Predicting Vulnerable Software Components with Dependency Graphs

Viet Hung Nguyen
University of Trento, Italy
vhnguyen@disi.unitn.it

Le Minh Sang Tran
University of Trento, Italy
tran@disi.unitn.it

ABSTRACT

Security metrics and vulnerability prediction for software have gained a lot of interests from the community. Many software security metrics have been proposed e.g., complexity metrics, cohesion and coupling metrics. In this paper, we propose a novel code metric based on dependency graphs to predict vulnerable components. To validate the efficiency of the proposed metric, we conduct a prediction model which targets the JavaScript Engine of Firefox. In this experiment, our prediction model has obtained a very good result in term of accuracy and recall rates. This empirical result is a good evidence showing dependency graphs are also a good option for early indicating vulnerability.

Categories and Subject Descriptors

H.4 [Information Systems Applications]: Miscellaneous; D.2.8 [Software Engineering]: Metrics—complexity measures, performance measures

General Terms

Security, Vulnerability, Prediction

1. INTRODUCTION

It is the fact that most of software have security problems during lifetime. This insecure phenomenon eventually comes from two reasons: complexity and motivations [15]. As requirements are continuously evolving and growing, software is going to include more and more functions while still guaranteeing the backward compatibility. This increases the complexity of the software. Meanwhile, the software market is the market of “lemons” [1, 15], as first comers take better advantages, and have a higher probability to success. Therefore, software vendors lack motivations for putting more testing effort on their products, instead they rush themselves to reduce the time to market.

A trade-off solution widely adopted is to focus only on the most vulnerable parts of software. Concentration on the rest will come later in the maintenance phase, in which discovered vulnerabilities are fixed by perioding patches. For this purpose, software developers can employ security metrics to prioritize testing effort. The product manager, on the other hand, also needs security metrics to determine whether company’s products have reached to an acceptable level of security so that they are ready to the market.

So far many software security metrics have been introduced, but they are inadequate. For example, estimating the complexity of a software system and its correlation with vulnerability could give an indicator to the security level of software’s components. However, the software complexity itself is very difficult to measure. The software complexity can be seen in two perspectives: structural and semantic. The structural complexity is represented in the code structure. Most existing complexity metrics such as total line of code, nesting levels fall into this category. The semantic complexity is the semantic of the code, which is the complexity of the problems, algorithms. Measuring semantic complexity then is more difficult than structural one. As a consequence, current complexity metrics could not perfectly reflect the real nature complexity of software system.

Contribution. We are interested in studying security metrics, in particular metrics for vulnerabilities prediction. In this work, we introduce a new prediction model using dependency graphs of software system. These dependency graphs are based on the relationship among software elements (i.e., components, classes, functions, variables) which can be obtained from a static code analyzers (e.g., Doxygen) or from a detail design specification. Therefore, the proposed model can be applied in both design phase and developing phase. The proposed model is supported by an experiment on JavaScript Engine of Firefox.

The rest of this paper is organized as follows. We briefly present the vulnerability prediction model in the next section (§2), we discuss issues that a prediction model needs to concern, and how to evaluate a prediction model. Then we describe the definition of dependency graph (§3) and our experiment (§4). Next we discuss the validity (§5) of our work, we discuss both internal and external factors that might impact the experiment result. Finally we concisely describe studies in this area for the last ten years (§6) and conclude the paper with future work (§7).

2. VULNERABILITY PREDICTION MODEL

This section discusses common research issues, objectives
of defect/vulnerability prediction (prediction, for short) papers. We also describe a common prediction framework as well as the way to evaluate the prediction models.

The objective of prediction studies is to prioritize explicitly or implicitly testing effort. A good prediction model will indicate vulnerable-prone components for testing. This helps to reduce testing time and provide a certain level of defect-free. To this purpose, there are two typical issues for a prediction model [22]:

- **Classification**: this issue concerns to the question “Are you vulnerable/defective?”
- **Ranking**: in other words, it is the question “How likely you are vulnerable?” or “Which are the most vulnerable components?”

According to [2], there are different approaches implementing the software fault prediction model such as genetic programming, neural networks, case-based reasoning, fuzzy logic, Dempster-Shafer networks, statistics and machine learning based. Also, [2] points out the trends in modern vulnerability researches.

Figure 1 describes a common solution using machine learning techniques. The approach has two phases: training and predicting. In the training phase, code base and vulnerability data are fed into a black box to produce the training dataset which includes code base metrics and vulnerability data. This dataset is used to train the predictor (a.k.a classifier). Table 1 describes some popular classification techniques.

The second phase is the predicting phase which classifies a new code module. The metrics for this module is also generated by the black box. And the trained classifier uses these values to determine whether the code module is vulnerable or not.

The outcome of a prediction model for classification can be true positives (TP), false positives (FP), true negatives (TN), and false negatives (FN) as depicted in Figure 2. To evaluate the prediction model, we use the Precision, Recall, F-measure and Accuracy. The Precision relates the number of true defective components to the number of components predicted as defective.

\[
Precision = \frac{TP}{TP + FP}
\]

![Figure 1: Basic setting for machine learning based prediction model.](image)

The Recall defines the rate of true defective components in comparison to the total number of defective components:

\[
Recall = \frac{TP}{TP + FN}
\]

A good prediction model should obtain high Precision and Recall [14]. However, high recall often comes at the cost for low precision and vice versa. Therefore F-measure is used to combine the precision and recall as a harmonic mean.

\[
F\text{-}measure = \frac{2 \times Recall \times Precision}{Recall + Precision}
\]

The Accuracy metric (Acc or success rate) measures the overall accuracy of the prediction. The Acc is defined as follows:

\[
Acc = \frac{TP + TN}{TP + TN + FP + FN}
\]

Back to the main objective, in order to effectively prioritize testing effort, the recall value should be as high as possible. It means that the model should be able to recognize as much as possible vulnerable components before release time. However, there should be a trade-off between precision and recall. Otherwise, a cheating model always answers vulnerable to any source module, and thus the recall rate reaches to 100%.

### 3. DEPENDENCY GRAPHS

The software code base is composed by many components which are compiling units. A component, depend on the programming language, could be located in a single source file e.g., `.java` in Java, or in multiple ones e.g., header file (.h) and body file (.c, .cpp) in C/C++ languages. We consider
two file, a header file (e.g., .h, .hpp) and its implementation (e.g., .c, .cpp) as a single component. In our work, we assume that header and implementation files share a same name with different extensions.

From a static point of view, a component is the aggregation of data items and executable code. In procedural-oriented languages, the former are global variables (or shared variables) and the later is procedures (or functions). Meanwhile, in object-oriented languages, data items are data members of classes and executable code is class methods. In the case of C++ which supports both procedural and object-oriented specification, data items are both shared variables (or shared data) and data members, and executable code is both functions and methods. Below, we formally present the definition of a software component.

**Definition 1. (Component)** A component is a pair \(\{DC, MC\}\), where \(DC\) is a set of shared variables and data members, and \(MC\) is a set of functions and methods.

A software system, from a static point of view, is a set of components and relations among members in these components. A relation is an ordered pair of members, in which data is flown from the source (first member) to the target (second member). There are four types of relation as follow: parameter passing, function return, data read, and data write. The two first relation types are between a method and a method, where the caller method calls and passes data as parameters to the callee, and might receive returned data from the callee. The two last types are between a method and a data item (data write), or a data item and a method (data read). The following presents a formal definition of a software system in this work.

**Definition 2. (Software system)** A software system is a quadruple \(\langle SC, DS, MS, R\rangle\) where \(SC\) is a set of components, \(DS = \bigcup C \in SC DC\) is a set of data items, \(MS = \bigcup C \in SC MC\) is a set of all functions, \(R \subseteq (DS \cup MS) \times (DS \cup MS)\) is a set of member relations.

Notably, there are two special cases. Firstly, we treat the call to zero-parameter functions as a null parameter-passing relation. Secondly, we treat all kinds of parameter passing as value-passing and ignore the side effect of other kinds of parameter passing e.g., reference-passing, name-passing.

### 3.1 Member Dependency Graph (MDG)

The member dependency graph (MDG) is a direct graph constructed from data items and methods of a software system. The edges connecting two nodes fall into three categories: call-edges, return-edges and data-edges. The two first edge types correspond to the parameter-passing and function-return relations of methods (or functions). Meanwhile, the data-edge can be data read or data write depending on the direction of the edge. Below, we present the formal definition of the MDG.

**Definition 3. (Member dependency graph)** The member dependency graph of a software system \(\langle SC, DS, MS, R\rangle\) is a direct graph \(\mathbf{G}_{MD}(\mathbf{V}_{M}^{d}, \mathbf{V}_{M}^{f}, \mathbf{E}_{M}^{d}, \mathbf{E}_{M}^{f})\) where

- \(\mathbf{V}_{M}^{d} \equiv DS\): is a set of data nodes,
- \(\mathbf{V}_{M}^{f} \equiv MS\): is a set of function nodes,
- \(\mathbf{E}_{M}^{d} \subseteq \mathbf{V}_{M}^{d} \times \mathbf{V}_{M}^{d}\): is a set of call-edges,
- \(\mathbf{E}_{M}^{f} \subseteq \mathbf{V}_{M}^{f} \times \mathbf{V}_{M}^{d}\): is a set of return-edges,
- \(\mathbf{E}_{M}^{f} \subseteq \mathbf{V}_{M}^{d} \times \mathbf{V}_{M}^{f}\): is a set of data-edges.


![Figure 3: Member dependency graph.](image)

The Figure 3 presents a sample member dependency graph of a software system. The round node represents a function, and rectangular node represents a data item. The solid line represents for both call-edge and data-edge regarding to the source and the target. The dash line represents the return-edge. The label on each edge denotes the number of data item transferred and the communication mean which could be an in-process communication or an out-process one e.g., RPC, RMI, WS/Http.

### 3.2 Component Dependency Graph

While the MDG focuses on the member-level of a software system, the component dependency graph (CDG) focuses on the component-level. For the sake of clarity, we use terms member node and member connection to refer to node and edge of MDG, and component node and component connection to refer to those of CDG. We also use the terms connection, edge and relation interchangeably throughout this work. For convenience, we use the notions \(V_{M}^{C}\) and \(E_{M}^{C}\) whenever all of member nodes and edges of an MDG are mentioned, respectively. Obviously, \(V_{C} = V_{M}^{C} \cup V_{M}^{f}\), and \(E_{C} = E_{M}^{d} \cup E_{M}^{f} \cup E_{M}^{d}\).

Basically, the CDG is generated from the MDG by gathering all member nodes of a same component into a component node. Connections between members which belong to two components and have a same direction are collapsed into a single connection between these two components.

**Definition 4. (Component Dependency Graph)** Given a software system \(\langle SC, DS, MS, R\rangle\) and its member dependency graph \(\mathbf{G}_{MD}(\mathbf{V}_{M}^{d}, \mathbf{V}_{M}^{f}, \mathbf{E}_{M}^{d}, \mathbf{E}_{M}^{f})\), the component dependency graph \(\mathbf{G}_{CD}(\mathbf{V}_{C}, \mathbf{E}_{C})\) is constructed as follow: \(V_{C} = \{SC: SC \in SC\}\) is a set of component nodes. \(E_{C} \subseteq \mathbf{S}_{C} \times \mathbf{S}_{C}\) is a set of component connections.

\[E_{C} = \{(C_{1}, C_{2})| C_{i} \in \mathbf{S}_{C}; \exists (m_{1}, m_{2}) \in E_{M}^{d}, m_{i} \in C_{i}, i = 1, 2\}\]

Figure 4 illustrates an example of a CDG generated from the MDG depicted in Figure 3. In this example, the first character of member node is the component name.

### 3.3 Predicting Vulnerable Components

To serve the predicting purpose, the elements of MDG and CDG are enriched with some extra information come from a static code analyzer. The MDG and CDG turn to be sorts of attributed graph in which edges and nodes are associated with certain attributes presented in Table 2. In this table, attributes of node, edge are divided into different groups. Attributes’ short name is presented in the first column, and
<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
<th>Formula</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Member Node Attributes</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LoC</td>
<td>line of code</td>
<td></td>
</tr>
<tr>
<td>LB</td>
<td>blank lines</td>
<td></td>
</tr>
<tr>
<td>V</td>
<td>McCabe complexity</td>
<td></td>
</tr>
<tr>
<td>P</td>
<td>the number of parameters in the method signature</td>
<td></td>
</tr>
<tr>
<td>E</td>
<td>the number of return points in the method</td>
<td></td>
</tr>
<tr>
<td><strong>Member Edge Attributes</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d</td>
<td>number of data items transferred in an edge</td>
<td>[d(e,m_1,m_2) = \begin{cases} P(m_2) &amp; m_2 \in V^m_M \ 1 &amp; \text{otherwise} \end{cases} ]</td>
</tr>
<tr>
<td><strong>Component Node Attributes</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>n</td>
<td>the number of member nodes that build up the component node</td>
<td></td>
</tr>
<tr>
<td>V-density</td>
<td>the McCabe density of a component C</td>
<td>V-density(C) = \frac{1}{n} \sum_{m \in C} V(m)</td>
</tr>
<tr>
<td>I-density</td>
<td>the interface complexity density of a component C</td>
<td>I-density(C) = \frac{1}{n} \sum_{m \in C} (P(m) + E(m))</td>
</tr>
<tr>
<td>e</td>
<td>the number of edges between two arbitrary nodes of a component.</td>
<td></td>
</tr>
<tr>
<td>r_{in}</td>
<td>the number of incoming connections to a component.</td>
<td></td>
</tr>
<tr>
<td>r_{out}</td>
<td>the number of outgoing connections from a component.</td>
<td></td>
</tr>
<tr>
<td>idf</td>
<td>the internal dataflow of a component.</td>
<td>idf(C) = \sum_{e \in \text{INTERNAL}(C)} d(e)</td>
</tr>
<tr>
<td>edf_{out}</td>
<td>the outgoing dataflow of a component.</td>
<td>edf_{out}(C) = \sum_{e \in \text{OUT}(C)} d(e)</td>
</tr>
<tr>
<td>edf_{in}</td>
<td>the incoming dataflow of a component.</td>
<td>edf_{in}(C) = \sum_{e \in \text{IN}(C)} d(e)</td>
</tr>
<tr>
<td>edf</td>
<td>the average internal dataflow</td>
<td>edf(C) = \frac{1}{n} \sum_{e \in \text{IN}(C)} d(e)</td>
</tr>
<tr>
<td>edf_{out}</td>
<td>the average outgoing dataflow</td>
<td>edf_{out}(C) = \frac{1}{n} \sum_{e \in \text{OUT}(C)} d(e)</td>
</tr>
<tr>
<td>edf_{in}</td>
<td>the average incoming dataflow</td>
<td>edf_{in}(C) = \frac{1}{n} \sum_{e \in \text{IN}(C)} d(e)</td>
</tr>
<tr>
<td>eir</td>
<td>the ratio between average outgoing and incoming dataflow</td>
<td>eir(C) = \frac{edf_{out}}{edf_{in}}</td>
</tr>
<tr>
<td>oir</td>
<td>the ratio between outgoing and incoming dataflow</td>
<td>oir(C) = \frac{edf_{out}}{edf_{in}}</td>
</tr>
</tbody>
</table>

Table 2: Attributes of the member node and component node.

Figure 4: The component dependency graph generated from the member dependency graph in Figure 3

the description about the attribute goes into the second column. The last column describes how the attribute value is obtained. The blank means the attribute value comes from a source code analyzer, otherwise it denotes the formula to calculate the value from others.

Depend on the granularity of the vulnerability database, each node will have an extra attribute denoting the number of security bugs of the underlying code entity.

The predictor discussed in section 2 is trained based on values of these attributes. In this study, we use two tuples \((r_{in}, r_{out}, oir, eir)\) and \((r_{in}, r_{out}, V-density, I-density, oir, eir)\), in which \(r_{in/out}\) are the number of incoming and outgoing relations of a component from/to others, \(oir\) is the ratio between the average outgoing and incoming dataflow, \(eir\) is the ratio between the average external and internal dataflow. \(V-density\) and \(I-density\) are, respectively, the density of McCabe and interface complexity. On one hand, we use the first tuple to determine the prediction power of CDG. On the other hand, we use the second one to see if the prediction power can be improved (or even worsened). The following presents the formulas for these attributes of a component.

\[
V-density = \frac{\sum_{m \in C} \text{McCabe complexity of } m}{\text{number of member nodes}}
\]

\[
I-density = \frac{\sum_{m \in C \cup V^d_M} \text{parameters of } m + \text{return points of } m}{\text{number of member nodes}}
\]

\[
oir = \frac{\sum (\text{outgoing dataflow}) \times r_{in}}{\sum (\text{incoming dataflow}) \times r_{out}}
\]

\[
eir = \frac{\sum (\text{external dataflow}) \times \sum (\text{internal edges})}{\sum (\text{internal dataflow}) \times (r_{out} + r_{in})}
\]

4. EXPERIMENT AND DISCUSSION

In this section we present an experiment that classifies and ranks vulnerable components of the JavaScript Engine (JSE) used in Mozilla Firefox. We address JSE because of numerous reasons. First, java script is one of the most attractive and vulnerable source by which browsers are being attacked, (see Figure 2 in [12]). Second, JSE is a part of a popular open source project, Mozilla, so the code base
of JSE is easy to access via CVS server. Third, vulnerability data of JSE is available at both vendor advisories: MFSA and Bugzilla, public vulnerability database (NVD, Bugtraq). Finally, prior papers of Shin and others [16–18] have done analysis on JSE so that we are able to compare our work to theirs.

4.1 Experiment setup

For the code base, two snapshots of JSE in Firefox 1.5 and 2.0 are taken into account. The vulnerability data is obtained by combining several sources as discussed in [3,12,18]. We obtain these data from the Vulnerability Database for Firefox [9] that we have built at the University of Trento. This database contains vulnerability information for Mozilla Firefox from version 1.0 to 3.5. The vulnerabilities of JSE are determined by components containing fixes for these vulnerabilities. If the component is located in JSE source folder (i.e., mozilla/js/src), the correlated vulnerability is supposed to belong to JSE. The granularity of the vulnerability is at component level.

Figure 5 summarizes steps for generating CDG from code base of JSE. Firstly, the code base is fed into Doxygen\(^1\), a source code documentation generation, and RSM\(^2\), a tool calculating standard source code metrics, to generate dependency information for these members. The outcome of Doxygen are analyzed to extract the dependency information (including dependency graph and variables, fields). These information is used to construct the MDG. On the other hand, the functional metrics are extracted from the outcome of RSM, and then embedded into the MDG. Finally, the nodes and edges of MDG are then synthesized to produce the CDG.

The constructed dataset of component nodes’ metrics is used to train and test the predictor. In this work, instead of implementing classification techniques from scratch, we employ WEKA, an open source data mining toolkit, which supports many learning techniques. We use 10-fold cross validation to validate the prediction models. During the validation, the whole dataset is partitioned into 10 folds (segments) randomly. Each fold in turn is held as the test data and a classification is trained on the rest nine folds. In this experiment, we use several classification techniques. Each classifier has its own recall, precision, F-measure and accuracy. We take the average of these values to estimate the quality of our proposed model.

4.2 Result and Discussion

Table 3 shows the 10-fold cross validation result of prediction models constructed from our dataset using four attributes generated from CDG \(\{r_{in}, r_{out}, o_{ir}, c_{ir}\}\). This ta-

---

Table 3: Classification result for JSE of Firefox 1.5 and 2.0

<table>
<thead>
<tr>
<th>Classifier</th>
<th>version 1.5</th>
<th></th>
<th></th>
<th>version 2.0</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Re (%)</td>
<td>Pr(%)</td>
<td>FP rate(%)</td>
<td>Acc(%)</td>
<td>Re (%)</td>
<td>Pr(%)</td>
</tr>
<tr>
<td>Bayesian Network</td>
<td>74.42</td>
<td>55.17</td>
<td>16.15</td>
<td>81.86</td>
<td>62.50</td>
<td>58.14</td>
</tr>
<tr>
<td>Naïve Bayes</td>
<td>69.77</td>
<td>65.22</td>
<td>9.94</td>
<td>85.78</td>
<td>62.50</td>
<td>55.56</td>
</tr>
<tr>
<td>Neural Network</td>
<td>48.84</td>
<td>72.41</td>
<td>4.97</td>
<td>85.29</td>
<td>55.00</td>
<td>81.38</td>
</tr>
<tr>
<td>Random Forest</td>
<td>65.12</td>
<td>62.22</td>
<td>10.56</td>
<td>84.31</td>
<td>57.30</td>
<td>52.27</td>
</tr>
<tr>
<td>SMO</td>
<td>39.53</td>
<td>85.00</td>
<td>1.86</td>
<td>85.78</td>
<td>62.50</td>
<td>55.56</td>
</tr>
<tr>
<td><strong>Average</strong></td>
<td><strong>59.53</strong></td>
<td><strong>68.01</strong></td>
<td><strong>8.70</strong></td>
<td><strong>84.61</strong></td>
<td><strong>60.00</strong></td>
<td><strong>60.60</strong></td>
</tr>
</tbody>
</table>

---

\(^1\)http://www.doxygen.org

\(^2\)http://msquaredtechnologies.com/
Table 4: Comparison of the prediction power between Nesting level metric (Shin et al. [17]) and CDG.

<table>
<thead>
<tr>
<th>Nesting level(*)</th>
<th>CDG(**)</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.5</td>
<td>v2.0</td>
</tr>
<tr>
<td>Accuracy</td>
<td>91.13%</td>
</tr>
<tr>
<td>FP rates</td>
<td>0.98%</td>
</tr>
<tr>
<td>FN rates</td>
<td>89.93%</td>
</tr>
</tbody>
</table>

(*) : values are taken from Shin et al. [17] report.
(**): the average values of techniques listed in Table 3.

5. VALIDITY

In this section we discuss both internal and external threats to the validity of our work.

- **Bugs in tools.** The code base of JSE written in C/C++. In this language, preprocessors such as macro and compiler conditions are widely used. Thus the difficulty of parsing the source code increases for third-party tools rather than C/C++ compiler itself. Therefore, tools like Doxygen, RSM might produce errors. For example, the existence of macros in function declaration misleads Doxygen and RSM in parsing function name and signature. This might cause the same function to be recognized differently by these tools.

- **Bugs in data collection.** The MDG is constructed by gathering reports generated by Doxygen and RSM. The code that parses these reports and constructs the MDG might be buggy and produce errors. This threat however can be mitigated by manual checking the outcome on a small amount of data. Faulty code was fixed and checked again to free the program from bugs as much as possible.

- **Quality of the Vulnerability database.** We collect vulnerability information from [9]. The quality of this database is not clear at the moment. Our work heavily relies on this database. Therefore the quality of this database strongly influences the quality of our prediction model.

6. RELATED WORK


Neuhaus et al. [12] introduce a new metric called import and function-call to construct the prediction model. The imports in programming languages (#include <> in C++, import in Java) allow developers to reuse service provided by another libraries. The import patterns indicate sets of imports that usually go together. Similarly, function-call patterns are groups of functions called together. The experiment conducted on the entire Mozilla code base (13,111 files) reports a good value of precision (70%) and recall (45%). To further estimate the model, the authors use a simple cost model. This model assumes that there are V vulnerabilities distributed arbitrarily among m components, and testers have a “testing budget” of T units. Spending 1 unit on a component will determine whether this component is vulnerable. Obviously, both V and T are less than m. The testers have to randomly choose components for the test. Hence, supported by WEKA, to estimate the vulnerable level of each component. The predicted results are tested against the actual number of vulnerabilities of each component by using Spearman rank correlation (ρ). The correlation strength is judged by the range of ρ [4], where the value that less than 0.3 indicates a weak relation, and one greater than 0.5 means strong relation, and a medium relation for otherwise. In this experiment, we obtain the value of ρ computed by [19] as 0.58 (uncorrected) and 0.45 (corrected). This means that our vulnerable ranking has a good accuracy.
the test efficiency is $TV/m$. With the guide of a prediction model with precision $p$, the test efficiency is $Tp$. Therefore, the prediction model is qualified if $p > V/m$. In practise, they estimate $V$ by $V'$, the number of known vulnerabilities. Applying this quality check, thier prediction model has $m = 10,452$, $V' = 424$, which obviously shows that the model is more than fifteen times better than random test strategy.

Menzies et al. [10] claim that choosing attribute metrics is less significant than choosing how to use these metric values. In the experiment on MDP datasets, they rank different metrics by using InformationGain values to select the metrics for the predictor. The ranking value of a metric is different across projects. The accuracy of the predictor is evaluated by a probability of prediction $(pd)$, and a probability of false alarm $(pf)$. However, Zhang et al. [20] point out that assessment using the Information Retrieval notions of precision and recall rate is better than $pd,pf$. Zhang et al. [21] replicate the work in [10], with a combination of three function-level complexity metrics to do the prediction.

Olague et al. [13] make a comparison of the prediction power of three different metric suites: CK, MOOD and QMOOD. Their experiment is conducted on six versions of Rhino, an open-source JavaScript implementation of Mozilla. The defect data are collected from Mozilla Bugzilla. The authors use logistic regression method to perform the prediction for each metric suite. The result shows that the CK suite is the superior prediction method for Rhino.

Zimmerman et al. [23] build a logistic regression model to predict post-release defects of Eclipse using several metrics in different levels of code base i.e., methods, classes, files and packages. The defect data are obtained by analyzing the log of the source version control. This method is detailed in [24], and employed by [12]. The final dataset is put in the PROMISE repository. In an other work, Zimmerman and Nagappan [22] exploit program dependencies as metrics for their predictor. However, they did not state clearly where the defect data used in their study come from.

Shin and Williams [16–18] raise a research question about the correlation between complexity and software security. In order to validate these hypotheses, the authors conduct an experiment on JSE. They mine the code base of four JSE’s versions for complexity metric values. Meanwhile, faults and vulnerabilities for these versions are collected from MFSA and Bugzilla. The prediction model is based on nesting level metric and logistic regression method. Although the overall accuracy is quite high, their experiment still misses a large portion of vulnerability.

Jiang et al. [8], in their work, study about which is the impact to predicti power of machine learning based vulnerability discovery models, metrics or learning methods. Their experiments are based on MDP datasets, using many metrics belonging to three categories: code level, design level and combination of code and design level, as well as several machine learning methods. The experiment results show that the metrics strongly impact the power of the models, meanwhile, there is not much difference among learning methods. Also, the most powerful metric is the combination of both code and design level metrics, and the design metrics are the most inferior ones.

Gegick et al. [5–7] use warnings of automatic source analysis tools (ASA), code churn and total line of code to implement their prediction model. However, the conducted experiments are based on private defect data sources. Their studies are not reproducible, and hence are less convincing.

Chowdhury and Zulkernine [3] combine complexity, coupling and cohesion metrics for a vulnerability prediction model. These values are fed to a trained classifier to determine whether the source code is vulnerable. In their experiment, the authors used a vulnerability dataset for Firefox, which is assembled from MFSA and Bugzilla.

7. CONCLUSION

In this work, we have proposed a new model to predict vulnerable component in order to prioritize testing effort. Our proposed model also uses existing machine learning technique based on a set of metrics generated from the component dependency graph (CDG) of a software system. The prediction quality of the model is backup by an evidence on JSE of Mozilla Firefox 1.5 and 2.0. The high accuracy and recall reported from the experiment show the possibility of using CDG in the field of vulnerability prediction. Since we can construct the CDG of a software system at the design phase, our model might be used as an early predictor for vulnerability.

In the future, we will replicate our work using nesting level metric to obtain a better comparison between this metric and ours. We will also conduct an experiment for entire code base of several versions of Firefox. And we might want to enrich the CDG with other metrics to improve the prediction quality.

8. ACKNOWLEDGEMENT

We would like to thank professor Fabio Massacci (University of Trento, Italy) and professor Mohammad Zulkernine (Queen’s University, Canada) for many valuable comments and suggestions.

9. REFERENCES


