulnerabilities are most likely to cause disturbances that affect end users. Besides, long-lingering defects are more expensive to fix. Despite the availability of static code analysis tools to detect security vulnerabilities, they are underused due to several reasons such as inadequate usability and the lack of integration support (Abeyrathna *et al*., 2020). For instance, such tools do not provide enough information (Johnson *et al*., 2013), produce faulty warning messages (Christakis & Bird, 2016), and miscommunicate with developers (Johnson *et al*., 2016). Finally, the results produced by security analysis tools must be understandable to developers who are not experts in software security (Tahaei *et al*., 2021).

Thus, it is believed that a better exploration and visualization can be performed on the data in software artifacts, to provide increased awareness of security aspects, such as vulnerability distribution, obtaining and understanding information regarding vulnerability mitigation, and monitoring the evolution of software security. Therefore, novel metaphoric software visualization models or notable extensions to the existing models are required to better comprehend the software projects (Assal *et al.*, 2016).
This paper presents a conceptual framework and a proof-of-concept implementation of a visualization framework

* Corresponding author (chaman@uom.lk; https://orcid.org/0000-0002-1124-425X

that enables software practitioners to visually observe and analyze various security-related perspectives of a software system. It facilitates drilling down the security issues into different abstraction levels. The visualization framework was built by adhering to the underlying concepts of the CodeCity metaphor. This paper, therefore, expects to address the following research questions (RQs):

RQ1: How to visualize class-level and method-level security vulnerabilities in the source code by extending the CodeCity metaphor based on a multi-layered abstraction mechanism?

RQ2: Can such a visualization give rise to an improvement over the state-of-the-art security analysis tools regarding the correctness, usability, and time efficiency when performing security-specific analysis tasks?

The remainder of the paper is organized as follows. The Materials and Methods is presented in Section 2 followed by the Results and Discussion in Section 3. Finally, the Conclusions and Future work are presented in Section 4.

MATERIALS AND METHODS

Software visualization is beneficial in the contexts of software maintenance, reverse engineering, and software evolution analysis. Literature reveals that software visualization has been effective in reducing the difficulty of understanding complex software systems (Merino *et al*., 2018). Different visualization techniques have been proposed in the literature to represent different types of information about software systems. Thus far, techniques to visualize software attributes such as source code (Qayum *et al*., 2022), architecture (Dashuber *et al*., 2021), evolutionary aspects (Hammad *et al*., 2021), dependencies (Liu *et al*., 2021) and static aspects such as hierarchy (Caserta & Zendra, 2010) have been reported.

More than a decade ago, one of the main observations was that efforts in software visualization are not directly aligned with the needs of software developers (Reiss & Renieris, 2005). However, a few attempts have been made to address this issue and inspire developers to adopt visualization by utilizing multiple visualization techniques. It was observed that the lack of association among different visualization approaches is a significant barrier to finding the appropriate visualization technique to use in practice (Shahin *et al*., 2014). Thus, it was noted that most of the software practitioners are unaware of existing visualization techniques. Several research attempts tried to address this issue by examining to which software engineering tasks distinct visualization techniques have been applied. Nevertheless, it was observed that most of these research attempts are still too coarse-grained to find a fitting visualization for the different needs of software practitioners (Merino *et al*., 2018; Wijesiriwardana & Wimalaratne, 2019).

Software visualization techniques

Software visualization techniques can be categorized based on different aspects of software systems. For example, Shahin *et al*. (2014) classified visualization techniques into five groups: graph-based, notation-based, metaphor-based, metrics-based, and other techniques. Early on, the visualization techniques were purely in two-dimensional space, but later evolved into three-dimensional space and virtual/augmented reality applications.

This section attempts to provide a simple, yet useful classification based on two-dimensional and three-dimensional techniques. Furthermore, this classification summarizes the different visualization attributes of software systems such as architecture, evolution, and code structure. Figure 1 provides a taxonomy of visualization techniques together with the corresponding software attributes. This classification would facilitate software researchers in identifying a suitable visualization approach and picking the most suitable tool.

City metaphor: City metaphor (Wettel, 2008) is formulated based on the concept of cities, and it is commonly used to provide a visual overview of the entities of a software system. It allows enhancements to represent information related to these entities. In the visual representation of city metaphor, the cities correspond to packages; the buildings correspond to classes, and objects inside buildings correspond to the methods. Existing streets represent the relationship between classes among buildings. The city metaphor lets developers solve high-level programme comprehension tasks on different versions of a target programme. Further, it was used to identify possible design issues and to study the evolution of software systems (Ardigò *et al*., 2022). The city metaphor has

also been utilized in combination with virtual reality-based interactive visualization tools in immersive 3D environments (Moreno-Lumbreras, 2021). Interactive visualization supported by navigation with the keyboard makes the City metaphor suitable for the visualization of large-scale software systems. Also, its ability to isolate elements allows focusing on different parts of the software system. However, the City metaphor is not suitable for representing complex relationships such as dependencies in a software system. Furthermore, it might encounter performance issues while increasing scalability. This research utilizes the core concepts of the city metaphor with some extensions.

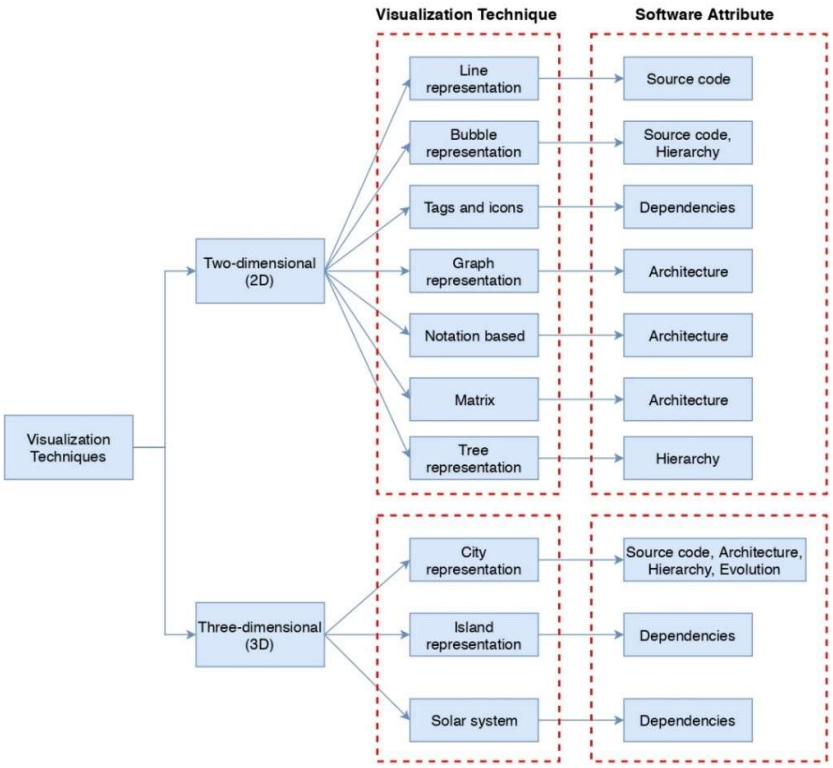

Figure 1: Taxonomy of software visualization techniques

Solar system metaphor: This technique visualizes a software system using a 3-dimensional solar system (Todorov *et al*., 2017). In this way of visualization, the software is represented by a virtual galaxy, consisting of several solar systems. This is capable of applying the consolidated knowledge of software engineers. However, for considerably large software systems that have long names for library packages, the ones placed near the core will become unreadable. Further, overlaps can be observed in the libraries that have data similarities, which could affect visibility.

Island metaphor: Island metaphor intends to visualize software entities such as classes, packages, and components. In this metaphor, an ocean with separate islands is used to represent the software system (Misiak *et al*., 2018). A notable advantage of the island metaphor is that it shows a lot of information in a single view, which is better than displaying information in multiple views. However, in the island metaphor, the information can only be presented on a 2-dimensional space, which makes displaying relationships between components problematic.

Graph representation: This technique (Müller, 2019) is capable of displaying various types of quantitative data of a software system, which is a 2D technique that utilizes graphics according to the plotted data. It can visualize software metrics and the evolution of specific metrics. The graph visualization concept enables software architects to analyze and design changes to software systems visually. Using a consolidated view of the hierarchy and the dependencies of the software system, software architects can reorganize the modular structure interactively.

Matrix-based visualization: This is a well-known 2-dimensional approach that allows for understanding the software evolution process. It is capable of keeping track of changes made to the source code across various versions (Rufiange, 2014). Thus, it is considered a suitable technique for the real-time monitoring of constraint-oriented programmes.

Bubble metaphor: This is a two-dimensional visualization environment, where the bubbles represent the code segments of a software product. The bubble metaphor is suitable for understanding complicated behavior in software systems. Also, it is identified as efficient in comparison to a box or convex hull (Merino *et al*., 2017).

Tree technique: The tree technique visualizes entities of software systems using trees (Yu, 2020). Thus, the hierarchical structure of the source code entities such as method, classes, and packages are represented by different types of trees. The colour, height, and width represent software metrics associated with these entities. The tree-based visualizations can be grouped into two types, called explicit and implicit techniques. For example, the tree map is considered implicit, whereas the node-link diagram is considered explicit. The tree technique provides ease in laying out different information in a structured manner for effective interpretation.

Based on the literature review, it was decided to select the City metaphor as the visualization technique to be used for visualizing security facets due to three main reasons. First, researchers have formulated the city metaphor in different ways based on how the software components and their characteristics are visually described through the city metaphor. Second, the city metaphor significantly affects the users' feelings and emotions. Finally, the users' thinking about city metaphor is positive and comparable with that of other traditional 3D implementations. Apart from the above-mentioned reasons, the city metaphor provides natural support in nailing down the different aspects such as code evolution and quality improvements of software products.

Security vulnerability detection

Static code analysis tools find code vulnerabilities in the source code automatically. While most of the tools give examples of how to resolve the security flaws, some may modify the code to remove the vulnerabilities. Some of the most widely used static code analysis tools are described below.

FindBugs: FindBugs mainly searches for potential problems by matching bytecode against a list of bug patterns. Some of its strengths are that it successfully finds real defects in code, and it has a low rate of detecting false bugs. Among its weaknesses is that it needs compiled code to work, which can be difficult for developers.

Rough auditing tool for security (RATS): RATS is a tool for scanning C, C++, Perl, PHP, and Python source codes. As stated in the name, it only performs a rough analysis of the source code. This tool will not find every error in the source code, and at times produces false positives.

LAPSE+: LAPSE is developed by OWASP, which detects security vulnerabilities specifically for suspicious data injection in Java applications. LAPSE+ facilitates developers as well as auditors to detect vulnerabilities in Java EE Applications. The level of complexity of this analysis grows when the software products are having thousands of lines of code.

Flawfinder: Flawfinder is a code analysis tool that examines C/C++ code and reports possible flaws sorted by risk level. Flawfinder is continually being updated and improved, and has many resources available to help developers use this tool. Flawfinder falls short in its speed in comparison to RATS.

Yasca: Yasca is available for use with many programming languages such as Java, C/C++, JavaScript, HTML, and Visual Basic. Yasca not only detects security vulnerabilities but deviation from best practices in programming.

Proposed approach

CodeCity provides a 3D view of the source code by using a city structure, which it makes sense to manipulate by using the size, shape, and colour. Secondly, it allows using different granularity levels to visualize multiple perspectives of software security, for example, cities having buildings and buildings having rooms. By extending

the 3D space into more levels, Secure CodeCity is also able to provide a game-like environment, thereby encouraging engagement. Secure CodeCity also supports effective visualization of vulnerabilities at different granularity levels. Figure 2 presents the proposed extensions to the CodeCity metaphor.

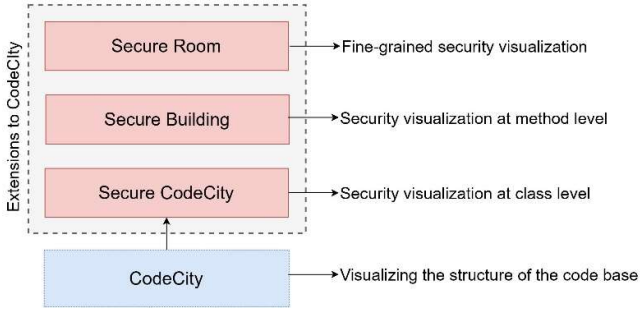

Figure 2: Proposed extensions to CodeCity

Conceptual model for Secure CodeCity

Figure 3 presents the conceptual model of the proposed visualization approach. The proposed model employs static code analysis to identify security vulnerabilities with respect to OWASP vulnerability types. Security aspects in complex software systems can be systematically categorized into different types based on the nature of the security vulnerabilities. This paper is intended to assist software developers by categorizing vulnerabilities by considering the structure of the source code, with a particular focus on object-oriented software systems. Besides, security vulnerabilities can be observed at different levels of a software product (*i.e.*, package level, class level, and method level in object-oriented software systems). Thus, different granularity levels are required to better comprehend the security issues at the aforementioned different levels. To overcome this issue, the proposed visualization model consists of three distinct visualizations namely, Secure CodeCity, Secure Building, and Secure Room.

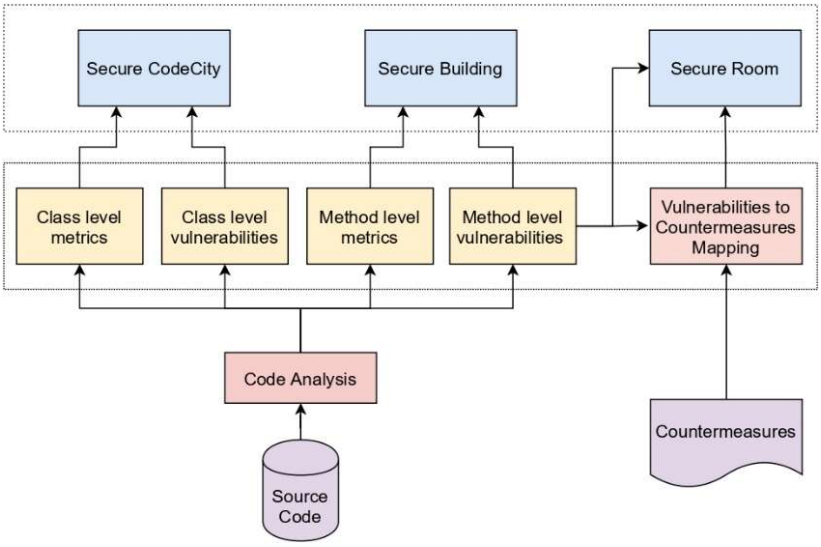

Figure 3: Architecture of visualization

Secure CodeCity : The first level view visualizes the software system as a three-dimensional city, where each building depicts a class of the software system investigated. The footprint size and the height of the building represent Cyclomatic Complexity and the Lines of Code, respectively. Once the city structure is built, the

CodeCity will be augmented with security-related information in the software project. For example, a building will be further augmented with different software metrics such as Severity level of vulnerabilities, Security Remediation Effort, Security Rating, and Cognitive Complexity. Besides, the operations such as zoom, move, and rotate could be performed on the 3D city view.

Secure building: The building view appears upon triggering a Class (*i.e.*, a building, in the city view). As stated previously, the first level is based on the CodeCity concept, where a building represents a Class in the software system. A Class consists of Methods, similar to a building consisting of rooms. Therefore, using a building view in the second level visualization allows users to generate a familiar mental model by minimizing the cognitive loads. This view can identify all the methods that have OWASP vulnerabilities. Besides, this view can share some other useful information such as the number of OWASP vulnerabilities in a particular class and vulnerability distribution among the selected class.

Secure room: Selection of a method in the second level view (*i.e.*, building view) allows navigation to the third level view. The walls of the room are used to represent different aspects of vulnerabilities. All the vulnerability details in the selected Method are shown in the third view, along with the suggested countermeasures to resolve the security bugs in the source code.

City generation and attribute mapping

Like CodeCity , in Secure CodeCity also, the classes are visualized as buildings and packages as districts. This choice is rooted in the fact that classes are the cornerstones of the object-oriented paradigm. However, the attributes mapping of Secure CodeCity is completely different from CodeCity. For example, as stated previously, the footprint size and the height of the building represent Cyclomatic Complexity and the Lines of Code, respectively. Besides, the colour of the building represents the severity level of the vulnerabilities of a particular class.

Positioning of the buildings:

An algorithm is designed to generate a 2D Grid, where the location of each package is determined by using (x,y) coordinates that are calculated based on the root level. Then the exact location of the class in a particular folder is also determined by using (x,y) coordinates that are calculated for the relevant package. Using the class ID and the folder structure, the classes that belong to each folder are separately grouped and positioned. Using the number of lines of codes and the number of attributes in each class, the space required to generate a particular class as a 3D model is calculated.

In the second level view (*i.e.*, the building view), the rooms are assigned a colour based on the following equation as depicted in OWASP risk ranking methodology.

$$Risk = likelihood * impact$$

The likelihood will be calculated based on the factors such as ease of discovery, ease of exploitation, awareness, and intrusion detection.

Secure CodeCity for security visualization

This section describes Secure CodeCity that is implemented to visualize vulnerabilities in software systems. The implementation goals of Secure CodeCity are two-fold: to visualize the security bugs that lie in the codebase and to visualize the interlinked security artifacts. The implementation is based on the theoretical foundation laid in Section 3. As described previously, Secure CodeCity consists of three views: the City view, the Building view, and the Room view to analyze the security vulnerabilities in different granularity levels. The Secure CodeCity framework was developed as a standalone application that renders in the browser. How the data are extracted from different sources and pre-processed to facilitate the visualization is described below.

Data extraction and preparation for visualization

Static code analysis with SonarQube : SonarQube is a static code analysis tool as well as a code quality measuring tool which is widely used among software practitioners and researchers. The Secure CodeCity Framework requires analyzing the source code using SonarQube to identify Security vulnerabilities. Besides, SonarQube provides the required software metrics of the software system. The vulnerabilities identified as security bugs are categorized with respect to OWASP T10 by this tool. Categorization of software bugs into OWASP T10 is the additional reason for selecting SonarQube.

Metrics pre-processor: SonarQube provides an API to get metric details related to the scanned project, where the extracted details could be used to generate the project structure. To store details about the extracted details from SonarQube, TreeElement is used, which is implemented using the TypeScript Classes. However, SonarQube API does not provide a way to get method details inside a class. Therefore, the source code is fed to a Java parser and the method name, size, and method lines details are extracted.

Issues pre-processor : SonarQube provides the facility to extract vulnerability details, Security Remediation Effort, and Security Rating Details of any given File via Sonar API. It sends output details as a JSON object. Each JSON object is processed, and security-related details are stored in the tree node. The details are processed according to the various needs of the project.

Different views in secure CodeCity

As described previously, the first level view visualizes the software system as a three-dimensional city. Figure 4a shows the City view of a software system, with buildings are represented in different colours, where less vulnerable classes are shown in Green colour and highly vulnerable classes are shown in Red colour.

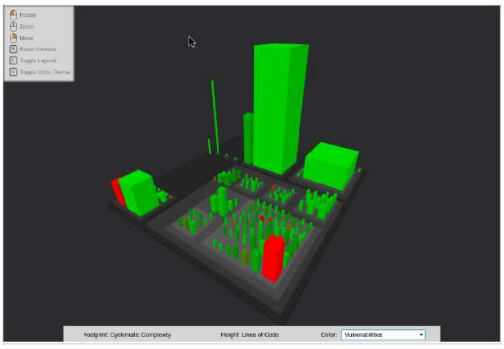

Figure 4a: Security vulnerability distribution of secure CodeCity (Less vulnerable class: Green building; Highly vulnerable class: Red colour)

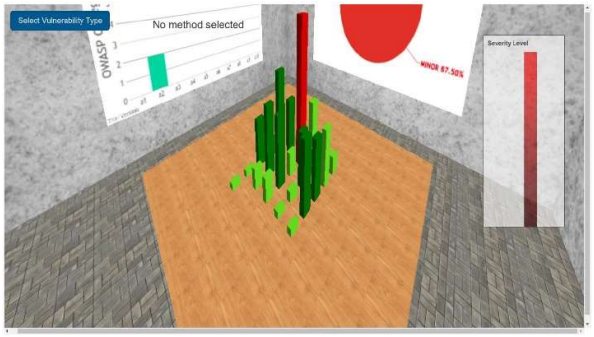

Figure 4b: Visualization of method level vulnerabilities

Upon triggering a building (*i.e.*, a Class) in the City view, the next level of visualization appears. Figure 4b shows the vulnerability distribution of the Methods of the triggered Class. Moreover, the Wall view is used to provide additional security information such as vulnerability categorized according to OWASP and percentage of critical vulnerabilities. Therefore, with the conceptual framework of Secure CodeCity and the proof-of-concept implementation, RQ1 can be addressed successfully.

RESULTS AND DISCUSSION

This paper presents a novel mechanism and a proof-of- concept implementation to effectively visualize software security facets in a three-dimensional metaphor. First, the evaluation is carried out to measure the overall correctness of Secure CodeCity when compared with a state-of-the-art security analysis tool. Then, the usability

of Secure CodeCity compared with the same tool is determined. In order to evaluate the above claims, three hypotheses (see Table 2) have been formulated, such that they are expected to answer RQ2: Can such a visualization witness an improvement over the state-of-the-art security analysis tools regarding the correctness, usability, and time efficiency when performing security-specific analysis tasks?

Table 2: Hypothesis

	Null hypothesis		Alternative hypothesis
H1o	The total correctness score for all tasks is the same across the experimental and control group.	H1	The total correctness score for all tasks is different across the experimental and control group.
H2o	The System Usability Scores are the same across the experimental and control group.	H2	The System Usability Scores are different from the experimental and control group.
H3o	The total completion time for all tasks is the same across the experimental and control group.	H3	The total completion time for all tasks is different from the experimental and control group.

Experimental design

A user study-based experiment has been designed by following the Between Subjects Design approach, which is a well-known experimental design strategy in Software Engineering (Kampenes, 2009). The goal of the experiment is to show whether Secure CodeCity's metaphoric visualizations provide better support to software practitioners in solving software security-related tasks than state-of-the-practice non-visual exploration tools. Thus, like other empirical evaluations of metaphoric software visualization approaches, the usability has been selected as one the dependent variables of the experiment along with the correctness and time efficiency.

SonarQube was selected as the baseline state-of-the-practice tool due to several reasons. Foremost, SonarQube is considered as a benchmark tool among the software engineering research community over the last decade (Wijesiriwardana & Wimalaratne, 2018; Marcilio *et al*., 2019; Lenarduzzi *et al*., 2020). Furthermore, SonarQube is well suited for software security-related analytics.

Two open-source benchmark Apache projects, Apache Ant and Struts 2, available in Github, are used for the experiment. It was decided to select the projects that are less than 500,000 Lines of Code (LOC) due to the time limitations in conducting the user study. The first project, Apache Ant, is an open-source project initiated by Apache Software Foundation; it is a software tool used to automate software build processes such as compile, run, test, and assemble Java-based applications. The second project, Apache Struts, is a free open-source solution to create Java web applications. It encourages developers to utilize a model-view-controller (MVC) architecture.

Table 3: Summary of the Open-Source Projects Used for the User Study

Project Name	No: of Classes	LOC
Apache Ant	1,277	112,503
Apache Struts 2	1,649	134,155

Controlled variables

The experiment is conducted with 23 subjects who are Masters students in Computer Science and Information Technology. The inclusion criteria of the subjects were based on working experience in the software industry and awareness of SonarQube. A minimum of two years of industry experience and a minimum of two years of experience in working with SonarQube were considered as the inclusion criteria. Of the 23 participants, 12 were allocated to the experimental group, and the remaining 11 were allocated to the control group. The allocation of the subjects to both experimental and control groups was done randomly. The experimental group was instructed to perform the tasks with Secure CodeCity, whereas the control group was instructed to use SonarQube to perform the tasks. Figure 5 shows the industry experience in years among the experimental and control groups and the experience in using SonarQube in years.

The experimental group and the control group were instructed to provide answers to the tasks as per their experience with Secure CodeCity and SonarQube, respectively while performing the tasks.

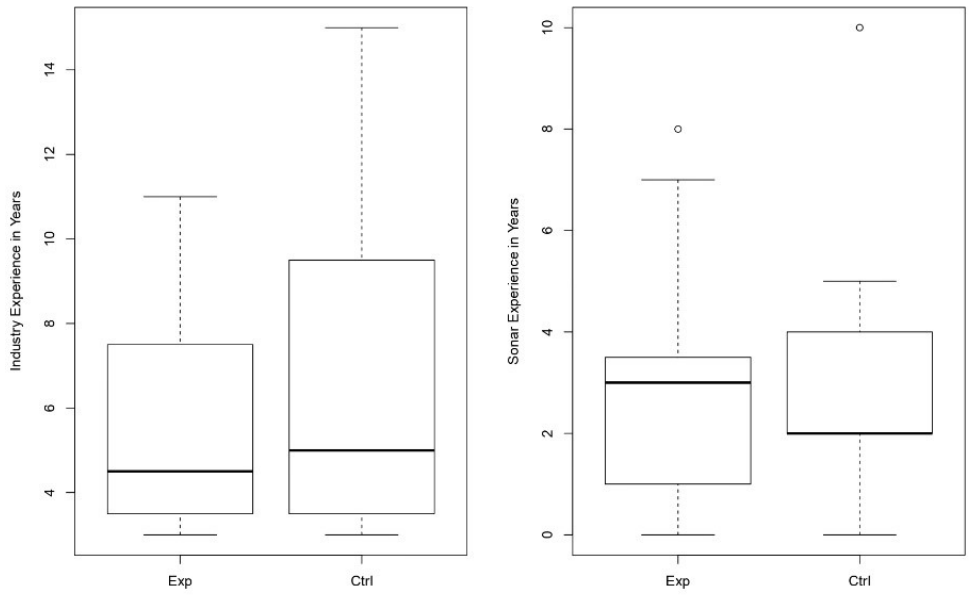

Figure 5: Industry experience (on the left) and SonarQube experience (on the right)

Selection of security-related tasks

The task selection was based on several visualization levels of the CodeCity framework. The initial focus is on the class level (first level) vulnerability information and other essential software metrics. For example, finding out "What is the most vulnerable class of a particular software project?" is a piece of useful information for a software practitioner. Then the tasks are gradually focused on the second level and third level visualizations.

Table 4: Security-related Tasks Used for the Evaluation

Task ID	Task Description
T1	What is the most vulnerable class in Apache Ant project?
T2	How many vulnerabilities are there in the ChainReadeRHelper.java class?
T3	How many security vulnerabilities are there in the verifySettings method in the DepthSelector.java class?
T4	What is the cyclomatic complexity of the DepthSelector.java class?
T5	What is the Security Remediation Effort in the DepthSelector.java class?
T6	What is the most critical method in the JakartaMultiPartRequest.java class?
T7	What is the percentage of MINOR issues in class Reg- ister.java?
T8	How many security vulnerabilities are there in the cleanUp method in the JakartaMultiPartRequest.java class?
T9	Does the system suggest countermeasures to rectify the first vulnerability in cleanUp method in the JakartaMultiPartRequest.java class?

Data analysis

This section presents the data collection and analysis process to evaluate the aforementioned two hypotheses and the experimental results gathered by the statistical analysis.

Correctness scores of the tasks for both experimental and control groups:

The correctness values for the tasks are obtained by a simple rating mechanism. For example, if the answer provided by a participant to a particular task is correct, two points are given. Thus, a maximum of 18 points can be obtained by a participant if all the tasks were correctly answered. Likewise, the wrong answers were allocated zero marks.

The mean correctness score of Secure CodeCity is 9.11 (with a standard deviation of 1.45), which is higher than the mean correctness score of 7.22 in the control group (with a standard deviation of 1.98). The correctness scores of the experimental and control groups are shown as a box plot in Figure 6. According to the Figure, the 25th percentile of the experimental group is above the 75th percentile of the control group. Thus, notable overall correctness of Secure CodeCity is observed over SonarQube in performing the security-related tasks. Besides, the non-parametric Independent Samples Mann-Whitney U (MWU) test is used with a significance level of 0.01 to test the first hypothesis. As per the experimental results, the MWU test rejects H1o at the 99 percent confidence level (with $p < 0.0001$), accepting the hypothesis H1: total correctness scores are different for the experimental and control groups.

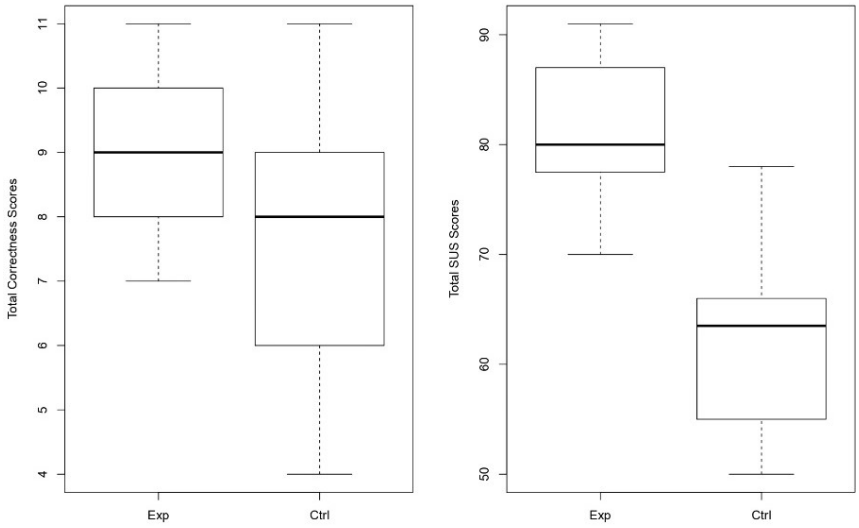

Figure 6: Total correctness score (on the left) and total SUS score (on the right)

Usability scores of the tasks for both experimental and control groups:

System usability score (SUS): 10 questions have been used to assess the usability of Secure CodeCity compared with the SonarQube in performing the aforementioned nine tasks. The subjects were requested to provide a score ranging from 1 to 5 for each of the ten SUS questions as per the level of satisfaction while performing the tasks. As per the SUS scoring mechanism, 1 point has been subtracted from the score for each odd-numbered question. For each even-numbered question, 5 points have been subtracted from the score. Then the total score was obtained by summing up the newly generated values. As the last step, the final SUS score was obtained by multiplying the total score by 2.5. However, the final SUS score is not a percentage value, instead, it gives the score out of 100, and still is considered an unambiguous method for the comparison of the results.

The mean SUS score of Secure CodeCity is 80.82 (with a standard deviation of 7.01), and it is 16.1 greater than the control group's SUS score of 64.70 (with a standard deviation of 8.02). It was observed that the 25^{th} percentile of the experimental group is above the 75^{th} percentile of the control group as shown in the box plots in Figure 7. Thus, it confirmed the acceptance of Secure CodeCity over the baseline tool, SonarQube used for the evaluation. Moreover, the MWU test rejects H2o at the 95 percent confidence level (with $p < 0.0001$), accepting hypothesis H2: the SUS scores are different for the experimental and control groups.

Overall completion time of the tasks for both experimental and control groups:

Based on the difficulty level of a few questions, study subjects were instructed not to spend more than 10 minutes on a single task. Upon the completion of each task, they were requested to note down the time spent on each task. Figure 7 presents the distribution of time across the experimental group and the control group. Based on that, it was observed that Secure CodeCity was capable of obtaining the results much faster than the baseline tools. However, a distinguishable benefit was not witnessed in Secure CodeCity in performing T2 and T5. The MWU test rejects the null hypothesis H3o at the 99 percent confidence level. The acceptance of the alternative hypothesis confirms that the distribution of the total completion times among the two groups is different.

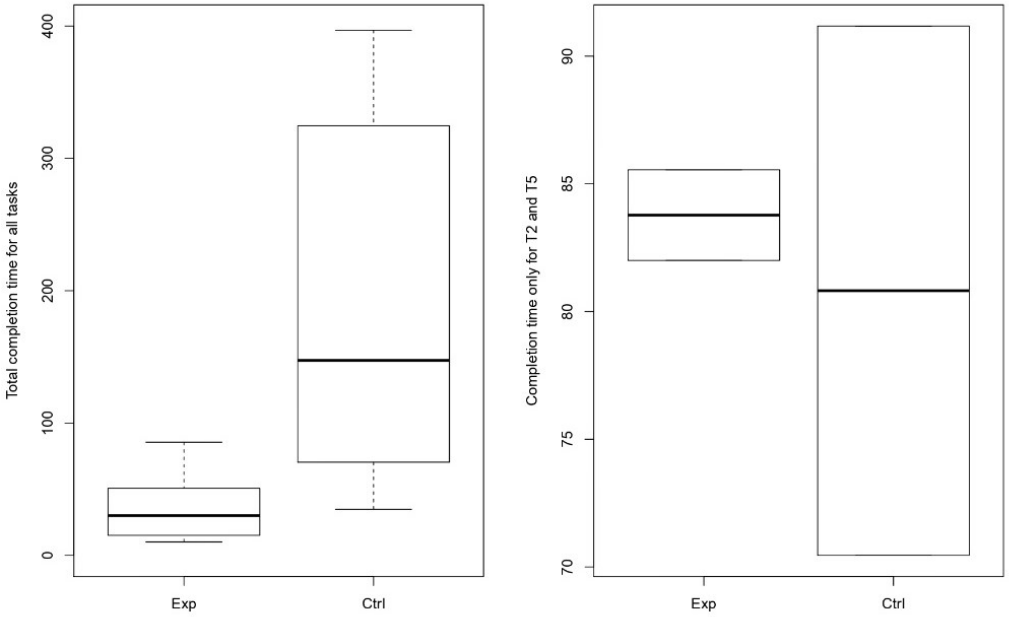

Figure 7: Overall completion time for all the tasks (on the left) and completion time only for T2 and T5 (on the right)

Thus, RQ2 can be successfully answered by accepting hypotheses H1, H2, and H3. Thus, secure CodeCity outperforms state-of-the-art tools regarding usability, accuracy, and time efficiency.

CONCLUSION

This work presents a novel mechanism and a proof-of-concept visualization tool, Secure CodeCity to facilitate security analytics in software projects. Secure CodeCity is capable of visualizing the security issues in software projects organized into different granularity levels: Secure CodeCity , Secure Building, and Secure Room. For example, Secure CodeCity presents the class-level vulnerabilities, and Secure Building presents the method-level vulnerabilities. A user study was conducted with 23 participants to evaluate Secure CodeCity in terms of correctness, usability, and time efficiency compared with the selected state-of-the-art tool SonarQube. Evaluation results confirmed the capability of Secure CodeCity in facilitating security analytics thus, this study would provide means for better software security analytics in developing less vulnerable software. More specifically, the main contributions (C1 and C2) of this paper are as follows:

C1: Metaphoric software visualization based on a multilayered abstraction mechanism.

As described in Section 3, a conceptual model of Secure CodeCity is introduced, which was organized into different abstractions to effectively visualize fine-grained security facets. Secure CodeCity is a natural extension of CodeCity, where three distinct views are provided namely Secure CodeCity, Secure Building, and Secure Room.

C2: Design, implementation, and evaluation of Secure CodeCity

As described previously, Secure CodeCity extends the CodeCity metaphor to facilitate understanding of security-related information of software projects. The evaluation results prove that Secure CodeCity is capable of addressing the limitations of existing static code analysis tools for security analysis.

As future work, it is expected to develop a security-specific positioning algorithm for Secure CodeCity. As of now, the positioning of the classes (*i.e.*, buildings) on top of a package (*i.e.*, 2D pane) is based on a positioning algorithm that avoids class overlaps and class displacements. However, a security-aware positioning algorithm would improve the usability of the Secure CodeCity by further minimizing the cognitive loads of the users. Besides, extensive pre-processing mechanisms could be investigated to facilitate the swift rendering of Secure CodeCity. However, performance is considered a common challenge that every software visualization tool needs to encounter. Therefore, a security-specific pre-processing of the source code is recommended prior to the visualization step.

Acknowledgment

The authors gratefully acknowledge the financial support provided by the National Research Council of Sri Lanka (Grant no: NRC15-74).

REFERENCES

Abeyrathna A., Samarage C., Dahanayake B., Wijesiriwardana C. & Wimalaratne P. (2020) A security specific knowledge modelling approach for secure software engineering. *Journal of the National Science Foundation of Sri Lanka* **48**: 93–98.
DOI: http://dx.doi.org/10.4038/jnsfsr.v48i1.8950

Ardigò S., Nagy C., Minelli R. & Lanza M. (2022). M3triCity: Visualizing evolving software and data cities. *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 21 - 29 May. Pittsburgh, Pennsylvania, USA, pp. 130–133.
DOI: http://dx.doi.org/10.1145/3510454.3516831

Assal H., Chiasson S. & Biddle R. (2016). Cesar: Visual representation of source code vulnerabilities. *Proceedings of the 2016 IEEE Symposium on Visualization for Cyber Security (VizSec)*, 24-24 October. pp. 1–8. Baltimore, MD, USA.
DOI: http://dx.doi.org/10.1109/VIZSEC.2016.7739576

Caserta P. & Zendra O. (2010). Visualization of the static aspects of software: A survey. *IEEE Transactions on Visualization and Computer Graphics* **17**(7): 913–933.
DOI: http://dx.doi.org/10.1109/TVCG.2010.110

Christakis M. & Bird C. (2016). What developers want and need from program analysis: an empirical study. *Proceedings of the 31st IEEE/ACM international conference on automated software engineering,* 3-7 September. Singapore, pp. 332–343.
DOI: http://dx.doi.org/10.1145/2970276.2970347

Dashuber V., Philippsen M. & Weigend J. (2021). A Layered Software City for Dependency Visualization. *Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP),* 8-10 February. pp. 15–26.
DOI: http://dx.doi.org/10.5220/0010180200150026

Hammad M., Abdul H., Jarzabek S. & Koschke R. (2021). Visualization of clones. In: *Code Clone Analysis*. (eds. K.Inoue & C.K. Roy), pp. 107–120. Springer, Singapore.
DOI: http://dx.doi.org/10.1007/978-981-16-1927-4_8

Johnson B., Song Y., Murphy-Hill E. & Bowdidge R. (2013). Why don't software developers use static analysis tools to find bugs?. *Proceedings of the 35th IEEE International Conference on Software Engineering (ICSE)*, 18-26 May. San Francisco,USA, pp. 672–681.
DOI: http://dx.doi.org/10.1109/ICSE.2013.6606613

Johnson B., Pandita R., Smith J., Ford D., Elder S., Murphy-Hill E. & Sadowski C. (2016). A cross-tool communication study on program analysis tool notifications. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 13-18 November. Seattle, USA, pp. 73–84.
DOI: http://dx.doi.org/10.1145/2950290.2950304

Kampenes V.B., Dybå T., Hannay J.E. & Sjøberg D.I. (2009). A systematic review of quasi-experiments in software engineering. *Information and Software Technology* **51**(1): 71–82.
DOI: http://dx.doi.org/10.1016/j.infsof.2008.04.006

Lenarduzzi V., Lomio F., Huttunen H. & Taibi D. (2020). Are sonarqube rules inducing bugs?. *Proceedings of the 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 18-21 February. Ontario, Canada, pp. 501–511.
DOI: http://dx.doi.org/10.1109/SANER48275.2020.9054821

Liu H., Tao Y., Huang W. & Lin H. (2021). Visual exploration of dependency graph in source code via embedding-based similarity. *Journal of Visualization* **24**(3): 565–581.
DOI: http://dx.doi.org/10.1007/s12650-020-00727-x

Marcilio D., Bonifácio R., Monteiro E., Canedo E., Luz W. & Pinto G. (2019). Are static analysis violations really fixed? a closer look at realistic usage of sonarqube. *Proceedings of the 2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, 25-26 May. Montreal, Canada, pp. 209–219.
DOI: http://dx.doi.org/10.1109/ICPC.2019.00040

Merino L., Ghafari M., Anslow C. & Nierstrasz O. (2017). CityVR: Gameful software visualization. *Proceedings of the 2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 17-22 September. Shanghai, China, pp. 633–637.
DOI: http://dx.doi.org/10.1109/ICSME.2017.70

Merino L., Ghafari M., Anslow C. & Nierstrasz O. (2018). A systematic literature review of software visualization evaluation. *Journal of Systems and Software* **144**: 165–180.
DOI: http://dx.doi.org/10.1016/j.jss.2018.06.027

Misiak M., Schreiber A., Fuhrmann A., Zur S., Seider D. & Nafeie L. (2018). IslandViz: A tool for visualizing modular software systems in virtual reality. *Proceedings of the 2018 IEEE Working Conference on Software Visualization (VISSOFT)*, Madrid, Spain, pp. 112–116.
DOI: http://dx.doi.org/10.1109/VISSOFT.2018.00020

Moreno-Lumbreras D., Minelli R., Villaverde A., González-Barahona J. M. & Lanza M. (2021). CodeCity : On-Screen or in Virtual Reality? *Proceedings of the 2021 Working Conference on Software Visualization (VISSOFT),* 27-28 September. Luxembourg, pp. 12–22.
DOI: http://dx.doi.org/10.1109/VISSOFT52517.2021.00011

Müller R. & Fischer M. (2019). Graph-based analysis and visualization of software traces. *Proceedings of the 10th Symposium on Software Performance*, 5 November. Würzburg, Germany, pp. 26–28.

Qayum A., Khan S.U.R. & Akhunzada A. (2022). FineCodeAnalyzer: Multi-perspective source code analysis support for software developer through fine-granular level interactive code visualization. *IEEE Access* **10**: 20496–20513.
DOI: http://dx.doi.org/10.1109/ACCESS.2022.3151395

Reiss S. P. & Renieris M. (2005). JOVE: Java as it happens. *Proceedings of the 2005 ACM Symposium on Software Visualization*, 14-15 May. St. Louis Missouri, USA, pp. 115–124.
DOI: http://dx.doi.org/10.1145/1056018.1056034

Rufiange S. & Melançon G. (2014). Animatrix: A matrix-based visualization of software evolution. *Proceedings of the 2014 Second IEEE Working Conference on Software Visualization*, 29-30 September. Victoria, Canada, pp. 137–146.
DOI: http://dx.doi.org/10.1109/VISSOFT.2014.30

Shahin M., Liang P. & Babar M.A. (2014). A systematic review of software architecture visualization techniques. *Journal of Systems and Software* **94**: 161–185.
DOI: http://dx.doi.org/10.1016/j.jss.2014.03.071

Soman S., Pareek P.K., Dixit S., Chethana R.M. & Kotagi V. (2021). Exploration study to study the relationships between variables of secure development lifecycle (SDL). In: *Emerging Technologies in Data Mining and Information Security (*eds. J.M.R.S.Tavares, S. Chakrabarti, A. Bhattacharya & S. Ghatak) pp. 641–649. Springer, Singapore.
DOI: http://dx.doi.org/10.1007/978-981-15-9774-9_59

Tahaei M., Vaniea K., Beznosov K. & Wolters M.K. (2021). Security notifications in static analysis tools: Developers' attitudes, comprehension, and ability to act on them. *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, 8-13 May. Yokohama, Japan, pp. 1–17.
DOI: http://dx.doi.org/10.1145/3411764.3445616

Todorov B., Kula R.G., Ishio T. & Inoue K. (2017). SoL Mantra: Visualizing update opportunities based on library coexistence. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)*, 18-19 September. Shanghai, China, pp. 129–133.
DOI: http://dx.doi.org/10.1109/VISSOFT.2017.23

Wettel R. & Lanza M. (2008). CodeCity : 3D visualization of large-scale software. In *Companion of the 30th international conference on Software engineering*, 10-18 May. Leipzig Germany, pp. 921–922.
DOI: http://dx.doi.org/10.1145/1370175.1370188

Wijesiriwardana C. & Wimalaratne P. (2018). Fostering real-time software analysis by leveraging heterogeneous and autonomous software repositories. *IEICE Transactions on Information and Systems* **101**(11): 2730–2743.
DOI: http://dx.doi.org/10.1587/transinf.2018EDP7094

Wijesiriwardana C. & Wimalaratne P. (2019). Software engineering data analytics: a framework based on a multi-layered abstraction mechanism. *IEICE Transactions on Information and Systems* **102**(3): pp. 637–639. DOI: http://dx.doi.org/10.1587/transinf.2018EDL8070

Yu, G. (2020). Using ggtree to visualize data on tree-like structures. *Current Protocols in Bioinformatics* **69**(1): e96. DOI: http://dx.doi.org/10.1002/cpbi.96