Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey

SEYED MOHAMMAD GHAFFARIAN and HAMID REZA SHAHRIARI,
Amirkabir University of Technology

Software security vulnerabilities are one of the critical issues in the realm of computer security. Due to their potential high severity impacts, many different approaches have been proposed in the past decades to mitigate the damages of software vulnerabilities. Machine-learning and data-mining techniques are also among the many approaches to address this issue. In this article, we provide an extensive review of the many different works in the field of software vulnerability analysis and discovery that utilize machine-learning and data-mining techniques. We review different categories of works in this domain, discuss both advantages and shortcomings, and point out challenges and some uncharted territories in the field.

CCS Concepts: • **General and reference** → **Surveys and overviews**; • **Security and privacy** → **Vulnerability management**; **Software and application security**; • **Software and its engineering** → *Software testing and debugging*;

Additional Key Words and Phrases: Software vulnerability analysis, software vulnerability discovery, software security, machine-learning, data-mining, review, survey

ACM Reference format:
Seyed Mohammad Ghaffarian and Hamid Reza Shahriari. 2017. Software Vulnerability Analysis and Discovery Using Machine-Learning and Data-Mining Techniques: A Survey. *ACM Comput. Surv.* 50, 4, Article 56 (August 2017), 36 pages.
https://doi.org/10.1145/3092566

1 INTRODUCTION

Nowadays, computer software is ubiquitous, and the modern human life is greatly dependent on a great variety of software. There are different forms of computer software running on different platforms and ranging from simple apps on hand-held mobile devices to sophisticated distributed enterprise software systems. These software are produced with many different methodologies, based on a vast variety of technologies, each having their own advantages and limitations. An important concern in this vast critical industry, and in the field of computer security, is the problem of *software security vulnerabilities*. To cite industry experts on the matter:

> "*In the context of software security, vulnerabilities are specific flaws or oversights in a piece of software that allow attackers to do something malicious: expose or alter sensitive information, disrupt or destroy a system, or take control of a computer system or program.*" Dowd et al. (2007)

Authors' addresses: S. M. Ghaffarian and H. R. Shahriari (corresponding author), Computer Engineering and Information Technology Department, Amirkabir University of Technology, 424 Hafez Avenue, Tehran, Islamic Republic of Iran; emails: {s.m.ghaffarian, shahriari}@aut.ac.ir.

https://doi.org/10.1145/3092566

The severity of the threat imposed by software vulnerabilities has varying degrees, depending on factors such as exploitation complexity and attack-surface (Nayak et al. 2014). Numerous examples and incidents exist in the past two decades in which software vulnerabilities have imposed significant damages to companies and individuals. To emphasize the significance of the issue, we mention a few examples in recent years. A prominent example is the situation of vulnerabilities in popular browser plugins that have threatened the security and privacy of millions of Internet users (e.g., Adobe Flash Player (US-CERT 2015; Adobe Security Bulletin 2015) and Oracle Java (US-CERT 2013)). Moreover, vulnerabilities in popular and fundamental open-source software have also threatened the security of thousands of companies and their customers around the globe (e.g., Heartbleed (Codenomicon 2014), ShellShock (Symantec Security Response 2014), and Apache Commons (Breen 2015)).

The aforementioned examples are only a few of the vast number of vulnerabilities that are reported each year. Due to the importance of this matter, many different mitigation approaches have been investigated by researchers of the academic community and the software industry. An extensive survey of different approaches to mitigate program security vulnerabilities, including testing, static analysis, and hybrid analysis, as well as secure programming, program transformation, and patching methods published between 1994 and 2010, is presented by Shahriar and Zulkernine (2012).

In addition to the well-known and well-researched approaches reviewed in Shahriar and Zulkernine (2012), a different class of approaches exists that are methods that utilize techniques from the fields of data science and artificial intelligence (AI) to address the problem of software vulnerability analysis and discovery. This interesting class of approaches is neglected in Shahriar and Zulkernine (2012), while it has received increasing attention by the research community in the following years (from 2011 onwards).

In this article, we present a categorized review about this class of approaches in the field of software vulnerability analysis and discovery that utilize data-mining and machine-learning techniques. First, we define the problem of software vulnerability analysis and discovery, as well as briefly introduce the conventional approaches in the filed. We also briefly introduce machine-learning and data-mining techniques and the motivation behind their utilization. Afterward, we will categorically review the many different works that utilize machine-learning and data-mining techniques for the problem of software vulnerability analysis and discovery. We present different categories for this class of works and discuss their advantages and limitations. At the end, we conclude the article with a discussion of the challenges in the field and point out some uncharted domains to inspire future work in this emerging research area.

2 BACKGROUND: SOFTWARE VULNERABILITY ANALYSIS AND DISCOVERY

2.1 Definition

We start by stating the definition of a *software security vulnerability*. In his PhD thesis on the matter of *software vulnerability analysis*, Ivan Krsul defines software vulnerability as:

> "*an instance of an error in the specification, development, or configuration of software such that its execution can violate the security policy.*" (Krsul 1998)

Almost a decade later, Ozment acknowledges Krsul's definition among others, yet suggests a slight modification:

> "*A software vulnerability is an instance of a mistake in the specification, development, or configuration of software such that its execution can violate the explicit or implicit security policy.*" Ozment (2007)

Ozment changes the word *error* to *mistake* and cites the IEEE Standard Glossary of Software Engineering Terminology (IEEE Standards 1990) to justify this. As mentioned earlier, industry experts have provided similar definitions:

> "*In the context of software security, vulnerabilities are specific flaws or oversights in a piece of software that allow attackers to do something malicious: expose or alter sensitive information, disrupt or destroy a system, or take control of a computer system or program.*" Dowd et al. (2007)

As we can see from the aforementioned definitions, different key terms are used to define software vulnerabilities. To clarify these terms and select the most suitable, we refer to the IEEE Standard Glossary of Software Engineering Terminology (IEEE Standards 1990). We lookup the definitions of four key terms: *"error," "fault," "failure,"* and *"mistake."* According to IEEE Standards (1990) the definition of *error* is: "the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition" (IEEE Standards 1990). A *fault* is: "an incorrect step, process, or data definition in a computer program" (IEEE Standards 1990). Faults are also known as *flaws* or *bugs*. A *failure* is: "the inability of a system or component to perform its required functions within specified performance requirements" (IEEE Standards 1990). Finally, a *mistake* is: "a human action that produces an incorrect result" (IEEE Standards 1990). A summary and clarification of the relation of these terms is to "distinguish between the human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error)" (IEEE Standards 1990).

From these definitions it's clear that the suitable key term to use for the definition of a software vulnerability is *"fault"* (also *flaw* or *bug*). More precisely:

> ***A software vulnerability is an instance of a flaw, caused by a mistake in the design, development, or configuration of software such that it can be exploited to violate some explicit or implicit security policy.***

The cause of a software vulnerability is a human mistake, which its manifestation is the flaw (fault or bug). The execution of faulty statements of a vulnerable software don't necessarily violate security policies; not until some specially crafted data (the *exploit code*) or some random data with certain conditions, reaches the faulty statements, which at this point, its execution can violate some security policies (the *exploitation* resulting the *security failure*).

The definition of a software vulnerability as a fault, and a mistake as its cause, has been previously acknowledged by others. Ozment states that "a vulnerability that results from a development mistake is a fault" (Ozment 2007), yet he distinguishes between vulnerabilities resulting from development mistakes, and vulnerabilities resulting from design or configuration mistakes; but doesn't provide an explanation for this difference. Dowd et al. also state:

> "*In general, software vulnerabilities can be thought of as a subset of the larger phenomenon of software bugs. Security vulnerabilities are bugs that pack an extra hidden surprise: A malicious user can leverage them to launch attacks against the software and supporting systems.*" Dowd et al. (2007)

2.2 Soundness, Completeness, and Undecidability

Program vulnerability analysis is the problem to decide whether a given program contains known security vulnerabilities (according to a security policy) or not. Based on the undecidability of Turing's Halting Problem and Rice's Theorem, it can be shown that many program analysis problems

are also undecidable in the general case (Landi 1992; Reps 2000). What undecidability means to the practitioner is that a *sound* and *complete* solution to the problem does not exist.

In mathematical logic, a proof system is *sound* if no invalid argument can be approved by the system. A proof system is *complete* if all valid arguments can be approved by the system. By corollary, a *sound and complete* proof system is one that can approve all valid arguments and disprove all invalid arguments (Xie et al. 2005).

In the context of software security, a vulnerability analysis system is *sound* if it never approves a vulnerable program (no vulnerabilities are missed). A vulnerability analysis system is *complete* if all secure programs can be approved (no false vulnerabilities). By corollary, a *sound and complete* vulnerability analysis system can approve all secure programs and disapprove all vulnerable programs (no missed vulnerabilities and no false vulnerabilities) (Xie et al. 2005). As mentioned earlier, such *sound and complete* system is known to be non-existent (Jhala and Majumdar 2009).

Besides vulnerability analysis, A more practically useful system is a program vulnerability discovery (or vulnerability reporting) system. In contrast to a vulnerability analysis system which approves or disapproves the security of a given program (i.e., binary output), a program vulnerability discovery system reports more detailed information (such as type, location, etc.) for each vulnerability discovered in a given program. This is the more useful and desirable system for the software industry, which aids developers and engineers to more easily detect and fix vulnerabilities. Again, a *sound and complete* software vulnerability discovery system (a system that reports no false vulnerabilities, and reports all actual vulnerabilities) is known to be non-existent.

2.3 Conventional Approaches

Despite the undecidable nature of the problem of software vulnerability analysis and discovery, a plethora of approaches have been investigated and proposed by practitioners both in the academic community and the software industry due to the critical importance of the matter. The proposed approaches are all inevitably approximate solutions; they all either lack soundness, or completeness, or both. Hence, all research efforts try to propose an improved approach compared to previous works, regarding a specific aspect of the process of software vulnerability analysis and discovery; for example, vulnerability coverage, discovery precision, runtime efficiency, and so on.

An extensive review of different approaches to mitigate program security vulnerabilities, including program vulnerability analysis and discovery methods published between 1994 and 2010, is presented by Shahriar and Zulkernine (2012). All program analysis approaches can be categorized in three main categories:

- **Static Analysis:** A given program is analyzed based on its source code, without the need to execute it. These approaches utilize a generalized abstraction to analyze the properties of a program, hence static analysis approaches at best can be *sound* (i.e., at best there are no missed vulnerabilities, but false vulnerabilities may be reported). The more accurate the generalization, the fewer reported false vulnerabilities. In practice, a trade-off has to be made between analysis precision and computational efficiency.
- **Dynamic Analysis:** A given program is analyzed by executing it with specific input data and monitoring its runtime behavior. In this approach, a set of input test-cases are used to analyze the properties of a program, and since there are often infinitely possible inputs and runtime states, hence a dynamic analysis system can not analyze the entire program's behavior. Therefore, dynamic analysis systems can at best be *complete* (i.e., approve all secure programs and not report false vulnerabilities), but they cannot be sound, since it's possible to miss some vulnerabilities that reside in unseen program states. There are practical shortcomings to dynamic analysis approaches, namely, the requirement for a working runtime

environment of the given program, and the possibly long time and high costs required for the processing of all input test-cases when analyzing large, complex software. Nevertheless, dynamic analysis approaches are greatly used in the software industry.

—**Hybrid Analysis:** A given program is analyzed with a mixture of static analysis and dynamic analysis techniques. A possible misconception based on previous discussions about static and dynamic analysis approaches, might be that hybrid analysis approaches can be *sound and complete* (hence, violating undecidability of the problem). Unfortunately this is not true, and while hybrid analysis approaches can benefit from the advantages of both static and dynamic analysis, they also suffer from the limitations of both approaches. A hybrid analysis approach can be either a static analysis system that leverages dynamic analysis to identify false vulnerabilities, or a dynamic analysis approach that leverages static analysis techniques to guide the test-case selection and analysis process.

It should be noted, though, not all static analysis systems are sound, and not all dynamic analysis systems are complete. Among the many different vulnerability discovery approaches, some are more established in the software industry; namely:

—**Software Penetration Testing:** a manual software security testing approach, carried out by a team of security experts (also referred to as *white-hat hackers*) (Arkin et al. 2005; Bishop 2007).

—**Fuzz-Testing:** also known as *random-testing*, where well-formed input data are randomly mutated and fed to the program under test at large, while monitoring for failures (Godefroid 2007; Godefroid et al. 2012).

—**Static Data-Flow Analysis:** also known as *"Tainted Data-flow Analysis,"* it is a static program analysis approach where untrusted data from input *sources* is marked as *tainted* and its flow to sensitive program statements known as *sinks* is tracked as a potential indicator of vulnerability (Evans and Larochelle 2002; Larus et al. 2004; Ayewah et al. 2008; Bessey et al. 2010).

3 EMPLOYING MACHINE-LEARNING AND DATA-MINING TECHNIQUES

In addition to the aforementioned approaches, there is a different class of works that utilize techniques from the fields of data science and artificial intelligence (AI) to address the problem of software vulnerability analysis and discovery. This interesting class of approaches is neglected in the review of Shahriar and Zulkernine (2012), while it has received increasing attention by the research community in the following years (from 2011 onwards).

Machine-learning techniques from the field of AI have proven to be effective in practice for many different application areas (Russell and Norvig 2009). This is also true for the domain of computer security and privacy, which many different applications have been addressed using these techniques (e.g., spam filtering (Guzella and Caminhas 2009; Caruana and Li 2012) and intrusion detection systems (Garcia-Teodoro et al. 2009; Zhou et al. 2010), to name a few).

As defined by Arthur Samuel in his seminal work, machine-learning is the field of study to develop computational techniques and algorithms that enable computer systems to gain new abilities without being explicitly programmed (Samuel 1959). Data mining is the computational process of extracting knowledge from large amounts of data, consisting of several steps: data extraction and gathering, data cleaning and integration, data selection and transformation, knowledge mining, and finally visualization and communication (Han et al. 2011). Machine-learning algorithms and techniques are often used in the process of data mining for pre-processing, pattern recognition, and generating prediction models.

Machine-learning techniques can be broadly categorized into three main approaches: *(1) Supervised Learning:* the learning system infers a desired function/model based on a set of labeled training examples, where each example consists of input data (typically a vector) and the desired corresponding output value (the label). *(2) Unsupervised Learning:* in situations where labeled training data is not available, the goal of the learning system is to identify patterns and structures in the given dataset. *(3) Reinforcement Learning:* the learning system is trained to achieve a certain goal, by receiving rewards and penalties through interacting with a dynamic environment.

3.1 Hopes and Fears

Although the use of machine-learning techniques for security applications dates back to several decades, the advancements and capabilities of machine-learning and data-mining techniques in recent years, and their success stories in addressing many difficult application problems have motivated researchers to more thoroughly investigate the effective utilization of these techniques for difficult problems in the domain of computer security and privacy. For example, Carl Landwehr shares his thoughts on the matter as follows:

> "*In their early days, computer security and artificial intelligence didn't seem to have much to say to each other … Security researchers aimed to fix the leaks in the plumbing of the computing infrastructure or design infrastructures they deemed leakproof … But the two fields have grown closer over the years, particularly where attacks have aimed to simulate legitimate behaviors … We might imagine systems that would have a degree of self-awareness about the data that they process. The notion of reflective systems (systems that can reference and modify their own behavior) has its origins in the AI community … Imagine a plumbing system that contained a system of smart pipes that could detect incipient leaks. A cyber-infrastructure that incorporated the analog of smart pipes would be of great interest.*" Landwehr (2008)

Other researchers have emphasized the influential role of AI-techniques to obtain solutions for complex problems in the realm of computer security and privacy. For example, Tyugu (2011) states: "*It has become obvious that many cyber-defence problems can be solved successfully only when methods of artificial intelligence are being used.*" A similar viewpoint is presented by Heinl (2014).

On the other hand, there are also concerns in the security community on the use of AI techniques. For example, despite the large amount of published research in the field of anomaly-based intrusion detection systems, some previous studies state these systems are seldom deployed in the intrusion detection industry (Sommer and Paxson 2010). Other studies have also challenged the anomaly-detection paradigm for network intrusion detection (Gates and Taylor 2006). The result of these studies highlights the fact that effective use of AI-techniques in the realm of computer security and privacy is not trivial and requires well-understanding about the characteristics of these techniques (Sommer and Paxson 2010). To achieve the best possible results, machine-learning and data-mining techniques should be tailored to suite the characteristics of security problems. On this matter, Morel states:

> "*Despite some work done in the past, AI does not play a central role in cybersecurity today and cybersecurity has not been an area of development of AI as intensely pursued as Robotics, and others … AI techniques are developed around applications. Cybersecurity has never been an area of concentration in AI … AI has made a lot of accomplishments and there is a lot to be learned relevant for cybersecurity and many new techniques suited for cybersecurity could be inspired from existing ones in AI.*" Morel (2011)

To that end, research on the design of machine-learning techniques tailored to computer security problems should be pursued.

3.2 Categorizing Previous Work

In the field of software vulnerability analysis and discovery, many research studies were published in previous years that investigate the use of machine-learning and data-mining techniques, which we have extensively reviewed in this article. We categorize the reviewed works in four main categories, which in summary are defined and discriminated as follows:

(1) **Vulnerability Prediction Models based on Software Metrics:** A large body of studies that utilize a (mostly supervised) machine-learning approach to build a prediction model based on well-known software metrics as the feature set, and then use the model to assess the vulnerability status of software artifacts based on measured software-engineering metrics.
(2) **Anomaly Detection Approaches:** This category of works, utilize an unsupervised learning approach to automatically extract a model of normality or mine rules from the software source code, and detect vulnerabilities as deviant behavior from the normal majority and rules.
(3) **Vulnerable Code Pattern Recognition:** This category of works, utilize a (mostly supervised) machine-learning approach to extract patterns of vulnerable code segments from many vulnerability code samples, and then use pattern-matching techniques to detect and locate vulnerabilities in software source code.
(4) **Miscellaneous Approaches:** A handful of notable recent works that utilized techniques from the fields of AI and data science for software vulnerability analysis and discovery, which do not fit in any of the aforementioned categories, nor they constitute a coherent category.

The rationale behind the proposed categorization is twofold: First, we discriminate works that analyze program syntax and semantics, from those that do not. The majority of works that do not analyze program syntax and semantics, use software engineering metrics for vulnerability prediction. On the other hand, among the large body of studies that are based on analyzing program syntax and semantics, we observe two major approaches: *Vulnerable Code Pattern Recognition* and *Anomaly Detection Approaches*. Figure 1 intuitively summarizes the categorization scheme.

While other criterion could also be used for the purpose of categorization (such as: supervised versus unsupervised learning paradigms, different learning and mining techniques, feature-representation schemes, etc.) they do not create semantically coherent categories of previous works. In our opinion, the proposed categorization results more meaningful families of studies, which allows for better comparison of approaches against one another, hence we believe it is a suitable choice for an organized survey of previous studies in the field.

4 VULNERABILITY PREDICTION BASED ON SOFTWARE METRICS

The first category of approaches that we review in this article is *"vulnerability prediction models,"* which utilize data-mining, machine-learning, and statistical-analysis techniques to predict vulnerable software artifacts (source code files, object oriented classes, binary components, etc.) based on common software engineering metrics. The main idea of these approaches is borrowed from the field of software quality and reliability assurance in the domain of software engineering, where limited resources for software testing and verification, demands a guiding model to enable more efficient software testing plans. To this end, *"fault prediction models"* (or *"defect prediction models"*) have been investigated and used in the industry (Khoshgoftaar et al. 1997). Fault

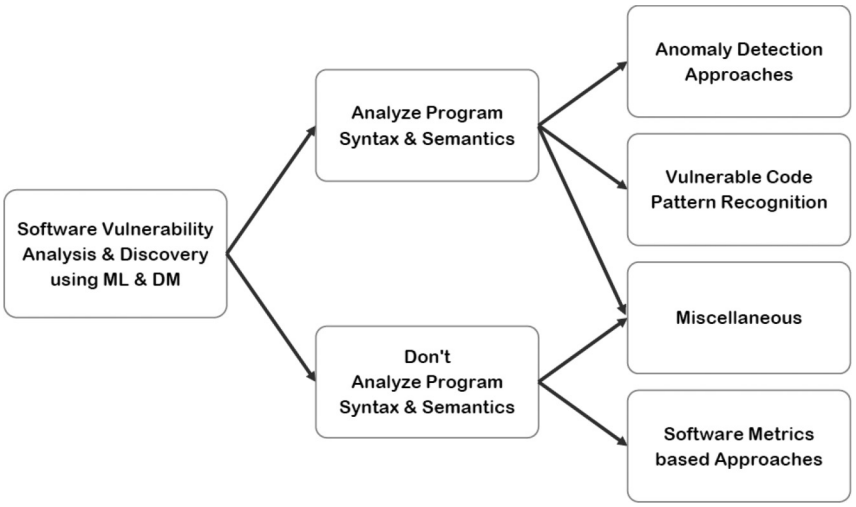

Fig. 1. Twofold categorization scheme of reviewed works.

prediction models are computational models that are trained based on historical data gathered from software projects, and provide a list of software artifacts that are more likely to contain faults to prioritize software testing efforts. The historical data is based on different software engineering metrics, such as source-code size, complexity, code-churn, and developer-activity metrics (Kaner and Bond 2004). According to the IEEE Standard Glossary of Software Engineering, the term *"metric"* is defined as *"a quantitative measure of the degree to which a system, component, or process possesses a given attribute"* (1990). Extensive research is performed and published on the subject of fault prediction models in the domain of software engineering that are reviewed in survey articles (Catal and Diri 2009; Malhotra 2015).

Vulnerability prediction models are proposed in the domain of software engineering based on a similar motivation as fault prediction models. Detection and mitigation of security vulnerabilities requires manual analysis by experts who are trained with a security mindset (Heelan 2011); yet, software quality and reliability assurance teams have limited resources that need to be guided for more efficient and effective security audit and testing efforts. Based on the fact that vulnerabilities are a specific type of fault, vulnerability prediction models have been proposed and investigated in the industry and academic community. Similar to fault prediction models, vulnerability prediction models are also built based on various software metrics and don't incorporate program analysis methods (i.e., analyzing program source code for certain properties). In the following, we review some of the latest works in this field.

4.1 Summary of Recent Works

Zimmermann et al. (2010) studied the possibility of predicting the existence of vulnerabilities in binary modules of a proprietary commercial product (Microsoft Windows Vista) based on classical metrics that have been used in prior research for defect prediction. As a first analysis, they computed the correlations between the metrics and the number of vulnerabilities per binary, using the Spearman's rank correlation. The results show that classical metrics have statistically significant correlation with the number of vulnerabilities; however, the effect is only small. Another analysis was to evaluate the prediction capabilities of these metrics. The authors used binary Logistic Regression for five groups of classical metrics (churn, complexity, coverage,

dependency, and organizational). Model evaluation was perform with tenfold cross-validation and computing precision and recall values. The authors report that most metrics predict vulnerabilities with an average to good precision (low false positives); however, the recall is very low (high false negatives or missed vulnerabilities) and that coverage metrics failed to produce any meaningful results. Results include precision below 67% and recall below 21%.

Meneely and Williams (2010), investigated the relationship between developer-activity metrics and software vulnerabilities. The developer-activity metrics under study include: number of distinct developers who changed a source file, number of commits made to a file, and number of geodesic paths containing a file in the contribution network. The authors performed the study on three open-source software projects. The gathered data set in each study included a label of whether or not a source code file was patched, and the developer-activity metrics from version control logs. Using statistical correlation analysis the authors report that statically significant correlation was found for each metric with the number of vulnerabilities; but the correlations vary and are not very strong. The authors used Bayesian network as the predictive model, with tenfold cross-validation to generate training and validation sets. According to the authors, the analysis shows that developer-activity can be used to predict vulnerable files; yet, precision and recall values are disappointing (precision between 12%–29%, and recalls between 32%–56%).

Doyle and Walden (2011) analyze relationships between software metrics and vulnerabilities in 14 of the most widely used open source web applications between 2006 to 2008, such as WordPress and Mediawiki. The authors used static analysis tools (e.g., Fortify Source Code Analyzer, PHP CodeSniffer, etc.) to measure a variety of metrics in source code repositories of these applications, including static analysis vulnerability density (SAVD), source-code size, cyclomatic complexity, nesting complexity, as well as another metric proposed by the authors named *Security Resources Indicator* (SRI). For predictability, Spearman's rank correlation was computed between SAVD and the other metrics. The results show that no single metric is suitable to distinguish high vulnerability web applications from low vulnerability ones; however, average cyclomatic complexity per function was an effective predictor for several applications, especially when used in combination with SRI scores to classify applications into high and low security focus applications. Since static analysis tools might produce high false positives, the authors manually reviewed the reports of one tool (Fortify SCA) for two of the selected web applications, which yielded a false-positive rate of 18%; concluding that the false-positive rate is acceptable and no threat to validity.

Shin and Williams (2013), investigated whether traditional fault prediction models based on complexity and code-churn metrics can be used for vulnerability prediction. For this purpose, the authors performed an empirical study on Mozilla Firefox, using 18 complexity metrics, 5 code churn metrics, and fault history metric. Several classification techniques were tested to predict faulty and vulnerable files; where the authors state that results from all techniques were similar. While faulty source code files were seven times more than vulnerable source files, the fault prediction model and the vulnerability prediction model performed similarly in vulnerability prediction; with a recall about 83% and precision of about 11%. Based on the results, the authors conclude that fault prediction models based upon traditional metrics can also be used for vulnerability prediction; however, future research is required to improve precision (reduce false positives) while retaining high recall.

In another work by the same authors, Shin and Williams (2011) investigated the use of execution complexity metrics as indicators of software vulnerabilities. According to the authors, the motivation behind this study is based on the intuition of security experts, which often hypothesize that *"software complexity is the enemy of software security."* To this end, the authors performed empirical case studies on two open source projects, to compare the effectiveness of execution complexity and static complexity metrics for vulnerability detection. In total, 23 complexity metrics were gathered

in this study. The authors performed both discriminative and predictive analysis on the metrics. For the discriminative analysis, the authors use Welch's t-test to compare the means of metric values for vulnerable files against neutral files. The results show that 20 of 23 metrics for one of the projects show statistically significant discriminative power; however, this is only true for about half of the metrics for the other project, including none of the execution complexity metrics. To evaluate the predictive power of the metrics, the authors performed binary classification using Logistic Regression, with tenfold cross-validation. To account for the redundant information in the many metrics, the authors performed feature-space reduction based on information gain ranking. Another issue is the heavy imbalance between the majority (neutral files) and minority class (vulnerable files), which the authors addressed by random under-sampling of the majority class. The end results are that for all three sets of metrics (code, dependency, all-combined) the recall rate is fair (67%–81%) but precision is disappointing (8%–12%). In conclusion, the results show that these metrics don't have statically significant discriminative power, and the predictive capability is not reliable.

Shin et al. (2011) performed a more extensive study on whether complexity, code-churn, and developer-activity (CCD) metrics can be used for vulnerability prediction. To this end, the authors performed empirical case studies on two open-source projects. A total of 28 CCD software metrics were analyzed in this study, including 14 complexity metrics, 3 code-churn metrics, and 11 developer-activity metrics. To evaluate the discriminative power of the metrics, the authors used Welch's t-test, where the test hypotheses were supported by at least 24 of 28 metrics for both projects. To evaluate the predictive power of the metrics, the authors tested several classification techniques, yet they only present the results of one, since all techniques provided similar performance. To validate the model's predictive capability, the authors performed next-release validation where several releases were at hand, and cross-validation where only a single release was available. Both univariate and multivariate prediction hypotheses were evaluated. Based on average values found from the fault prediction literature, the authors select thresholds as at least 70% for recall, and at most 25% for false-positive rate to support the prediction hypotheses. Only 2 of 28 univariate models, and 3 of 4 multivariate models using development history–based metrics predicted vulnerable files with high recall and low false-positive rates for both projects. The authors conclude that development history metrics are stronger indicators of vulnerabilities compared to code complexity metrics collected in this study.

Moshtari et al. (2013) mention three main limitations of previous studies on vulnerability prediction models, hence they propose a new approach to predict vulnerable locations in software based on complexity metrics by resolving the limitations of previous studies. The authors proposed a semi-automatic analysis framework to detect software vulnerabilities, and used its output as vulnerability information, instead of reported vulnerabilities, which the authors claim that provides more complete information about vulnerabilities in a software. Unlike previous studies that only studied within-project vulnerability prediction, this study investigated both within-project and cross-project vulnerability prediction based on data gathered from five open-source projects. Various classification techniques were used for the experiments. A set of 11 unit complexity metrics and 4 coupling metrics were measured at file-level granularity. The reported results for within-project prediction performed on Mozilla Firefox are impressive for various classification techniques (recall above 90% and false-positive rate below 10%). The authors claim that the more complete vulnerability information contributes to this improvement, and justify their claim by comparing the proposed approach with a replication of a previous work by Shin et al. (2011). Cross-project experiments were performed on randomly selected releases of the five projects, where one project was considered as test data and the model was trained on four other projects. According to the reported

F2-measures, the best models for cross-project prediction achieved a detection rate of about 70%, with about 26% false positives.

Meneely et al. (2013) explored the properties of vulnerability-contributing commits (VCC) by tracing more than 65 vulnerabilities in the Apache HTTPD web server back to the version control commits that originally contributed the vulnerable code. The authors manually found 124 VCCs, spanning 17 years, which they analyzed based on code-churn and developer-activity metrics using statistical analysis techniques. Based on the results of this exploratory study, they present several findings: (1) code-churn metrics are empirically associated with VCCs, in a way that bigger commits can potentially introduce vulnerabilities; (2) commits that affected more developers were more likely to be VCCs; and (3) commits by a new developer to a source file are more likely to be VCCs.

Bosu et al. (2014) performed a similar empirical study, where they analyzed more than 260,000 code review requests from 10 open source projects, identifying more than 400 vulnerable code changes using a three-stage semi-automated process. Their aim was to identify characteristics of vulnerable code changes, and identify characteristics of developers likely to introduce vulnerabilities. Some key findings include: (1) changes by less experienced contributors were significantly more likely to introduce vulnerabilities; (2) vulnerability likelihood increases with size of the change (more lines changed); and (3) new files are less likely to contain vulnerabilities compared to modified files.

Perl et al. (2015) studied the effects of employing the meta-data contained in code repositories alongside code metrics to identify vulnerability contributing commits. The authors assert the fact that software grows incrementally, and most open-source projects employ version control systems, hence, commits are natural units to check for vulnerabilities. With this motive, the authors compile a dataset containing 170,860 commits from 66 C/C++ GitHub projects, including 640 vulnerability-contributing commits (VCCs) mapped to relevant CVE IDs. The authors select a set of code-churn and developer-activity metrics, as well as GitHub meta-data from different scopes (project, author, commit, and file) and extract these features for the gathered dataset. Based on this dataset, the authors evaluate their proposed system, named *VCCFinder*, which uses a Support Vector Machine (SVM) classifier to identify VCCs from neutral commits. For evaluation, the system is trained on data up to the end of 2010, and tested against CVEs reported in 2011 to 2014. The authors compare the results of their proposed system against the results of the FlawFinder static analysis tool; which at the same level of recall (recall = 24%), FlawFinder reaches precision of only 1%, while VCCFinder reaches precision of 60%, producing far less false positives. The aforementioned dataset is publicly published by the authors as a contribution to the research community.

Walden et al. (2014) performed a study to compare the performance of predicting vulnerable software components based on software metrics versus text-mining techniques. For this purpose, the authors first built a hand-curated dataset of vulnerabilities gathered from three large and popular open-source PHP web applications (Drupal, Moodle, PhpMyAdmin), containing 223 vulnerabilities. This dataset is offered to the research community as a contribution. For vulnerability prediction based on software metrics, a set of 12 code complexity metrics were selected for this study. For text mining, each PHP source file was first tokenized, unnecessary tokens were either removed or transformed (comments, punctuations, string and numeric literals, etc.), and the frequencies of the final tokens were counted. The well-known "bag-of-words" technique was used to construct numerical feature-vectors from textual tokens of each PHP source file. Based on previous experience with vulnerability prediction studies, the authors select the Random-Forest model as the primary classification algorithm. For model evaluation, the authors used stratified three-fold cross-validation. The authors also addressed the issue of imbalance class data by performing

random under-sampling on the majority class (non-vulnerable code). According to various experiments by the authors, the prediction technique based on text mining performed better on average (i.e., higher recall and precision) and the difference was statistically significant. The authors also tested cross-project vulnerability prediction, yet cross-project prediction performance was generally poor for both approaches. The poor performance of cross-project prediction was expected, since the authors did not account for the unequal distribution of data between applications.

Morrison et al. (2015) state that while defect prediction models are adopted by Microsoft teams, this is not the case with vulnerability prediction models (VPMs). To explain this discrepancy, the authors try to replicate a VPM proposed by Zimmermann et al. (2010) for two newer releases of the Microsoft Windows operating system. The authors reproduced binary-level prediction precision of about 75% and recall about 20%; however, the binaries often are very large for practical inspection, and source file level prediction is preferred by engineers. Hence, the authors built the same model for source file level granularity, which yielded precision below 50% and recall below 20%. Based on these results, the authors conclude that *"VPMs must be refined to achieve actionable performance, possibly through security-specific metrics."*

Younis et al. (2016) try to identify the attributes of code that contain vulnerabilities that are more likely to be exploitable. To this end the authors gather 183 vulnerabilities from the Linux kernel and Apache HTTPD web server projects, which includes 82 exploitable vulnerabilities. The authors select eight software metrics from four different categories to characterize these vulnerabilities, and use Welch's t-test to examine the discriminative power of each metric. Results of the discriminative power of metrics is mixed; some metrics have statistically significant discriminative power while others do not. The authors also investigate whether there is a combination of metrics that can be used as predictors of exploitable vulnerabilities, where three different feature-selection methods and four different classification algorithms are tested. The best performing model is a Random Forest classifier with Wrapper Subset selection approach, which achieves an F-measure of 84%.

4.2 Discussion

In the previous subsection, we reviewed several recent studies in the field of vulnerability prediction models based on software metrics. A summary of all the articles reviewed in this section is presented in Table 1, where we have specified the key differentiating factors of each work.

One might question the basic decision of trying to predict the existance of software vulnerabilities based on software engineering metrics, as an instance of *confusing symptoms and causes* (Zeller et al. 2011). This criticism is reinforced by some studies (e.g., Walden et al. (2014)) that show basic text mining on software source code can yield better results in terms of prediction performace compared to vulnerability prediction models based on some software metrics; although, this empirical study can't be generalized to all software projects and all software metrics. On the other hand, a recent study by Tang et al. (2015) criticises the conclusions of Walden et al. (2014), since they did not consider the size of individual components that affects the code inspection effort; hence, they compare the predictive power of these two kinds of prediction models in the context of effort-aware vulnerability prediction and conclude that the two metrics perform similarly.

A justification to use software metrics for vulnerability prediction is that these metrics are often either readily available or easily obtained in software engineering projects. Moreover, software fault/defect prediction models are already being used in some software projects and building vulnerability prediction models does not require additional expertise. On the other hand, the purpose of these systems is to act only as a guiding model for better planning and allocation of resources in software engineering teams. Therefore, vulnerability prediction models based on software metrics are a line of research in industry and academia.

Table 1. Summary of Recent Works on Vulnerability Prediction Models Based on Software Metrics

Paper	Metrics	Granularity	Within/ Cross-project	Vulnerability info
(Zimmermann et al. 2010)	Code-churn, complexity, coverage, dependency, organizational	Binary modules	Within-project	Public advisories
(Meneely and Williams 2010)	Developer-activity	Source file	Within-project	Public advisories
(Doyle and Walden 2011)	Code complexity, Security Resources Indicator	Source file	Within-project	Tool-based detection
(Shin and Williams 2013)	Complexity, code-churn, fault-history	Source file	Within-project	Public advisories
(Shin and Williams 2011)	Code complexity, dependency network complexity, execution complexity	Source file	Within-project	Public advisories
(Shin et al. 2011)	Complexity, code-churn, developer-activity	Source file	Within-project	Public advisories
(Moshtari et al. 2013)	Unit complexity, coupling	Source file	both	Self-developed detection framework
(Meneely et al. 2013)	Code-churn, developer-activity	Code commits	Within-project	Public advisories
(Bosu et al. 2014)	Developer-activity	Code commits	Within-project	Public advisories
(Perl et al. 2015)	Code-churn, developer-activity, GitHub meta-data	Code commits	Cross-project	Public advisories
(Walden et al. 2014)	Code complexity	Source file	both	Public advisories
(Morrison et al. 2015)	Code-churn, complexity, coverage, dependency, organizational	Binary modules, source file	Within-project	Public advisories
(Younis et al. 2016)	Code complexity, Information Flow, Functions, Invocations	Functions	Within-project	Public advisories

Based on the previous works reviewed above, it is clear that the field of vulnerability prediction models based on software metrics has yet to mature. Some concluding points, including both challenges and possible future works, can be drawn from the review of previous studies:

—A statistical challenge in the field of vulnerability prediction models is introduced by the fact that vulnerabilities are few and sparse in the datasets. In the field of data mining and machine learning, this issue is known as *imbalance class data*, which can greatly hinder the performance of machine-learning algorithms and there are practices to address the issue (Domingos 2012). Some of the previous works reviewed in this section have addressed the imbalance class data issue with random under-sampling the majority class. This is an important issue that should not be neglected in any study that utilizes machine-learning and data-mining techniques.

—In contrast to the majority of previous studies, Moshtari et al. (2013) used a semi-automated framework for vulnerability detection, which they used instead of information available via public advisories and vulnerability databases (e.g., NVD). They achieved significantly higher recall and precision values compared to previous works (even in the cross-project settings). This could be a promising approach that future researches can also follow to gather more complete vulnerability information and achieve improved results.

—Cross-project studies in the filed of vulnerability prediction models are few, hence an area for future work. Specially compared to the field of defect prediction models, cross-project vulnerability prediction is highly under-researched. Cross-project prediction introduces additional challenges, which arise from the fact that the distribution of data in the training and test sets can vary significantly and hinder the performance of traditional machine-learning and statistical-analysis techniques. This challenge has been addressed in the field

of machine-learning research as *"inductive transfer"* (or *"transfer learning"*) techniques (Pan and Yang 2010), which their use has been investigated in software defect prediction studies (Ma et al. 2012; Nam et al. 2013). These studies can be a basis for future research in the field of cross-project vulnerability prediction models.

— Most studies in the field of vulnerability prediction based on software metrics report poor results. One possible conclusion is that traditional software metrics are not suitable indicators for software vulnerabilities. This conclusion is explicitly discussed by Morrison et al. (2015). Henceforth, defining security-specific metrics, such as the Security Resources Indicator (SRI) proposed by Doyle and Walden (2011) is another area for future studies.

— An uncharted area in this field is using deep-learning methods for vulnerability prediction. Deep learning is a newly emerged topic in the field of machine-learning research, which has made great achievements in several application domains, and is gaining increasing attention from both researchers and practitioners (LeCun et al. 2015). Yang et al. (2015) presented a study on applying deep-learning methods for just-in-time software defect prediction. This is another promising area for future studies in the field of vulnerability prediction models.

5 ANOMALY DETECTION APPROACHES

In this section, we review a category of works that employ an anomaly detection approach using machine-learning and data-mining techniques for software vulnerability analysis and discovery. Anomaly detection refers to the problem of finding patterns in data that do not conform to the normal and expected behavior; often referred to as *anomalies* or *outliers* (Chandola et al. 2009). This problem has been extensively studied in many diverse research areas and application domains; including the domain of software defect and vulnerability discovery.

In the context of software quality assurance, anomaly detection methods are aimed to identify software defects, by finding locations in source code that do not conform to the usual or expected code patterns for application programming interfaces (APIs). Simple examples of such API usage patterns are the function-call pair of `malloc` and `free`, or `lock` and `unlock`. Besides these simple well-known patterns, each API has it's own share of rules and patterns, which can also be quite complicated and not very well-documented. Not conforming to the intended rules and usage patterns of an API can cause software defects and also possibly software vulnerabilities.

Another area where anomaly detection methods are applied for software quality assurance, is for detecting neglected conditions or missing checks. Missing checks are the source of many software defects and vulnerabilities. These checks can be broadly categorized in two types: (1) checks required for proper API usage; and (2) checks that implement program logic. Both types can lead to either software defects or software vulnerabilities. An example of the first type is a check for the proper type or value of input data used as parameters of an API function call. Missing such checks could lead to a crash, undefined, or undesired behavior by the software (e.g., division by zero). The failure can also have security consequences (e.g., buffer-overflow, SQL-injection, etc.). An example of the second type is missing to check the rights or permissions of a subject when accessing a resource object. Again, these logical defects can have security consequences, resulting security logic vulnerabilities (e.g., confidentiality and integrity access control).

Anomaly detection approaches have been proposed for both detecting incorrect API usage patterns, and missing checks. An important aspect of these approaches is the automatic extraction of normal behavior, or in other words, automatic extraction of specifications, rules, and patterns, which are then used as the basis for detecting deviant behaviors. The automatic extraction of normal behavior is crucial to the applicability and success of these approaches, whereas if the normal behavior or specifications were to be provided by human users it would greatly hinder

the efficiency of the approach, since: (1) writing specifications is a hard and tedious task; and (2) human mistakes can lead to inaccurate specifications that result in incorrect outcomes.

In the following, we review and summarize previous works in chronological order, starting with early works from the past decade to the latest works in recent years in the field of anomaly detection approaches for software defect and vulnerability detection. Note that some of the reviewed works focus on security vulnerabilities, while others do not; yet we still review these works, since the proposed approaches don't exclusively aim at non-security defects.

5.1 Summary of Previous Works

Engler et al. (2001) state that a major obstacle to finding program errors is knowing what correctness rules the system must obey, which are often undocumented or specified in an ad hoc manner. To address this problem they demonstrate a technique to automatically extract programmer beliefs implied by program source code. To this end, they discuss the concept of *bugs as deviant behavior*, and propose an approach that extracts programmer beliefs by tailoring *"rule templates"* to the source code (e.g., function calls that fit the rule template *"<a> must be paired with <b>"*). Two types of rules are extracted: (1) Must-beliefs and (2) May-beliefs. Must-beliefs are certain well-known programming rules (e.g., *"a pointer dereference implies that a programmer must believe the pointer is non-null"*). May-beliefs are cases where some code features suggest a belief, but may instead be a coincidence. To distinguish valid May-beliefs from coincidences, the violations (or errors) of these beliefs are found using a statistical analysis techniques called Z-ranking, to rank and sort the errors. For evaluation, the authors checked various rule-templates on complex software systems such as Linux and OpenBSD projects. The results show varied false-positive rates for different scenarios, ranging from 4% to 57%. The authors also test a security checker using rules for well-known security vulnerabilities, which resulted in finding 35 security holes in Linux and OpenBSD.

Livshits and Zimmermann (2005) proposed a tool named DynaMine, that analyzes source code check-ins from revision histories to automatically extract application-specific coding patterns, based on highly correlated method calls. DynaMine analyzes incremental changes, which helps achieve more precise results. The proposed approach starts by pre-processing software revision histories for method-calls that have been inserted, and storing this information in a database to be mined. The mining approach is based on a modified version of the classical *a priori* algorithm, which uses a set of items as its input and produces frequent item-sets and strong association rules among items. The modifications improve the algorithm's runtime and allows the approach to scale for analyzing large software systems. Furthermore, the authors apply several ranking strategies to the patterns that are mined by the algorithm. The extracted patterns are presented to the user for assessment. After patterns are selected by the user, a dynamic analysis tool is used for further validation of selected patterns and detecting violations. The approach was evaluated on two large Java projects. According to the authors, the proposed mining approach discovered 56 previously unknown, highly application-specific patterns. According to experimental results, 32 of 56 (57%) patterns were hit at runtime, which 21 of 32 (66%) of these patterns were confirmed as very likely valid patterns. Furthermore, more than 260 pattern violations were found, yet the authors do not evaluate the false-positive rate.

Li and Zhou (2005) assert that programs usually follow many implicit and undocumented programming rules, which violating these rules by unaware programmers easily introduces defects. The authors propose a method called PR-Miner to extract implicit programming rules from large source code, without prior knowledge about the software and little effort from programmers. PR-Miner was designed to extract programming rules in general forms (without fixed templates), containing multiple elements of different types such as functions, variables and data types. In

summary, PR-Miner starts by parsing and pre-processing program source code, removing unnecessary elements such as keywords, constant data values, and so on, and renaming local variables in functions and data fields in structs to similar names based on their data type. After pre-processing, all program elements are hashed into a number, and function definitions are mapped into a set of numeric hash values of all elements inside the function, inserted as a row into an item-set database. This database is mined using a frequent itemset mining algorithm (namely FPclose), to find frequent program elements that appear together. These frequent sets of program elements are called programming patterns. An efficient rule generation algorithm is used to extract programming rules from the frequent programming patterns. The programming rules are used to detect violations, which is based on the idea that programming rules are usually followed in most cases and violations seldom occur. Evaluation is performed on three large open-source projects. The results indicate that PR-Miner found thousands of rules in these software projects, which the authors were unable to validate all, only discussing a few samples. PR-Miner also reported many violations that the authors managed to manually assess only the top 60 reports, revealing that false-positive rate is high (between 73%–90%).

Wasylkowski et al. (2007) reassert the fact that interacting with objects often requires following a model or protocol, which are not always documented, and violations can lead to defects. To automatically extract typical object usage models, the authors propose mining sequences of method calls in program source code, which are then used to find deviations as defect candidates. At first, object usage models are extracted from Java bytecode, which are finite state automata with anonymous states, and feasible method calls as state transitions. This model is extracted using intraprocedural static analysis for each single method. Afterwards, temporal properties over method calls (e.g., *"method* `next()` *can precede* `hasNext()` *"*) are mined from the usage models, using frequent itemset mining on closed patterns. These frequent patterns express normal object usage and are used to train a classifier to identify violations of these patterns, which are regarded as likely defect locations. Furthermore, the authors introduce a defect indicator, which ranks anomalies based on several factors. The proposed approach was implemented as a tool called JADET (for Java Anomaly Detector). Evaluation was performed on five popular open-source Java programs. According to the reported results, among the top 77 anomalies detected by JADET in all 5 projects, there are 40 (52%) false positives, 5 defects, 5 code-smells, and 27 hints.

Acharya et al. (2007) state that previous approaches are unable to capture some useful ordering information among API usage patterns, especially when multiple APIs are involved across different procedures. The authors propose an automatic approach to extract "frequent partial orders" from API client code. The approach consists of four main steps and is based on a previous work by a few of the authors named MAPO (Xie and Pei 2006). In the first step, a push-down model checking (PDMC) process is employed to extract interprocedural control-flow-sensitive static traces related to the APIs. In the second step, an algorithm is used on a given trace to separate different usage scenarios, so each scenario could be mined separately. In the third step, a method named FRECPO is used for mining "frequent closed partial orders" (FCPOs) from the set of scenarios extracted from each static execution trace. The FCPOs might not be universal patterns, and can only be specific to the analyzed client code. To address this issue an algorithm named Mine-Verify is introduced, which uses two randomly split disjoint sets of clients to verify the patterns. For evaluation, the framework was applied on 72 client programs of the X11 UNIX windowing system. For each experiment, 36 clients were randomly chosen as mine clients and the remaining 36 clients were used as verify clients. The authors don't provide an evaluation of their experimental results, instead they only present an evaluation of a single example; for which, 5 of 6 known patterns was successfully detected by the approach and only one false pattern was reported.

Chang et al. (2008) emphasize the importance of neglected conditions as a difficult-to-find class of defects by performing a preliminary study on bug fixes in versions 1.0 and 1.5 of the Firefox project, which show that among 167 selected bugs, 109 (65%) involved one or more neglected conditions. To this end, the authors present an approach combining data-mining techniques with static program analysis to extract implicit conditional rules in a code base and to detect neglected conditions as rule violations. Accordingly, a program is represented by a "system dependence graph (SDG)," as a collection of modified "procedure dependence graphs (PDG)," called "Enhanced PDG" (EPDG). The EPDGs are augmented with so called "shared data dependence edges" (SDDE), which link program elements that use the same variable definition in a control-flow path. Potential rules are represented by EPDG minors, which can be thought of as subgraphs of an EPDG in which some paths have been contracted to edges. A "heuristic maximal frequent subgraph mining (HMFSM)" algorithm is used to find recurring graph minors in a database of "near transitive closures (NTCs)" of EPDG subgraphs. After extraction and confirmation of valid rules, the NTC graph database is searched for rule violations (neglected conditions) using a heuristic graph-matching algorithm. For evaluation, the authors performed experiments by applying the approach on four open source projects. More than 1200 candidate rules were detected by the approach in all four projects, which on average less than 25% of detected rules were invalid (not semantically meaningful). The heuristic graph-matching algorithm was fully successful in finding rule instances with a 100% precision, yet the results for violation detection were not impressive, where 79% of reported violations on all four projects were false positives. The authors claim *"about half of false positives were caused by semantically equivalent statements being given different labels"* and present some suggestions to improve the situation for future works.

Thummalapenta and Xie (2009) proposed a novel approach to reduce the false-positive rate of automatically mining programming rules. To this end, they introduce the notion of "*alternative patterns*," where various frequent patterns of an API call are considered together. For example, an alternative pattern can be of the form "P1 or P2" where both P1 and P2 are frequent, and P2 is a semantic alternative for P1. Another source of false positives is the case of *imbalanced alternative patterns*, where P1 and P2 are semantically valid alternatives but P1 is highly frequent and P2 is not much frequent in an entire code base. These imbalanced alternatives denoted as "P1 or P2" are much more challenging to find using traditional mining techniques. An approach called "Alattin" is proposed that includes a new mining algorithm called "ImMiner," which uses an iterative mining strategy to mine both balanced and imbalanced patterns and a technique for detecting neglected conditions. At first Alattin extracts reused APIs in the source code, and feeds them to a code search engine to gather additional relevant code examples. Frequent itemset mining is performed on the gathered database to extract frequent patterns. Then for each frequent pattern, the input database is split into two negative and positive databases, where the negative database includes all pattern candidates that do not conform to the pattern, whereas the positive database includes all conforming candidates. Frequent itemset mining is again applied on the negative database to construct imbalanced alternative patterns. These mined patterns are then used to detect violations at method call sites. For detecting neglected condition, Alattin extracts all conditional checks around API call sites. As each mined pattern consists of several alternatives, Alattin reports a violation only if a call site doesn't satisfy any of the alternatives. For the purpose of evaluation, empirical experiments were performed on 6 Java libraries, where 144 patterns in total were mined with a runtime of about one hour. The authors manually evaluated 90 of the 144 patterns, which indicated 75 (83%) are real rules, 7 are partial rules, and 8 (9%) are false rules. Experiments for violation detection on the same libraries resulted a 28% reduction in false positives compared to similar approaches.

Gruska et al. (2010) investigate the possibility of cross-project anomaly detection. For this purpose, the authors introduce a lightweight, language-independent parser applicable for analyzing programs written in several languages with similar syntax (including C, C++, Java, and PHP). Since the proposed method is based on program structure and function calls, many of the details present in source code can be disregarded and the parser only parses selected parts of source code; this is what makes the parser lightweight and language independent. The parsing process consists of several steps, including creating tokens, identifying structure, and extracting function calls. A specially designed generic abstract representation similar to an abstract syntax tree (AST) is used to store the information extracted by the parser. The abstract representations are used to create function models that are finite state machines where the states represent locations in the code and the transitions are function calls. This model is used to extract all possible sequences of function calls in a specific function with other relevant information, such as name of the function being called, number of parameters, arguments, return value, and target. Afterwards, each function model is transformed into a set of temporal properties that represent the flow of values between function calls. This is achieved by using the JADET tool from a previous work by the authors (Wasylkowski et al. 2007). The JADET tool is extended to support the aforementioned model and used to mine frequent temporal properties from the function models. A concept analysis approach is used for anomaly detection similar to the approach used in the JADET tool. Finally, detected anomalies are ranked and filtered before presented to the user. For evaluation, more than 6000 open source Linux projects were mined to extract 16 million temporal properties reflecting normal API usage. The authors randomly chose 20 projects from the distribution and applied anomaly detection. From the total 138 detected anomalies, only the top 25% were manually evaluated by the authors, which resulted in only 4 defects, 7 code smells, and 39 (78%) false positives. The authors also present their analysis system as a web-based service hosted at `checkmycode.org`, which is currently out of service, as of late 2011.

Yamaguchi et al. (2013) propose a system named Chucky for automatic detection of missing checks in source code, aimed at assisting manual code audit. Chucky combines machine-learning techniques with static program analysis to determine missing checks. The authors discriminate two types of security checks in source code: (1) Checks implementing security logic (e.g., Access Control); and (2) Checks ensuring secure API usage (e.g., checking buffer size). Chucky employs a five-step procedure, executed for each source and sink chosen by the auditor. The analysis starts with robust parsing based on an island grammar, where conditions, assignments, and API symbols are extracted for each function definition. Second, neighborhood discovery is performed using a nearest-neighbor and bag-of-words techniques, based on the similarity of API symbols in functions. Third, lightweight tainting is performed to determine only those checks associated with the target source or sink. Fourth, all functions and their neighbors are embedded in a vector space based on the tainted conditions. And finally, anomaly detection is performed by first computing the center of mass of all embedded neighbor vectors as a model of normality, and then an anomaly score is computed for each function based on the distance of its vector with the normal model of its neighborhood. The final results are sorted and presented to the user based on the anomaly score. Qualitative and quantitative evaluations are performed to demonstrate the efficacy of the approach, where they analyze missing checks in the code of several open-source projects. The authors report that Chucky identified several missing checks in all projects, where almost all of the top 10 reported anomalies for each function either contain a defect or a security vulnerability. Moreover, 12 previously unknown vulnerabilities (i.e., zero-days) were also discovered using Chucky in the course of this research.

Table 2. Summary of Reviewed Works on Anomaly Detection for Vulnerability Discovery

Paper	Type	Approach	Within/ Cross-project	Security focused
(Engler et al. 2001)	API usage pattern	Template-based rule extraction	Within	Yes
(Livshits and Zimmermann 2005)	API usage pattern	Association rule mining	Within	No
(Li and Zhou 2005)	API usage pattern	Frequent closed itemset mining	Within	No
(Wasylkowski et al. 2007)	API usage pattern	Frequent closed itemset mining	Within	No
(Acharya et al. 2007)	API usage pattern	Frequent partial-order itemset mining	Cross	No
(Chang et al. 2008)	Missing checks	Maximal frequent sub-graph mining	Within	No
(Thummalapenta and Xie 2009)	API usage pattern + Missing checks	Imbalanced frequent itemset mining	Cross	No
(Gruska et al. 2010)	API usage pattern	Frequent closed itemset mining	Cross	No
(Yamaguchi et al. 2013)	Missing checks	k-Nearest neighbors + bag-of-words	Within	Yes

5.2 Discussion

In the previous subsection, we reviewed and summarized several studies in the field of anomaly detection for discovering software defects and vulnerabilities as deviant behaviors. A glance summary of all the articles reviewed in this section is also presented in Table 2, where we have specified the key differentiating factors of each work.

As reviewed earlier, anomaly detection approaches can be used to both find technical software vulnerabilities due to improper API usage, as well as logic vulnerabilities due to neglected conditions or missing checks. An important fact to note is that this process can be performed automatically by a tool, without the need to specify security policies or security specifications. In our opinion, this is the most promising aspect of anomaly detection approaches. However, there are limitations to anomaly detection approaches for software defect and vulnerability discovery:

— Anomaly detection approaches are only effectively applicable for mature software systems. This limitation is because of the basic assumption that missing checks or improper API usages are rare events and the majority of conditions applied to security objects as well as API usages in a software project are correct. This assumption mainly holds true in mature software projects.
— A conditional check or API usage must be frequent in the code-base to be detected as a pattern by mining algorithms. Rare checks or API usages are unlikely to be mined as a pattern, consequently deviations cannot be detected. All works that utilize a frequent itemset mining approach suffer from this limitation (e.g., Li and Zhou (2005), Wasylkowski et al. (2007), and Gruska et al. (2010)).
— Often times, anomaly detection approaches are unable to specify the type of defect or vulnerability, since these approaches can only find that a given code doesn't conform to any of the normal rules or patterns, and it could be a violation of any. Of course there might be cases where an instance is clearly violating a single rule or pattern and not any other, which in this case the type and even the fix for the defect can be specified by the system.

—The high false-positive rates of previous approaches shows that these systems are not yet reliable and the outputs require careful manual audit, which limits the usability of anomaly detection systems. Some notable works suffering from high false-positive rates include (Li and Zhou 2005; Wasylkowski et al. 2007; Chang et al. 2008).

The limitations of anomaly detection approaches are not exclusive to the domain of software defect and vulnerability discovery, and many other application domains also suffer from such shortcomings. The anomaly detection paradigm has been challenged in the field of network intrusion detection; for example Gates and Taylor (2006) present a provocative discussion, where they question some of the assumptions commonly made by researchers in the field. Sommer and Paxson (2010) also provide a through discussion on the challenges of adopting the anomaly detection approach for network intrusion detection. A more general discussion on the challenges and different aspects of anomaly detection systems in different application domains is provided by Chandola et al. (2009).

It is clear that there is still room for much further progress in the field of vulnerability discovery using anomaly detection. Some possible future works drawn from the review of previous studies is as follows:

—A prevalent problem in previous works on anomaly detection for defect and vulnerability discovery is the high false-positive rate. High false positives in the output of an analysis tool renders it useless by overwhelming its users. Improving the accuracy by reducing the false-positives rate is of high importance for future works. The work by Thummalapenta and Xie (2009) is an example of research in this direction that presents interesting contributions.
—A problem in security focused approaches such as Chucky (Yamaguchi et al. 2013), is the inability to distinguish security relevant anomalies (anomalies that are vulnerabilities) from those that are not. These approaches can successfully find both missing check and improper API usage defects, yet not all of these defects are security vulnerabilities. Future works can propose novel approaches to help distinguish defects from security vulnerabilities to make the results more relevant to the security analyst and improve usability.
—Graphs are rich representations that are extensively used in the domain of program analysis, software testing, and software vulnerability discovery. Chang et al. (2008) investigated the use of graph-mining and graph-matching techniques to discover missing-check defects in software. As reviewed earlier, this work achieves promising results in extracting rules and matching the instances, yet it produced high false-positive rates. A recent survey is published by Akoglu et al. (2015), which reviews recent advances in the field of graph-based anomaly detection and description. Further research in the area of graph-based anomaly detection for vulnerability discovery can be pursued in future works.

6 VULNERABLE CODE PATTERN RECOGNITION

In this section, we review another category of studies that utilize data-mining and machine-learning techniques to automatically extract features and patterns of vulnerable code segments, and discover software vulnerabilities via pattern-matching techniques.

Similar to the anomaly detection paradigm discussed in the previous section, this category of approaches also analyze and extract features from program source code (either high-level source code or binary machine code). However, in contrast to the anomaly detection category where the aim was to extract models and rules of normality, this category of works strives to extract models and patterns of vulnerable code.

The common theme in this category of studies is to gather a large dataset of software vulnerability samples, process them to extract feature vectors from each sample, and utilize machine-learning

algorithms (mostly supervised learning) to automatically learn a pattern recognition model for software vulnerabilities. To this end, different approaches are used to process and extract features from program source code; including: conventional code parsers, static data-flow and control-flow analysis, dynamic analysis, text mining on program source code, and so on.

In the following, we provide a review summary of recent works in chronological order for this category.

6.1 Summary of Recent Works

Yamaguchi et al. (2011, 2012) propose a method for assisted discovery of vulnerabilities, by introducing the concept of *"vulnerability extrapolation,"* which aims at identifying unknown vulnerabilities based on programming patterns observed in known security vulnerabilities. The incentive for *vulnerability extrapolation* is based on the observation that a recurrent scenario in a security analyst's daily work cycle is to search for similar instances of a recently discovered vulnerability, in the rest of the code base (Heelan 2011). To this end, the authors propose a four step process: first, a robust parser based on an island grammar is employed to extract abstract syntax trees (AST) for each function in C/C++ code, even without a working build environment. Second, the ASTs of the functions are embedded in a vector space, by a method similar to the bag-of-words technique. This is accomplished by first discarding irrelevant nodes in the AST and representing each function as a vector of contained subtrees. The subtrees are represented by vectors indicating the existence of certain API symbols. Third, latent semantic analysis is used on this vectorial representation to identify structural patterns in the code corresponding to subtrees frequently occurring together in ASTs of the code base. This is achieved by first aggregating all vectorial representations of functions into a large sparse matrix and applying the singular value decomposition (SVD) technique, which outputs a matrix whose rows describe all functions as mixtures of structural patterns. Fourth, vulnerability extrapolation is performed by comparing the rows of the output matrix from the previous step using a suitable distance function (e.g., the cosine distance). For the purpose of evaluation, the authors perform experiments on four open-source projects: LibTIFF, Pidgin, FFmpeg, and Asterisk. The authors used some of the latest reported vulnerabilities in each project as seeds for extrapolation, which resulted in discovering several zero-day vulnerabilities.

Shar and Tan (2012, 2013) propose a set of static code attributes based on data-flow analysis of PHP web applications that can be used to predict program statements that are vulnerable to SQL-injection (SQLI) and cross-site scripting (XSS) attacks. A total of 20 static code attributes are proposed by the authors that reflect different data-flow aspects of a code segment, such as: the number of statements that input data from various sources (e.g., HTTP requests, files, databases); the type of input data; the number of different output sink statements (e.g., database queries, HTML outputs); and the number of different input validation and sanitization statements.

To evaluate the effectiveness of the attributes, the authors developed a prototype tool called PhpMinerI and performed experiments on eight open source web applications based on PHP. PhpMinerI extracts the control-flow (CFG) and data-flow graphs (DFG) of a given PHP program and performs backward data-flow analysis on target sink statements to a reaching input source statement. With this backward analysis, the aforementioned attributes are calculated and each sink is represented as a 20-dimensional vector. These vectors along with their known vulnerability status are fed as training data to different classification models. The resulting classifiers are used to predict the vulnerability status of unseen sink statements. A 10×10 cross-validation method is employed for evaluation on each open-source project. The best experimented model reported by the authors (i.e., the MLP classifier) achieved an average result of 93% recall and 11% false-positive rate for SQLI, and 78% recall and 6% false-positive rate for XSS vulnerabilities. The authors also compare the performance of PhpMinerI against Pixy, an open-source static tainted data-flow

analysis tool for PHP web-applications aimed at detecting SQLI and XSS vulnerabilities, which is also the base of PhpMinerI. On average, Pixy discovered more vulnerabilities but also produced much more false positives compared to PhpMinerI.

Shar et al. (2013) extend their previous work to address two major drawbacks of the approach: (1) limited prediction accuracy of the static attributes; and (2) the need for a labeled training dataset. To this end, they propose a set of additional attributes that are gathered via dynamic analysis of program execution traces. The proposed dynamic attributes count the number of statements in an execution trace that invoke certain input validation and sanitization functions. The authors also make changes to the list of static attributes. The final proposed attributes set is composed of 15 static and 7 dynamic attributes. In addition to the modified attributes set, the authors propose an unsupervised prediction approach to make the system applicable in the absence of labeled training data. The authors employ some data processing techniques, such as normalization and dimensionality reduction with principal components analysis (PCA). The authors evaluated the new attribute set by updating the PhpMinerI tool and experimenting on six open-source PHP web applications. For the supervised setup, two classification models were tested: MLP and Logistic Regression. For the unsupervised setup, the well-known k-Means clustering algorithm was used with k = 4 and the Euclidean distance function. The authors report 90% recall and 85% precision rates on average for the supervised setup. The unsupervised setup achieved 76% recall and 39% precision on average.

Shar et al. (2015) further extend their previous work, with several additions and modifications: first, in addition to SQLI and XSS, they extend the vulnerability scope by adding *remote code execution* (RCE) and *file inclusion* (FI) web vulnerabilities. Second, they employ static backward program slicing and leverage control dependency information in order the extract different execution paths from program slices. Third, the input validation and sanitization (IVS) attributes set were again modified to include 10 static and 22 dynamic attributes. And fourth, instead of the previous unsupervised approach, the authors experiment a semi-supervised approach along side the supervised approach for vulnerability prediction. The modified approach is implemented in the renamed *PhpMiner* tool, and evaluated on seven open-source PHP web applications with known vulnerabilities. In the supervised setup the Logistic Regression and Random Forest classification techniques are used for experimentation. In the semi-supervised setup the Co-Forest algorithm, with 20% labeled and 80% unlabeled data is used. The supervised setup recall rates vary for different types of vulnerabilities: 92% for SQLI, 72% for XSS, 76% for FI, and 64% for RCE. The average reported false-positive rate is less that 10%. With shortage of labeled training data, the semi-supervised experiments show the advantage of such learning schemes compared to supervised approaches, where 25% improvement in recall and 3% reduction in false-positive rates are achieved.

Hovsepyan et al. (2012) presented a preliminary study on predicting vulnerable software components using text analysis techniques. A more elaborate version of the work is presented by Scandariato et al. (2014). The authors investigate the possibility of predicting the vulnerability status of software components using text-mining techniques. The proposed method is mainly based on the bag-of-words technique, where a vectorial representation for each software component (i.e., source code file) is constructed based on the frequency of different textual tokens in the source code. After constructing the vectorial representations, five classification techniques are used to train prediction models based on a labeled training dataset of 10 open-source Android applications. To provide the labels of the training dataset, the authors have used a commercial program vulnerability analysis solution. An extensive set of experiments are reported by the authors, including: within-project experiments, subsequent-versions experiments, and cross-project experiments. The authors set the acceptance performance thresholds as precision and recall values above or equal to 80 percent. According to the reported results, the best performing classification models

are obtained via Random Forest. The end results are that, within-project and subsequent-versions vulnerability prediction is acceptable, yet cross-project vulnerability prediction results are not.

Yamaguchi et al. (2014, 2015) propose a new graph representation named the *Code Property Graph*, for modeling and discovering vulnerabilities by means of traversals over this graph. Yet designing effective traversals to detect complicated vulnerabilities can be highly difficult, hence the authors propose an automated approach to infer sanitization patterns for a given sensitive sink, and use these patterns as traversal queries over the code property graph to efficiently locate unsanitized data flows resulting taint-style vulnerabilities. For this purpose the authors propose a four-step procedure: first, for each call of a selected sink, so-called *"definition graphs"* are generated by analyzing the code property graph, which compactly encode both argument definitions and sanitizations in a special structure to enable easy enumeration of feasible invocations. Second, the definition graphs are decomposed into individual invocations and cluster analysis is performed for all call sites to obtain patterns of common argument definitions. Third, the generated patterns are extended by adding potential sanitization checks for restricting the flow of data to arguments. Fourth and finally, the inferred patterns are expressed as a graph traversal query over the code property graph for detecting taint-style vulnerabilities. The proposed method was implemented as a prototype tool named *"Joern"* and used to perform experiments on five popular open-source projects. The results indicate that the proposed method is able to construct effective queries for a set of known vulnerabilities, including the infamous Heartbleed bug, and reducing the amount of code review by 95%. The extracted search pattern also lead to the discovery of five previously unknown (zero-day) vulnerabilities.

Pang et al. (2015) investigate the possibility of predicting vulnerable software components using the *N-gram* text-mining technique. Previous studies such as Scandariato et al. (2014) tested the bag-of-words technique, which is the simplest form of N-gram analysis. A major concern when performing N-gram analysis is dealing with high dimensionality, which could greatly hinder the performance of classification models. To overcome this challenge, the authors employ a hybrid approach, combining N-gram analysis and statistical feature selection for predicting vulnerable software components. As a pre-processing phase, certain tokens in program source code were excluded from the N-gram analysis, including operators, separators, and comments. For feature selection, the authors use a statistical hypothesis testing approach based on the Wilcoxon rank-sum test to rank the features and exclude a large number of less important features. The SVM classification algorithm is selected for building the prediction models. To evaluate the approach, the authors perform empirical experiments on four open-source Android applications based on Java. For vulnerability information the authors use the information published by Scandariato et al. (2014) where a commercial program vulnerability analysis solution was used to label each Java source file. The authors extracted all N-gram features for $(1 \leq N \leq 5)$ from all Java source files and performed their statistical feature selection technique, where only one fifth of features were used to train the SVM model and the rest were discarded. The authors used fivefold cross-validation to evaluate the prediction performance. On average, the model achieved almost 96% for precision and 87% for recall for within-project setup. Results for cross-project experiments were on average 67% for precision and 63% for recall.

Grieco et al. (2015) emphasize the lack of tools and methodologies for OS scale program testing with limited time budget, and assert on the need for fast and efficient techniques to quickly identify programs that are more likely to be vulnerable, and direct program fuzzing campaigns. Hence the authors propose a scalable vulnerability analysis approach based on machine-learning techniques for binary code. The proposed approach uses lightweight static and dynamic features to predict if a binary program is likely to contain easily exploitable memory corruption vulnerabilities. The static features are the set of finite sequence calls to the standard C library in the program, which

Table 3. Summary of Reviewed Works on Vulnerable Code Pattern Recognition

Paper	Code Processing Approach	Learning Approach	Static/ Hybrid	Source/ Binary
(Yamaguchi et al. 2011, 2012)	Extracting AST with parser	Supervised (classification)	Static	Source
(Shar and Tan 2012, 2013)	Static data flow analysis	Supervised (classification)	Static	Source
(Shar et al. 2013, 2015)	Static program slicing and control flow analysis	Semi-supervised and supervised (classification)	Hybrid	Source
(Scandariato et al. 2014)	Bag-of-words extraction from program source text	Supervised (classification)	Static	Source
(Yamaguchi et al. 2014, 2015)	Extracting Code Property Graph	Unsupervised (clustering)	Static	Source
(Pang et al. 2015)	N-gram analysis on program source text	Supervised (classification)	Static	Source
(Grieco et al. 2015)	N-gram analysis on function call sequences	Supervised (classification)	Hybrid	Binary

is gathered using a linear sweep disassembly and then performing a very lightweight static analysis to extract the set of desired function calls. The dynamic features are obtained by analyzing program execution traces and extracting calls to the C standard library with its arguments and the final state of the process. To utilize machine-learning techniques, the authors considered each call trace as a text document and used two text-mining techniques for vectorization of data: N-grams and word2vec. The proposed method was implemented as a prototype tool named VDiscover and tested on more than 1000 test cases from the Debian Bug Tracker, distributed over almost 500 Debian packages. To address the class imbalance issue, random oversampling technique was used, and for the overfitting issue, the dropout training technique was used. The authors tested several machine-learning algorithms, such as Logistic Regression, MLP, and Random Forest. Each algorithm was trained either with static or dynamic features separately. Moreover, each vectorization technique was used separately for training the models. At the end, the best performing model was a Random Forest trained with dynamic features, vectorized with 2- or 3-grams, achieving an average test error (average of false negative and false positive) of 31%. Both the dataset and the VDiscover tool are published under open licenses.

6.2 Discussion

In the previous subsection, we reviewed and summarized recent works in the field of vulnerable code pattern recognition for vulnerability analysis and discovery. A brief glance summary of all the articles reviewed in this section is presented in Table 3, where we have specified the key differentiating factors of each work.

Vulnerable code pattern recognition is a promising approach to be used for the problem of vulnerability discovery, yet previous works in this area suffer from some important limitations. In our opinion, a few aspects must be provisioned in such works to produce effective vulnerability discovery systems, which are missing from many of the previous works in this field:

— An effective vulnerability discovery system must be able to *identify the type of a vulnerability*. This is important, since security analysts in software engineering teams need to prioritize their tasks due to limited resources. Effective prioritization can be done based on the severity of vulnerabilities, which is related to the type of vulnerabilities. Many studies reviewed in this section lack this ability where they only present models to distinguish vulnerable code from non-vulnerable code, but the specific type of vulnerability is not determined.

—Another aspect of an effective vulnerability discovery system is the ability to *precisely point out the location of vulnerabilities*. This is related to the granularity of the analysis approach, and most studies suffer from utilizing coarse granularity. Some studies analyze programs at software components granularity, others at source file granularity, others at function level granularity, and others at program slice or execution paths leading to sensitive operations. The more a system provides finer locations for vulnerabilities, the more useful the system is.
—Previous studies (Yamaguchi et al. 2014) have shown that effective modeling and discovery of vulnerabilities requires information about different aspects of software, including: syntactic information, control-flow information, and data-flow information. Some previous studies employ naive approaches based on very limited or shallow information about the program (e.g., Scandariato et al. (2014)).

Based on the aforementioned limitations, we present a few areas to be pursued in future works:

—Investigation for vulnerable code pattern recognition approaches with the above mentioned aspects is a major area for future studies. As mentioned earlier, the ability to identify the type of vulnerabilities, and specifying ever more precise locations of vulnerabilities is highly desirable. These aspects can be achieved by researching on specific attributes for different types of vulnerabilities and by finer analysis granularity with suitable matching techniques.
—Another valuable area for future studies is to define rich feature extraction and description techniques to achieve higher precision and recall for vulnerability discovery; also known as *feature engineering*. Feature engineering is domain-specific (Domingos 2012), and thus each application domain demands its own specific features; hence, future research should pay more attention to the feature engineering aspect. An example of recent research on feature engineering is the work by Long and Rinard (2016) on automatic patch generation for self-correcting software; where a system named *Prophet* is proposed, which learns a probabilistic model of correct code based on a novel set of features for capturing deep semantic properties of correctness across different applications (Long and Rinard 2016). The works by Yamaguchi et al. (2012, 2015) are examples of feature engineering based on rich graphical program representations. Further research can be pursued in this area inspired by other fields such as graph-mining and graph-learning (Aggarwal and Wang 2010; Cheng et al. 2014; Foggia et al. 2014).
—As discussed in earlier sections of this article, using *deep-learning* methods (LeCun et al. 2015) to build powerful cross-project vulnerable code pattern recognition systems is another promising area for future works. Peng et al. (2015) present a pioneering work in this field that uses deep-learning methods for program classification (a program analysis application); although this work is not in the field of software vulnerability analysis and discovery, it is related to the field of program analysis applications and demonstrates the potentials of utilizing deep-learning approaches in this field.

7 MISCELLANEOUS APPROACHES

In this section, we review and discuss some other notable works that utilize different techniques from the fields of AI and data science, for software vulnerability analysis and discovery, yet they do not fit in any of the aforementioned categories, nor they constitute a coherent category. Nevertheless, these studies employ novel approaches that could be valuable to inspire future studies in this field. In the following, we review and summarize each study in chronological order. At the end, a brief summary of all reviewed studies in this section is presented in Table 4.

Table 4. Summary of Reviewed Miscellaneous Approaches

Paper	Approach Summary
(Sparks et al. 2007)	Used Genetic Algorithm (GA) for intelligently guiding the input selection process of black-box fuzz testing
(Wijayasekara et al. 2012, 2014)	Used text mining (bag-of-words) on bug reports in open bug databases for identifying hidden impact bugs (HIBs)
(Alvares et al. 2013)	Used a hybrid of static data-flow analysis and computational intelligence (GA and FSS) techniques for discovering exploitable memory corruption vulnerabilities
(Medeiros et al. 2014)	Used classification techniques on the output of static tainted data-flow analysis for web application vulnerability discovery to identify false-positive reports
(Sadeghi et al. 2014)	Used a probabilistic rule ranking approach based on the information contained in categorized software repositories to improve the efficiency and scalability of static vulnerability analysis tools

Sparks et al. (2007) investigated the use of *Genetic Algorithm* (GA) for intelligently guiding the input selection process of black-box fuzz testing. For this purpose, the authors propose to treat the transition of input data through program statements as an estimated parameter, and develop a *"dynamic absorbing Markov process"* to model this behavior. This Markov process is used to construct a fitness function for the GA, such that based upon dynamic feedback concerning the success of previously tried input values, it leads to the execution of less probable paths in the control flow graph of the program. For the evolution of genomes through generations, a grammatical evolution process is used where the genomes are variable-length strings encoding the production rules of the context-free grammar used to generate input values for the program. The proposed approach was implemented under the open source PAIMEI reverse engineering framework. For evaluation, the tool was tested on the `tftpd.exe` Windows server program and compared it to a random exploration. The results, averaged over 3,000 generations for 50 runs, indicate the GA achieved 85% coverage while random selection reaches 50% coverage. Both approaches found two vulnerable `strcpy()` functions, where the first was reached by the GA in 224 generations and the second after 227 generations, compared to the random selection that reached the first after 2,294 generations and the second after more than 9,000 generations.

Wijayasekara et al. (2012) present a preliminary study on identifying hidden impact vulnerabilities by mining and analyzing public bug databases. *Hidden impact vulnerabilities* (HIVs) or *hidden impact bugs* (HIBs) are *"software bugs that are disclosed to the public via bug databases before being identified as having a security impact and being labeled as vulnerabilities"*; thus, even though the bug is known to the community and the public, it may not be as quickly fixed by developers, because the security implication are not understood. Due to the importance of this matter, the authors perform a study on public bug databases of two large open-source projects: Linux kernel and MySQL database; the reported results indicate that a significant portion of discovered vulnerabilities were initially reported as HIBs.

In a later work by the same authors (Wijayasekara et al. 2014), a methodology is proposed for identifying HIBs by performing text mining and analysis of bug reports in open bug databases. The proposed method consists of several steps: first, both short and long textual descriptions of bug reports are extracted, tokenized, compressed by combining synonyms and hyponyms, and

finally stemmed. After these pre-processing operations, the most frequently appearing words are extracted, and each bug report is vectorized based on these frequent words using a bag-of-words approach. Finally, these feature vectors are used to train a classifier for identifying HIBs. The proposed method is evaluated based on bug reports of the Linux kernel project from January 2006 to April 2011, containing 6,000 regular bugs and 73 HIBs. The authors tested three different classification algorithms, yet even the best performing model didn't achieve acceptable results. Note that the authors did not address the severely imbalanced class data issue.

Alvares et al. (2013) investigate a hybrid approach using program analysis and computational intelligence techniques from the field of AI for finding exploitable memory corruption vulnerabilities. To this end the authors proposed a three step method: first, program source code is segmented to individual functions, and some information about variables is extracted. Second, all possible assignment operations are extracted using a reachability analysis based on abstract interpretation and intraprocedural data flow analysis. The result of this step is the set of all reaching definition traces for each function, which can be seen as a multi-dimensional search space for possible states of local variables. Third, computational intelligence techniques are used to identify input data that can trigger inconsistent states for local variables. For this matter, a suitable fitness function is defined by the authors and two different computational intelligence algorithms are tested: Fish School Search (FSS) and Genetic Algorithm (GA). The authors empirically evaluate the proposed method with a single known vulnerable C function as a case study. The proposed method is able to find suitable inputs that trigger the vulnerability in reasonable time.

Medeiros et al. (2014) assert that static tainted data flow analysis could be an effective mechanisms for vulnerability discovery if only a solution can be used to remedy the high false-positives rate. To this end, the authors investigated a novel hybrid approach for automatic discovery and correction of web application vulnerabilities using data-mining and machine-learning techniques to predict false positives. The authors performed a manual analysis on some instances of web vulnerabilities and selected a set of 14 attributes that lead to false positives. Based on these attributes, a dataset of 76 vulnerability reports including 32 false positives and 44 real vulnerabilities was used to train several classification models for detecting false-positive reports. The best performing model was the Logistic Regression algorithm. The proposed approach is implemented as a proof of concept prototype tool named *Web Application Protection* (WAP), and consists of several steps: first, static global, interprocedural and context-sensitive tainted data flow analysis is performed on source code of PHP web applications to identify possible XSS, SQLI, RFI, local file inclusion (LFI), and OS command injection (OSCI) vulnerabilities. Second, the trained classifier is used to identify and exclude false-positive reports. Third, a fixed set of code templates are used to correct the detected vulnerable sinks in the source code. Finally, the corrected code along side additional information about the vulnerabilities is provided as feedback to the web developer for further confirmation and educational purposes. For evaluation, the authors performed a set of experiments on 35 open source PHP web applications and compared the results against two similar tools: Pixy and PhpMiner. The reported results indicate that WAP performs significantly better than the other tools, although the complete details of false-positive and false-negative rates for the WAP tool is not presented.

Sadeghi et al. (2014) emphasize the importance to utilize static analysis techniques at large scale for discovering vulnerabilities in large software repositories, such as "Google Play Store" and "Apple App Store"), yet a solution is required to overcome the deficiencies and improve scalability of static analysis. To this end, the authors state that the fact that a majority of applications provisioned on categorized repositories are built using a common application development framework (ADF) presents additional opportunities, and thus they investigate the utility of such information in favor of security inspection and analysis of software. The authors propose an analysis

framework consisting of two main parts: (1) rule ranking and (2) vulnerability analysis. In the first part, a categorized software repository is inspected by a set of static analysis tools using a set of vulnerability detection rules. The output report is analyzed to find the likelihood of occurrence for each vulnerability in each category, using a probabilistic rule classification approach. The result of this analysis is a ranking for the vulnerability detection rule sets. In the second part, the ranking results are used to improve the efficiency of static analysis. To this end, a rule selector uses the rule rankings and a given software category, to select only the most effective subset of detection rules. Finally, the static analysis tools are used with the rule subsets to efficiently analyze a given category of software. To assess the effectiveness of the main assumption and evaluate the proposed framework, the authors perform comparative experiments on a set of 460 Android apps against a set of 441 Java apps. The results indicate that a 68% reduction of rule set size is possible in Android apps without any loss of coverage, while this reduction is less that 37% for Java apps. In terms of computational cost, this reduction results in 67% speed up of the analysis process on average.

8 ANALYSIS OF APPLIED TECHNIQUES

In this section, we present an analysis of the applied machine-learning and data-mining techniques in the reviewed work, and draw some conclusions. The analysis is presented in three sub-sections, corresponding to the three main steps of applying machine-learning techniques.

8.1 Feature Engineering and Representation

The process of feature extraction, selection, and description is known as *feature engineering*, which is considered the key to the success of machine-learning systems (Domingos 2012). The performance of machine-learning models is greatly dependent on the quality of features extracted from data (i.e., discriminative and expressive power of features). As mentioned earlier, feature engineering is domain-specific (Domingos 2012), and thus each application domain demands its own specific features. In this section, we analyze the feature-engineering applied in previous work.

In the first category, which vulnerability prediction models where constructed based on measured software engineering metrics, little work is done on feature engineering. This is mainly because software engineering metrics are readily available quantitative values, which are easily embeddable in flat vectorial representations. Only a few studies applied statistical or information theoretic analysis techniques for feature selection, to reduce dimensionality (i.e., Shin et al. (2011) and Younis et al. (2016)). As discussed earlier, most studies in this category report poor results, which could be related to the poor feature-engineering done in this field.

In other categories, different feature engineering and representations are used to tackle the problem at hand. In the anomaly detection category, various types of features are defined and used for the purpose of specifying the normal behavior, including: rule-templates for method-call sequences, numeric records based on hashing of program elements, sequences and partial orders of method-calls, augmented program dependence graphs, and vectorial representation of program elements using a bag-of-words approach. Earlier works in this category (such as Engler et al. (2001) and Livshits and Zimmermann (2005)) are more focused on the order of method-calls as API-usage rules and report poor results. More recent studies (such as Chang et al. (2008) and Yamaguchi et al. (2013)) utilize program elements and the flow of information and program control among them, which report more promising results. A conclusion to be drawn from this observation is that data-flow and control-flow among program elements is a more effective feature type for specifying program behavior.

In the third category, which the aim is to recognize patterns of vulnerable code, two types of feature representation are applied: (1) *Vectorial representation* and (2) *Graphical representation*. Text-mining approaches (such as Scandariato et al. (2014)) utilize the vectorial representation. Moreover,

the input sanitization and verification (ISV) features proposed by Shar et al. (2015) also use the vectorial representation. A few works such as Yamaguchi et al. (2014, 2015) utilize the graphical representation, incorporating syntactic, control-flow, and data-flow information about the program under analysis.

From the review of previous work, it is clear that quality feature-engineering with rich representations contribute to the success of machine-learning and data-mining approaches to the problem of software vulnerability analysis and discovery. As discussed earlier, some of the vectorial representation are considered naive approaches with shallow information about the program, while other representations (such as graphs) are more rich and capable of modeling many aspects of a given program.

8.2 Model Construction

In this section, we analyze the model-construction techniques used in previous reviewed works.

In the category of vulnerability prediction models based on software metrics, all previous studies utilize supervised classification algorithms to construct the target model, which is actually trivial, since by definition the problem of vulnerability prediction is a classification problem. The exclusive use of supervised learning algorithms can be related to the readily accessible software metrics in large amounts, which provides for labeled training data. Various classification algorithms have been experimented in previous studies; including: Logistic Regression, Bayesian Network, Multi-Layer Perceptron (also known as Feed-Forward Neural Network), Support Vector Machines, and Random Forest. Most studies report similar performance by all the classification algorithms; hence, it can be concluded that the classification algorithm is not a major factor in the performance of vulnerability prediction models based on software metrics.

In the category of anomaly detection approaches, different mining and unsupervised learning algorithms are used. The dominant approach with absolute majority in this category is the use of *frequent itemset mining* algorithms. The reason for this dominance can be related to the problem definition, where the aim is to automatically extract implicit programming rules, API-usage patterns, and missing checks from the software source code itself (instead of documentations). Other unsupervised learning schemes (such as cluster analysis, and self organizing maps) have not been experimented, which might be useful.

In the third category, the dominant approach with absolute majority is the use of supervised classification algorithms. Again, this is clearly related to the target problem, which by definition the recognition of vulnerable code patterns is a classification problem. Various classification algorithms have been experimented in previous studies; including: Logistic Regression, Multi-Layer Perceptron, Support Vector Machines, and Random Forest. In contrast to software metrics–based prediction, the performance results varies for different classification algorithms, and researchers have selected a best-choice based on empirical experiments. Yet, the reported best-choice varies in different studies and a single common best-choice does not exist. This is no surprise, since the performance of learning algorithms is greatly dependent on the feature-engineering properties, and different studies in this category have proposed widely different feature-engineering schemes.

A few of the reviewed studies in the category of vulnerable code pattern recognition have experimented unsupervised and semi-supervised learning schemes. As discussed earlier, semi-supervised learning schemes are useful in situations where limited amounts of labeled training data is at hand, and experiments by Shar et al. (2015) show the effectiveness of their use. Unsupervised learning is also utilized in this category, where Yamaguchi et al. (2015) uses cluster analysis as a pre-processing phase; while Shar et al. (2013) experimented with k-Mean clustering as one of the main algorithms for model construction in the absence of labeled training data.

In the miscellaneous approaches category, a more wide variety of techniques are experimented, including text classification of bug-reports, evolutionary computing algorithms (e.g., Genetic Algorithm), and probabilistic ranking of detection rules.

8.3 Model Evaluation

The final step is to evaluate the performance of the constructed model. The dominant evaluation methodology observed in the reviewed works that employed supervised or semi-supervised learning schemes is the *tenfold cross-validation* technique, which enables the calculation of a *confusion-matrix*, and reporting *precision* and *recall* values. This is a well-known and well-established evaluation methodology, adopted by the absolute majority of previous studies in the first and third categories of this survey.

A few studies used a slightly different evaluation methodology. For example, Younis et al. (2016) report a single *F-measure* value, instead of reporting precision and recall values. Moshtari et al. (2013) report the *F2-measure*, which they consider to be a more suitable criteria for security applications. Walden et al. (2014) perform stratified threefold cross-validation, and Peng et al. (2015) perform fivefold cross-validation. Shar et al. (2015) perform 10×10-fold cross-validation, which is considered a more reliable evaluation methodology compared to tenfold cross-validation.

For the anomaly detection category where unsupervised learning and mining schemes are employed, the evaluation criteria is to only examine the reported anomalies, and calculate the false-positive rate. Calculation of a full confusion-matrix is not possible, since the true outputs are not known, hence the rate of false negatives cannot be calculated.

Note that we did not include the performance percentages in any of the summary tables, since, due to the lack of standard benchmark datasets, different studies have chosen different sets of software vulnerability data, hence comparing the performance results is meaningless.

9 CONCLUSION AND FUTURE WORKS

Data-mining and machine-learning techniques have been successfully used in many different application domains, including the domain of computer security. In this article, we extensively reviewed previous work that applied data-mining and machine-learning techniques for software vulnerability analysis and discovery. We organized previous studies in four main categories. For each category, we provided a short yet sufficiently detailed summary of each work, enabling researchers to take a grasp of the key steps, techniques, and evaluation methodology in each category in brief time. Moreover, we presented a table at the end of each section that briefly summarized at a glance the key aspects of various works. In addition to the summary tables, we discussed achievements and limitations of each category, as well as open areas for future studies specific to each category.

Our main intent with this categorized review of previous studies is to provide an organized overview of the achievements and shortcomings of previous studies for prospective researchers in this emerging field. The utilization of data-mining and machine-learning techniques for software vulnerability analysis and discovery presents many opportunities, and we discussed many open problems and open areas for future work in each category.

The main conclusion that we draw from the study of previous approaches is the yet immature state of research in the field of software vulnerability analysis and discovery using machine-learning and data-mining techniques. The majority of studies provide extended empirical case studies, which try to apply well-known techniques and best-practices from the fields of machine-learning and data mining on software vulnerability data, to investigate the answer of some research questions. Learning from other successful application domains (such as Computer Vision), and a few outstanding works reviewed in this survey, we believe that fruitful research areas include:

Table 5. List of Previous Studies with Contributions in Feature Engineering or Data-Mining Algorithms

Area of Contribution	Work Citation
Feature Engineering	(Gruska et al. 2010; Doyle and Walden 2011), (Yamaguchi et al. 2011, 2012), (Shar and Tan 2012, 2013), (Yamaguchi et al. 2013), (Shar et al. 2013, 2015), (Medeiros et al. 2014), (Yamaguchi et al. 2014, 2015), (Grieco et al. 2015)
Feature Engineering + Mining Algorithm	(Livshits and Zimmermann 2005; Li and Zhou 2005; Wasylkowski et al. 2007; Acharya et al. 2007; Chang et al. 2008; Thummalapenta and Xie 2009)

(1) Engineering rich features with high discriminative and expressive power for various types of software vulnerabilities;
(2) Designing new machine-learning and data-mining algorithms, tailored to the characteristics of the problem of software vulnerability analysis and discovery;

As discussed earlier, *feature engineering* is the design of techniques for the extraction and description of quality features from a given dataset, which is known to be a key to the success of machine-learning systems (Domingos 2012). In the category of vulnerability prediction models based on software metrics, this means designing new software security focused metrics. In the other categories, this means designing feature extraction techniques to incorporate syntactic and semantic program features, such as rich graphical program representations, which embody such information on a given program.

Regarding the design of new machine-learning and data-mining algorithms, an often desired but unachieved property in classification and prediction models is *"comprehensibility,"* or the ability for humans to understand the knowledge acquired by machine-learning and data-mining algorithms, to gain trust and be able to perform modifications to the model as needed. In more critical applications (such as security) this property is even more desirable. Comprehensibility is a subject in machine-learning research (Chorowski 2012; Freitas 2014), yet few studies have been published in this area (Van-Assche and Blockeel 2007). A few studies have investigated achieving comprehensibility with machine-learning and data-mining models in some application areas, specifically, software fault prediction models are one of the few (Moeyersoms et al. 2015). Comprehensibility of machine-learning models can be a valuable property in security systems; hence, this is also an area for future work in software vulnerability discovery.

We also mentioned *"transfer learning"* (or *"inductive transfer"*), which is a field of machine-learning research where learning algorithms are designed to properly handle situations where the distribution of data in the training and test sets can vary significantly (a challenge in cross-project vulnerability analysis and discovery) (Pan and Yang 2010).

The results of a simple exploratory analysis revealing the status of reviewed works regarding their contribution area is presented in Figure 2. The figure shows that more than half of previous studies reviewed in this article do not provide novelties in either feature engineering, nor in mining/learning algorithms, and only investigate the answer to some research questions by empirical studies based on previously known feature attributes, and well-known data-mining and machine-learning algorithms and best practices. Table 5 presents the list of previous studies that do present novel contributions in feature engineering or learning/mining algorithms.

Apart from the aforementioned aspects, one major barrier for the proper advancement of such approaches is the lack of standard benchmarking datasets. The current state of evaluations by researchers, where they report performance results on self-gathered datasets, is not a reliable

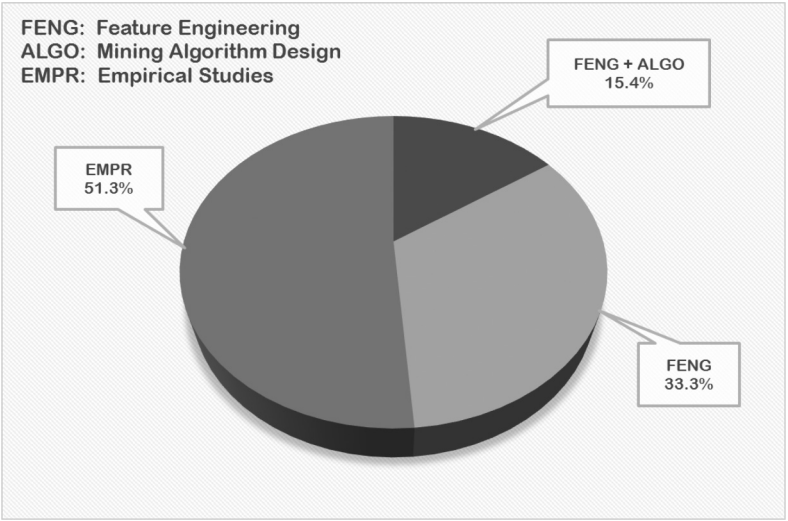

Fig. 2. Status of reviewed works regarding contribution areas.

indicator of the proposed approaches' success or failure. A standard benchmarking dataset that is carefully crafted by onsidering many different aspects of a suitable software vulnerability dataset, provides the means for proper evaluation and comparison of different approaches. The NIST's Software Assurance Reference Dataset (SARD),[1] including the Juliet C/C++ and Java test suits (Boland and Black 2012), is a step in this direction, yet there is still much more work to be done.

The Juliet dataset is suited for evaluating static tainted data-flow analysis tools and lacks the proper characteristics of a dataset to evaluate the effectiveness of utilizing data-mining and machine-learning techniques for vulnerability analysis and discovery. A suitable dataset must include a large number of samples from different software vulnerability types, with adequately diverse forms of the vulnerability type, preferably, based on various programming languages. The better the samples resemble real-world software vulnerabilities, the more useful it can be for evaluations. The creation of such a dataset is also another important research effort and a valuable contribution to the community and industry.

REFERENCES

Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. 2007. Mining API patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering (ESEC/FSE'07)*. ACM, 25–34.

Adobe Security Bulletin. 2015. APSA15-05: Security Advisory for Adobe Flash Player. Retrieved from https://helpx.adobe.com/security/products/flash-player/apsa15-05.html.

Charu C. Aggarwal and Haixun Wang. 2010. A survey of clustering algorithms for graph data. In *Managing and Mining Graph Data*. Springer, 275–301.

Leman Akoglu, Hanghang Tong, and Danai Koutra. 2015. Graph based anomaly detection and description: A survey. *Data Min. Knowl. Discov.* 29, 3 (2015), 626–688.

Marcos Alvares, Tshilidzi Marwala, and Fernando Buarque De Lima Neto. 2013. Applications of computational intelligence for static software checking against memory corruption vulnerabilities. In *Proceedings of the IEEE Symposium on Computational Intelligence in Cyber Security (CICS'13)*. IEEE, 59–66.

Brad Arkin, Scott Stender, and Gary McGraw. 2005. Software penetration testing. *IEEE Security & Privacy* 3, 1 (2005), 84–87.

[1] http://samate.nist.gov/SARD/.

Nathaniel Ayewah, David Hovemeyer, J. David Morgenthaler, John Penix, and William Pugh. 2008. Using static analysis to find bugs. *IEEE Softw.* 25, 5 (2008), 22–29.

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A few billion lines of code later: Using static analysis to find bugs in the real world. *Commun. ACM (CACM)* 53, 2 (2010), 66–75.

Matt Bishop. 2007. About penetration testing. *IEEE Security & Privacy* 5, 6 (2007), 84–87.

Tim Boland and Paul E Black. 2012. Juliet 1.1 C/C++ and java test suite. *Computer* 45, 10 (2012), 88–90.

Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. 2014. Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings of the 22nd ACM International Symposium on Foundations of Software Engineering (FSE'14)*. ACM, 257–268.

Stephen Breen. 2015. What Do WebLogic, WebSphere, JBoss, Jenkins, OpenNMS, and Your Application Have in Common? This Vulnerability. Retrieved from http://foxglovesecurity.com/2015/11/06/what-do-weblogic-websphere-jboss-jenkins-opennms-and-your-application-have-in-common-this-vulnerability/.

Godwin Caruana and Maozhen Li. 2012. A survey of emerging approaches to spam filtering. *ACM Comput. Surveys (CSUR)* 44, 2 (2012), 9.

Cagatay Catal and Banu Diri. 2009. A systematic review of software fault prediction studies. *Expert Syst. Appl.* 36, 4 (2009), 7346–7354.

Varun Chandola, Arindam Banerjee, and Vipin Kumar. 2009. Anomaly detection: A survey. *ACM Comput. Surveys (CSUR)* 41, 3 (2009), 15.

Ray Yaung Chang, Andy Podgurski, and Jiong Yang. 2008. Discovering neglected conditions in software by mining dependence graphs. *IEEE Trans. Softw. Eng.* 34, 5 (2008), 579–596.

Hong Cheng, Xifeng Yan, and Jiawei Han. 2014. Mining graph patterns. In *Frequent Pattern Mining*. Springer, 307–338.

Jan Chorowski. 2012. *Learning Understandable Classifier Models*. Ph.D. Dissertation. University of Louisville.

Codenomicon. 2014. The Heartbleed Bug. Retrieved from http://heartbleed.com/.

Pedro Domingos. 2012. A few useful things to know about machine learning. *Commun. ACM (CACM)* 55, 10 (2012), 78–87.

Mark Dowd, John McDonald, and Justin Schuh. 2007. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities*. Addison-Wesley Professional.

Maureen Doyle and James Walden. 2011. An empirical study of the evolution of PHP web application security. In *Proceedings of the 3rd International Workshop on Security Measurements and Metrics (MetriSec'11)*. IEEE, 11–20.

Dawson Engler, David Yu Chen, Seth Hallem, Andy Chou, and Benjamin Chelf. 2001. Bugs as deviant behavior: A general approach to inferring errors in systems code. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*. ACM, 57–72.

David Evans and David Larochelle. 2002. Improving security using extensible lightweight static analysis. *IEEE Software* 19, 1 (2002), 42–51.

Pasquale Foggia, Gennaro Percannella, and Mario Vento. 2014. Graph matching and learning in pattern recognition in the last 10 years. *Int. J. Pattern Recogn. Artific. Intell.* 28, 1 (2014), 1450001.

Alex A. Freitas. 2014. Comprehensible classification models: A position paper. *ACM SIGKDD Explor. News.* 15, 1 (2014), 1–10.

Pedro Garcia-Teodoro, J. Diaz-Verdejo, Gabriel Macia-Fernandez, and Enrique Vazquez. 2009. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Comput. Secur.* 28, 1 (2009), 18–28.

Carrie Gates and Carol Taylor. 2006. Challenging the anomaly detection paradigm: A provocative discussion. In *Proceedings of the New Security Paradigms Workshop (NSPW'06)*. ACM, 21–29.

Patrice Godefroid. 2007. Random testing for security: Blackbox vs. whitebox fuzzing. In *Proceedings of the 2nd International Workshop on Random Testing (RT'07)*. ACM, 1.

Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox fuzzing for security testing. *Queue* 10, 1 (2012), 20.

Gustavo Grieco, Guillermo Luis Grinblat, Lucas Uzal, Sanjay Rawat, Josselin Feist, and Laurent Mounier. 2015. *Toward Large-scale Vulnerability Discovery Using Machine Learning*. Technical Report. The Free International Center of Information Sciences and Systems (CIFASIS), National Council for Science and Technology of Argentina (CONICET).

Natalie Gruska, Andrzej Wasylkowski, and Andreas Zeller. 2010. Learning from 6,000 projects: Lightweight cross-project anomaly detection. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA'10)*. ACM, 119–130.

Thiago S. Guzella and Walmir M. Caminhas. 2009. A review of machine-learning approaches to spam filtering. *Expert Syst. Appl.* 36, 7 (2009), 10206–10222.

Jiawei Han, Micheline Kamber, and Jian Pei. 2011. *Data Mining: Concepts and Techniques* (3rd ed.). Morgan Kaufmann.

Sean Heelan. 2011. Vulnerability detection systems: Think cyborg, not robot. *IEEE Secur. Privacy* 9, 3 (2011), 74–77.

Caitriona H. Heinl. 2014. Artificial (intelligent) agents and active cyber defence: Policy implications. In *Proceedings of the 6th International Conference on Cyber Conflict (CyCon'14)*. IEEE, 53–66.

Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen, and James Walden. 2012. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th International Workshop on Security Measurements and Metrics (MetriSec'12)*. ACM, 7–10.

IEEE Standards. 1990. *IEEE Standard Glossary of Software Engineering Terminology*. IEEE Std. 610.12-1990.

Ranjit Jhala and Rupak Majumdar. 2009. Software model checking. *ACM Comput. Surveys (CSUR'09)* 41, 4 (2009), 21.

Cem Kaner and Walter P. Bond. 2004. Software engineering metrics: What do they measure and how do we know? In *Proceedings of the 10th International Symposium on Software Metrics (METRICS'04)*. IEEE.

Taghi M. Khoshgoftaar, Edward B. Allen, John P. Hudepohl, and Stephen J. Aud. 1997. Application of neural networks to software quality modeling of a very large telecommunications system. *IEEE Trans. Neural Netw.* 8, 4 (1997), 902–909.

Ivan Victor Krsul. 1998. *Software Vulnerability Analysis*. Ph.D. Dissertation. Purdue University.

William Landi. 1992. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst. (LOPLAS)* 1, 4 (1992), 323–337.

Carl Landwehr. 2008. Cybersecurity and artificial intelligence: From fixing the plumbing to smart water. *IEEE Secur. Privacy* 6, 5 (2008), 3–4.

James R. Larus, Thomas Ball, Manuvir Das, Robert DeLine, Manuel Fahndrich, Jon Pincus, Sriram K. Rajamani, and Ramanathan Venkatapathy. 2004. Righting software. *IEEE Softw.* 21, 3 (2004), 92–100.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. 2015. Deep learning. *Nature* 521, 7553 (2015), 436–444.

Zhenmin Li and Yuanyuan Zhou. 2005. PR-miner: Automatically extracting implicit programming rules and detecting violations in large software code. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with the 13th ACM International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*. ACM, 306–315.

Benjamin Livshits and Thomas Zimmermann. 2005. DynaMine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference held Jointly with the 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'05)*. ACM, 296–305.

Fan Long and Martin Rinard. 2016. Automatic patch generation by learning correct code. In *Proceedings of the 43rd Symposium on Principles of Programming Languages (POPL'16)*. ACM, 298–312.

Ying Ma, Guangchun Luo, Xue Zeng, and Aiguo Chen. 2012. Transfer learning for cross-company software defect prediction. *Info. Softw. Technol.* 54, 3 (2012), 248–256.

Ruchika Malhotra. 2015. A systematic review of machine-learning techniques for software fault prediction. *Appl. Soft Comput.* 27 (2015), 504–518.

Iberia Medeiros, Nuno F. Neves, and Miguel Correia. 2014. Automatic detection and correction of web application vulnerabilities using data mining to predict false positives. In *Proceedings of the 23rd International Conference on World Wide Web (WWW'14)*. ACM, 63–74.

Andrew Meneely, Harshavardhan Srinivasan, Afiqah Musa, Alberto Rodriguez-Tejeda, Matthew Mokary, and Brian Spates. 2013. When a patch goes bad: Exploring the properties of vulnerability-contributing commits. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'13)*. IEEE, 65–74.

Andrew Meneely and Laurie Williams. 2010. Strengthening the empirical analysis of the relationship between linus' law and software security. In *Proceedings of the ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'10)*. ACM, 9.

Julie Moeyersoms, Enric Junque de Fortuny, Karel Dejaeger, Bart Baesens, and David Martens. 2015. Comprehensible software fault and effort prediction: A data-mining approach. *J. Syst. Softw.* 100 (2015), 80–90.

Benoit Morel. 2011. Artificial intelligence: A key to the future of cybersecurity. In *Proceedings of the 4th ACM workshop on Artificial Intelligence and Security (AISec'11)*. ACM, 93–98.

Patrick Morrison, Kim Herzig, Brendan Murphy, and Laurie Williams. 2015. Challenges with applying vulnerability prediction models. In *Proceedings of the Symposium and Bootcamp on the Science of Security (HotSoS'15)*. ACM, 4.

Sara Moshtari, Ashkan Sami, and Mahdi Azimi. 2013. Using complexity metrics to improve software security. *Comput. Fraud Secur.* 2013, 5 (2013), 8–17.

Jaechang Nam, Sinno Jialin Pan, and Sunghun Kim. 2013. Transfer defect learning. In *Proceedings of the International Conference on Software Engineering (ICSE'13)*. IEEE, 382–391.

Kartik Nayak, Daniel Marino, Petros Efstathopoulos, and Tudor Dumitras. 2014. Some vulnerabilities are different than others. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'14)*. Springer, 426–446.

Andy Ozment. 2007. Improving vulnerability discovery models. In *Proceedings of the 2007 ACM workshop on Quality of Protection (QoP'07)*. ACM, 6–11.

Sinno Jialin Pan and Qiang Yang. 2010. A survey on transfer learning. *IEEE Trans. Knowl. Data Eng.* 22, 10 (2010), 1345–1359.

Yulei Pang, Xiaozhen Xue, and Akbar Siami Namin. 2015. Predicting vulnerable software components through N-gram analysis and statistical feature selection. In *Proceedings of the 14th International Conference on Machine Learning and Applications (ICMLA'15)*. IEEE, 543–548.

Hao Peng, Lili Mou, Ge Li, Yuxuan Liu, Lu Zhang, and Zhi Jin. 2015. Building program vector representations for deep learning. In *Proceedings of the 8th International Conference on Knowledge Science, Engineering and Management (KSEM'15)*. Springer, 547–553.

Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. 2015. VccFinder: Finding potential vulnerabilities in open-source projects to assist code audits. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS'15)*. ACM, 426–437.

Thomas Reps. 2000. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst. (TOPLAS)* 22, 1 (2000), 162–186.

Stuart Russell and Peter Norvig. 2009. *Artificial Intelligence: A Modern Approach* (3rd ed.). Pearson.

Alireza Sadeghi, Naeem Esfahani, and Sam Malek. 2014. Mining the categorized software repositories to improve the analysis of security vulnerabilities. In *Proceedings of the 17th International Conference on Fundamental Approaches to Software Engineering (FASE'14) Part of the European Joint Conferences on Theory and Practice of Software (ETAPS'14)*. Springer, 155–169.

Arthur L. Samuel. 1959. Some studies in machine learning using the game of checkers. *IBM J. Res. Dev.* 44, 1 (1959), 210–229.

Riccardo Scandariato, James Walden, Aram Hovsepyan, and Wouter Joosen. 2014. Predicting vulnerable software components via text mining. *IEEE Trans. Softw. Eng.* 40, 10 (2014), 993–1006.

Hossain Shahriar and Mohammad Zulkernine. 2012. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Comput. Surveys (CSUR)* 44, 3 (2012), 11.

Lwin Khin Shar, Lionel C Briand, and Hee Beng Kuan Tan. 2015. Web application vulnerability prediction using hybrid program analysis and machine learning. *IEEE Trans. Depend. Secure Comput.* 12, 6 (2015), 688–707.

Lwin Khin Shar and Hee Beng Kuan Tan. 2012. Predicting common web application vulnerabilities from input validation and sanitization code patterns. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12)*. IEEE, 310–313.

Lwin Khin Shar and Hee Beng Kuan Tan. 2013. Predicting SQL injection and cross site scripting vulnerabilities through mining input sanitization patterns. *Info. Softw. Technol.* 55, 10 (2013), 1767–1780.

Lwin Khin Shar, Hee Beng Kuan Tan, and Lionel C. Briand. 2013. Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis. In *Proceedings of the 35th International Conference on Software Engineering (ICSE'13)*. IEEE, 642–651.

Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason Osborne. 2011. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Trans. Softw. Eng.* 37, 6 (2011), 772–787.

Yonghee Shin and Laurie Williams. 2011. An initial study on the use of execution complexity metrics as indicators of software vulnerabilities. In *Proceedings of the 7th International Workshop on Software Engineering for Secure Systems (SESS'11)*. ACM, 1–7.

Yonghee Shin and Laurie Williams. 2013. Can traditional fault prediction models be used for vulnerability prediction? *Empir. Softw. Eng.* 18, 1 (2013), 25–59.

Robin Sommer and Vern Paxson. 2010. Outside the closed world: On using machine learning for network intrusion detection. In *Proceedings of the 31st IEEE Symposium on Security and Privacy (SP'10)*. IEEE, 305–316.

Sherri Sparks, Shawn Embleton, Ryan Cunningham, and Cliff Zou. 2007. Automated vulnerability analysis: Leveraging control flow for evolutionary input crafting. In *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC'07)*. IEEE, 477–486.

Symantec Security Response. 2014. ShellShock: All you need to know about the Bash Bug vulnerability. Retrieved from http://www.symantec.com/connect/blogs/shellshock-all-you-need-know-about-bash-bug-vulnerability.

Yaming Tang, Fei Zhao, Yibiao Yang, Hongmin Lu, Yuming Zhou, and Baowen Xu. 2015. Predicting vulnerable components via text mining or software metrics? An effort-aware perspective. In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS'15)*. IEEE, 27–36.

Suresh Thummalapenta and Tao Xie. 2009. Alattin: Mining alternative patterns for detecting neglected conditions. In *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09)*. IEEE, 283–294.

Enn Tyugu. 2011. Artificial intelligence in cyber defense. In *Proceedings of the 3rd International Conference on Cyber Conflict (CyCon'11)*. IEEE, 1–11.

US-CERT. 2013. Oracle Java Contains Multiple Vulnerabilities. Retrieved from https://www.us-cert.gov/ncas/alerts/TA13-064A.

US-CERT. 2015. Adobe Flash and Microsoft Windows Vulnerabilities. Retrieved from https://www.us-cert.gov/ncas/alerts/TA15-195A.

Anneleen Van-Assche and Hendrik Blockeel. 2007. Seeing the forest through the trees: Learning a comprehensible model from an ensemble. In *Machine Learning: Proceedings of the 18th European Conference on Machine Learning (ECML'07) (Lecture Notes in Computer Science (LNCS))*. Springer, Berlin, 418–429.

James Walden, Jeffrey Stuckman, and Riccardo Scandariato. 2014. Predicting vulnerable components: Software metrics vs text mining. In *Proceedings of the 25th International Symposium on Software Reliability Engineering (ISSRE'14)*. IEEE, 23–33.

Andrzej Wasylkowski, Andreas Zeller, and Christian Lindig. 2007. Detecting object usage anomalies. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE'07)*. ACM, 35–44.

Dumidu Wijayasekara, Milos Manic, and Miles McQueen. 2014. Vulnerability identification and classification via text mining bug databases. In *Proceedings of the 40th Annual Conference of the IEEE Industrial Electronics Society (IECON'14)*. IEEE, 3612–3618.

Dumidu Wijayasekara, Milos Manic, Jason L. Wright, and Miles McQueen. 2012. Mining bug databases for unidentified software vulnerabilities. In *Proceedings of the 5th International Conference on Human System Interactions (HSI'12)*. IEEE, 89–96.

Tao Xie and Jian Pei. 2006. MAPO: Mining API usages from open source repositories. In *Proceedings of the International Workshop on Mining Software Repositories (MSR'06)*. ACM, 54–57.

Yichen Xie, Mayur Naik, Brian Hackett, and Alex Aiken. 2005. Soundness and its role in bug detection systems. In *Proceedings of the Workshop on the Evaluation of Software Defect Detection Tools (BUGS'05)*.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and discovering vulnerabilities with code property graphs. In *Proceedings of the 35th IEEE Symposium on Security and Privacy (SP'14)*. IEEE, 590–604.

Fabian Yamaguchi, Felix Lindner, and Konrad Rieck. 2011. Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. In *Proceedings of the 5th USENIX Workshop on Offensive Technologies*. USENIX Association.

Fabian Yamaguchi, Markus Lottmann, and Konrad Rieck. 2012. Generalized vulnerability extrapolation using abstract syntax trees. In *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC'12)*. ACM, 359–368.

Fabian Yamaguchi, Alwin Maier, Hugo Gascon, and Konrad Rieck. 2015. Automatic inference of search patterns for taint-style vulnerabilities. In *Proceedings of the 36th IEEE Symposium on Security and Privacy (SP'15)*. IEEE, 797–812.

Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. 2013. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 20th ACM SIGSAC Conference on Computer & Communications Security (CCS'13)*. ACM, 499–510.

Xinli Yang, David Lo, Xin Xia, Yun Zhang, and Jianling Sun. 2015. Deep learning for just-in-time defect prediction. In *Proceedings of the International Conference on Software Quality, Reliability and Security (QRS'15)*. IEEE, 17–26.

Awad Younis, Yashwant Malaiya, Charles Anderson, and Indrajit Ray. 2016. To fear or not to fear that is the question: Code characteristics of a vulnerable function with an existing exploit. In *Proceedings of the 6th ACM Conference on Data and Application Security and Privacy (CODASPY'16)*. ACM, 97–104.

Andreas Zeller, Thomas Zimmermann, and Christian Bird. 2011. Failure is a four-letter word: A parody in empirical research. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*. ACM.

Chenfeng Vincent Zhou, Christopher Leckie, and Shanika Karunasekera. 2010. A survey of coordinated attacks and collaborative intrusion detection. *Comput. Secur.* 29, 1 (2010), 124–140.

Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. 2010. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST'10)*. IEEE, 421–428.

Received August 2016; revised April 2017; accepted May 2017