Journal of Systems Architecture 57 (2011) 294–313

Contents lists available at ScienceDirect

Journal of Systems Architecture

journal homepage: www.elsevier.com/locate/sysarc

Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities

Istehad Chowdhury [a,*], Mohammad Zulkernine [b]

[a] Department of Electrical and Computer Engineering, Queen's University, Kingston, Ontario, Canada K7L3N6
[b] School of Computing, Queen's University, Kingston, Ontario, Canada 7KL3N6

ARTICLE INFO

Article history:
Received 13 October 2009
Received in revised form 30 January 2010
Accepted 2 June 2010
Available online 19 June 2010

Keywords:
Vulnerability prediction
Software metrics
Complexity
Coupling
Cohesion

ABSTRACT

Software security failures are common and the problem is growing. A vulnerability is a weakness in the software that, when exploited, causes a security failure. It is difficult to detect vulnerabilities until they manifest themselves as security failures in the operational stage of software, because security concerns are often not addressed or known sufficiently early during the software development life cycle. Numerous studies have shown that complexity, coupling, and cohesion (CCC) related structural metrics are important indicators of the quality of software architecture, and software architecture is one of the most important and early design decisions that influences the final quality of the software system. Although these metrics have been successfully employed to indicate software faults in general, there are no systematic guidelines on how to use these metrics to predict vulnerabilities in software. If CCC metrics can be used to indicate vulnerabilities, these metrics could aid in the conception of more secured architecture, leading to more secured design and code and eventually better software. In this paper, we present a framework to automatically predict vulnerabilities based on CCC metrics. To empirically validate the framework and prediction accuracy, we conduct a large empirical study on fifty-two releases of Mozilla Firefox developed over a period of four years. To build vulnerability predictors, we consider four alternative data mining and statistical techniques – C4.5 Decision Tree, Random Forests, Logistic Regression, and Naïve-Bayes – and compare their prediction performances. We are able to correctly predict majority of the vulnerability-prone files in Mozilla Firefox, with tolerable false positive rates. Moreover, the predictors built from the past releases can reliably predict the likelihood of having vulnerabilities in the future releases. The experimental results indicate that structural information from the non-security realm such as complexity, coupling, and cohesion are useful in vulnerability prediction.

1. Introduction

There is an increasing number of critical processes supported by software systems in the modern world. Think of the current prevalence of air-traffic control and online banking. When combined with the growing dependence of valuable assets (including human health and wealth, or even human lives) on the security and dependability of computer support for these processes, we see that secure software is a core requirement of the modern world. Unfortunately, there is an escalating number of incidences of software security failures. A security failure is a violation or deviation from the security policy, and a security policy is "a statement of what is, and what is not, allowed as far as security is concerned" [1]. WhiteHat Security Inc. found that nine out of ten websites had at least one security failure when they conducted a security assessment of over 600 public-facing and pre-production websites between January 1, 2006 and February 22, 2008 [2]. The number of security-related software failures reported to the Computer Emergency Response Team Coordination Center (CERT/CC) has increased five-fold over the past seven years [3].

Security failures in a software system are the mishaps we wish to avoid, but they could not occur without the presence of vulnerabilities in the underlying software. "A vulnerability is an instance of a fault in the specification, development, or configuration of software such that its execution can violate an implicit or explicit security policy" [31]. A fault is an accidental condition that, when executed, may cause a functional unit to fail to perform its required or expected function [18]. We use the term 'fault' to denote any software fault or defect, and reserve vulnerability for those exploitable faults which might lead to a security failure.

Vulnerabilities are generally introduced during the development of software. However, it is difficult to detect vulnerabilities until they manifest themselves as security failures in the operational stage of the software, because security concerns are not always addressed or known sufficiently early during the Software

* Corresponding author. Tel.: +1 613 583 4771; fax: +1 613 533 6513.
E-mail addresses: istehad@cs.queensu.ca, istehadc@yahoo.com (I. Chowdhury).

1383-7621/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.sysarc.2010.06.003

Development Life Cycle (SDLC). Therefore, it would be very useful to know the characteristics of software artifacts that can indicate post-release vulnerabilities – vulnerabilities that are uncovered by at least one security failure during the operational phase of the software. Such indications can help software managers and developers take proactive action against potential vulnerabilities. For our work, we use the term 'vulnerability' to denote post-release vulnerabilities only.

Software metrics are often used to assess the ability of software to achieve a predefined goal [4]. A software metric is a measure of some property of a piece of software. Complexity, coupling, and cohesion (CCC) can be measured during various software development phases and are used to evaluate the quality of software [21]. The term software complexity is often applied to the interaction between a program and a programmer working on some programming task [69]. In this context, complexity measures typically depend on program size and control structure, among many other factors. High complexity hinders program comprehension [69]. Coupling refers to the level of interconnection and dependency among software entities. Entities are said to be highly coupled when they depend on each other to such an extent that a change in one necessitates changes in others dependent upon it. Moreover, highly coupled entities are difficult to understand in isolation and reuse because dependant entities must be included. Cohesion refers to the degree that a particular entity provides a single functionality to the software system as a whole [21]. Highly cohesive entities, which have only one responsibility, are more desirable than weakly cohesive entities that do many operations and therefore are likely to be less maintainable and reusable.

Complexity, *coupling* and *cohesion*-related structural measurements pertain to *software architecture* because "the software architecture of a system is the structure or structures of the system, which comprises software elements, the externally visible properties of those elements, and the relationships among them" [71]. These metrics provide complementary solutions that are potentially useful for architecture evaluation [73], leading to more secured software design and code, and eventually more secured and dependable software.

Numerous studies [6–17,21,69] show that high complexity and coupling and low cohesion make understanding, developing, testing, and maintaining software difficult, and, as a side effect, may introduce faults in software systems. Our intuition is that these may, as well, lead to introduction of vulnerabilities – weaknesses that can be exploited by malicious users to compromise a software system. In fact, in one of our previous studies, we have shown that high coupling is likely to increase damage propagation when a system gets compromised [35].

Although CCC metrics have been successfully employed to indicate faults in general [6–17], the efficacy of these metrics to indicate vulnerabilities has not yet been extensively investigated. A very few works associate complexity and coupling with vulnerabilities. Shin and William [31–33] investigate how vulnerabilities can be inferred from (only) code complexity. A study by Traroe et al. [30] uses the notion of "service coupling", a measurement specific to service-oriented architecture. The effect of cohesion on vulnerabilities has never been studied before.

In this work, we explore how the likelihood of having vulnerabilities is affected by *all three* aforementioned aspects – complexity, coupling, and cohesion. This study incorporates some standard and traditional CCC metrics to CCC metrics for object-oriented architecture. Our objective is to investigate whether structural information from the non-security realm such as complexity, coupling, and cohesion metrics can be helpful in automatically predicting vulnerabilities in software.

The principal contributions of this research can be summarized as follows. First, a systematic framework to automatically predict vulnerability-prone entities from CCC metrics is proposed. Second, statistical and machine learning techniques are used to build the vulnerability predictors. In doing so, we compare the prediction performances of four alternative techniques, namely C4.5 Decision Tree, Random Forests, Logistic Regression and Naïve-Bayes. Among these, C4.5 Decision Tree, Random Forests, and Naïve-Bayes have not been applied in any kind of vulnerability prediction before. Third, an extensive empirical study is conducted on fifty-two releases of Mozilla Firefox [39] to validate the usefulness of CCC metrics in vulnerability prediction. In doing so, we provide a tool to automatically map vulnerabilities to entities by extracting information from software repositories such as security advisories, bug databases, and concurrent version systems.

The major implications of this research are as follows. First, automatic predictions of vulnerabilities will assist software practitioners in taking preventive actions against potential vulnerabilities during the early stages of the software lifecycle. Therefore, there will be a shift from reactive to proactive approach to deal with vulnerabilities. Another implication of this research is that techniques to automatically predict fault-prone entities from CCC metrics can be adopted or leveraged to automatically predict vulnerable-prone entities as well, which has not been systematically done as of now. However, the results might not necessarily be the same as for software fault prediction. Although vulnerabilities can be viewed as exploitable faults in software, there is a need to specifically investigate the efficacy of predicting vulnerabilities from CCC metrics. Research has shown that vulnerable entities have distinctive characteristics from faulty-but-non-vulnerable entities in terms of code characteristics [32–34]. Moreover, it has been found that prediction of vulnerable functions from all functions provides better results than prediction of vulnerable functions from faulty functions [35]. Finally, it is implied that robust architecture, and quality design and code are important for security and dependability. Hence, a relationship with the CCC metrics to the vulnerabilities can lead to conception of more secured software architecture, design and code, and eventually more secured and dependable software.

The rest of the paper is organized as follows. In Section 2, we present the framework to predict vulnerability using CCC metrics. In Section 3, we provide background on CCC metrics and give brief overviews of the statistical and machine learning techniques used for vulnerability prediction. In Section 4, we discuss in detail how to predict vulnerability-prone entities using the framework. In Section 5, we report the vulnerability prediction results and discuss the implications of the results. Section 6 compares and contrasts the related work on fault and vulnerability prediction. Finally, we conclude the paper, discuss some limitations of our approaches, and outline avenues for future work in Section 7.

2. Overview of vulnerability-prediction framework

There are two main approaches to software vulnerability prediction. First, count-based techniques focus on predicting the number of vulnerabilities in a software system. Managers can use these predictions to determine if the software is ready for release or if it is likely to have many lurking vulnerabilities. An example of such work is [28]. Second, classification[1]-based techniques emphasize predicting the entities in a software system that are vulnerability-prone or likely to have vulnerabilities. A vulnerability-prone entity can be a function, file, class or other component defined by a manager or a software security engineer, which is likely to have at least one vulnerability in a release. These predictions can assist managers

[1] Classification is one of the most common inductive learning tasks that categorizes a data item into one of several predefined categories [14].

in focusing their resource allocation in a release by paying more attention to vulnerability-prone entities. Examples of such studies are [31–33].

In this study, we treat vulnerability prediction as a classification problem – predicting whether an entity is vulnerability-prone or not. We observed in our preliminary investigation that a very small proportion of entities contain more than one vulnerability in a given release. Various previous studies have also made similar observations and therefore have treated fault and vulnerability prediction as a classification problem [6–12,31–33]. Formal validation and verification and security testing are important requirement of software security [5,71], but they are also resource intensive. Therefore, these activities should be guided where they are needed most. That is, to facilitate efficient vulnerability detection during the SDLC, we need to identify those areas of the software which are most likely to have vulnerabilities (vulnerability-prone), and thus require most of our attention. This approach has the potential to find more vulnerabilities with the same amount of effort. We can draw an analogy between this approach and weather prediction. We are predicting the areas that are likely to experience rainfall (which might include areas that did not experience rainfall before) as opposed to predicting the amount of rainfall.

In this paper, we present a framework for how CCC metrics can be used to automatically predict vulnerability-prone entities, illustrated in Fig. 1. In the step *Map Post-Release Vulnerabilities to Entities,* one can map vulnerabilities (reported in the vulnerability reports) to fixes (in the version archives) and thus to the locations in the code that caused the problem [13,27,29,32]. This mapping is the basis for automatically associating metrics with vulnerabilities. To map vulnerabilities to entities, we find out how many vulnerabilities an entity has had in the past. In doing so, we outline how to extract vulnerability information from security advisories, locate the related entries in bug repositories, and then trace the changes in the source code meant to mitigate vulnerabilities via analyzing the version archives. The locations of vulnerability fixes help us to determine the number of vulnerabilities that were present in those entities. A methodical description of this step is provided in Section 4.3. In *Compute CCC Metrics*, a set of standard CCC metrics is computed for each entity. There are numerous tools available for automatically computing CCC metrics from the source code. A detailed description of how we have computed the metrics and what tools we have used for our study is provided in Section 4.4. Then these CCC metrics and the vulnerability history of entities can be used to build and train the vulnerability predictor, which is done in the *Build Predictor from Vulnerability History and CCC Metrics* step. The resulting *Predictor* takes the CCC metrics of newly developed or modified entities as inputs and generates the probability of a vulnerability occurring in the entity. Based on the calculated probability, it is possible to automatically predict the vulnerability-prone entities, or, in other words, classify entities whether they are vulnerability-prone or not. A detail account of building predictors (training) and the result of applying them (testing) is provided in Section 5.

Several statistical and machine learning techniques are considered to build vulnerability predictors so that the conclusions drawn from the prediction results are not overly influenced by any specific technique. We first use a basic but popular decision-tree-based data mining technique called C4.5 Decision Tree. Then, we compare its prediction capability with Random Forests, an advanced form of decision tree technique. We also consider Logistic Regression, a standard statistical technique, and Naïve-Bayes, a simple but often useful probabilistic prediction technique. Logistic regression has been previously used in vulnerability prediction

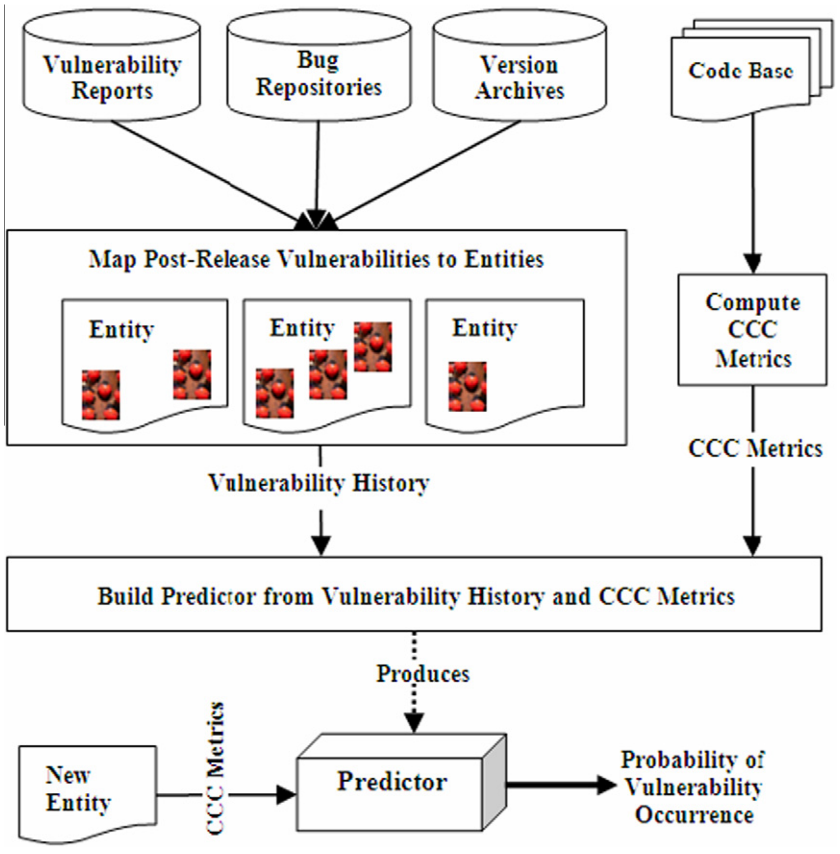


Fig. 1. A Framework to predict vulnerabilities from CCC metrics.

Table 1
CCC metrics that are hypothesized to indicate vulnerabilities.

Metrics	Description and rationale
McCabe's [22]	McCabe's cyclomatic complexity: the number of independent paths through a program unit (i.e., number of decision statements plus one). For a file or class, it is the average cyclomatic complexity of the functions defined in the file or class. The higher this metric the more likely an entity is to be difficult to test and maintain without error.
Modified [22]	Modified cyclomatic complexity: identical to cyclomatic complexity except that each case statement is not counted; the entire switch statement counts as 1.
Strict [22]	Strict cyclomatic Complexity: identical to cyclomatic complexity except that the AND (&& in C/C++) and OR (|| in C/C++) logical operators are also counted as 1.
Essential [22]	Essential cyclomatic complexity: a measure of the code structuredness by counting cyclomatic complexity after iteratively replacing all structured programming primitives with a single statement.
CountPath [21]	CountPath complexity: the number of unique decision paths through a body of code. A higher value of the *CountPath* metric represents a more complex code structure. According to the experience of the software engineers of SciTools Inc. [44], it is common to have a large value for the count path metric, even in a small body of code [45].
Nesting [25]	Nesting complexity: the maximum nesting level of control constructs (if, while, for, switch, etc.) in the function. Highly nested control structures might make a program entity complex and hence difficult to comprehend by a programmer.
SLOC [21]	Source line of code: the number of executable lines of source code. Although SLOC measures size, prior research has found that SLOC highly correlates with complexity.
FanIn [26]	FanIn: the number of inputs a function uses. Inputs include parameters and global variables that are used (read) in the function. FanIn measures information flow.
FanOut [26]	FanOut: The number of outputs that are set. The outputs can be parameters or global variables (modified). FanOut also measures information flow.
HK [26]	Henry Kafura(HK): HK = (SLOC in the function) $\times$ (FanIn $\times$ FanOut)^2. HK measures information flow relative to function size.
WMC*	Weighted Methods per Class (WMC): the number of local methods defined in the class. WMC is related to size complexity. Chidamber et al. empirically validated that the number of methods and complexity of the methods involved is an indicator of development and maintainability complexity [20].
DIT*	Depth of inheritance tree (DIT): the maximum depth of the class in the inheritance tree. The deeper the class is in the inheritance hierarchy, the greater the number of methods it is likely to inherit, making it more complex to predict its behavior [20]. DIT measures the number of potential ancestor classes that can affect a class, i.e., it measures inter-class coupling due to inheritance.
NOC*	Number of children (NOC): the number of immediate sub-classes of a class or the count of derived classes. NOC measures inheritance complexity. The more children a class has, the more methods and instance variables it is likely to be coupled to. Therefore, NOC measures coupling as well.
CBC*	Count of base classes: the number of base classes. Like NOC, CBC measures inheritance complexity and coupling.
RFC*	Response set for a class (RFC): the set of methods that can potentially be executed in response to a message received by an object of that class. RFC is simply the number of methods in the set, including inherited methods. RFC measure both complexity and coupling.
CBO*	Coupling between object classes (CBO): the number of other classes coupled to a class *C*.
LCOM*	Lack of Cohesion Of Methods (LCOM): the LCOM value for a class *C* is defined as $\mathrm{LCOM}(C) = (1 - \lvert E(C)\rvert \div (\lvert V(C)\rvert \times \lvert M(C)\rvert)) \times 100\%$, where $V(C)$ is the set of instance variables, $M(C)$ is the set of instance methods, and $E(C)$ is the set of pairs (v,m) for each instance variable v in $V(C)$ that is used by method m in $M(C)$.

* Indicates CCC metrics applicable only for OO architecture. For more details, consult [20].

using code complexity metrics [32]. The other machine learning techniques have never been applied to vulnerability prediction before.

To empirically validate the prediction accuracy and usefulness of the framework, we conduct case studies on fifty-two Mozilla Firefox releases developed over a period of four years. We are able to predict majority of the vulnerability-prone entities in Mozilla Firefox with tolerable false positive rates. The results of our experiments (Section 5) show that the predictor developed from one release performs consistently in predicting vulnerability-prone entities in the following release.

3. Background

This section provides background on complexity, coupling, and cohesion (CCC) metrics that are hypothesized to affect vulnerability-proneness. It also furnishes brief overviews of the statistical and machine learning techniques used in this study to predict vulnerabilities.

3.1. Complexity, coupling, and cohesion metrics

Complexity, coupling, and cohesion (CCC) related metrics are designed to measure structural quality of software. Table 1 summarizes the CCC metrics. The metrics annotated by "*" in Table 1 are commonly known as Chidamber–Kemerer (CK) metric suite, and the detail descriptions of these metrics can be found in [20]. For further details about rest of the metrics, readers are directed to [21], unless otherwise specified in the table. The rest of the metrics are the standard and traditional CCC metrics selected from prior research on fault and vulnerability prediction [13–16,31–33].

In this study, we are interested primarily in the structural complexity metrics that we can compute from software artifacts such as source code. We consider McCabe's c1yclomatic complexity [22] metric and its several variations to capture different attributes of program complexity. As already mentioned, some metrics such as Weighted Methods per Class (WMC) and Depth of Inheritance Tree (DIT) are applicable only when the software is developed based on OO architecture.

Coupling metrics measure the relationships between entities [24]. "In ontological terms two objects are coupled if and only if at least one of them acts upon the other" [20]. Therefore, if a method invokes another method, then the two methods are coupled. If a class C_1 has an instance of another class C_2 as one of its data members, C_1 and C_2 are coupled. Two entities are also coupled when they act upon the same data (i.e., read or write the same global variables) or communicate though data passing (parameter passing in case of functions and message passing in case of objects). We can observe from the examples that coupling metrics measures information flow complexity. FanIn and FanOut are coupling metrics that measure information flow (e.g., by parameter passing). Between the two general approaches to calculating FainIn and FanOut (informational[2] vs. structural), we take the informational approach [24]. Response set For a Class (RFC) is another example of a coupling metric that measures the design coupling in OO architecture.

Cohesion metrics measure the relationships among the elements within a single module. For example, in object-oriented

[2] Informational approach considers data communication (e.g., through parameter passing), whereas structural approach considers exchange of program control (e.g., via function calls or method invocations).

architecture, if the methods that serve a given class tend to be similar in many aspects, the class is said to have high cohesion. Cohesion is decreased if the methods of a class have little in common and/or methods carry out many varied activities, often using unrelated sets of data. Lack of Cohesion Of Methods (LCOM), initially proposed by Chidamber et al. [20], is one of most common metrics that measure cohesion. We use a slightly modified version of LCOM refined by Henderson-Sellers et al. [75], and a method to measure this LCOM is described in Table 1. Lack of cohesion in an entity implies that it should be split into two or more sub entities. Non-cohesive entities are difficult to reuse, and an attempt to reuse them might introduce problems [20]. This is the rationale for including the LCOM metric. Some of the other cohesion metrics are Tight Class Cohesion and Loose Class Cohesion more commonly known as TCC and LCC, respectively [21]. TCC and LCC are not included in this study because these two metrics are not computed by the metric-computation tool we have used.

3.2. Overview of statistical and data mining techniques

This section provides brief overviews (based on the detail descriptions available in [60]) of the four alternative statistical and data mining techniques used to build vulnerability predictors. The predictors basically classify software entities as vulnerability prone or not vulnerability-prone, which are the categories[3] of the classification problem at hand. The independent variables, also called exploratory variables or features, are the CCC metrics.

3.2.1. Decision Tree

A decision tree technique, such as a C4.5 Decision Tree (DT), generates predictors in the form of an abstract tree of decision rules. The decision rules accommodate non-monotonic and nonlinear relationships among the combinations of independent variables. DT generates predictors that are easy to interpret (logical rules associated with probabilities). Therefore, they are easy to adopt in practice as practitioners can then understand why they get a specific prediction.

3.2.2. Random Forests

Random Forests (RF) is an advanced form of decision-tree-based technique. In contrast to a simple decision-tree-based technique such as C4.5 Decision Tree, RF builds a large number of decision trees. The category of a new sample is determined based on the voting from the generated trees. We consider RF because it is more robust to noise such as inter-correlated features compared to DT. This is important advantage as we are aware of the fact that a number of metrics in our study are actually inter-correlated (e.g., all the complexity metrics). Therefore, we expect RF to maintain a higher prediction accuracy as it is typically found to outperform basic decision trees and some other advanced machine learning techniques in prediction accuracy [36,62].

3.2.3. Logistic Regression

Logistic Regression (LR) a way to classify data into two groups depending on the probability of occurrence of an event for given values of independent variables. In our case, LR computes the probability that an entity is vulnerability-prone for given values of CCC metric. If the probability is greater than a certain cut-off point (e.g., 0.5), then the entity is classified as vulnerability-prone, otherwise not. We include LR in our study because it is a standard statistical classification technique and it is has been used in several earlier studies on predicting fault-prone and vulnerability-prone entities [9,10,13,30,32].

3.2.4. Naïve-Bayes

Naïve-Bayes (NB) is a rule generator based on Bayes' rule of conditional probability. Although Naïve-Bayes tends to be unreliable when there are many inter-related attributes, this simple technique often yields very accurate results. It is also computationally efficient. We consider NB because it has often outperformed more sophisticated classification methods [59].

4. Predicting vulnerabilities

This section describes how to predict vulnerability-prone entities in software as outlined by the framework initially presented in Fig. 1 of Section 2. As an empirical evaluation of the framework, we conduct case studies on Mozilla Firefox to predict its vulnerability-prone files. This section begins by providing an overview of Mozilla Firefox (the source of data for our empirical evaluation). Then, in Section 4.2, we explain the dependent and independent variables of the prediction task at hand. This is followed by a detail account on how different steps in the vulnerability-prediction framework are applied on Mozilla Firefox. First, we describe how we map vulnerabilities in Mozilla back to its files (Section 4.3). Note that, we apply the *Map Post-Release Vulnerabilities to Entities* step of the vulnerability-prediction framework in Section 4.3. Then we specify how we compute complexity, coupling, and cohesion metrics from the code base of Mozilla Firefox in Section 4.4, which elaborates on the *Compute CCC Metrics* step of the framework. Once we have the vulnerability history and CCC metrics, we then describe the implementation and parameter initialization of the statistical and machine learning techniques are be used to build the predictors (as outlined in *Build Predictor from Vulnerability History and CCC Metrics step*). Finally, in Section 4.6, we provide an overview of the tool we have developed to automatically map vulnerabilities to entities in Mozilla Firefox, i.e., to automate the *Map Post-Release Vulnerabilities to Entities* step of the vulnerability-prediction framework.

4.1. Overview of Mozilla Firefox

We chose to conduct our case study on Mozilla Firefox [39], an open source browser. With an approximate user-base of 270 million, it is one of the most popular internet browsers [46]. The Mozilla Firefox project is large not only in terms of user base but also in terms of source code: the code-base of each release of the browser measures more than 2 million lines of code. Moreover, it has a rich history of publicly available vulnerability fixes over a period of four years (January 26, 2005 to April 27, 2009). The integrated nature of the Mozilla repositories enables us to accurately map vulnerabilities to their locations in the source code.

We have conducted case studies on different releases of Mozilla Firefox (releases are analogous to versions). At the time of data collection (March 1, 2009), fifty-two releases of Mozilla Firefox have had vulnerability fixes, from Release 1.0 (R1.0) to Release 3.0.6 (R3.0.6). To validate this study, we have collected vulnerability information from all fifty-two releases as the vulnerabilities are distributed amongst all the releases.

Like [65], we use a file as a logical unit of analysis based on the belief that a file is a conceptual unit of development where developers tend to group related entities such as functions, data types, etc. Other fine-grained units of code can be used, such as individual functions or larger/smaller code chunks that can be determined by a person (system designer or programmer) with good knowledge

[3] In the data mining and machine learning field, the categories are called classes. However, we avoid using "class" to denote "category", because readers may confuse it with a class as a software entity of OO architectures.

of the system. Sub-system or module level[4] analysis is also possible. To facilitate redesign, one might want to analyze from a higher level of abstraction such as sub-systems or modules. By contrast, to facilitate unit testing and code inspection, lower level of abstractions such as file or class level analysis is more appropriate [57]. Analysis at the file level is logical not only because files tend to represent developers' organizational choices, but also it is particularly convenient for automated analysis as it does not require parsing the source code to identify the unit of analysis.

To identify the vulnerable files, we count the number of files that were changed in the course of vulnerability fixes. This can be done by following the links through the Mozilla Foundation Security Advisories (MFSAs) [42] (see Section 4.3 for details). We automate this vulnerability-collection approach initially introduced by Shin and Williams [31]. They count the number of vulnerabilities per function in the Javascript Engine component of Mozilla Firefox. We do the mapping at the file level and for the entire Mozilla Firefox code-base, not just for one component. We only consider the source files (i.e., files with .c, .cpp, .java, and .h extensions) from which the metrics are computed. The various configuration and scripting files to build or test Mozilla Firefox are not examined as they do not represent the source code.

Over the aforementioned period of 4 years and 52 releases, 718 (6.4%) of the total of 11,139 files have had vulnerability fixes. In total, these 718 files suffered from about 1450 vulnerability fixes ranging from one vulnerability fix per file to more than five vulnerability fixes. Fig. 2 presents a histogram of the number of vulnerability fixes per file in Mozilla Firefox. It shows that 454 files have had one vulnerability fix; 141 files have had two vulnerability fixes; 46 files have had three vulnerability fixes; and so on. We can observe that the majority (454) of the 718 vulnerable files have just one vulnerability. Therefore, instead of predicting how many vulnerabilities a file is going to have, we find it logical to predict the vulnerability-prone files i.e., the files that are likely to have one or more vulnerabilities. Hence, we treat the problem of vulnerability prediction as a classification problem, i.e., categorizing files as vulnerability-prone or not.

From Fig. 2, there is another interesting point to make. The vulnerability distribution in the histogram directly contradicts the folklore in fault prediction that says that entities that had problems in the past will likely have problems in the future. As the data from Mozilla Firefox suggests, this might not be true in the case of vulnerabilities (i.e., security-related faults). If that were truly the case, the histogram would show ascending numbers of files with ascending numbers of vulnerabilites. In fact, there are about twice as many files (454) with one vulnerability fix than all files (264) with two or more vulnerability fixes combined. This implies that majority of the files are not repeat offenders, i.e., they do not have vulnerability fixes in the subsequent releases. This observation has been already made by Neuhaus et al. [29] and our findings also confirm this. In fact, when Neuhaus et al. took the concurrent version system (CVS) logs from July 24, 2007 – encompassing changes due to vulnerability reports from February 23, 2007 to July 17, 2007 – they found that 149 components[5] were changed in response to those vulnerability reports. Of these newly fixed components, 81 were repeat offenders, having at least one vulnerability-related fix before January 4, 2007. The remaining 68 components had never had a security-related fix. Therefore, predicting only on the basis of past vulnerability fixes would miss all the aforementioned 68 components. One implication of this observation is that, to make

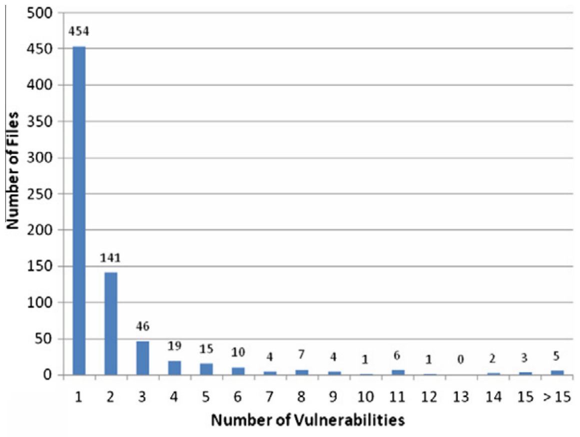


Fig. 2. A histogram showing the number of vulnerabilities per file in Mozilla Firefox.

accurate predictions, we have to go beyond vulnerability history and consider additional factors (such as CCC metrics).

4.2. Dependent and independent variables

A *variable* is any measureable characteristic, whether of a person, a situation, a piece of software, or anything else. In the context of prediction, a *dependent* variable is one about which we make a prediction; an *independent* variable is one that is used to make the prediction. For example, the average age of death could be a dependent variable, and age and gender might be used as independent variables to predict average age of death.

In our case, we are trying to predict whether a file in Mozilla Firefox is vulnerability-prone or not (dependent variable). Since we identify the location of a vulnerability from the location of its fix (i.e., which file the fix is in), predicting the occurrences of vulnerability fixes is equivalent to predicting the likelihood of having post-release vulnerabilities in that file. It is typical for a vulnerability fix to involve several files, and we therefore count the number of vulnerability fixes that were required in that file for developing the *next* release $n + 1$. This aims at capturing the vulnerability-proneness of a file in the current release n. Furthermore, as reported in the preceding section, only a very small portion of files undergo more than one vulnerability fix for a given release, so finding vulnerability-proneness in release n is treated as a classification problem and is estimated as the probability that a given file will undergo one or more vulnerability corrections in release $n + 1$.

The independent variables are the CCC metrics already defined in Table 1. The fundamental hypothesis underlying our work is that the vulnerability-proneness of entities may be affected by their complexity-, coupling-, and cohesion-related structural characteristics.

The Pearson correlations coefficients (denoted by correlation in short) between complexity, coupling, and cohesion (CCC) metrics and the number of vulnerabilities in each file of the five major releases of Mozilla Firefox are presented in Table 2 and Fig. 3. We can observe that the complexity metrics are generally positively correlated to the number of vulnerabilities in Mozilla Firefox for all the five releases. The correlations between many of the CCC metrics and vulnerabilities are slightly above 0.5 for all the releases, and the remaining correlations are less than 0.5 and more than 0.4. Therefore, generally CCC metrics are moderately correlated to vulnerabilities in Mozilla Firefox. The Number of Children (NOC) measuring coupling and complexity due to inheritance is strongly correlated (0.6–0.7) to vulnerabilities. Therefore, it may be a good indicator of vulnerabilities. The overlapping lines (one line for each

[4] A module is generally comprised of several source files implementing a set of related tasks.

[5] Neuhaus et al. define a Mozilla Firefox component as a collection of similarly named source (*.c*/*.cpp*) and header (*.h*) files providing a specific service.

Table 2
Correlations between CCC metrics and vulnerabilities in Mozilla Firefox.

Metrics	Correlations with vulnerabilities				
	R3.0.6	R3.0	R2.0	R1.5	R1.0
McCabe's	0.510	0.513	0.513	0.514	0.510
Modified	0.511	0.514	0.514	0.515	0.511
Strict	0.509	0.513	0.512	0.514	0.510
Essential	0.512	0.514	0.514	0.515	0.512
CountPath	0.497	0.504	0.503	0.504	0.502
Nesting	0.532	0.541	0.541	0.542	0.538
SLOC	0.514	0.518	0.541	0.515	0.514
FanIn	0.532	0.537	0.537	0.538	0.537
FanOut	0.514	0.520	0.520	0.521	0.518
HK	0.529	0.535	0.536	0.536	0.537
WMC	0.429	0.437	0.437	0.442	0.460
DIT	0.459	0.455	0.456	0.475	0.488
NOC	0.642	0.663	0.663	0.662	0.714
CBC	0.457	0.463	0.463	0.467	0.502
RFC	0.434	0.434	0.434	0.439	0.457
CBO	0.454	0.458	0.458	0.462	0.471
LCOM	0.438	0.444	0.444	0.447	0.486

For all the correlations, $p < 0.05$.

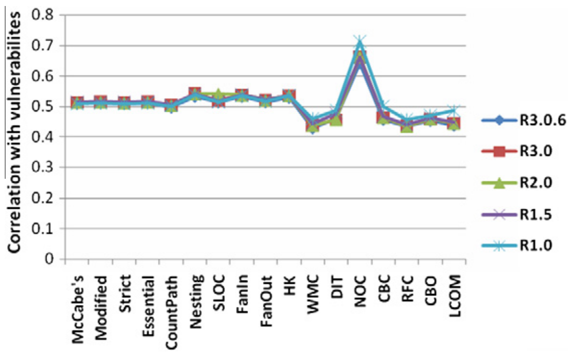


Fig. 3. Plot of correlations between CCC metrics vs. the number of vulnerabilities in Mozilla Firefox.

major release) in Fig. 3 illustrates that the correlation patterns are consistent across releases.

The observed correlations are statistically significant as indicated by the small p-value of t-test ($p < 0.05$). We chose t-statistics over z-score because z-score is used when the population mean and population standard deviation are known for the original population. If such information is not available, it is better to use t-test with sample data [38]. We do not know about the entire population of vulnerabilities in Mozilla Firefox as vulnerabilities before version 1.0 (Release 0.9 and earlier) are not documented.

We can make two inferences from these observations. First, CCC metrics may be useful in vulnerability prediction as they demonstrate moderate but significant correlations to vulnerabilities. Second, because the correlations are positive, highly complex and coupled and non-cohesive files are likely to have more vulnerabilities than less complex and coupled and cohesive files.

4.3. Mapping vulnerabilities to files

To map vulnerabilities to files, we need three types of data: vulnerability reports, a history of source code changes, and a method of tracing from vulnerabilities to the original and changed code of a file. The vulnerabilities in Mozilla Firefox are reported as Mozilla Foundation Security Advisories (MFSAs) [41]. The source code change history is available from concurrent version system (CVS) archives. However, there is no direct link from MFSA to CVS. The trace from vulnerabilities to the original and changed code of an entity can be obtained via Bugzilla, a bug tracking system [40]. A bug tracking system is likely to contain all sorts of entries ranging from corrective to perfective or even specifying preventive maintenance [27]. To avoid confusion, in this section, we refer to any problem posted on Bugzilla as an *issue* and use the term *bug*[6] only for issues requiring corrective maintenance. Our approach of mapping vulnerabilities to files can be summarized in the following three steps which are further explained in the next three subsections:

(1) Retrieve the vulnerabilities from the Mozilla Foundation Security Advisories (MFSAs).
(2) For each vulnerability found in Step 1, indentify the link(s) to the associated bug(s) in Bugzilla.
(3) For each bug found in Step 2, determine the file(s) that were modified to fix the bug and increment the vulnerability count for those file(s).

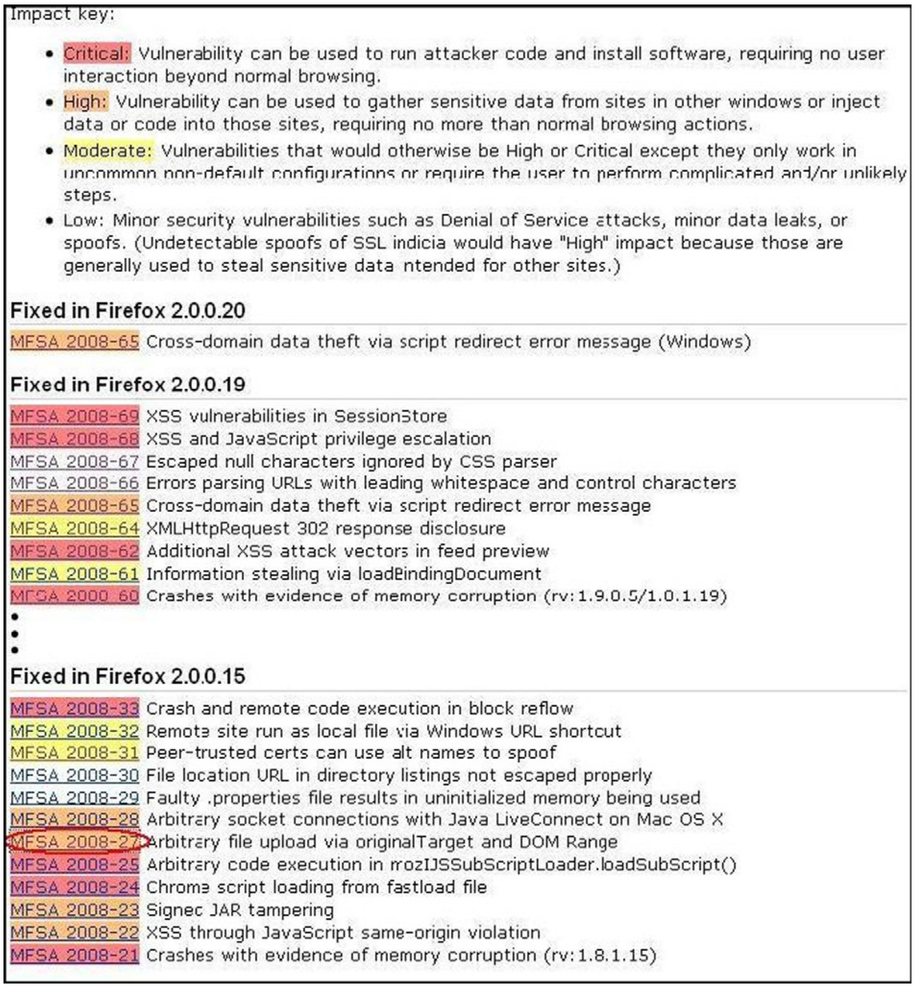
4.3.1. Extracting vulnerability information from MFSA

Vulnerabilities are announced in security advisories that provide users workarounds or pointers to fixed versions and help them avoid security problems. For example, Common Vulnerabilities and Exposures (CVE) [53] lists publicly known information security vulnerabilities and exposures. In the case of Mozilla, the mitigated vulnerabilities are also posted in the Mozilla Foundation Security Advisories (MFSA) page. Fig. 4 shows a section of the list of vulnerabilities posted in MFSA page. Each vulnerability has a unique *MFSA Identifier*, i.e., each MFSA entry corresponds to a vulnerability. For example, *MFSA* 2008-27 (circled in Fig. 4) means that it is the 27th vulnerability discovered in the year 2008. The listing also shows which vulnerabilities have been fixed in which releases. Note that some vulnerabilities can affect multiple releases. For example, the vulnerability *MFSA* 2008-65 was fixed in releases 2.0.0.20 and 2.0.0.9.

Following the MFSA link for each MFSA entry (or vulnerability), we go to the page where the vulnerability is reported in detail in the MFSA (shown in Fig. 5). The *title* and *description* section of the entry tell us that the vulnerability concerns arbitrary file uploads which allows malicious content to force the browser into uploading local files to the remote server. This could be used by an attacker to steal files from known locations on a victim's computer. The References section shown in Fig. 5 contains links to Bugzilla reports which can be used in Step 2 to find the bug(s) associated with the vulnerability.

4.3.1.1. Locating the related bug reports in Bugzilla. Each MFSA has references to bugs corresponding to the vulnerability. These references to bugs can be found in the *reference* section (underlined in Fig. 5). Details about the bugs in Mozilla Firefox can be found in the Bugzilla database. MFSAs have references to the Bugzilla database that typically take the form of links to its web interface, such as *https://bugzilla.mozilla.org/show_bug-.cgi?id=423541*. The six-digit number at the end of the URL is the *bug identifier*. Therefore, from Fig. 5, we know that vulnerability with MFSA ID *MFSA* 2008-27 corresponds to the bug *423541*. For each MFSA page, we extract the links to the Bugzilla bug report using a simple regular expression. In python's raw regular expression syntax, the regular expression looks like: r"\"https://bugzilla.mozilla.org/show_bug.cgi\?id=\d+\"". Fig. 6 shows a screen-shot of a bug report in Bugzilla. Bugzilla records include a description of identified bugs, related components, developers, and the current status of bug fixes and verifications. Bugzilla also provides a link to the modified code for each bug fix. The bug report in

[6] It is important not to confuse this *bug*-a Bugzilla entry indicating a corrective maintenance requirement to fix a vulnerability - with the general definition of bug- a fault in the code [47].



Impact key:

- Critical: Vulnerability can be used to run attacker code and install software, requiring no user interaction beyond normal browsing.
- High: Vulnerability can be used to gather sensitive data from sites in other windows or inject data or code into those sites, requiring no more than normal browsing actions.
- Moderate: Vulnerabilities that would otherwise be High or Critical except they only work in uncommon non-default configurations or require the user to perform complicated and/or unlikely steps.
- Low: Minor security vulnerabilities such as Denial of Service attacks, minor data leaks, or spoofs. (Undetectable spoofs of SSL indicia would have "High" impact because those are generally used to steal sensitive data intended for other sites.)

Fixed in Firefox 2.0.0.20

MFSA 2008-65 Cross-domain data theft via script redirect error message (Windows)

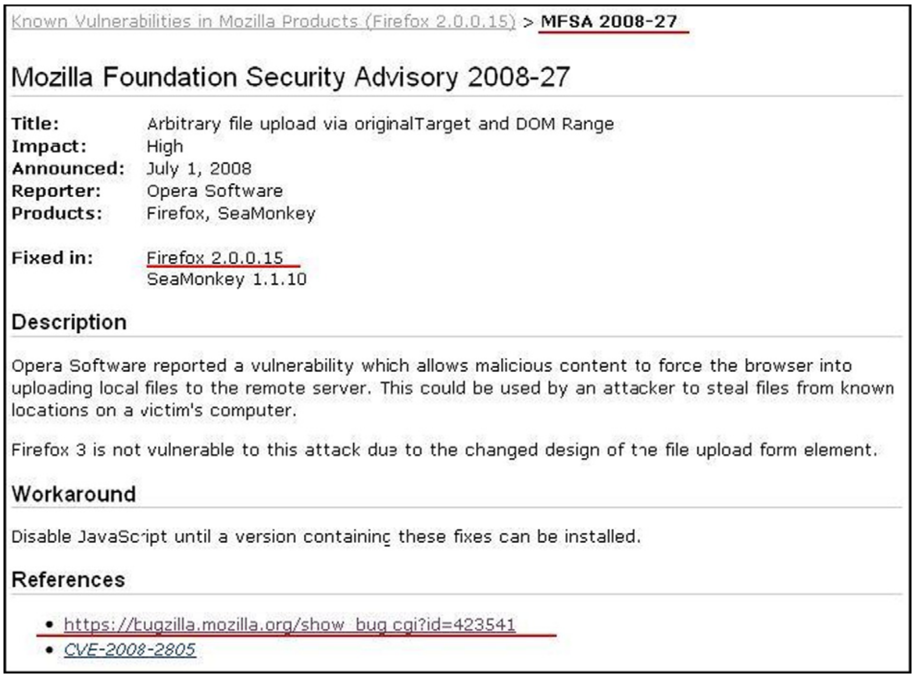
Fixed in Firefox 2.0.0.19

MFSA 2008-69 XSS vulnerabilities in SessionStore
MFSA 2008-68 XSS and JavaScript privilege escalation
MFSA 2008-67 Escaped null characters ignored by CSS parser
MFSA 2008-66 Errors parsing URLs with leading whitespace and control characters
MFSA 2008-65 Cross-domain data theft via script redirect error message
MFSA 2008-64 XMLHttpRequest 302 response disclosure
MFSA 2008-62 Additional XSS attack vectors in feed preview
MFSA 2008-61 Information stealing via loadBindingDocument
MFSA 2008-60 Crashes with evidence of memory corruption (rv:1.9.0.5/1.8.1.19)
•
•
•

Fixed in Firefox 2.0.0.15

MFSA 2008-33 Crash and remote code execution in block reflow
MFSA 2008-32 Remote site run as local file via Windows URL shortcut
MFSA 2008-31 Peer-trusted certs can use alt names to spoof
MFSA 2008-30 File location URL in directory listings not escaped properly
MFSA 2008-29 Faulty .properties file results in uninitialized memory being used
MFSA 2008-28 Arbitrary socket connections with Java LiveConnect on Mac OS X
MFSA 2008-27 Arbitrary file upload via originalTarget and DOM Range
MFSA 2008-25 Arbitrary code execution in mozIJSSubScriptLoader.loadSubScript()
MFSA 2008-24 Chrome script loading from fastload file
MFSA 2008-23 Signed JAR tampering
MFSA 2008-22 XSS through JavaScript same-origin violation
MFSA 2008-21 Crashes with evidence of memory corruption (rv:1.8.1.15)

Fig. 4. A section of the list of vulnerabilities posted in MFSA [48].



Known Vulnerabilities in Mozilla Products (Firefox 2.0.0.15) > **MFSA 2008-27**

Mozilla Foundation Security Advisory 2008-27

Title:	Arbitrary file upload via originalTarget and DOM Range
Impact:	High
Announced:	July 1, 2008
Reporter:	Opera Software
Products:	Firefox, SeaMonkey
Fixed in:	Firefox 2.0.0.15 SeaMonkey 1.1.10

Description

Opera Software reported a vulnerability which allows malicious content to force the browser into uploading local files to the remote server. This could be used by an attacker to steal files from known locations on a victim's computer.

Firefox 3 is not vulnerable to this attack due to the changed design of the file upload form element.

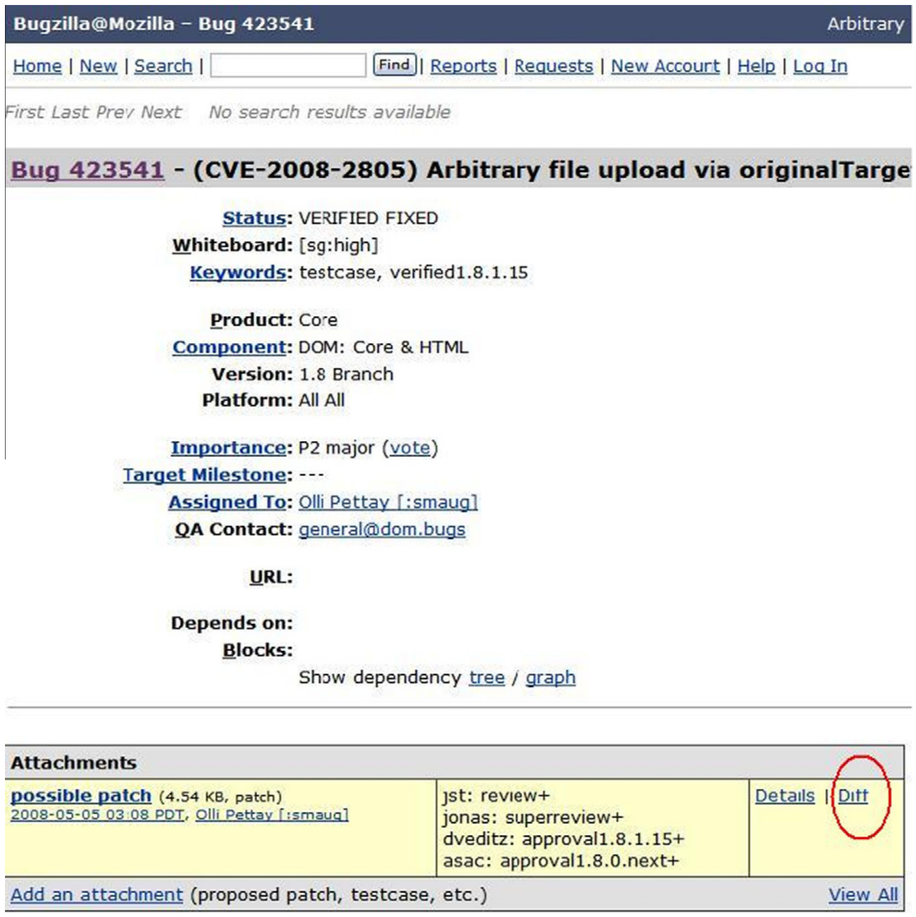
Workaround

Disable JavaScript until a version containing these fixes can be installed.

References

- https://bugzilla.mozilla.org/show_bug.cgi?id=423541
- *CVE-2008-2805*

Fig. 5. An example of a vulnerability report in MFSA [49].


Bugzilla@Mozilla – Bug 423541 Arbitrary

Home | New | Search | Find | Reports | Requests | New Account | Help | Log In

First Last Prev Next No search results available

Bug 423541 - (CVE-2008-2805) Arbitrary file upload via originalTarge

Status: VERIFIED FIXED
Whiteboard: [sg:high]
Keywords: testcase, verified1.8.1.15

Product: Core
Component: DOM: Core & HTML
Version: 1.8 Branch
Platform: All All

Importance: P2 major (vote)
Target Milestone: ---
Assigned To: Olli Pettay [:smaug]
QA Contact: general@dom.bugs

URL:

Depends on:
Blocks:
Show dependency tree / graph

Attachments		
possible patch (4.54 KB, patch) 2008-05-05 03:08 PDT, Olli Pettay [:smaug]	jst: review+ jonas: superreview+ dveditz: approval1.8.1.15+ asac: approval1.8.0.next+	Details | Diff
Add an attachment (proposed patch, testcase, etc.)		View All

Fig. 6. An example of a bug report in Bugzilla [50].

Fig. 6 shows that Bug 423541 is found in the DOM component and the bug fix is verified for the release 1.8.1.15.

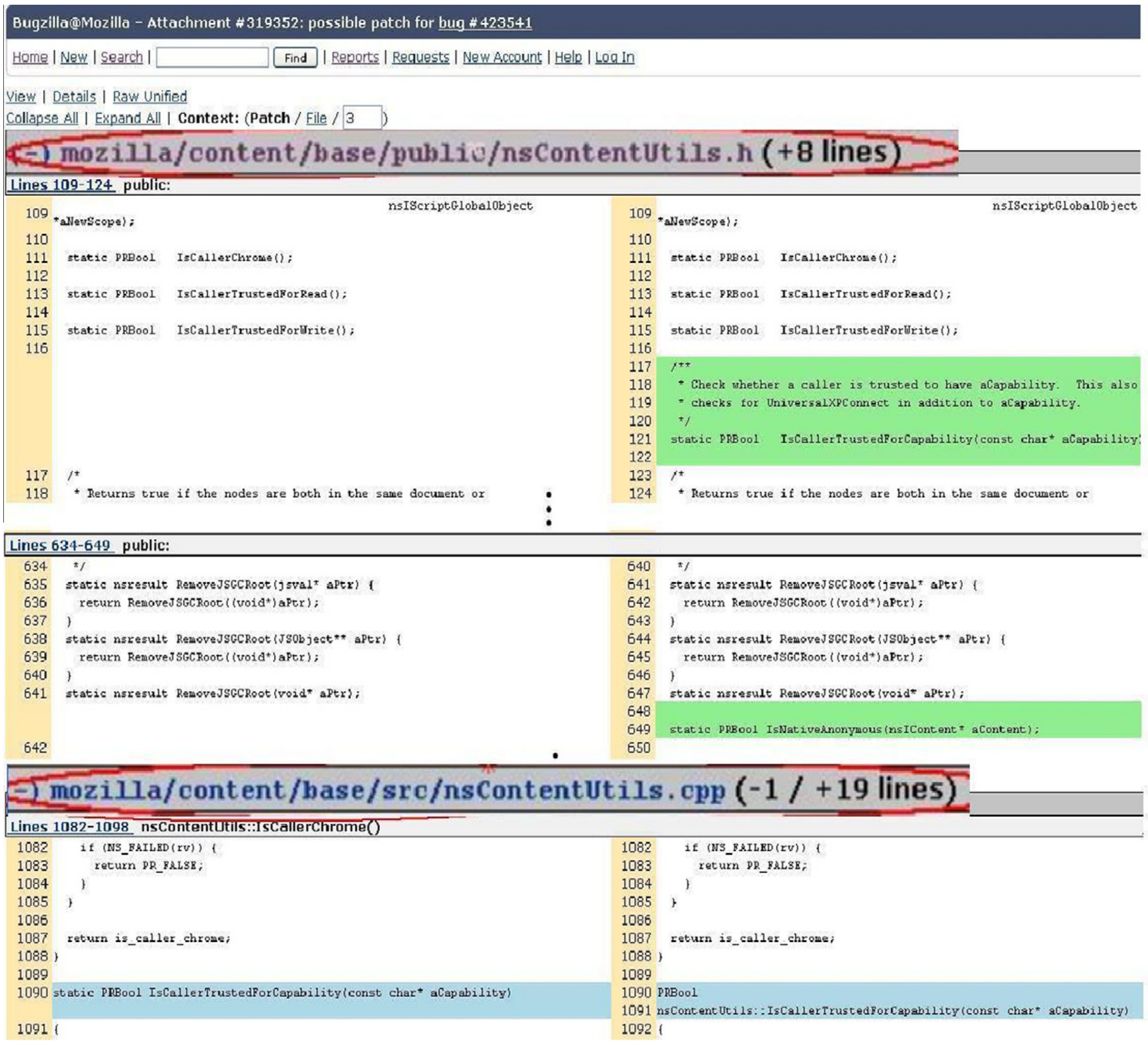
4.3.1.2. Locating the files with vulnerability fixes. The code changed for a bug fix (which corresponds to a vulnerability fix) can be found by following the *Diff* link (circled at the bottom-right corner of Fig. 6). The *Diff* page is shown in Fig. 7, where specific sections of the webpage are magnified to highlight the affected files. Fig. 7 reveals that the files *mozilla/content/base/public/nsContentUtils.h* and *mozilla/content/base/src/nsContent-Utils.cpp* are changed to fix the bug with *bug id* 423541 which corresponds to the vulnerability with MFSA ID *MFSA 2008-27*. Therefore, the vulnerability counts (i.e., how many vulnerabilities a file has had in the past) of the above stated files are increased by one. If these files have had no vulnerability fixes before the *MFSA 2008-27* vulnerability fix, the vulnerability counts of these two files are incremented from zero to one after this vulnerability fix. If the same files are modified to fix another vulnerability (which is represented by a different MFSA ID, e.g., *MFSA 2008-28*), then the vulnerability counts of these files are again incremented by one. That is, the vulnerability counts become two. However, we make sure that the same vulnerability fixes are not counted multiple times. For example, from Fig. 3.3, we know that *MFSA* 2008-27 vulnerability is fixed in release 2.0.0.15. If the same vulnerability is also fixed in release 2.0.0.16, i.e., the files *mozilla/content/base/public/nsContentUtils.h* and *mozilla/content/base/src/nsContentUtils.cpp* are Also modified in release 2.0.0.16 to fix *MFSA* 2008-27, we do not increment the vulnerability counts of these files again as this fix had been already counted in the previous release. Once we have identified the fixes of bugs to mitigate vulnerabilities, we can find out how many vulnerabilities a file has had in the past. Note that a security advisory reporting a vulnerability may refer to multiple bugs, and a bug fix can involve several files.

4.4. Computing complexity, coupling, and cohesion metrics

The CCC metrics are computed by statically analyzing the source code of Mozilla Firefox which can be obtained either by downloading a source archive or by using a source control system like a CVS client. Because we do not intent to check-in (submit) any code back into the source archives, we simply downloaded the source code of specific releases. The source code for a release can be found on the Mozilla FTP (File Transfer Protocol) server [42]. A detailed manual on how to acquire the source code of a specific release via FTP is provided in the Developer Guide [43].

We used a commercial tool called Understand C++ [44] to automatically calculate the metrics from the source code. This tool is used because it is user friendly and it has a good set of APIs to interact with programming languages such as C++, Perl, and Python.

In this study, we compute the metrics at file-level granularity. The metrics for measuring complexity, coupling, and cohesion specifically in OO architecture (i.e., the C–K metric suite annotated by "*" in Table 1) are computed for each class and aggregated to the file level. For example, the file *nsAEClassDispatcher.h* has the classes *AEDispatchHandler* and *AEDispatchTree* with Lack of Cohesion of Methods (LCOM) scores of 55% and 33%, respectively. We take the average LCOM of these two classes, 44%, to be the representative LCOM of the file. In case of all other design-level OO metrics, we add the metric values to aggregate to the file level. For example, the values of the Weighted Methods per Class (WMC) metric



Bugzilla@Mozilla – Attachment #319352: possible patch for bug #423541

Home | New | Search | Find | Reports | Requests | New Account | Help | Log In
View | Details | Raw Unified
Collapse All | Expand All | Context: (Patch / File / 3)

(-) mozilla/content/base/public/nsContentUtils.h (+8 lines)

Lines 109-124 public:

```
109                               nsIScriptGlobalObject   109                               nsIScriptGlobalObject
    *aNewScope);                                             *aNewScope);
110                                                        110
111   static PRBool   IsCallerChrome();                    111   static PRBool   IsCallerChrome();
112                                                        112
113   static PRBool   IsCallerTrustedForRead();            113   static PRBool   IsCallerTrustedForRead();
114                                                        114
115   static PRBool   IsCallerTrustedForWrite();           115   static PRBool   IsCallerTrustedForWrite();
116                                                        116
                                                           117   /**
                                                           118    * Check whether a caller is trusted to have aCapability.  This also
                                                           119    * checks for UniversalXPConnect in addition to aCapability.
                                                           120    */
                                                           121   static PRBool   IsCallerTrustedForCapability(const char* aCapability)
                                                           122
117   /*                                                   123   /*
118    * Returns true if the nodes are both in the same document or    124    * Returns true if the nodes are both in the same document or
```

Lines 634-649 public:

```
634    */                                                  640    */
635   static nsresult RemoveJSGCRoot(jsval* aPtr) {        641   static nsresult RemoveJSGCRoot(jsval* aPtr) {
636     return RemoveJSGCRoot((void*)aPtr);                642     return RemoveJSGCRoot((void*)aPtr);
637   }                                                    643   }
638   static nsresult RemoveJSGCRoot(JSObject** aPtr) {    644   static nsresult RemoveJSGCRoot(JSObject** aPtr) {
639     return RemoveJSGCRoot((void*)aPtr);                645     return RemoveJSGCRoot((void*)aPtr);
640   }                                                    646   }
641   static nsresult RemoveJSGCRoot(void* aPtr);          647   static nsresult RemoveJSGCRoot(void* aPtr);
                                                           648
                                                           649   static PRBool IsNativeAnonymous(nsIContent* aContent);
642                                                        650
```

(-) mozilla/content/base/src/nsContentUtils.cpp (-1 / +19 lines)

Lines 1082-1098 nsContentUtils::IsCallerChrome()

```
1082     if (NS_FAILED(rv)) {                              1082     if (NS_FAILED(rv)) {
1083       return PR_FALSE;                                1083       return PR_FALSE;
1084     }                                               1084     }
1085   }                                                   1085   }
1086                                                       1086
1087   return is_caller_chrome;                            1087   return is_caller_chrome;
1088 }                                                     1088 }
1089                                                       1089
1090 static PRBool IsCallerTrustedForCapability(const char* aCapability)    1090 PRBool
                                                           1091 nsContentUtils::IsCallerTrustedForCapability(const char* aCapability)
1091 {                                                     1092 {
```

Fig. 7. An example *Diff* page showing the changed files as a result of a bug fix [51].

(which is basically the number of methods defined in a class) of *AEDispatchHandler* and *AEDispatchTree* are 9 and 6, respectively. The WMC for the file *nsAEClassDispatcher.h* is thus 15.

Now that we have the CCC metrics and vulnerability history for each file in Mozilla Firefox, we can use this information to implement the *Build Predictors from Vulnerability History and CCC Metrics* step of our proposed framework for vulnerability prediction. The next section is on how we have implemented this step.

4.5. Building vulnerability predictors

The vulnerability predictors are developed by implementing the four statistical and machine learning techniques or algorithms discussed in Section 3.2. Waikato Environment for Knowledge Analysis (WEKA) is a popular, open source toolkit implemented in Java for machine learning and data mining tasks [52]. We chose WEKA for implementing the four statistical and machine learning techniques. The parameters for each of the investigated techniques are initialized mostly with the default settings of the WEKA toolkit as follows.

4.5.1. C4.5 Decision Tree (DT)

We use the well-known J48 WEKA-implementation of the C4.5 algorithm to generate a decision tree. The confidence factor used for pruning is set at 25% and the minimum number of instances per leaf is set at 10. A higher confidence factor incurs more pruning of a decision tree, where pruning refers to discarding one or more sub-trees and replacing them with leaves to simplify a decision tree (while not increasing error rates).

4.5.2. Random Forest (RF)

The number of trees to be generated is set at 10; the number of input variables randomly selected at each node is set at 2; and each tree is allowed to grow to the largest extent possible, i.e. the maximum depth of the tree is unlimited.

4.5.3. Logistic Regression (LR)

In WEKA, LogitBoost[7] with simple regression functions as base learners is used for fitting the logistic models. The optimal number of LogitBoost iterations to perform is cross-validated, which leads to automatic attribute/feature selection. The heuristicStop is set to

[7] "LogitBoost is a boosting based machine learning algorithm that uses a set of weak predictors to create a single strong learner. A weak learner is defined to be a classifier which is only slightly correlated with the true classification. In contrast, a strong learner is a classifier that is arbitrarily well correlated with the true classification" [74]. Interested readers are directed to [74] for more information about LogitBoost and logistic models.

50. If heuristicStop > 0, the heuristic for greedy stopping while cross-validating the number of LogitBoost iterations is enabled. This means LogitBoost is stopped if no new error minimum has been reached in the last heuristicStop iterations. It is recommended to use this heuristic as it gives a large speed-up especially on small datasets. The maximum number of iterations for LogitBoost is set at 500. For very small/large datasets, a lower/higher value might be preferable.

4.5.4. Naïve-Bayes (NB)

The useSupervisedDiscretization is set to *False* so that continuous, numeric attributes (in our case the CCC metrics) are not converted to discretized or nominal ones. Discretization refers to the process of converting continuous features or variables to nominal or categorical features. NB does not require any numeric parameters to be initialized.

In addition to the above parameter initializations, a default threshold (cut-off) of 0.5 is used for all techniques to classify an entity as vulnerability-prone if the predicted probability is higher than the threshold.

There is scope to improve the prediction performance by experimenting with these parameters. However, as we will soon see, it is possible to predict vulnerability-prone files with reasonable accuracy even with the aforementioned default parameters. This proves our point that CCC metrics can be useful additions in vulnerability prediction. The main objective for considering several learning techniques is to demonstrate the efficacy of vulnerability prediction using CCC metrics irrespective of what technique is used, and not necessarily to determine what the best learning technique is. Therefore, in this work, we do not attempt to optimize the parameters for higher accuracy.

4.6. Tool implementation

We have developed a tool to automate the aforementioned method of mapping vulnerabilities to entities by extracting information from the Mozilla repositories. The tool is basically a collection of Python scripts, which currently generates text output. The source code and detailed documentation can be found at our website [56]. To extract information from the html files (i.e., Mozilla repository web-interfaces), we use python's regular expressions library and *BeautifulSoup* [54], a HTML/XML parser module for python.

The tool can be viewed as a combination of logical components as illustrated in Fig. 8. The *Parse MFSA* component parses the Mozilla Foundation Security Advisories (MFSA) web pages (html files) to create a list of vulnerabilities reported in Mozilla Firefox (snapshots of the MFSA pages are shown in Figs. 4 and 5). For each vulnerability, the tool extracts information such as the MFSA link to the vulnerability report, description of the *bug-ids* that correspond to the vulnerability. The *Retrieve Bugzilla Entries for Each Vulnerability* component queries Bugzilla to retrieve the bug reports that resulted from or were associated with the vulnerability advisory i.e., MFSA. The bug reports (html files, example shown in Fig. 6) are parsed to retrieve the links to code fixes in the CVS repository. Then the *Trace Code-change for Bug Fix* component follows the links to the CVS repository to identify the changed code and files that were modified to fix the bug (and hence the vulnerability). Note that, the components *Parse MFSA, Retrieve Bugzilla Entries for Each Vulnerability,* and *Trace Code-change for Bug Fix* and *Map Vulnerabilities to File* implement Steps 1, 2, and 3, respectively, described in Section 4.3. The entire source tree or source code of a specific release is downloaded by the *Download Source Tree* component. The downloaded source code is then fed into *Compute CCC Metrics Using Understand 2.0.* Understand 2.0 is the third-party, commercial tool used to automatically compute CCC metrics (the component is grayed-out to indicate the use of third-party tools). An

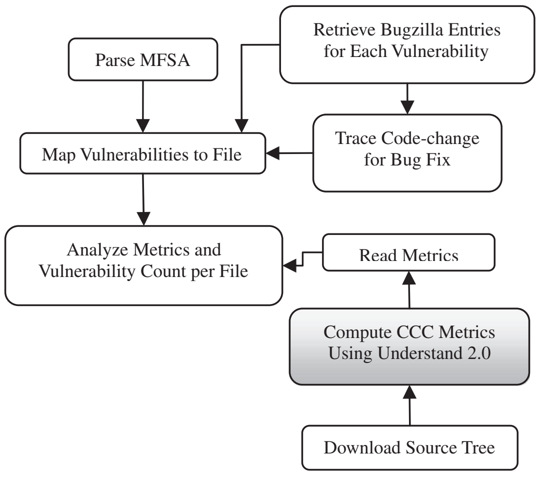

Parse MFSA
Retrieve Bugzilla Entries for Each Vulnerability
Map Vulnerabilities to File
Trace Code-change for Bug Fix
Analyze Metrics and Vulnerability Count per File
Read Metrics
Compute CCC Metrics Using Understand 2.0
Download Source Tree

Fig. 8. An overview of the tool to automatically extract vulnerability data.

interface module, *Read Module,* is written so that our tool can read, preprocess, and aggregate the metrics at file level from the raw *csv* (comma separated values) files generated by Understand 2.0. Finally, *Analyze Metrics and Vulnerability Count per File* component also generates the appropriate data set to train and test the vulnerability predictors used in this study. It also performs the statistical analyses to compute the correlations between CCC metrics and vulnerabilities in Mozilla Firefox files (reported in Table 2, Fig. 3, and in our previous research [37]). The statistical analysis is done via *stats.py* [55], a python library developed at Cornell University, USA.

Now that we have the predictors, the next section discusses how we can quantitatively evaluate the performance of the predictors.

4.7. Prediction performance measures

So far we have discussed how to extract relevant information and build vulnerability predictors. Before applying the predictors, we need to know how to evaluate their performance. The performance of a predictor can be measured in several ways. Most frequently used measures are accuracy, recall, precision, false positive rate, and false negative rate. For the two-class problem (e.g., vulnerability-prone or not vulnerability-prone), these performance measures are explained using a confusion matrix, shown in Table 3.

The confusion matrix shows the actual vs. the predicted results where:

- True Negative (TN) = The number of files predicted as not being vulnerability-prone where no vulnerability is discovered in those files.
- False Positives (FP) = The number of files incorrectly predicted as vulnerability-prone when they are not vulnerable.
- False Negative (FN) = The number of files predicted as not being vulnerability-prone which turn out to have a vulnerability.

Table 3
Confusion matrix.

Actual	Predicted as	
	Not vulnerable	Vulnerable
Not vulnerable	TN = True Negatives	FP = False Positives
Vulnerable	FN = False Negatives	TP = True Positives

- True Positives (TP) = The number of files predicted as being vulnerability-prone which are in actual fact vulnerable.

From the confusion matrix, several of the prediction performance measures such as accuracy, precision, recall, F-measures, false positive rate, and false negative rate can be derived as follows.

4.7.1. Accuracy

Accuracy is also known as overall correct classification rate. It is defined as the ratio of the number of files correctly predicted to the total number of files as shown in Eq. (1):

$$Accuracy = \frac{TP + TN}{TP + FP + TN + FN} \quad (1)$$

4.7.2. Precision

Precision, also known as the correctness, measures the efficiency of prediction. It is defined as the ratio of the number of files correctly predicted as vulnerability-prone to the total number of files predicted as vulnerability-prone, as shown in Eq. (2).

$$Precision = \frac{TP}{TP + FP} \quad (2)$$

4.7.3. Recall

Recall is the vulnerable entity detection rate which quantifies the effectiveness of a predictor. It is defined as the ratio of the number of files correctly predicted as vulnerability-prone to the total number of files that are actually vulnerable. The formulae to calculate recall is given in Eq. (3).

$$Recall = \frac{TP}{TP + FN} \quad (3)$$

Both precision and recall are important performance measures. The higher the precision, the less effort wasted in testing and inspection, and the higher the recall, the fewer vulnerable files go undetected. Unfortunately, there is a trade-off between precision and recall. For example, if a predictor predicts only one file as vulnerability-prone and this file is actually vulnerable, the precision will be 100%. However, the predictor's recall will be low if there are other vulnerable files. In another example, if a predictor predicts all files as vulnerable, the recall will be 100%, but its precision will be low. Therefore, a measure is needed which combines recall and precision in a single efficiency measure.

4.7.4. F-measure

F-measure can be interpreted as a weighted average of precision and recall [60]. For convenient interpretation, we also express it in terms of a percentage like our other performance measures so it reaches its best value at 100 and its worst at 0. The general formula for F-measure is given in Eq. (4), where F_β-measure "measures the effectiveness of prediction with respect to a user who attaches β times as much importance to recall as precision" [67]

$$F_\beta - Measure = \frac{1 + \beta^2 \times Precision \times Recall}{(\beta^2 \times Precision) + Recall} \quad (4)$$

The traditional *F*-measure, denoted by F_1-measure, gives equal importance to both precision and recall by taking their harmonic mean [60]. Two other commonly used *F*-measures are the F_2-measure and $F_{0.5}$-measure. F_2-measure weighs recall twice as much as precision whereas $F_{0.5}$-measure weighs precision twice as much as recall.

We believe that it is more important to identify the vulnerable files, even at the expense of incorrectly predicting some non-vulnerable files as vulnerability-prone. To draw an analogy, consider well-known philosophical quote that states, "Better ten guilty persons go free than one innocent person is punished" [66]. In the case of vulnerability prediction, this quote can be rephrased as "Better ten non-vulnerable files are investigated than one vulnerable file going unnoticed". This is because a single vulnerable file may lead to serious security failures. Given that, we think more weight should be given to recall than precision. Therefore, we include the F_2-measure, which weights recall twice as much as precision, to evaluate a predictor.

Some researchers choose to use the false positive rate (FP rate) and the false negative rate (FN rate) instead of precision and recall. Ostrand and Weyuker [58] in particular argue that false positive rate (Type I misclassifications) and false negative rate (Type II misclassifications) are the most important measures. We also believe that these measures are effective in evaluating vulnerability prediction models. The FP rate and FN rate are defined in Eqs. (5) and (6), respectively.

A high FN rate indicates that there is a risk of overlooking vulnerabilities, whereas a high FP rate indicates effort may be wasted in investigating the predicted vulnerable entities. These notions are highly related to recall and precision; in fact, *recall* = 1–*FN rate* and precision is inversely proportional to FP rate. Therefore, it is redundant to use all of them to indicate prediction performance.

$$FP\ rate = \frac{FP}{FP + TN} \quad (5)$$

$$FN\ rate = \frac{FN}{TP + FN} \quad (6)$$

In this study, we use accuracy, recall and FN rate as employed in [61]. All these measures are expressed in percentages. In addition, we use F_1-measure and F_2-measure to combine the precision and recall into a single measure.

The next section describes the result of predicting vulnerability-prone files, and the accuracies of predictions using different techniques are quantitatively evaluated and compared using the aforementioned performance measures.

5. Results and discussion

This section presents the results of predicting vulnerability-prone files in Mozilla Firefox based on their complexity, coupling, and cohesion (CCC) metrics. These results will help us quantitatively evaluate the usefulness of using CCC metrics for vulnerability prediction.

5.1. Need for a balanced training set

To build and evaluate the predictors, file-level CCC metrics and vulnerability data from the 52 releases of Mozilla Firefox are used. There are 718 vulnerable files (minority category) as opposed to 10,421 non-vulnerable files (majority category) in the obtained dataset (a total of 11,139 instances). Training the predictors on such an imbalanced data set would produce biased predictors towards the more dominant category [60] (in this case, the non-vulnerable files). This phenomenon is known as over fitting. The result of training predictors on an imbalanced data set is shown in Table 4 for C4.5 Decision Tree (DT) technique. Because the predictor becomes biased towards the overwhelming majority category, it predicts many vulnerable files as non-vulnerable files. Consequently, it misses many vulnerable files. This is reflected by the low recall of 23.3%. Naturally, the FP rate is low as files are predicted very rarely as vulnerable-prone by the biased predictor. Notably, the high overall accuracy of 93.21% is misleading. It would be possible to be just as accurate by blindly predicting all files as non-vulnerable, because 93% of the instances in the data set correspond to non-vulnerable files. Even with such high accuracy and low FP-

Table 4
Performance of a biased predictor built from imbalanced data set.

Accuracy	Recall	FP rate	F_1-measure
93.21	23.30	1.10	23.30

rate, such a biased predictor is practically useless because it would miss a majority of the vulnerable files. Similar results are obtained for all other techniques, and, for simplicity of description, we report only the result for DT in Table 4.

To facilitate the construction of unbiased predictors, we create a balanced subset (1463 instances) from the complete training set which consists of the 718 instances representing the vulnerable files and a random selection of 718 instances representing non-vulnerable files. Many prior studies have also under-sampled the majority category instances to obtain a balanced training set [57,61,62]. The problem with under-sampling is that we lose data we can learn from. This is a significant problem when one works with a small sample size, which is not a major problem in our case. The other possible technique would be to perform over-sampling of the minority category, i.e., duplicating the instances representing vulnerable files. Over-sampling the minority category can raise their weight to decrease their error rate. The same result can be achieved with an under-sampling technique, which also speeds up the predictor building significantly by reducing the size of the dataset.

As opposed to these sampling techniques, cost-based prediction can be adopted to deal with the imbalanced data set. In cost-based prediction, we could set higher weights to instances representing vulnerable files. Then the error in predicting the minority class would decrease at the expense of the increase in the overall prediction error. In addition, the interpretation of standard prediction performance measures would become unintuitive [60].

With a balanced data set, the proportion of vulnerable and non-vulnerable files is exactly 50%. Therefore, a random prediction is likely to be correct 50% of the time. For the predictors to prove useful, they should be correct more than 50% of the time when classifying into either vulnerable or non-vulnerable files. We will be able to evaluate the usefulness of the predictors from the results presented in the next section.

5.2. Prediction performance of different techniques

We have considered four alternative data mining and statistical techniques – C4.5 Decision Trees (DT), Random Forests (RF), Logistic Regression (LR), and Naïve-Bayes (NB) – and compared their prediction performances. Using DT as the baseline scheme, we perform a statistical significance test to determine whether the differences in performance measures by other techniques are statistically significant. The term statistical significance refers to the result of a pair-wise comparison of schemes using either a standard *t*-test or the corrected re-sampled *t*-test [37]. We used the latter *t*-test, because the standard *t*-test can generate too many significant differences due to dependencies in the estimates (in particular when anything other than one run of an *x*-fold cross-validation is used). As the significance level decreases, the confidence in the conclusion increases. Traditionally, a significance level of less than or equal to 0.05 (the 95% confidence level) is considered statistically significant. Therefore, we have performed the corrected re-sampled *t*-test at 0.05 significance.

The results are obtained by performing a *10-fold cross-validation* to reduce variability in the prediction. *Cross-validation* is a technique for assessing how accurately a predictive model will perform in practice [60]. The dataset is randomly partitioned into 10 bins of equal size. For 10 different runs, 9 bins are picked to train the predictors and the remaining bin is used to test them, each time leaving out a different bin for testing. We use stratified sampling to make sure that roughly the same proportion of vulnerable and non-vulnerable files are present in each bin while making the random partitions. In addition, to reduce the chances for the predictors to be overly influenced by peculiarities of any given release, we randomly select training sets across the releases. Finally, we compute the mean and the standard deviation for each performance measure over these 10 different runs (40 runs in total for four techniques). The mean and standard deviation (StDev) in accuracy, recall, FP rate, and F_1-measure for each technique are presented in Fig. 5. The significance column (Sig.?) in Table 5 reports the results of the statistical significance test. The annotation "Yes+ " or "Yes–" indicates that a specific result is statistically better (+) or worse (−) than the baseline scheme (in this case, DT) at the specified significance level (0.05). On the other hand, "No" means the specific performance measure of the two techniques might be different, but the difference is statistically insignificant.

There are few points that we would like to emphasize from Table 5. First, for most techniques, the predictions are more than 70% accurate. This demonstrates the efficacy of using CCC metrics in vulnerability prediction, irrespective of what learning or prediction technique is used. Second, we are able to correctly predict 74% of the vulnerability-prone files with an overall accuracy of 73%. The results are promising given that we are trying to predict something about security using information from the non-security realm and learning techniques with the default parameter settings of the WEKA tool. Third, the standard deviations in the different performance measures are very low. The low standard deviations indicate that there is little fluctuation in the prediction accuracies in different runs of the experiments.

For a detailed assessment of the performances of different prediction techniques, the accuracy, recall, FP rate and F_1-measure of the different techniques are compared graphically in Fig. 9. In Fig. 9a, b, and d, the closer a technique appears to the top left corner, the better the technique in terms of the overall accuracy, recall, and F_1-measure, respectively. In Fig. 9c, the closer a technique is to the bottom left corner, the better the technique as far as FP rate is concerned. From Table 5 and Fig. 9a, we can observe that RF's overall prediction accuracy is slightly higher than DT and significantly better than LR and NB. However, RF's variance in the overall accuracy is higher than that of DT and LR. Nonetheless, the differences in mean and standard deviation in overall accuracy are not statistically significant. As far as recall is concerned, DT outperforms all other techniques as shown in Table 5 and Fig. 9b. Therefore, DT can be an efficient technique in predicting a maximum number of vulnerability-prone files in Mozilla Firefox. In contrast, many vulnerable entities may remain unnoticed if prediction is made using a posterior probability based technique such as NB. The higher recall in DT is also statistically significant compared to LR and NB. It is surprising that DT can detect more vulnerability-prone files with a lower variance in detection rate than its more advanced counterpart, RF. In the case of false positive rate (Fig. 9c), we see the opposite scenario. The false positive rate is at a maximum in DT and at a minimum in NB. This means that although DT can detect more vulnerable files, it is also likely to raise more false alarms. As already mentioned, there is a trade-off between recall and false positive rate. This issue is further explored in Section 5.2.1. One may have to tolerate a number of false positives to ensure that a minimum number of vulnerability-prone files go unpredicted. From Fig. 9d, we observe that DT strikes a higher balance between recall and FP rate, i.e., between efficiency and effectiveness as its F_1-measure is higher than other techniques.

It will be interesting to observe how accurately different techniques predict the vulnerability-prone entities when we put more emphasis on recall by analyzing the F_2-measures of the four

Table 5
Prediction performance of different techniques.

Technique	Accuracy			Recall			FP rate			F_1-measure		
	Mean	StDev	Sig.?	Mean	StDev	Sig.?	Mean	StDev	Sig.?	Mean	StDev	Sig.?
DT	72.85	1.46		74.22	0.03		28.51	0.05		73.00	0.01	
RF	72.95	1.57	No	69.43	0.04	No	23.53	0.04	No	72.00	0.02	No
LR	71.91	1.33	No	59.39	0.03	Yes-	15.58	0.03	Yes+	68.00	0.02	Yes-
NB	62.40	1.29	Yes-	29.18	0.03	Yes-	4.39	0.01	Yes+	44.00	0.03	Yes-

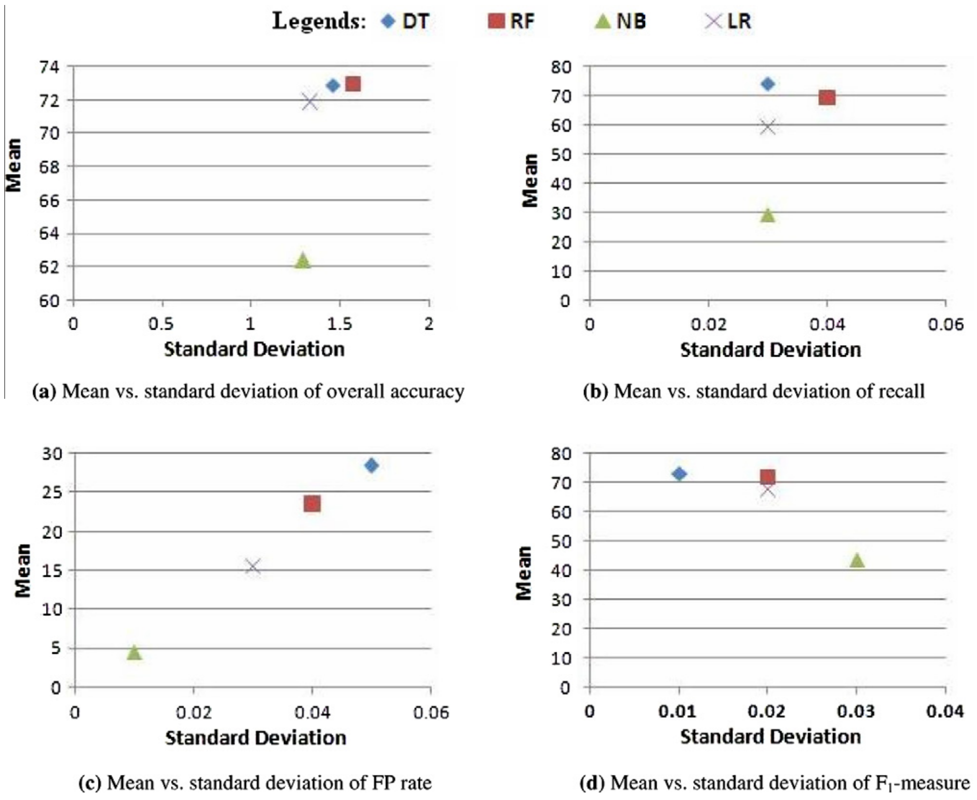
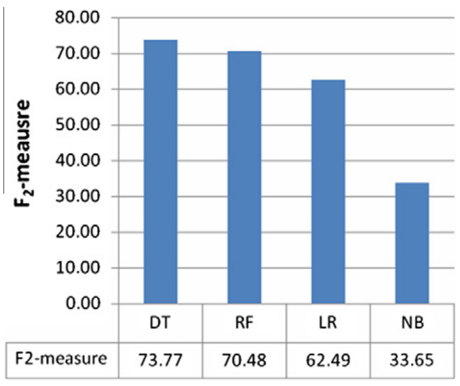


Fig. 9. Mean vs. standard deviation of different prediction measures.

techniques. Recollect that the F_2-measure evaluates the accuracy of predicting vulnerability-prone files while considering recall to be twice as important as precision. Fig. 10 compares the F_2-measures of DT, RF, LR and NB techniques, with DT showing the best performance. Therefore, if the objective is to correctly predict a higher percentage of vulnerable files, then DT promises to be the preferred technique, although in overall prediction accuracy RF performs better.

5.2.1. Trade-off between recall and FP rate

One would ideally like to achieve high recall and a low false positive rate (FP rate). Unfortunately, the FP rate typically increases with any increase in recall. To investigate the trade-off between the vulnerability-prone file detection rate (recall) and the false positive rate (FP rate), we plot a graph of recall vs. FP rate (labeled as *recall* in Fig. 11). Such plots are also called Receiver Operating Curve (ROC). ROC curves are often used to visualize the performance of a predictor in detecting the true class (in our case vulnerability-prone files) [61,62]. For ease of explanation, we first present the ROC curve of a single technique, namely C4.5 Decision Tree, in Fig. 11. Fig. 11 illustrates that we can correctly identify about 60% of the vulnerability-prone files while keeping the false positive rate below 20%, 74% of the vulnerability-prone files with a false positive rate of below of 30%, and so on.



	DT	RF	LR	NB
F2-measure	73.77	70.48	62.49	33.65

Fig. 10. Comparison of F_2-measures of different techniques.

As mentioned before, it is more important to correctly predict most of the vulnerable files, even at the expense of incorrectly predicting some non-vulnerable files as vulnerability-prone. Therefore, reasonably high FP rates are tolerable. However, the FP rate also cannot be too high; otherwise a predictor will be deemed useless in practice. The F_1-measure helps to identify the optimum

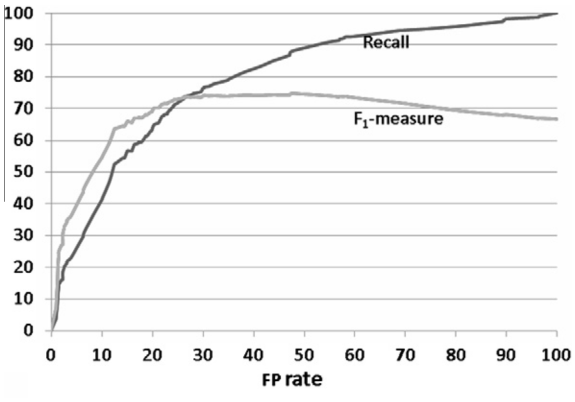


Fig. 11. Plot of recall and F_1-measure vs. FP rate.

point of operation, i.e., to find a point of balance between efficiency and effectiveness. This is why we also place the graph of F_1-measure vs. FP rate (labeled as F_1-measure) on Fig. 11. The optimum point of operation can be obtained at the point of intersection of the recall vs. FP rate and F_1-measure vs. FP rate curves as shown in Fig. 11. The intersection point is termed as optimum because beyond that FP rate the overall accuracy deteriorates.

Fig. 12 presents the ROC curves for all four techniques, where we can again observe the trade-off between recall and FP rate. A good predictor would achieve high recall with a low FP rate. These ROC curves can be used to visualize predictors' performances in terms of correctly predicting the vulnerability-prone files. If the ROC curve of a technique T_A lies above that of another technique T_B, then T_A performs better than T_B in predicting the vulnerability-prone files at the same FP rate. It can be seen that different techniques performs better than others in different regions of the curves. The operative region is highlighted by the dashed rectangle (about where the curves of recall vs. FP rate bend, changing from having tangents with slopes greater than one to tangents with slopes less than one). This is because if recall is too low (e.g., lower than 50% or 60%) would make a predictor ineffective no matter how low the FP rate is. Similarly, a high FP rate (e.g., higher than 50%) would make it inefficient no matter how high the recall is. In the operative region, the decision-tree-based techniques generally perform better than LR and constantly perform better than NB. Between the decision-tree-based techniques, DT occasionally performs better than RF, and vice versa.

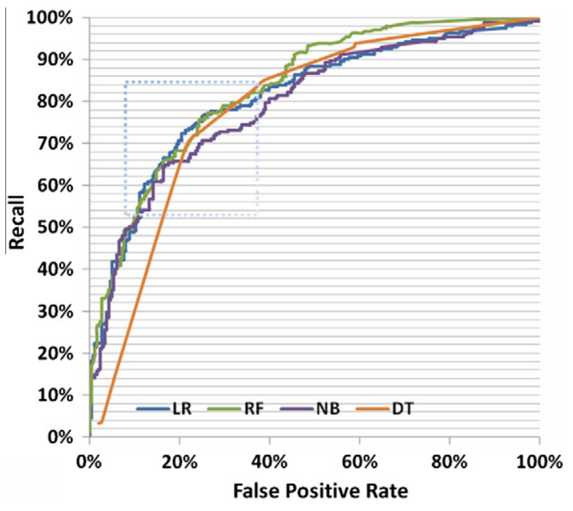


Fig. 12. Plot of recall vs. FP rate.

5.2.2. Next-release validation

We conduct a next-release validation to investigate whether a predictor trained from past releases can predict vulnerability-prone entities in future releases. As already mentioned in Section 4.1, 52 releases of Mozilla Firefox have had vulnerability Fixes starting from Release R1.0 to Release R3.0.6 at the time of data collection (March 1, 2009). Data from the first 32 releases (R1.0 to R2.0.0.9) are used to train the predictors, and the data from the remaining 20 releases (R2.0.0.10 to R3.0.6) are used to test them. The data set is preprocessed to obtain a balanced training set. The accuracy, recall, FP rate, and F_1-measure for the next-release validation using the C4.5 Decision Tree technique are presented in Table 6. We notice a drop in the vulnerability-prone file detection rate (recall) and an increase in the false positive rate when predicting future vulnerabilities based on learning from the past. This is because the predictor is trained on a balanced data set but tested on a very unbalanced dataset containing about 99 times more non-vulnerable files than vulnerable files. We have tested on an unbalanced data set to mimic the real world situation. Nevertheless, we can observe that predictors obtained from past releases can be used to predict vulnerability-prone entities in the follow-up releases without any significant variance from the results presented in Table 6.

5.2.3. Comparison of results with other works

This section compares the results of our vulnerability prediction in Mozilla Firefox with other studies that used the same case study. Shin and Williams [32,33] predicted vulnerabilities in Mozilla Firefox using (only) code complexity metrics as independent variables and Logistic Regression (LR) as the prediction technique reports the accuracy, recall, and FP rate of their studies on vulnerability prediction. The average of FP-rate and recall of those studies using LR are compared with that of our predictions using LR in Fig. 13. In the figure, we also compare the FP-rate and recall of our next-release validation using C4.5 Decision Tree (DT). We compare their results with ours next-release validation test because Shin and Williams also have used the information from the past releases to predict vulnerabilities in future releases. Note that, they have used vulnerability history up to release 2.0.0.4 (the last release available during their study conducted in February 29, 2008), while we have used vulnerability history up to release 3.0.6 (the last release available during our study conducted in March 1, 2009). Some variations in the predictions are inevitable because the two studies (ours and Shin and Williams) use different datasets. We explain in the following paragraphs that the variations in the predictions are not mainly because of the use of different datasets but mainly because of the different sampling techniques used to train the predictors.

As it can be observed from Table 7 and Fig. 13, Shin and William achieve very low FP rates (which are desirable) but suffer from very low recalls (which are extremely undesirable). From the discussion about the need for a balanced training set presented in Section 5.1, it is clear that they used a highly imbalanced data set to train the logistic-regression predictors which resulted in biased predictors. In our file-level data set, the number of non-vulnerable files is about 94% more than that of vulnerable files. Shin and Wiliams analyzed vulnerabilities at the function level where the imbalance is much higher. Although the FP rate is almost zero, identifying only 13% (on average) of the vulnerable locations reduced the usefulness of their predictors as they would miss 87%

Table 6
Prediction performance of DT in next-release validation.

Accuracy	Recall	FP rate	F_1-measure
69.61%	69.52%	30.30%	66.00%

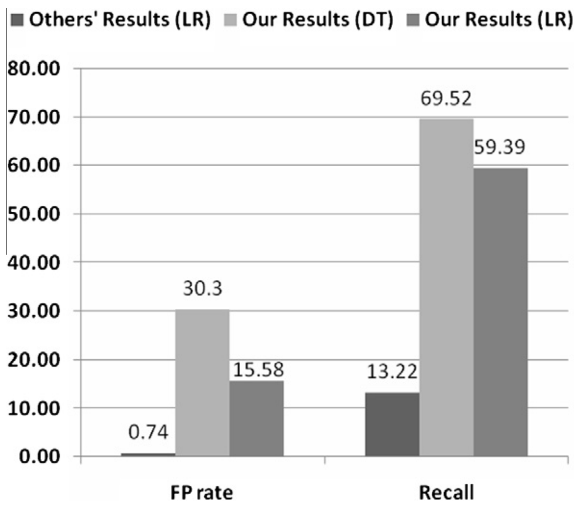


Fig. 13. Comparison of our FP rates and Recalls with other studies [32,33].

Table 7
Vulnerability prediction result of [32] and [33].

Release	Accuracy	FP rate	Recall
R1.0.2	90.98	0.90	11.9
R1.0.7	91.36	0.89	13.82
R1.5	91.13	0.98	10.07
R1.5.0.8	96.34	1.63	20.45
R2.0	96.81	0.06	4.92
R2.0.0.4	99.52	0.00	18.18

(on average) of the vulnerabilities. In contrast, we train the predictors on balanced data and achieve recalls of about 59% using the same technique (LR) at the expense of about 15% FP rate. Recollect that, using decision-tree based technique, our recall is more than 70% with false positive rate of about 25% (Table 5). Even in next-release validation using C4.5 Decision Tree, we achieve 69% recall at an expense of 30% FP rate (Table 6). We believe such a trade-off makes a vulnerability predictor more useful as the goal should be to correctly predict most of the vulnerable locations while keeping the FP rate at a reasonable level. Thus, their higher accuracy (see Table 7) does not necessarily indicate better results. That is why we do not compare the "accuracy" side by side with our results in Fig. 13. It would have been possible to be just as accurate by blindly predicting all the functions as non-vulnerable, because less than 1% of functions are vulnerable ones in the test set used in [32] and [33]. Under these circumstances, it is more important to identify those vulnerable entities to take effective proactive actions against potential vulnerabilities. Our predictors with higher recalls are more suitable for that task. Had we known the "precision" of their predictions, we could compare their results with ours using F_1-measure and F_2-measure that combine precision and recall into a single performance measure.

6. Related work

The related research is presented in three parts. First, we describe the research on fault prediction using complexity, coupling, and cohesion metrics [6–12]. Second, we compare and contrast recent work that predicts vulnerabilities from complexity and coupling metrics [30,31–33]. Finally, we describe some studies that use other phenomena (e.g., import patterns or past vulnerabilities) to identify the vulnerable components in a software system [28,29].

6.1. Fault prediction using complexity, coupling, and cohesion metrics

Several prior studies [6–12] have used the C–K metric suite [20] (indicated by the annotation "*" in Table 1) to identify fault-prone entities (e.g., fault-prone classes) in software developed in Object-Oriented (OO) architecture. These studies build statistical models with those metrics as independent variables and faults as the dependent variables. In general, it has been shown that these metrics, measuring structural complexity, coupling, and cohesion, can be used to predict the fault-prone modules or the number of faults. Janes et al. [8] identify that coupling metrics such as Response Set for a Class (RFC) and Coupling Between Object classes (CBO) are Good fault predictors. Succi et al. [9] report that metrics such as Number of Children (NOC) and Depth of Inheritance Tree (DIT) measuring inheritance complexity and coupling can also be used as indicators of the fault-prone classes. Basili et al. [11] discover that five of the metrics of the C–K metrics suite are useful indicators of fault-prone classes, and in fact, they are better predictors than the best set of "traditional" code metrics (the metrics without any "*" annotation in Table 1).

We use the C–K metric suite in our work as well. However, unlike [6–12], we use these metrics to predict vulnerabilities (security-related faults), not general faults. Moreover, many of those studies [6,7,9,11] use pre-release faults (i.e., those faults identified during the testing phase), whereas we use post-release vulnerabilities (i.e., the security-related faults found during the operational phase). Although Janes et al. [8] use post-release faults, they use the number of revisions (i.e., how many times a class has changed so far) as a proxy for the number of faults. Using the sheer number of revisions as a proxy for the number of faults in a software entity can be misleading, because changes can be made for perfective, adaptive, or corrective reasons. Perfective changes are made to improve the product effectiveness (e.g., to add functionality, to decrease the response time). Adaptive changes are made in response to the changes in the product's operational environment. Corrective changes are made to remove faults, leaving the specifications unchanged. Therefore, only corrective changes can be used as a proxy for the number of faults, not the other types of changes. In this study, we have precisely indentified those corrective changes resulting from vulnerability fixes (see Section 4.3 for details).

6.2. Vulnerability prediction using complexity and coupling metrics

Failure prediction using complexity, coupling, and cohesion metrics has been the subject of much research in software engineering. However, vulnerability prediction using complexity and coupling metrics is a fairly new area, and the applicability of cohesion metrics in vulnerability prediction has never been studied before.

From our preliminary investigation [37], we have observed that CCC metrics are moderately correlated with vulnerabilities in Mozilla Firefox. Such findings coined the idea of employing CCC metrics to predict vulnerabilities. However, in [37], we have performed a univariate analysis, finding correlation between vulnerabilities and each metric separately to identify the individual importance of each metric. Monotonic models based on correlations on the raw or monotonically transformed data cannot be used effectively to identify defect-prone modules under many situations [9,11,17]. This is because complexity, coupling, and cohesion may interact in the way that they may collectively affect vulnerability-proneness. For example, larger files may be more vulnerability-prone when they are more complex and less cohesive but may not be as vulnerability-prone when they are highly cohesive. The statistical and data mining techniques used in this study accounts for such multivariate interactions.

6.2.1. Complexity and vulnerability

Recently, there have been a few attempts at identifying vulnerability-prone functions using some traditional code-level complexity metrics by Shin and William [31–33]. Their results show weak correlation between code complexity metrics and vulnerabilities. Therefore, our study incorporates some coupling and cohesion metrics which are not considered in [31–33]. Additionally, we include some metrics specifically designed to measure CCC in OO architecture. They use Logistic Regression to build vulnerability predictors whereas we apply three additional statistical and data mining techniques to build the predictors. We have replicated their case studies on Mozilla Firefox, and our improvements over their prediction performances are already discussed in detail in Section 5.2.3 where we present our results.

6.2.2. Coupling and vulnerability

The experimental investigations by Traroe et al. [30] somewhat substantiates the common intuition that service coupling affects Denial of Service (DoS) attackability (the *R*-squared values of their Logistic Regression analyses are 70% which means that their model explains about 70% of the variation in attackability). Attackability is defined as the likelihood that a software system or service can be compromised under an attack. By coupling, they mean service coupling which is measured as "the number of shared components between a given pair of services". Their concept of service coupling can only be applied in a service-oriented architecture paradigm, whereas we measure coupling within traditional and object oriented paradigms. Even though service coupling is found to be a good explanatory factor for DoS attackability, there might be other independent variables such as complexity and cohesion, and we include those factors in our analysis. Moreover, their attackability data is obtained from simulated lab experiments whereas our vulnerability data is gathered from real-life attacks. Furthermore, they considered only DOS attacks whereas our analyzed vulnerability reports include a broader spectrum of attacks.

6.2.3. Vulnerability prediction using other metrics and techniques

Neuhaus and Zimmerman [29] have found that vulnerabilities in Mozilla Firefox components can be inferred from import and function-call patterns. A component is defined as a collection of similarly named source and header files providing a specific service. They identify common patterns of imports (#include in C/C++) and function calls in the vulnerable components using pattern mining techniques, whereas we identify patterns of CCC metric-values using statistical and machine learning techniques. There is a potential to combine these two approaches to build more accurate vulnerability predictions. They predict about half of the vulnerable components in Mozilla Firefox and about two-third of these predictions are correct. We have achieved better results in predicting vulnerability-prone files in Mozilla Firefox. In should be mentioned that, we analyze vulnerabilities in Mozilla Firefox at a different level of granularity (see Section 4.1 for explanations), not at the component level. Moreover, the way we map vulnerabilities to entities is slightly different from that of Neuhaus and Zimmerman. Their vulnerability data is as of January 4, 2007. We work on an up-to-date vulnerability data as of March 1, 2009 (the time of our data collection). However, their vulnerability data has been very helpful in cross checking the accuracy of our vulnerability mapping technique.

Alhazmi et al. [27] investigate whether the number of vulnerabilities latent in a software system can be predicted from the vulnerabilities which have already been discovered. They study the Windows and Red Hat Linux operating systems and model the future trends of vulnerability discovery which they call the vulnerability discovery rate. As Shin and William note, "Their approach can be useful for estimating the effort required to identify and correct undiscovered security vulnerabilities, but cannot identify the location of the vulnerabilities in the source code" [31]. By location, Shin and William mean the entity (e.g., file or function) where the vulnerability is present.

7. Conclusions, limitations, and future work

7.1. Conclusions

In this work, we investigate the efficacy of applying complexity, coupling, and cohesion metrics to automatically predict vulnerability-prone entities in software systems. We use four alternative statistical and machine learning techniques to build vulnerability predictors that learn from the CCC metrics and vulnerability history. The techniques are C4.5 Decision Trees, Random Forests, Logistic Regression, and Naïve-Bayes. We conduct an extensive empirical study on Mozilla Firefox to demonstrate the efficacy of employing CCC metrics in vulnerability prediction. The empirical study is extensive in the sense that we have studied vulnerability history of more than four years and fifty-two releases of Mozilla Firefox. Overall, we are able to correctly predict almost 75% of the vulnerability-prone files, with a false positive rate of below 30% and an overall prediction accuracy of about 74%. Notably, this reasonable accuracy in predicting vulnerability-prone files is obtained with default parameter setting of the learning methods. By experimenting with the parameters, it would be possible to achieve higher accuracy. We have made a number of additional observations from our study as described in the following paragraphs.

First, for most of the aforementioned statistical and machine learning techniques (except Naïve-Bayes), the predictions are more than 70% accurate. This result is obtained using the default parameter settings of the WEKA tool. This indicates that CCC metrics can be used in vulnerability prediction, irrespective of what prediction technique is used.

Second, the standard deviations in the performance measure obtained from 10-fold-cross validation are low. This implies that the prediction would perform consistently when applied to a different dataset. However, it is necessary to train the predictors on a balanced data set. Otherwise, it produces biased results towards the more dominant category resulting in poor recall and misleading overall accuracy.

Third, decision-tree-based techniques, such as C4.5 Decision Tree and Random Forests, significantly out-perform Logistic Regression and Naïve-Bayes in terms of correctly identifying vulnerability-prone files and in overall prediction accuracy. These techniques also achieve a balanced false positive rate and recall, striking a balance between efficiency and effectiveness in vulnerability prediction.

Fourth, a basic C4.5 Decision Tree technique performs as well as the advanced Random Forests technique. Although there are differences in the prediction performances achieved by these two techniques, the differences are not statistically significant. Moreover, the C4.5 Decision Tree performs better than Random Forests when more emphasis is given on correctly predicting the vulnerable files at the expense of a slightly elevated FP rate.

Fifth, predictors built from one release can be reliably used to predict vulnerability-prone entities in future releases.

Finally, we observe an improvement over similar studies [32,33] on vulnerability prediction in Mozilla Firefox. The improvement is mainly in recall or vulnerability-prone file detection rate. Moreover, our results are as good as the results of another study ([29] initially discussed in Section 6.2.3 on related work) using different input features and approaches. Therefore, CCC metrics should be included in the input feature set in vulnerability prediction attempts.

The aforementioned observations substantiate that CCC metrics can be useful and practical addition to the framework of automatic vulnerability prediction. Such automatic predictions will allow software practitioners take preventive actions against potential vulnerabilities early in the software lifecycle. The proactive actions can be software architecture and design reevaluation to control complexity, coupling, and cohesion for avoiding problems at the first place. Then, the prediction of vulnerability-prone entities can guide vulnerability detection efforts such as validation and verification, code refactoring, and/or security testing where they are needed most.

7.2. Limitations

We recognize that there are certain limitations to the results and conclusions we have presented in this paper, and we discuss several of them in the following paragraphs.

First, we are aware of the fact there are many other factors that can lead to vulnerabilities in software systems. Therefore, by no means, we imply that CCC metrics should be the sole consideration when trying to predict potential vulnerabilities early in the software lifecycle. Instead, our results suggest that complexity, coupling, and cohesion can be some of the major factors to be kept in mind during security assessment of software artifacts, but there is certainly no requirement to limit oneself to these factors only.

Second, our research relies on vulnerabilities which have already been discovered and reported. Vulnerabilities that have not been discovered or publicly announced yet are not used for our study even though that information might contribute to a more precise analysis. In particular, it is impossible to ever know the true error rates, as certain false positives may be true positives if the files identified contain vulnerabilities that have not been noticed yet.

Third, there is always an element of randomness and variance in the results produced by statistical and machine learning techniques. In order to confidently lessen the effects of algorithmic bias, we have performed 10-fold cross validation, a way of performing repeated training and testing. The main purpose of this study is to investigate the applicability of using CCC metrics to predict vulnerable entities. In doing that, we tried several techniques so that the results are not overly influenced by any specific technique. We were not attempting to identify the most effective technique or most effective set of parameters.

Finally, we acknowledge that some conclusions drawn from our experiments may not apply to all software of different domains. Nevertheless, we have substantiated our findings over a wealth of vulnerability data by analyzing fifty-two releases developed over a period of four years. Researchers gain confidence in a theory as similar results are observed in different settings. In this regard, our findings provide supportive evidence about the how complexity, coupling and cohesion metrics relate to vulnerabilities in software.

7.3. Future work

Our future work will concentrate on the following issues.

7.3.1. More metrics

Currently, we only use CCC metrics in structured and OO programming paradigms. It would be interesting to investigate the relationship between vulnerabilities with other CCC metrics such as service coupling and complexity in service-oriented architecture [30,70]. There is also the need to investigate the effects of software-development-process complexity and coupling, i.e., metrics that focus on a code-change process instead of on the code properties [63,71]. For example, Hassan et al. conjecture that entities that are part of large, complex modifications or entities that are being modified frequently and/or recently are likely to have faults [63]. Similar studies can be conducted in the context of security patches and vulnerabilities.

7.3.2. Granularity of analysis

Zimmermann et al.'s study [64] reveals that the degree of correlation between faults and complexity measures is different depending on the granularity of analysis. In this study, we analyze the effect of CCC metrics on vulnerability-proneness at the file-level. Analyzing at various granularities (e.g., at module or component level) might reveal some more interesting information.

7.3.3. More automation

We want to integrate third-party statistical and machine learning tools with our tool so that we can automatically obtain predictors from software archives without much human intervention. The next step would be to integrate these predictors into development environments supporting the decisions of software engineers and managers.

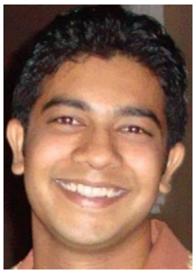
Acknowledgments

This research is partially funded by the Natural Sciences and Engineering Research Council of Canada (NSERC). The authors wish to thank Stephan Neuhaus of Saarland University, Saarbrücken, Germany for sharing his dataset (as of January 4th, 2007) on vulnerabilities in Mozilla Firefox and for his suggestions on how to obtain an updated dataset. We also thank Yonghee Shin of North Carolina State University, Raleigh, NC, USA for taking the time to answer our queries about her technique of extracting vulnerability data from Mozilla archives. We would like to acknowledge Scientific Toolworks, Inc. at St. George, UT, USA for their generous four-month-extension of the trial license of Understand 2.0, the tool used to automatically compute the metrics.

References

[1] M. Bishop, Computer Security: Art and Science, Addison-Wesley, Boston, MA, 2003.
[2] J. Grossman, Website Vulnerabilities Revealed: What everyone knew, but afraid to believe, White Hat Security Inc., <http://www.whitehatsec.com/home/assets/presentations/PPTstats032608.pdf> (accessed July 2009).
[3] Computer Emergency Response Team Coordination Center (CERT/CC), <http://www.cert.org/stats/cert_stats.html> (accessed July 2009).
[4] Jaquith, Security Metrics: Replacing Fear, Uncertainty, and Doubt, Pearson Education Inc. Upper Saddle River, NJ, 2007.
[5] E. Damiani, C.A. Ardagna, N.E. Ioini, Open Source Systems Security Certification, Springer, 2009.
[6] G. Koru, J. Tian, An empirical comparison and characterization of high defect and high complexity modules, Journal of Systems and Software 67 (2003) 153–163.
[7] M. Cartwright, M. Shepperd, An empirical investigation of an object-oriented software system, IEEE Transactions on Software Engineering 26 (8) (2000) 786–796.
[8] Janes, M. Scotto, W. Pedrycz, B. Russo, M. Stefanovic, G. Succi, Identification of defect-prone classes in telecommunication software systems using design metrics, Journal of Systems and Software 176 (2006) 3711–3734.
[9] G. Succi, W. Pedrycz, M. Stefanovic, J. Miller, Practical assessment of the models for identification of defect-prone classes in object-oriented commercial systems using design metrics, Journal of Systems and Software 65 (2003) 1–12.
[10] K. El Emam, W. Melo, J.C. Machado, The prediction of faulty classes using object-oriented design metrics, Journal of Systems and Software 56 (2001) 63–75.
[11] V. Basili, L. Briand, W. Melo, A validation of object-oriented design metrics as quality indicators, IEEE Transactions on Software Engineering 22 (1996) 751–761.
[12] K.O. Elish, M.O. Elish, Predicting defect-prone software modules using support vector machines, Journal of Systems and Software 81 (2008) 649–660.
[13] N. Nagappan, T. Ball, A. Zeller, Mining metrics to predict component failures, in: Proceedings of the 28th International Conference on Software Engineering, Shanghai, China, May 2006, pp. 452–461.

[14] T. Menzies, J. Greenwald, A. Frank, Data mining static code attributes to learn defect predictors, IEEE Transactions on Software Engineering 33 (9) (2007) 2–13.
[15] H. Zhang, X. Zhang, M. Gu, Predicting defective software components from code complexity measures, in: Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing, Melbourne, Australia, December 2007, pp. 93–96.
[16] W.M. Evanco, W.W. Agresti, A composite complexity approach for software defect modelling, Software Quality Journal 3 (1994) 27–44.
[17] N. Fenton, P. Krause, M. Neil, A probabilistic model for software defect prediction, IEEE Transactions on Software Engineering 2143 (2001) 444–453.
[18] IEEE, IEEE Std. 982.1-1988 IEEE Standard Dictionary of Measures to Produce Reliable Software, The Institute of Electrical and Electronics Engineers, June 1988.
[20] S.R. Chidamber, C.F. Kemerer, A metrics suite for object oriented design, IEEE Transactions on Software Engineering 20 (6) (1994) 476–493.
[21] N.E. Fenton, S.L. Pfleeger, Software Metrics: A Rigorous and Practical Approach, PWS Publishing Co., Boston, MA, USA, 1997.
[22] T.J. McCabe, A complexity measure, IEEE Transactions on Software Engineering 2 (4) (1976) 308–320.
[24] G.J. Myers, Composite/Structured Design, Van Nostrand Reinhold Company, New York, 1978.
[25] W.A. Harrison, K.I. Magel, A complexity measure based on nesting level, ACM Sigplan Notices 16 (3) (1981) 63–74.
[26] S. Henry, D. Kafura, Software structure metrics based on information flow, IEEE Transactions on Software Engineering (1981) 510–518.
[27] K. Ayari, P. Meshkinfam, G. Antoniol, M. Di Penta, Threats on Building Models from CVS and Bugzilla Repositories: the Mozilla Case Study, in: Proceedings of the 2007 Conference of the Center for Advanced Studies on Collaborative Research, Richmond Hill, Ontario, Canada, October 2007, pp. 215–228.
[28] O.H. Alhazmi, Y.K. Malaiya, I. Ray, Measuring, analyzing and predicting security vulnerabilities in software systems, Computers and Security 26 (3) (2007) 219–228.
[29] S. Neuhaus, T. Zimmermann, A. Zeller, Predicting vulnerable software components, in: Proceedings of the 14th ACM Conference on Computer and Communications Security, Alexandria, Virginia, USA, October 2007, pp. 529–540.
[30] M.Y. Liu, I. Traore, empirical relations between attackability and coupling: a case study on DoS, in: Proceedings of the 2006 ACM SIGPLAN Workshop on Programming Languages and Analysis for Security, Ottawa, Canada, June 2006, pp. 57–64.
[31] Y. Shin, L. Williams, Is complexity really the enemy of software security? in: Proceedings of the Fourth ACM Workshop on Quality of Protection, Alexandria, Virginia, USA, October 2008, pp. 47–50.
[32] Y. Shin, L. Williams, An empirical model to predict security vulnerabilities using code complexity metrics, in: Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement, Kaiserslautern, Germany, October 2008, pp. 315–317.
[33] Y. Shin, Exploring complexity metrics as indicators of software vulnerability, in: Proceedings of the Third International Doctoral Symposium on Empirical Software Engineering, Kaiserslautem, Germany, October 2008, available from the author's website <http://www4.ncsu.edu/~yshin2/papers/esem2008ds_shin.pdf> (accessed July 2009).
[34] M. Gegick, L. Williams, M. Vouk, Predictive models for identifying software components prone to failure during security attacks, Technical Report, Department of Computer Science, North Carolina State University, USA, October 28, 2008.
[35] I. Chowdhury, B. Chan, M. Zulkernine, Security metrics for source code structures, in: Proceedings of the Fourth International Workshop on Software Engineering for Secure Systems, Leipzig, Germany, May 2008, pp. 57–64.
[36] H. Malik, I. Chowdhury, H.M. Tsou, Z. Ziang, A.E. Hassan, Understanding the rationale for updating a function's comment, in: Proceeding of the 24th International Conference on Software Maintenance, Beijing, China, September 2008, pp. 167–176.
[37] I. Chowdhury, M. Zulkernine, Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities? in: Proceedings of the 25th ACM Symposium on Applied Computing, Sierre, Switzerland, March 22–26, 2010.
[38] J. Cohen, Statistical Power Analysis for the Behavioral Sciences, second ed., Academic Press, New York, 1988.
[39] Mozilla Firefox, <http://www.mozilla.com/en-US/firefox> (accessed July 2009).
[40] Bugzilla, <http://www.bugzilla.org> (accessed July 2009).
[41] Mozilla Vulnerabilities, <http://www.mozilla.org/projects/security/knownvulnerabilities> (accessed July 2009).
[42] Index of Mozilla FTP Server, <ftp://ftp.mozilla.org/pub/mozilla.org/> (accessed July 2009).
[43] Mozilla Developer Guide, <https://developer.mozilla.org/en/Download_Mozilla_Source_Code#Releases> (accessed July 2009).
[44] SciTools Inc., <http://www.scitools.com> (accessed July 2009).
[45] SciTools Inc. Blog, <http://scitools.com/blog/2008/10/tip-understand-the-countpath-metric.html> (accessed July 2009).
[46] Browser Statistics, <http://www.3schools.com/browsers/browsers_stats.asp> (accessed July 2009).
[47] The Linux Information Project, <http://www.linfo.org/bug.html> (accessed July 2009).
[48] Security Advisory for Firefox 2.0, <http://www.mozilla.org/security/known-vulnerabilities/firefox20.html> (accessed July 2009).
[49] Mozilla Foundation Security Advisory 2008-27, <http://www.mozilla.org/security/announce/2008/mfsa2008-27.html> (accessed July 2009).
[50] Bug 423541 – (CVE-2008-2805) "Arbitrary file upload via originalTarget and DOM Range", https://bugzilla.mozilla.org/show_bug.cgi?id=423541 (accessed July 2009).
[51] Diff Page for Bug 423541 in Mozilla Firefox, <https://bugzilla.mozilla.org/attachment.cgi?id=319352&action=diff> (accessed July 2009).
[52] WEKA Toolkit, <http://www.cs.waikato.ac.nz/ml/weka> (accessed July 2009).
[53] Common Vulnerabilities and Exposures, <http://cve.mitre.org/> (accessed July 2009).
[54] BeautifulSoup, <http://www.crummy.com/software/BeautifulSoup/documentation.html> (accessed July 2009).
[55] StatPy: Statistical Computing with Python, <http://www.astro.cornell.edu/staff/loredo/statpy/> (accessed July 2009).
[56] Documentation of the Implemented Tool, <http://www.research.cs.queensu.ca/~istehad/research/html/index.html> (accessed July 2009).
[57] E. Arisholm, L.C. Briand, M. Fuglerud, Data mining techniques for building fault-proneness models in telecom Java software, in: Proceedings of the 18th IEEE International Symposium on Software Reliability Engineering, Trollhättan, Sweden, November 2007, pp. 215–224.
[58] T.J. Ostrand, E.J. Weyuker, How to measure success of fault prediction models, in: Proceedings of the Fourth International Workshop on Software Quality Assurance, Dubrovnik, Croatia, September 2007, pp. 25–30.
[59] Y. Ma, L. Guo, B. Cukic, A Statistical Framework for the Prediction of Fault-Proneness, Advances in Machine Learning Application in Software Engineering, Idea Group Inc., 2006. pp. 237–265.
[60] I.H. Witten, E. Frank, Data Mining: Practical Machine Learning Tools and Techniques (2nd ed.), Morgan Kaufmann, San Francisco, 2005.
[61] L. Kuang, M. Zulkernine, An anomaly intrusion detection method using the CSI-KNN algorithm, in: Proceedings of the 23rd Annual ACM Symposium on Applied Computing, Fortaleza, Brazil, March 2008, pp. 921–926.
[62] J. Zhang, M. Zulkernine, A. Haque, Random forest-based network intrusion detection systems, IEEE Transactions on Systems Man and Cybernetics 38 (5) (2008) 648–658.
[63] A.E. Hassan, Predicting faults using the complexity of code changes, in: Proceedings of the 31st International Conference on Software Engineering, Vancouver, Canada, May 2009, pp. 45–56.
[64] T. Zimmermann, R. Premraj, A. Zeller, Predicting defects for eclipse, in: Proceedings of the Third International Workshop on Predictor Models in Software Engineering, Washington, DC, USA, May 2007, pp. 9–15.
[65] A.E. Hassan, Mining Software Repositories to Assist Developers and Support Managers, PhD. Thesis, University of Waterloo, 2004.
[66] M. Philips, The inevitability of punishing the innocent, Journal of Philosophical Studies, Springer, Netherlands 48 (3) (1985) 389–391.
[67] J. Rijsbergen, Information Retrieval, Second ed., Butterworth-Heinemann, 1979.
[69] J.K. Kearney, R.L. Sedlmeyer, W.B. Thompson, M.A. Gray, M.A. Adler, Software complexity measurement, ACM Communications 29 (11) (1986) 1044–1050.
[70] M. Comuzzi, C. Kotsokalis, G. Spanoudakis, R. Yahyapour, Establishing and monitoring SLAs in complex service based systems, in: Proceedings of the Seventh IEEE International Conference on Web Services, CA, USA, July 2009, pp.783–790.
[71] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice, second ed., Addison-Wesley, Boston, 2003. p. 21.
[73] L.C. Briand, S.J. Carrière, R. Kazman, J. Wüst, COMPARE: a comprehensive framework for architecture evaluation, Lecture Notes in Computer Science, Springer-Berlin 1543 (1998) 594.
[74] N. Landwehr, M. Hall, E. Frank, Logistic model trees, J. Mach. Learn. 59 (1–2) (2005) 161–205.
[75] B. Henderson-Sellers, L. Constantine, I. Graham, Coupling and cohesion (towards a valid metrics suite for object-oriented analysis and design), Object-Oriented Systems 3 (3) (1996) 143–158.



Istehad Chowdhury is currently a research intern in Cloakware Inc., Canada. He received his M.Sc. degree from the Department of Electrical and Computer Engineering of Queen's University, Canada in 2009, where he was a research assistant and a member of Queen's Reliable Software Technology (QRST) research group. He received his B.Sc. degree in Computer Science from Independent University, Bangladesh in 2005. Before joining Queen's, he was a lecturer in the Department of Computer Science of Stamford University, Bangladesh. He has been a member of ACM since 2001. His main research interest lies in the area of software engineering with special interest in software reliability and security, software metrics, empirical software engineering, and mining software repositories. More information about his research and publications can be found at http://www.cs.queensu.ca/~istehad.

Mohammad Zulkernine is a faculty member of the School of Computing of Queen's University, Canada, where he leads the Queen's Reliable Software Technology (QRST) research group. He received his B.Sc. in Computer Science and Engineering from Bangladesh University of Engineering and Technology in 1993. Dr. Zulkernine received an M. Eng. in Computer Science and Systems Engineering from Muroran Institute of Technology, Japan in 1998. He received his Ph.D. from the Department of Electrical and Computer Engineering of the University of Waterloo, Canada in 2003, where he belonged to the university's Bell Canada Software Reliability Laboratory. Dr. Zulkernine's research focuses on software engineering (software reliability and security), automatic software monitoring and intrusion detection, methods and tools for reliable and secure software. His research work are funded by a number of provincial and federal research organizations of Canada, while he is having an industry research partnership with Bell Canada. He is a senior member of the IEEE and a member of the ACM. Dr. Zulkernine is also cross-appointed in the Department of Electrical and Computer Engineering of Queen's University, and a licensed professional engineer of the province of Ontario, Canada.