Poster: Toward A Code Pattern Based Vulnerability Measurement Model

John Heaps, Rocky Slavin, Xiaoyin Wang
The University of Texas at San Antonio
{john.heaps,rocky.slavin,xiaoyin.wang}@utsa.edu

ABSTRACT

Many access control patterns, both positive and negative, have been identified in the past. However, there is little research describing how to leverage those patterns for the detection of access control bugs in code. Many software bug detection models and frameworks for access control exist, however most of these approaches and tools are process-based and suffer from many limitations. We propose a framework to detect access control bugs based on code pattern detection. Our framework will mine and generate bug patterns, detect those patterns in code, and calculate a vulnerability measure of software. Based on our knowledge we are the first pattern-based model for the detection and measurement of bugs in software. As a proof of concept, we perform a case study of the relational database access control pattern "Improper Authorization".

ACM Reference Format:
John Heaps, Rocky Slavin, Xiaoyin Wang. 2018. Poster: Toward A Code Pattern Based Vulnerability Measurement Model. In *SACMAT '18: The 23rd ACM Symposium on Access Control Models & Technologies (SACMAT), June 13–15, 2018, Indianapolis, IN, USA.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3205977.3208948

1 INTRODUCTION

The security and vulnerabilities of software systems are essential in making decisions for real-world problems. Many process-based approaches and models have been developed to mitigate and quantify software vulnerabilities. However, these approaches, while useful, have many limitations. First, the existing models require data from software processes to estimate model parameters, but fine-grained monitoring of software processes is not always possible. Second, these models cannot always handle new or upgraded software components. Third, these models often cannot differentiate between different types of defects. To address these issues, the proposed project will develop a pattern-based vulnerability measurement model, which checks software artifacts for the existence of negative patterns to estimate the risk of software failures and data security, their impact, and determine overall software vulnerability.

The model will quantitatively estimate the vulnerability of software based on negative pattern instances in software artifacts, as well as the importance and the activation probability of the patterns. Recent studies show that software reuse and code clones are prevalent throughout software systems. In addition, software projects are continually becoming more based on existing software frameworks, which have a limited number of usage patterns. Therefore, it is reasonable to assume that most of the design fragments and code portions of a new software product follow existing patterns. Furthermore, the vulnerability of a software application can be predicted by combining effects of all instances of negative bug patterns. The project will yield a **learning engine** to mine online bug repositories and project hosting websites; a **pattern checker** to detect the existence and invocation of patterns; and a **vulnerability model** for the estimation of the vulnerability of a software project based on the detected patterns and their invocation probabilities.

The project will achieve the following major objectives: 1) Estimate the vulnerability of a software project based on code patterns; 2) Support separate estimation of different aspects of software vulnerability, enabling fine-grained prediction of the effect of software failures; 3) Confirm the existence of negative patterns (i.e., identify the location of access control bugs) using test-coverage-based approach; 4) Evaluate the feasability of the model's ability to perform software vulnerability estimation on real-world software projects.

2 FRAMEWORK

In this section we will discuss our framework for the identification of bug patterns, detection of bugs in code, and proposed measurement model, as shown in Figure 1.

2.1 Learning Engine

The learning engine acts as a repository of known patterns. These patterns will be used to identify bugs in code during pattern detection. To create this initial repository, we first surveyed popular online bug repositories, such as Github, Bugzilla, Common Weakness Enumeration (CWE)[1], Jira, etc. We decided to initially support CWE and Github, as they often have code examples and code fixes linked with their bug reports. Further, they both offer very large, robust data sets of access control patterns and bugs.

CWE is a database that catalogs and categorizes known bugs. Our method crawls bug reports from CWE and categorizes them according to the common keywords in their descriptions and CWE classification categories. We are thus able to extract bug descriptions, code examples, and code solutions.

The goal of bug collection for Github is to identify the most common bugs on Github. We obtain thousands of issues from Github projects by making requests to Github's REST API v3. For each of these issues we are able to extract bug descriptions, solutions, and sometimes sections of code from before and after an issue was resolved. In our initial implementation we searched the top 1,000 Java-based repositories, and used the most recent 100 closed issues with the label "bug" from each.

[1]CWE website URL: https://cwe.mitre.org/

SACMAT '18, June 13–15, 2018, Indianapolis, IN, USA

ACM ISBN 978-1-4503-5666-4/18/06.
https://doi.org/10.1145/3205977.3208948

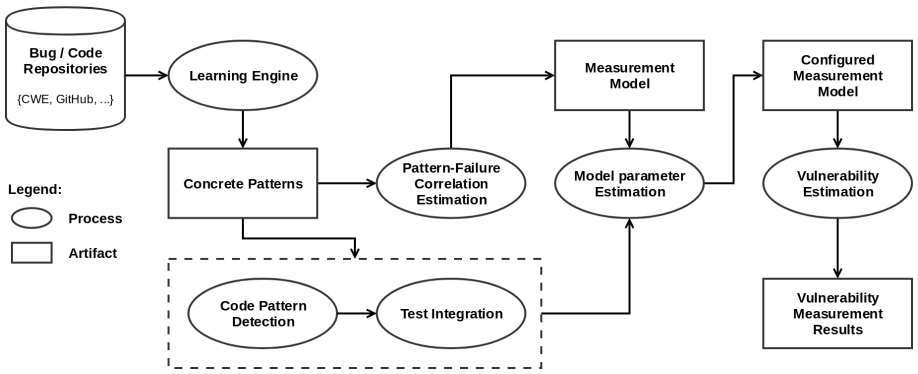

Figure 1: Bug Pattern Framework

After we collected the data from CWE and Github, we applied the clone detection tool CCFinder [2] to detect clones from the set of code commits and identify common buggy code and their fixes. Then, for the most popular common code bugs we manually extracted concrete code patterns from them, which were stored to be used in pattern detection and code analysis.

2.2 Pattern Detection

There is limited literature available on bug pattern detection. Work by Joseph Near and Daniel Jackson [5] describes the tool "SPACE" which is able to detect violations of access control vulnerability patterns. However, this tool requires that a user completely define and map software components and design to the role-based access control model. Not all software systems specifically use role-based access control, though. It can be difficult, or not possible, for systems that use other access control designs to define that system in the role-based model. Further, this approach is based on seven access control patterns, not code-level bug patterns on access control. Other work by Guangtai Liang et al. [3] introduces the tool "PatBugs" which detects temporal bugs in cross-platform mobile applications, focusing on API usage. The scope of this tool is too narrow for our purposes, it's focus being solely on temporal bug patterns in mobile applications.

To conduct the pattern detection in our framework, we utilized SpotBugs[2], a fork of the static analysis tool FindBugs [1]. The tool analyzes Java bytecode and detects the existence of bug patterns.

To study the effectiveness of SpotBugs on detecting bug code patterns, we downloaded the top 1,000 Java GitHub repositories that are under 50MB for a total of 826 repositories. From these repositories, we were able to compile 763 repositories using a program for automatic compilation of various types of Java projects. For each compiled repository, we ran SpotBugs to generate a list of bug types, which detected 217 bug pattern instances.

After identifying bug patterns, they further needed to be linked to abstract quality aspects to calculate a vulnerability, or risk, measure. Specifically, we consider four major aspects of vulnerability including: (1) **Control Integrity** which measures how likely the software may incorrectly interact with its users; (2) **Data Integrity** which measures how likely the software may provide incorrect output; (3) **Data Confidentiality** which measures how likely the software may release data to entities not authorized to receive it; and (4) **Data Availability** which measures how likely the software may not be able to provide data that should be in storage. Through linking bug code patterns with high-level quality aspects, we are able to estimate the vulnerability, or risk, on different aspects based on detected patterns related to them.

Finally, the detection of a bug pattern is not enough to determine that a bug actually exists, only that a bug possibly exists. We integrate testing techniques to test if the section of code identified by the pattern actually produces an error, showing it truly is a bug.

2.3 Measurement Model

The model used to calculate a vulnerability score must consider as many different types of bugs (related to access control) as possible, and must cover the abstract quality aspects mentioned previously. We began by considering existing measurement models (e.g., Common Vulnerability Scoring System (CVSS) [4], Common Weakness Scoring System (CWSS) [3], etc.), however these models only calculate a scoring for a single bug and not for an entire system. Further, the amount of possible automation in these models is also very limited and requires manual parameter assignments. Therefore, we created our own measurement model.

The basic idea of our measurement model is to estimate the vulnerability of a software based on the detected instances of code patterns in its code base. For a bug pattern instance, we determine the impact it will have on the software in relation to the identified abstract quality aspects. Further, we determine how likely the instance will be triggered at runtime, which was found from the testing performed previously. The more instances of bug patterns detected and the more likely those instances will be triggered, the higher (or worse) the vulnerability score should be.

At the most abstract level, for multiple detected instances of bug patterns, we use the following formula to generate a vulnerability value, normalized to the range [0, 1]. Here *Detected* is the set of bug instances detected in the code using patterns and *Risk(b)* denotes the risk value of a given bug *b*.

$$Vulnerability = 1 - \frac{R}{R + \sum_{Detected} Risk(b)} \quad (1)$$

R is a constant, which is the average risk sum per software project, which can be estimated using a large number of training software projects. With this formula if there are no bugs in a software project,

[2]SpotBugs URL: https://spotbugs.github.io/

[3]CWSS URL: https://cwe.mitre.org/cwss/cwss_v1.0.1.html

the vulnerability score will be 0. If the risk sum of all bugs in a project is R, the vulnerability score will be 0.5. So vulnerability scores above 0.5 indicate above average vulnerability, and lower than 0.5 indicate below average vulnerability. When the risk sum goes very high, the vulnerability value will be close to 1.

The current, top-level formula for the risk of a bug is:

$$Risk = Impact * Susceptibility \quad (2)$$

Impact represents how the behavior and data of the software are affected by the bugs present in it. *Susceptibility* defines how easy or often those bugs are executed.

We divide *Impact* into four different sub-aspects: Control Integrity, Data Integrity, Data Confidentiality, and Data Availability, which cover our abstract aspects. They are modeled by the equation:

$$Impact = A * Integrity_{Control} + B * Integrity_{Data} + C * Confidentiality_{Data} + D * Availability_{Data} \quad (3)$$

A, B, C, and D, model the relative importance (or weight) of $Integrity_{Control}$, $Integrity_{Data}$, $Confidentiality_{Data}$, and $Availability_{Data}$, respectfully. The weights of different aspects may be changed according to the actual usage scenarios. It should be noted that, although the formula currently models only negative patterns, positive patterns and mitigations can also be considered in the same way by changing the *Impact* value in a negative way.

Susceptibility indicates how likely the bug may be triggered during runtime. That is, when a software is executed, a bug that is triggered very often is more of a risk than a bug that is triggered rarely. *Susceptibility* can be estimated using the results from the integrated testing mentioned in Section 2.2. The more tests that trigger the bug, the higher the *Susceptibility* should be.

3 IMPROPER AUTHORIZATION EXAMPLE

In this section we briefly discuss how an access control pattern would move through the framework. We will use the "Improper Authorization" bug example, shown in Figure 2, to help explain the flow of our framework.

```
public ResultSet runEmployeeQuery(Connection conn, String name){
    PreparedStatement stmt = conn.prepareStatement("SELECT * " +
        "FROM employees WHERE name = ?");
    stmt.setString(1, name);
    ResultSet rs = stmt.executeQuery()
    return rs;
}
...

//Both "dbConn" and "employeeName" have been defined elsewhere
ResultSet employeeRecord = runEmployeeQuery(dbConn, employeeName);
```

Figure 2: Improper Authorization Bug Code Example

We begin by mining the bug from CWE. Improper Authorization is ID 285 in CWE[4] and occurs when software does not perform, or incorrectly performs, an authorization check when an actor attempts to access a resource or perform an action. After being stored in our learning engine repository, a concrete pattern is developed. For most bugs, multiple code patterns are needed to cover and identify as many variations of the bug as possible. These concrete code patterns are then fed into SpotBugs.

SpotBugs will then execute over a system and when it encounters a piece of code that matches one of the Improper Authorization patterns, it will mark its location in the code. Based on the location we are able to do testing, specifically targeting that piece of code to find if any input would lead to errors, especially errors that correlate with our abstract aspects from Section 2.2. The output of the testing is then used as input to the model.

The model receives all identified and tested bugs. The *Impact* is determined by the possible errors each bug can produce. For example, the Improper Authorization can effect Data Confidentiality and Data Availability. Based on testing, *Susceptibility* of each bug is determined. The overall *Risk* for each bug is then calculated, and finally the Vulnerability score for the system is produced.

[4]https://cwe.mitre.org/data/definitions/285.html

4 CONCLUSION AND FUTURE WORK

We have developed an initial vulnerability model and corresponding tool set to support automatic measurements of software vulnerability of access control violations. While initial studies are promising, the project can be further enhanced and extended.

Our current learning engine is based on bug reports collected from Github and CWE. We plan to enlarge our dataset to mine bugs from a more broad variety of bug datasets. Further, we are currently producing bug patterns manually, which is a tedious and slow process. We will investigate possible machine learning applications to help automatically generate bug patterns.

We are currently using Spotbugs to detect code patterns. One limitation of this tool is that it is Java specific, and requires built software projects to work with. To overcome this limitation, we plan to consider PMD, which takes source code as input and works with many different programming languages (which will also allow us to extend our learning engine to include other programming languages). One issue with PMD is that it may not support some bug code patterns that are currently supported in Spotbugs. Therefore, we plan to adapt such code patterns to PMD.

Our current integration with test coverage is rather preliminary. Currently, we need manual generation and execution of test cases for each software feature. In the next phase, we will automate test execution and the insertion of test coverage build plug-ins.

Finally, we plan to detect positive code patterns to estimate the mitigation of risks in the software project. Specifically, we will first reverse engineer code to class diagrams, and then extract design patterns from those class diagrams.

ACKNOWLEDGEMENTS

We would like to thank Dr. Jianwei Niu, Rodney Rodriguez, MSI STEM Research & Development Consortium (Award #D01_W911SR-14-2-0001-0012), and National Science Foundation (Award #1736209) for their contributions to this project.

REFERENCES

[1] Nathaniel Ayewah, David Hovemeyer, J David Morgenthaler, John Penix, and William Pugh. 2008. Using static analysis to find bugs. *IEEE software* 25, 5 (2008).
[2] Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670.
[3] Guangtai Liang, Jian Wang, Shaochun Li, and Rong Chang. 2014. PatBugs: A Pattern-Based Bug Detector for Cross-platform Mobile Applications. In *Mobile Services (MS), 2014 IEEE International Conference on*. IEEE, 84–91.
[4] Peter Mell, Karen Scarfone, and Sasha Romanosky. 2006. Common vulnerability scoring system. *IEEE Security & Privacy* 4, 6 (2006).
[5] Joseph P Near and Daniel Jackson. 2016. Finding security bugs in web applications using a catalog of access control patterns. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*. IEEE, 947–958.