SecureChange: An Automated Framework to Guide Programmers in Fixing Vulnerability

Sayem Mohammad Imtiaz*[1], Kazi Zakia Sultana†[2] and Tanmay Bhowmik‡[3]

[1]Department of Computer Science, Iowa State University, IA, USA
[2]Department of Computer Science, Montclair State University, NJ, USA
[3]Department of Computer Science and Engineering, Mississippi State University, MS, USA

Abstract

When developers fix a defect, they may change multiple files. The number of files changed for resolving the defect depends on how strongly the files are coupled with each other. In earlier works, researchers leveraged this coupling for better understanding and analyzing software as well as for guiding developers to quickly find all probable code areas to complete fixing a defect. In some studies, researchers generated association rules reflecting the coupling among files and built tools to automate the discovery of the related changes in the files. Such tools, however, do not consider the type of defects resolved earlier for generating the rules as a result of which many unrelated files may come up while changing a file in later releases for resolving a specific type of defect. Therefore, in our study, we consider only security defects or vulnerabilities to generate the rules and then automate the finding process of other related files while fixing a vulnerability. Our tool "SecureChange" suggests the developers a number of related files that might need to be changed while fixing a particular vulnerability based on the mined association rules from the revision history. This approach will have a significant role in guiding the developers in fixing a vulnerability. Furthermore, this will be an effective endeavor for training new developers based on the vulnerability history of a system, which will in turn help them to develop secure code. The proposed approach will also be helpful in educating new developers about software vulnerabilities. Finding all the related files which have been modified to fix a vulnerability, the new developers will be able to learn how the faults in a file can be the root cause of a vulnerability and how it can propagate to other related files and ultimately emerge as a vulnerability to the outside world. As a demonstration of our approach, we generate association rules based on the revision history of three systems: Android, Mozilla Firefox, and Apache Tomcat. The average precision and recall of 44% and 44% respectively for three systems indicate the feasibility of our approach.

1. Introduction

Coupling among files in a software becomes crucial for program understanding and resolving issues in software maintenance [1–3]. Developers fix some software issues in every revision and record the related changes which are stored as revision histories. Mining this revision history can be a good source of discovering coupling among the files. Revision history also tells us how the program evolves over time, which can later be used for identifying the versions having or not having a particular issue. Researchers have used such historical data to support code navigation [4]. In [5,6], coupling has been used to analyze and get insights of the program. In order to guide developers in fixing defects, [7] suggested a technique to provide related changes by mining revision histories.

The state of the art does not focus on the revision histories that are targeted to fix particular types of issues (general bugs or vulnerabilities or adding new functionalities) [1–3, 7]. Here, a potential drawback is that there is a higher probability of increased false positives if the developer is not fixing the same type of issue compared to the earlier versions. Therefore, following such a generalized association rule based technique could be misleading for a developer as he/she might end up wasting time concentrat-

*sayem@iastate.edu
†sultanak@montclair.edu
‡tbhowmik@cse.msstate.edu

DOI reference number: 10.18293/SEKE2020-132

ing on irrelevant files considering the task at hand. In order to address this gap, in this paper, we consider the revisions that were made for resolving vulnerabilities and then extract the files that were changed together. We leverage the extracted components to generate association rules so that we can guide the developers later on fixing vulnerabilities. Software vulnerability is a mistake in software that can be directly used by a hacker to gain access to a system or network[1]. As a vulnerability can put the security of a software at risk, in our study, we considered software vulnerability and applied the framework for fixing the security issues.

Let us consider an example. According to the revision log[2] of Apache Tomcat (an open-source Java Servlet Container developed by the Apache Software Foundation (ASF)), $NamingContextListener.java$ has been modified 50 times since 2006 as a part of fixing different issues. But it was coupled with only one file $ResourceLinkFactory.java$ in Revision 1757271[3] where the developer fixed the "Unrestricted Access to Global Resources" vulnerability. This vulnerability[4] has been fixed by the developers in 2016. A developer does not need to consider all files related to $NamingContextListener.java$ since 2006 for fixing a vulnerability in this file. This story tells us that mining the revision log of a system could be of no use if we do not consider the type of issue to be resolved. Therefore, we concentrate on the revisions related to fixing vulnerabilities so that when the developers will try to fix any vulnerability later, they can be guided based on only the vulnerability-fixing related revisions of that file.

Another motivation of the paper comes from the need to educate new developers on vulnerability for a specific system. As a new developer may be unfamiliar with the system she is working on, this can be an effective guidance for her to fix the faults in all the related files and help her to be acquainted to the new system.

This paper, primarily motivated by the work of Zimmermann et al. [7], uses the concept of association rule mining to produce the list of coupled files from the revision history. In contrast to the state of the art, it focuses on quick fixing software vulnerability and ensuring secure coding. The objectives of the paper are as follows:

1. to obtain association rules reflecting the coupling among the related vulnerable files so that programmers can be guided in fixing vulnerability in later releases of the same system.

2. to assist the developers in finding and fixing vulnerabilities in an efficient and effective way, thereby ensuring secure software evolution.

3. to identify coupling among vulnerable files and thus increase the ability to understand and analyze vulnerable code for software maintenance.

4. to evaluate the proposed approach that suggests related changes that might be needed for fixing a particular type of vulnerability in a system.

Section 2 shows the related works. In Section 3, we discuss Apriori algorithm which we used to generate association rules from the vulnerability revision histories. Section 4 presents the methodology followed in *SecureChange* to perform the experiments. Finally, Section 5 presents the results and Section 6 concludes the study.

2. Related Work

In [8], Ying et al. applied data mining techniques in the change history and determined sets of files that were frequently changed together in the past. They hypothesized that the change patterns (pertinent set of files) can be recommended to the developers performing a modification task. They revealed valuable dependencies among the files in the Eclipse and Mozilla open source projects and evaluated the performance of the recommendations that were produced by their approach for actual modification tasks. Xing et al. [9] used association rule mining at the design level on versions of UML diagrams to detect class co-evolution. They presented three potential applications of class co-evolution discovery in the context of software maintenance: finding the scope of future maintenance activities, guiding refactoring activities and identifying system instabilities. Although they showed promising initial results of their approach, it still lacks in large scale evaluation. In [10], the authors investigated how a change in one source code entity propagates to other entities. They applied several heuristics to predict change propagation and validated their approach in five open source software systems. Gall et al. [2] first used release history of a system to uncover logical dependencies and common change patterns among modules in order to detect potential structural shortcomings. The CVS history has also been used to detect more fine-grained logical coupling between classes [5], files, and functions [6]. In [11–13], authors used inductive learning (a relevance relation identifying two files that are updated together) to learn different concepts between logically coupled files.

Michail [14, 15] applied data mining technique to discover library reuse patterns (for example, how library functions are used together or how library functions are overwritten by the applications classes). [15] considered the inheritance relationship and generated generalized association rules to find how the descendent classes have been invoked or instantiated. In SecureChange, we use association

[1] https://cve.mitre.org/about/terminology.html

[2] https://svn.apache.org/viewvc/tomcat/trunk/java/org/apache/catalina/core/NamingContextListener.java?view=log&pathrev=1757271

[3] https://svn.apache.org/viewvc?view=revision&revision=1757271

[4] http://cve.mitre.org/cgi-bin/cvename.cginame=CVE-2016-6797

rule mining for mining vulnerability revision histories and guiding developers in secure software development.

3. Introducing the Apriori Algorithm

In order to suggest relevant co-occurring changes for a particular change, *SecureChange* leverages an association rule learning technique known as Apriori algorithm [16]. Let us assume that we have a set of transactions, $T = \{T_1, T_2, T_3, ..., T_n\}$, where each transaction records a set of co-occurring change items, $C = \{C_1, C_2, C_3, ...\}$. Apriori algorithm generates a set of frequent items based on a support threshold. The frequent item set is then leveraged to learn association rules among individual items, C_i, in the set of all transactions. A rule is denoted as follows:

$$X \rightarrow Y \; where \; X, Y \subseteq C$$

Here, X is called antecedent and Y is called consequent. In other words, if X occurs, then Y follows. The rules are derived based on some parameters:

- **Support Count:** Support count indicates the popularity of an item. It measures how frequently a particular item appears in all transactions. It can be expressed as:

$$Support(X) = \frac{|\{t \in T, X \subseteq t\}|}{|T|}$$

- **Confidence:** Confidence measures the relative importance of the consequent in a rule. It indicates how frequently a consequent appears in all transactions that contain antecedent.

$$Confidence(X \rightarrow Y) = \frac{Support(X \cup Y)}{Support(X)}$$

  Higher the confidence, better the likelihood of occurring Y, given X.

- **Lift:** Lift takes the popularity of both antecedent and consequent into account. It is possible that an item is generally very popular. It may occur in many transactions without maintaining a particular pattern with any antecedent. In such a case, confidence provides a poor feedback on a rule. However, lift resolves this problem by considering popularity of both the antecedent and consequent. In other words,

$$Lift(X \rightarrow Y) = \frac{Support(X \cup Y)}{Support(X) \times Support(Y)}$$

  A lift value of 1 indicates that item-sets are independent of each other. Whereas, a value higher than 1 indicates a tie between item-sets. In other words, the higher the lift, the stronger the tie. On the other hand, lift value lower than 1 indicates a negative tie.

Table 1: Datasets

System	# Transactions	# Curated Transactions	Versions
Tomcat	76	48	5 - 8.5.38
Firefox	595	249	3.6 - 62.0.2
Android	1485	381	4.4 - 9.0

4. Methodology

This section describes the methodology of *SecureChange* in details.

4.1 Research Question

The primary challenge with mining association rules for secure development is that the occurrence of a vulnerability is very infrequent compared to regular bugs. As a result, vulnerability-fixing transactions are less in amount compared to other general transactions. Abundance of data is very crucial for the success of a data mining technique such as association rule mining. Therefore, in this study, we attempt to answer the following research question (RQ):

Given the limited availability of the vulnerability-fixing transactions, is association rule mining for secure software development as effective as it has been shown in guiding general software change tasks [7]?

4.2. Training Data Preparation

To evaluate *SecureChange*, we have performed case studies on three widely known real world open-source software systems, namely, Apache Tomcat[5], Mozilla Firefox[6], and Android Open Source Project (AOSP)[7]. The vulnerabilities detected and fixed in these systems are publicly accessible. We begin by mining respective security advisories of all the systems. A security advisory typically records detailed information of past vulnerabilities, including the source code changes for fixing the vulnerability and time of the fix. We treat set of all files changed for fixing a vulnerability as a single transaction.

Table 1 shows the demography of the collected data. We found that most of the transactions have only one change. These transactions can be easily identified by *Apriori* algorithm, hence leading to a bloated estimation of the performance. For instance, in our experiment, *SecureChange* provided two or three times better performance estimation than the one trained without such transactions. Therefore, we further curated the dataset to eliminate transactions with

[5] http://tomcat.apache.org/
[6] https://www.mozilla.org/
[7] https://source.android.com/

a single change and then assessed the effectiveness of *SecureChange* in predicting co-occurring changes.

4.3. Validation Technique

We have validated *SecureChange* in a repeated validation setup. The experiment has been repeated 100 times on a randomly shuffled dataset for each project. In each experiment, *Apriori* algorithm was trained with 90% of the data and remaining 10% was retained for testing. The data were randomly shuffled before each experiment. The random shuffling and repeating many times minimized the bias in the reported result. During every experiment, we performed the following steps:

1. Shuffle dataset randomly and split into two 90%-10% folds.
2. Train *SecureChange* with *Apriori* algorithm on the fold with 90% data.
3. After generating the rules or training, iterate through every transaction in the test dataset. For every transaction in the test dataset, we evaluate every pair of rules. For instance, if a certain transaction contains three files which have been changed to fix a vulnerability, $T_i = \{File_1, File_2, File_3\}$, then following rules are evaluated: $File_1 \rightarrow File_2$, $File_2 \rightarrow File_3$, $File_1 \rightarrow File_3$, $File_2 \rightarrow File_1$, $File_3 \rightarrow File_2$, and $File_3 \rightarrow File_1$. Then we obtain the average of the performance metrics for all queries in the test dataset.
4. Final performance metrics (i.e., Recall, Precision and Feedback) are reported by taking average over all repeated experiments.

4.4. Performance Metrics

The performance of *SecureChange* is evaluated based on the following three metrics:

- **Precision:** For a given transaction, precision refers to the percentage of rules predicted correctly out of all predictions. For instance, consider, a transaction contains two files, $T_i = \{File_1, File_2\}$. This transaction suggests two rules:

$$File_1 \rightarrow File_2 \quad (1)$$

$$File_2 \rightarrow File_1 \quad (2)$$

  Rule 1 tells us that if $File_1$ is changed, then $File_2$ changes and the impact is bidirectional; therefore, we have the rule 2.

  Assume, following rules have been predicted by *SecureChange* for $File_1$ and $File_2$:

$$File_1 \rightarrow File_2 \quad (3)$$

$$File_1 \rightarrow File_3 \quad (4)$$

$$File_2 \rightarrow File_4 \quad (5)$$

  The rules 3 and 4 basically tell us that with a certain support, confidence and lift, if $File_1$ is changed, then $File_2$ and $File_3$ are also changed. The rule 5 can be interpreted similarly.

  A transaction containing N items or files will have N number of queries. For example, we have two queries to validate: $File_1$ and $File_2$ as in the rules 1 and 2 generated from the transaction T_i. For query 1, only the rule 3 is correct out of two predictions made for $File_1$. Therefore, the precision, in this case, is 50%. Similarly, the precision for query 2 is 0 since the prediction is wrong for $File_2$. Finally, an average precision for all queries (25%) is reported by *SecureChange*. If *SecureChange* generates no rules for a query, the precision is considered to be 100%. However, considering such queries distorts the average precision. Therefore, we did not measure precision and recall for such queries and also did not consider them in performance evaluation.

- **Recall:** For a given transaction, recall refers to the percentage of rules predicted correctly out of all ground truth rules. In the example, for query 1, the recall is 100% as one out of one possible original changes for $File_1$ has been correctly predicted. Similarly, for query 2, it is 0. Finally, an average recall (50%) is reported by *SecureChange*.

- **Feedback:** A query can be left unreported by *SecureChange*. For instance, consider the transaction consisting of two files in the above example. Assume, following rules have been predicted by *SecureChange*:

$$File_1 \rightarrow File_2 \quad (6)$$

$$File_1 \rightarrow File_3 \quad (7)$$

  As we can see, rule 2 has not been predicted by *SecureChange*. The precision, in this case, is 100% since no false positive reported and recall is 0 since no actual change has been picked. Similarly, there can be many queries for which *SecureChange* may not respond. It would take a toll on the final average precision and recall and undermine the actual performance of the *SecureChange*. Therefore, we omit such queries in the final performance evaluation and instead incorporate another complementary performance metric,

Table 2: Performance Overview

System	Feedback	Precision	Recall
Firefox	42%	20%	24%
Android	37%	49%	56%
Tomcat	44%	62%	51%
Average	41%	44%	44%

feedback. Feedback informs us about the fraction of queries that have been responded to by *SecureChange*. It allows us to evaluate the actual performance of the *SecureChange* even in the case of inadequate training data. It also highlights the overall responsiveness of the framework. The feedback for this example is 50% since one query out of two queries has been responded.

5. Results and Implications

To answer the research question mentioned in Section 4.1, we have conducted case studies on three open-source software systems, Firefox, Tomcat, and Android Open Source Project, which publicly report vulnerabilities detected and fixed in their system. For three systems, *SecureChange* achieved approximately 41% feedback and 44% precision and recall on average as in Table 2. Different values for support, confidence, and lift parameters provide different results in each system as shown in Figure 1.

Zimmermann et al. [7] achieved 66% feedback in transactions where any kind of changes by the developers were considered. Although their feedback is better than the feedback we found for vulnerability-fixing transactions, they achieved 33% precision and 29% recall on average. The feedback found in our study can be explained from the fact that, considering all kind of changes allowed them to include more transactions in the experiments. On the other hand, security-related transactions are less frequent, and therefore, our curated security transactions are significantly less than the transactions considered in [7]. Intuitively, more the transactions, better the response rate would be.

However, the better recall and precision we obtained on a comparatively smaller set of transactions suggest that the vulnerability fixes often involve same set of files, hence they are frequently co-occurring. It implies that correlation studies on finer-granularities (e.g. statements, functions, classes etc.) is worth-exploring which could provide more precise vulnerability localization in the source code.

Figure 1 presents a comparative overview of the precision, recall, and feedback at different minimum supports and confidences for *Firefox*, *Android*, and *Tomcat* respectively. A general observation is that precision and recall tend to increase as the support and confidence increase. Feedback, on the other hand, tends to decrease as the support and confidence increase. A higher support and confidence ensures that only highly relevant rules are considered, therefore, as these parameters go up, the number of queries responded decreases for stricter filtering conditions. However, on the positive side, it only returns highly likely changes with lower false positives. Therefore, there is an obvious trade-off between the provided results and the accurate results.

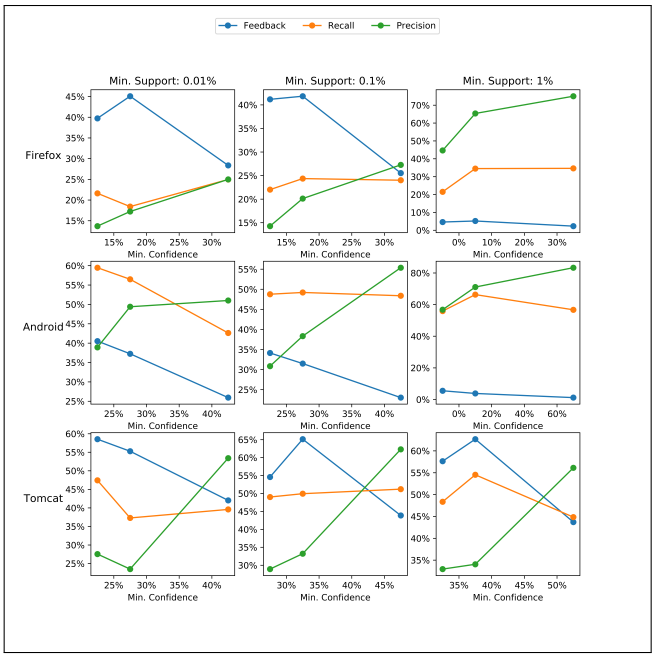

Figure 1: Precision, Recall, and Feedback for varying support and confidence (First row showing result for Firefox, second row for Android, and third row for Tomcat)

Our results imply that we can effectively apply association rule mining to guide developers in secure software development.

6. Threats to Validity and Conclusion

In this study, we first analyzed vulnerability revision histories of three large systems: Android, Firefox, and Tomcat. Then we generated association rules to figure out which files are frequently changed together in a vulnerability fix. When a developer starts to fix a vulnerability by making changes in a vulnerable file, our approach will suggest relevant set of files to be modified based on the generated association rules. We found precision and recall of 44% and 44% respectively on an average for three systems.

In our experiment, a threat to internal validity could be, to what extent, the reported result can be trusted or whether it is biased. To mitigate this threat, we have introduced randomization in the experimental procedure and repeated

the experiments 100 times. On the other hand, an external threat to *SecureChange* is whether it generalizes across application domains. In order to mitigate this threat, we have performed experiments on three different kinds of open-source systems, namely a web server (Tomcat), a desktop application (Firefox), and an operating system (Android). However, it is possible that *SecureChange* might not replicate the similar performance in closed-source systems for different architecture or other open-source systems for the quality of the captured vulnerability data. Also, a reasonable amount of preexisting vulnerability data has to be available to have useful feedback from *SecureChange*.

In future, we plan to extend this research for other systems so that our proposed framework, *SecureChange*, can be used by the developers of all systems. In addition, we will work on building a plug-in of the proposed framework for different source code editors (e.g., Eclipse, Netbeans) so that developers can find it more useful for secure coding.

References

[1] T. Ball, J. min Kim, A. A. Porter, and H. P. Siy, "If your version control system could talk..." 1997.

[2] H. Gall, K. Hajek, and M. Jazayeri, "Detection of logical coupling based on product release history," in *Proceedings. International Conference on Software Maintenance*, Nov 1998, pp. 190–198.

[3] J. M. Bieman, A. A. Andrews, and H. J. Yang, "Understanding change-proneness in oo software through visualization," in *11th IEEE International Workshop on Program Comprehension*, May 2003, pp. 44–53.

[4] D. Čubranić and G. C. Murphy, "Hipikat: Recommending pertinent software development artifacts," in *Proceedings of the 25th International Conference on Software Engineering*, ser. ICSE, 2003, pp. 408–418.

[5] H. Gall, M. Jazayeri, and J. Krajewski, "Cvs release history data for detecting logical couplings," in *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, Sep. 2003, pp. 13–23.

[6] T. Zimmermann, S. Diehl, and A. Zeller, "How history justifies system architecture (or not)," in *Sixth International Workshop on Principles of Software Evolution, 2003. Proceedings.*, Sep. 2003, pp. 73–83.

[7] T. Zimmermann, P. Weissgerber, A. Zeller, and S. Diehl, "Mining version histories to guide software changes," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 429–445, June 2005.

[8] A. T. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Softw. Eng.*, vol. 30, no. 9, pp. 574–586, Sep. 2004.

[9] Z. Xing and E. Stroulia, "Data-mining in support of detecting class co-evolution," in *The 16th International Conference on Software Engineering and Knowledge Engineering*, Alberta, Canada, June 20-24, 2004, 2004, pp. 123–128.

[10] A. E. Hassan and R. C. Holt, "Predicting change propagation in software systems," in *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, Sep. 2004, pp. 284–293.

[11] J. S. Shirabad, T. C. Lethbridge, and S. Matwin, "Supporting maintenance of legacy software with data mining techniques," in *Proceedings of the 2000 Conference of the Centre for Advanced Studies on Collaborative Research*, ser. CASCON '00, 2000, pp. 11–.

[12] ——, "Mining the maintenance history of a legacy software system," in *International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings.*, Sep. 2003, pp. 95–104.

[13] ——, "Mining the software change repository of a legacy telephony," in *Proceedings of International Workshop on Mining Software Repositories (MSR '04*, 2004, pp. 53–57.

[14] A. Michail, "Data mining library reuse patterns in user-selected applications," in *14th IEEE International Conference on Automated Software Engineering*, Oct 1999, pp. 24–33.

[15] ——, "Data mining library reuse patterns using generalized association rules," in *Proceedings of the 2000 International Conference on Software Engineering.*, June 2000, pp. 167–176.

[16] R. Agrawal, R. Srikant *et al.*, "Fast algorithms for mining association rules," in *Proc. 20th int. conf. very large data bases, VLDB*, vol. 1215, 1994, pp. 487–499.