ABSTRACT

Title of document: Continuous, Effort-Aware Prediction of Software Security Defects

Jeffrey Stuckman, Doctor of Philosophy, 2015

Directed by: Professor James Purtilo
Department of Computer Science

Software security defects are coding flaws which allow for a system's security to be compromised. Due to the potential severity of these defects, it is important to discover them quickly; therefore, they are a good focus for software quality improvement efforts such as code inspection. Our research focuses on vulnerability prediction models, which use machine learning to identify code that has an elevated likelihood of containing these defects. In particular, we study continuous prediction models, which repeatedly search for vulnerable code over a period of time, rather than being used at just one particular moment. To empirically evaluate the prediction methodologies that we define, we collected a fine-grained dataset of vulnerabilities in PHP applications. We then defined and implemented a method for defining families of features, or metrics, which characterize both the change in code over time and the state of the code at a given moment, enabling a systematic and fair comparison of continuous and traditional prediction models. We also introduce a methodology for effort-sensitive learning, which optimizes to minimize the expected cost of inspecting the code that is ultimately flagged by the model.

Our results show that the security defects in our dataset were long-lived, with a median lifetime of 871 days. Continuous prediction more readily discriminated

vulnerable from non-vulnerable code than traditional static prediction did, and prediction was more efficient when changes were broken apart by file than when they were aggregated together. However, high code churn negated some of the efficiency gains of continuous predictors in simulations, and the optimal prediction method in a given scenario depended on making a tradeoff between speed of detection and cost savings. As an additional contribution, we have released the fine-grained defect dataset – the first of its kind – to the public, in order to encourage future work in this field.

CONTINUOUS, EFFORT-AWARE PREDICTION OF SOFTWARE SECURITY DEFECTS

by

Jeffrey Charles Stuckman

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2015

Advisory Committee:
Professor James Purtilo, Chair
Professor Rance Cleaveland
Professor Michel Cukier
Professor Denny Gulick
Professor Marvin Zelkowitz

Acknowledgments

First and foremost, I thank my family and friends for supporting me throughout graduate school and the dissertation process. Your continued presence and support throughout the years, especially during the final months, was invaluable for me, and I thank you for this.

I thank James Purtilo, my advisor, for giving me the freedom and resources to pursue my interests throughout graduate school. In addition, I thank my collaborators, Riccardo Scandariato and James Walden, for further refining my work and inspiring new research directions. I also thank all of the graduate and undergraduate students who assisted with this work in large and small ways, including Moshe Katz, Gary Nilson, Kent Wills, James Wills, Connor Fox, Elizabeth Halper, Daniel Laurence, and many others who are too numerous to be named.

I also thank the United States Office of Naval Research, which partially supported this research under contract N000141210147.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AST	Abstract Syntax Tree
AUEC	Area Under the Effort Curve
AUCEC	Area Under the Cost-Effectiveness Curve
CSRF	Cross-Site Request Forgery
CVE	Common Vulnerabilities and Exposures
FP	False Positives
I	Inspection
IR	Inspection Ratio
R	Recall
TP	True Positives
XSS	Cross-Site Scripting

Chapter 1

Introduction

Software security vulnerabilities are software defects which allow for malicious users to compromise the security of a system. Successful attacks often entail the exploitation of one or more of these vulnerabilities; for this reason, finding vulnerabilities in systems, and preventing them from being created in the first place, is of considerable interest. High-profile vulnerabilities in commonly used software have the potential to impact the security of countless systems; for example, the Heartbleed vulnerability in OpenSSL affected a large proportion of Linux systems connected to the internet [172]. Although detecting and preventing attacks (i.e. exploits against vulnerabilities) could serve as an alternative method of preventing these security breaches, we exclusively focus on preventing vulnerabilities in this work, as these vulnerabilities are the ultimate cause of the problem.

Software security vulnerabilities are a subset of software defects, and many of the techniques designed for analyzing and preventing defects can also be applied to vulnerabilities. Vulnerabilities are distinct from weaknesses, which are common design or implementation issues that can lead to vulnerabilities [95]. For example, in the CWE (Common Weakness Enumeration), which is a standard taxonomy of weaknesses, "failure to handle incomplete element" is an implementation issue indicating a failure to handle incomplete input properly. Then, in the CVE (Common Vulnerabilities and Exposures) (a namespace for identifying specific vulnerabilities in specific products) one entry related to this weakness describes a denial-of-service vulnerability where a specific software project hangs when it receives an incomplete HTTP message. Because vulnerabilities ultimately originate from programming errors, much guidance has been produced for developers on how to avoid making the

most common of these errors [6]. Despite this, the flood of new software vulnerabilities continues. Such vulnerabilities often persist undiscovered for long periods of time [115], opening a large window of opportunity for unseen attackers to take control of systems. It is clear that a better way to prevent vulnerabilities, and discover existing vulnerabilities more quickly, is needed.

One approach to remediating vulnerabilities is to implement protective measures on the level of the platform or operating system, such as Control Flow Integrity [1] or Address-Space Layout Randomization [132], to disrupt patterns of activity that are almost always indicative of a vulnerability being exploited. However, only a few classes of vulnerabilities can be remediated in such ways (for instance, none of the scripting language vulnerabilities that we study in this work would benefit from this). Other, more ambitious, schemes, require the developer to adopt a new, more secure platform that makes vulnerabilities more difficult to create [160]. Other methods modify the semantics of an existing language or platform in order to subvert common exploit-related behaviors while attempting to render normal behaviors unaffected. While such approaches are highly effective in some circumstances [149], there will inevitably be cases where enhanced security measures disrupt legitimate behavior in hard-to-diagnose ways, leading developers to mistrust and shy away from these approaches.

In this work, we take a different approach for preventing vulnerabilities. Rather than introducing additional countermeasures to prevent exploits, we examine measures that can be taken during the software engineering process, such as code reviews, to prevent the vulnerabilities from making it into production in the first place. More specifically, we study *vulnerability prediction models*, or machine learning models which can point developers and testers to areas of the code where vulnerabilities are likely to exist. Although some work in this field [66] takes a qualitative approach to finding types of code where vulnerabilities are common, our approach is *quan-*

titative, in the sense that the relationships between the presence of vulnerabilities and some numerical attributes of the source code are analyzed.

1.1 Vulnerability prediction models and software quality

Three major steps must be taken in order to build such a vulnerability predictive model: (1) Attributes (sometimes known as features or metrics) must be computed for each file or each code change; (2) A model must be trained by looking for associations between these attributes and the locations of *known* vulnerabilities in the past; (3) The resulting model will guide a user of the model toward similar parts of the codebase where *unknown* vulnerabilities are likely to currently exist, or are likely to exist in the future. The traditional approach for computing these attributes has been with software metrics (and this is the approach that we take in this work).

Historically, software metrics were intended to serve as a standardized approach for quantifying, or indicating, software quality [21]. One common way of validating these metrics (for their soundness as quality indicators) was to associate them with the presence of software defects [89]. This was done by demonstrating that a software metric *is associated with* (is directly correlated with) or *predicts* (via a model) the presence of defects (or other *quality factors*), with the primary goal of establishing the metric's validity in general.

Later on, in the field of *statistical defect prediction*, the focus shifted from the validation of individual metrics toward the practical utility of identifying defect-prone areas of code. Practitioners of defect prediction note that the "best" metrics for predicting defects vary from project to project [92, 159]. However, identifying good candidates beforehand is not necessary, as machine learning algorithms are capable of building "black-box" models which simply treat software metrics (or other features, such as tokens in the source code [165]) as raw data for the prediction

algorithms, estimating which portions of the source code are more likely to contain undiscovered defects.

Despite the differing focus of these two problems (quality metric validation and statistical defect prediction), both share the common goal of statistically linking some characteristic of the source code to the presence of defects (or vulnerabilities, which can be treated as a subset of defects in general). In other words, one or more independent variables (which vary from study to study) are associated with the occurrence of software defects, the dependent variable. (In this work, note that we study software metrics from both perspectives – as a means for predicting vulnerabilities and as an object of study in their own right.)

In the statistical defect prediction field, it is generally expected that prediction will be used for directing defect-detection activities, such as code inspection, to areas of the code identified by the predictive model. Although such models are rarely "explainable" (it is difficult to work backward from the model and explain exactly how the metrics and vulnerabilities are related), they have nevertheless proven useful in practice. One study found that machine learning models performed better than expert human judgment when identifying portions of a codebase most likely to contain undiscovered defects [154]. Defect prediction tools have seen practical use in industrial, government, and commercial organizations [93], such as Microsoft [29]. Although the predictive models are almost always based on software metrics of some kind, various transformations of metrics, many of which attempt to incorporate historical data to enhance predictions, have been extensively studied [38].

Vulnerability prediction is a specific kind of statistical defect prediction which focuses only on vulnerabilities. Vulnerability prediction experiments are usually set up to determine which files have an elevated likelihood of containing vulnerabilities at one specific moment in time [104]. Software metrics have been used to predict vulnerabilities, as well as non-metric information such as code churn or information

on developers [139]. The rationale for vulnerability prediction – that quality improvement resources should be focused on "risky" parts of the codebase – is similar to that of standard defect prediction [48]. Although most vulnerability prediction studies are done on a file level (looking for whole files that are more likely to be vulnerable than others), some studies have been done on the level of commits [90] or changesets [23], although these change-level studies were exploratory in nature, and predictive models were never actually built.

Predictive models are not the only technique that has been proposed for finding latent vulnerabilities in a codebase. Other vulnerability discovery techniques exist, and prediction, which used properly, can complement these techniques. Static analysis tools such as [67, 168, 173] look for situations in code that are indicative of specific, identifiable vulnerabilities. However, these tools are limited (in the sense that they can only find the specific vulnerability patterns that they were developed to find) and suffer from high false positive rates [4, 5, 10, 162]. Fuzz testing [52], a dynamic method, involves using tools to find specific vulnerabilities by searching for inputs that induce crashes. However, as with static analysis, fuzz testing tools can only find certain types of vulnerabilities that they were designed to find. Penetration testing tools, another approach to discovering vulnerabilities, automatically explore applications from the outside. However, these tools tend to find only a small proportion of latent vulnerabilities [4, 5, 10, 14, 45]. Although these other techniques can serve as a useful adjunct to traditional quality improvement methods such as manual testing, because they can find vulnerabilities that traditional methods would have missed [162], the limited range of vulnerabilities that they can find (in relation to manual testing) and high false-positive and false-negative rates mean that they are not adequate on their own.

Predictive models, by their very nature, make no attempt to find specific instances of specific vulnerabilities or weaknesses in a codebase. Rather, they identify

the portions of a codebase where such vulnerabilities are most likely to appear. Because the characteristics of these portions of the codebase are learned during the training process (rather than coded into the tool from the start), vulnerability prediction models will seek to find patterns similar to those encountered during the model training process, allowing for future use of a tool to direct the user toward types of vulnerabilities that weren't even known at the time the tool was built. Also note that predictive models cannot actually verify the presence or absence of a vulnerability; for example, if a model correctly indicates that a certain portion of the code is likely to contain a vulnerability, it will probably continue to indicate this even after the vulnerability is fixed, because fixing the vulnerability will not significantly change the attributes that triggered the prediction in the first place. However, the unique way that vulnerability prediction works makes it possible to synergistically *combine* vulnerability prediction models with other quality improvement techniques, such as code inspection, traditional software testing, or vulnerability discovery tools such as the ones discussed previously. For example, one study [125] found that vulnerability discovery tools and vulnerability predictive models conferred similar benefits when used in isolation, but in some settings, the use of both techniques together was more effective than using either alone. In this related study, predictive models were used to prioritize static analysis alerts by directing the user to preferentially respond to the alerts which occurred in portions of the codebase which were flagged by the predictive model. It is also possible to use static analysis tools in conjunction with predictive models in a different way, by feeding the alerts into the model as another kind of attribute, allowing the model to take the alerts into consideration when deciding which parts of the code are most likely to contain true vulnerabilities. (The justification for this can be found in a previous study [163] linking statically found vulnerabilities to confirmed vulnerabilities in a codebase.)

1.2 Current limitations of vulnerability prediction models

Despite the fact that vulnerability prediction is an active research area, all current work in vulnerability prediction is focused on predicting at the *file level*, meaning that characteristics of files are used to predict the current, or future presence of vulnerabilities in the same files. (This includes our previously published predecessor to this work [165].) Although, as mentioned previously, several studies do examine vulnerability-containing commits or changesets, they do not use the same kinds of metrics that have successfully been used for file-level prediction, and they are not evaluated in a way that would allow for file-level and change-level studies to be compared. In fact, as we explore in the more thorough literature review in Chapter 3, this same limitation is present in state-of-the-art defect prediction studies as well.

Limiting prediction to the file level has several drawbacks:

- *Misalignment with the goal of maximizing the cost-benefit of a quality assurance task*: Many studies on improving quality assurance activities start with the premise that an organization has a *fixed total effort budget* which can be spent on the activity, and that the activity must be optimized to derive the maximum benefit from this budget. For example, one study [84] analyzed the tradeoff between the number of people assigned to a software testing task with a fixed man-hour budget and the amount of time that it would take to complete the task. Another study [130] found that static analysis provided more of a benefit than penetration testing when examining the effort associated with each technique. One similar study [11] found that static analysis tools resulted in cost savings over using no technique at all, due to vulnerabilities being found earlier, and hence being easier to remediate. Finally, it was found [10] that automated penetration testing was more cost-effective

at finding vulnerabilities than traditional manual testing. However, file-level vulnerability prediction techniques are traditionally evaluated based on their ability to discriminate vulnerable and non-vulnerable files, and this does not reflect the amount of effort required to find the vulnerabilities in a flagged file. (For example, longer files may take more effort to inspect.)

- *Failure to distinguish significant effects from trivial ones*: As we previously stated, traditional file-level prediction methodologies do not consider that inspecting different files requires varying amounts of effort. This is more problematic than it seems, because ignoring effort can also lead to the recommendation of predictive models which confer no real-world benefit at all. For example, one set of experiments [87] found that a random (trivial) model could appear to be beneficial if the wrong evaluation criteria were used, and that simply flagging the largest files in a product for inspection appears to work surprisingly well, as larger files have more code and more opportunities to contain vulnerabilities. In other words, evaluating predictive models by counting vulnerable and non-vulnerable files fails to establish the credibility of these models as useful tools for quality improvement.

- *Failure to fit into typical software engineering processes*: File-level prediction models are evaluated with the presumption that the model is run at just one point in time and is used to flag vulnerabilities that may have persisted for many months or years. However, it may be preferable to instead utilize these models *continuously*, picking out individual changes (or portions of changes) to inspect for *new* vulnerabilities. This kind of continuous prediction has been explored to an extent for general defect prediction; however, these studies did not use the same kinds of features (metrics) that are used in file-level prediction, and the comparative benefits of file-level and change-level prediction

have never been examined. Given the general goal of maximizing benefit while minimizing effort, it would be desirable to compare the benefits and effort associated with file-level and change-level prediction and choose a technique accordingly.

- *Attributes must be defined at the file level*: If prediction is only done at the file level, then all attributes (or features) used for prediction must also be defined at the level of individual files. Furthermore, the statistical relationships between the attributes and the vulnerabilities must be such that the trend in the attribute in one file contributes to the presence of vulnerabilities in the same file (as opposed to a different file). It is possible to avoid both of these limitations by performing non-file-level prediction which incorporates features not defined at the file level. We discuss this issue in more detail in Chapter 7 and in our previously published work [83].

Overcoming these deficiencies which stem from an over-reliance on file-level prediction motivates the major theme of this thesis: predicting vulnerabilities at multiple levels, such as at the file level, the change level, and the release level, with each level having its own associated experimental setup. To empirically study how each of these prediction setups works in practice, we experimentally evaluate each technique by using historical vulnerability data extracted from PHP web applications. This facilitates a series of *retrospective* experiments which estimate how helpful these prediction techniques would have been in real-world scenarios where vulnerabilities were present.

1.3 Definitions and assumptions

In this document, we utilize the following terminology related to defects and vulnerabilities:

Defect: We define a defect to be an error made during the programming or specification process that could potentially lead to unintended or erroneous operation in some runtime scenario. Note that a specification error may lead to behavior which was intended by the developers but is unsuitable or erroneous in practice. For example, a specification may describe a password system that has no password length requirements; this would lead to an insecure system, therefore, this behavior would be considered a defect, despite the fact that the specification was implemented faithfully. Also, a *defect* entails the simple presence of a programming or specification error in a software product – in other words, an instance of a defect is not necessarily tied to a specific software failure at a specific time.

Vulnerability: We define vulnerabilities to be the subset of software defects that can allow for an attacker to compromise the security of a system; for example, an integrity, reliability, or confidentiality property. Vulnerabilities are a subset of defects. As with defects, vulnerabilities are not necessarily tied to specific security breaches which occurred at specific times. Furthermore, although the existence of an exploit (a series of inputs or interactions that allows for the system's security to be compromised) is necessary for a vulnerability to exist (because otherwise the security breach in question wouldn't be possible), the specific inputs or interactions involved in such an exploit need not be formally defined.

Security defect: Synonym for *vulnerability*. Although the prediction techniques introduced in this work were exclusively evaluated on vulnerabilities, they would also be applicable to defects in general.

In addition, the following assumptions regarding vulnerabilities are made throughout this work:

Vulnerabilities come from the CVE/NVD: In each of the experiments in this work, we utilize the National Vulnerability Database [110] as the sole source of vulnerabilities which we analyze. The implications of this are that (1) we did not

attempt to independently discover any latent vulnerabilities in the products that we analyzed; and (2) we deferred to the judgment of the vendors (who issued security advisories, which act as the primary sources of vulnerabilities for the studied products in the NVD) when it comes to determining which defects can be considered vulnerabilities and which defects cannot. The practical implications of this assumption are discussed more detail in Chapter 6.

Latent vulnerabilities and known vulnerabilities are similar: It is very likely that unknown (latent) vulnerabilities exist in the products that we studied. We assume that the character of these vulnerabilities doesn't significantly differ from the character of the vulnerabilities in our dataset. If the known vulnerabilities are different from the unknown ones, the prediction techniques which we evaluated may not work in the same way on the true set of all vulnerabilities.

All vulnerabilities are equal: We consider all vulnerabilities to be of equal severity, and we do not credit some vulnerabilities as being more valuable than others when comparing the effectiveness of various prediction techniques. Note that CVSS scores, one common method for characterizing the severity of vulnerabilities, are not associated with the risk that a vulnerability will be exploited in practice [3] – in other words, ranking vulnerabilities by severity is not as straightforward as it may seem.

1.4 Document organization

The rest of this document is organized as follows:

In **Chapter 2**, we explore how the same *software metrics* can be computed at multiple levels. Historically, file-level predictive models have used a far richer set of features than change-level models, because many more static code metrics than change metrics have been defined. In this chapter, we introduce the concept of *metric families*, which are sets of static code metrics and change metrics which

share a common functional definition. We then introduce a *constructive* method for building these sets, and algorithms for computing these metrics, by simply supplying the function which characterizes the family. Next, we re-implement a number of commonly used metrics using this system. Finally, we prove that a subset of metrics in each metric family will satisfy a set of axioms (or properties) which are based on the properties of mathematical pseudometric spaces. The metric family work in this chapter unifies and extends the concepts of code delta and code churn which have previously been used in software analysis [56], yielding change metrics which work in the same way as traditional code churn measurements while simultaneously characterizing concepts such as complexity or coupling.

In **Chapter 3**, we move on to the problem of how these metrics can be made into predictive features and fused with vulnerability data in order to build an "example set", or a data structure that can be used to train machine learning algorithms. Because we have a special interest in diverse experimental setups (in order to predict at multiple levels), we pay special attention to the varying ways that previous defect prediction studies have set up their experiments. To this end, we first perform an extensive literature review focusing on defect prediction methodologies, resulting in a taxonomy of methods used to adapt and transform features for predictive problems. We then choose a subset of these feature adaptations and transformations that may prove useful for vulnerability prediction. Finally, we formally define a set of operators to build example sets with these adaptations and transformations, defining how the experiments in later chapters will be performed.

In **Chapter 4**, we then explore the machine learning algorithms that build predictive models by finding the patterns in the example sets used to train them. Although most aspects of machine learning are addressed by off-the-shelf software packages, we explore two specific problems unique to our experimental needs: building a model that is calibrated toward expending a predetermined amount of *effort*

(making the burden of quality assurance activities more predictable), and the need to optimize the problem such that a maximal number of vulnerabilities is discovered while expending a minimal amount of effort (rather than optimizing some other error statistic, as off-the-shelf algorithms would without adaptation). To this end, we introduce a novel meta-learning algorithm that can "wrap" most off-the-shelf machine learning algorithms to accomplish both of these goals, taking an approach similar to several meta-learning algorithms in other domains [36, 174].

In **Chapter 5**, we turn our attention to the *datasets* and *tools* that will be required to perform a study that predicts vulnerabilities at multiple levels. First, we build a vulnerability dataset consisting of two PHP web applications and data on a set of known vulnerabilities in each one. Having this information on known vulnerabilities enables a *retrospective* evaluation of vulnerability prediction, testing how well the predictive techniques would have caught real-life vulnerabilities which had occurred in the past. Although a number of similar software engineering research datasets have been made available to the public [94], our study required fine-grained data on how each historical vulnerability was introduced, repaired, and evolved over the time when it was present in the codebase. No existing public data contained this granularity of detail for vulnerabilities (or even for defects in general). For this reason, we collected a vulnerability dataset (through a largely manual process) and made it available to the research community through our previously published work [165]. We then turn our attention to building *tools* which are capable of developing and computing the code metrics in accordance with the metric definition system and algorithms described in Chapter 2. We introduce a novel metric development and visualization tool which allows a wide variety of metrics to be developed and visualized in the context of evolving codebases. This tool was then used to develop and compute the metrics used in the prediction experiments that follow.

Finally, in **Chapter 6**, we discuss the results of our prediction experiments,

utilizing the metrics, predictive features, datasets, and prediction algorithms which were introduced previously, predicting vulnerabilities at three levels: the file level (which files are most likely to contain vulnerabilities at a given point in time), the change level (which individual file changes are more likely to introduce vulnerabilities than others), and the release level (which versions of a product have an elevated likelihood of introducing vulnerabilities). These results show that, in general, the machine learning algorithms and meta-learning algorithms have a large impact on the performance of the resulting predictor, while, in contrast, the complex transformations and adaptations which were used in previous studies (and formalized in Chapter 3) rarely perform better than a simple set of metrics in this context. Our use of effort-based evaluation criteria demonstrates that the discriminative power of the algorithms is best when the algorithms measure changes (as opposed to static files); however, the large amount of code churn (or code that would have been deleted for unrelated reasons before any vulnerability was found in it) diminishes or negates the apparent advantage of change-based prediction. Finally, we perform *simulations* which depict, over time, the effect of allocating quality assurance effort to static file-based inspection, change inspection, or a combination of the two. The results of these simulations indicate that the "best" prediction strategy or setup will depend on which goal is to be accomplished: (1) quickly finding vulnerabilities during the development process before they are found by the general public; or (2) just maximizing the rate at which vulnerabilities are found by any means, whether they are found by the general public or during a quality assurance task.

Chapter 2

Defining families of code and change metrics

In past research, software metrics have been used for various purposes, such as characterizing various aspects of source code, predicting the occurrence of software defects and vulnerabilities, and building machine learning features to classify source code artifacts. Typical defect prediction studies use one of the following classes of metrics:

- **Code metrics**: Metrics applied to a single source code file or module. Examples include the Halstead software science metrics, the McCabe complexity metrics, and various object-oriented metric suites. *Code metrics* as defined in this chapter are equivalent to *static code attributes* as defined in [92].
- **Change metrics**: As outlined in [100], some metrics characterize differences across multiple versions of a module or product, such as *"change history of source files, changes in the team structure, testing effort, or technology and other human factors to software defects"* [100]. We refer to these metrics as *change metrics*. Code churn (summing the number of added, deleted, and modified lines between two versions) is the most common metric in this category.

In the context of defect prediction, code metrics and change metrics have historically been seen as separate entities. Studies tend to utilize code metrics for file-level prediction and change metrics for commit-level prediction, and there is little overlap between between the measurement goals of each of these sets of metrics. For example, cyclomatic complexity [86] is an extremely common code metric for defect prediction, but defect prediction studies rarely attempt to characterize the change

in cyclomatic complexity with change metrics. Similarly, two somewhat common change metrics for defect prediction are code churn and the number of *else*-parts added or removed [51]; however, neither of these metrics have a precise analog in the set of static code metrics that are used for this purpose.

Because the goal of this study is to fairly compare multiple level of prediction (such as file-level and change-level prediction), this poses two problems:

- Because a much broader variety of code metrics are available for defect prediction than change metrics, the effectiveness of change-based predictors may be hindered due to the lower-dimensionality feature vector that results.

- Because the available code and change metrics are so dissimilar, it is difficult to fairly compare the performance of static (file-based) predictors and change-based predictors, because any performance difference encountered may be an artifact of the features used.

In this work, we introduce a method for defining and constructing *families* of code metrics and change metrics, which are sets containing code and change metrics that characterize the same underlying concept. The members of these sets, which we will refer to as *metric family members*, measure the concept at a static point in time or characterize various aspects of how the concept changes over time. The specific member that's chosen for a vulnerability prediction study will depend on what is needed for the particular experiment in which the metrics are used. For example, one metric family could be centered on the concept of methods in object-oriented programs. Members of this metric family could include:

- A code metric counting the number of methods defined in a file at a particular revision

- A change metric measuring the number of methods that were added to a file over a particular revision

- A change metric measuring the number of methods in a file that were modified over a particular revision

Metric families unify existing code and change metrics using a formalism (labeling functions) that emphasizes the specific aspect of software that is to be measured, rather than the implementation details of how it is measured. Through this system of metric families, we replicate many common, existing code metrics while also defining new kinds of change metrics which would be difficult to construct otherwise; for example, we can build a change metric that shares some characteristics of both code churn metrics and source code coupling metrics.

In Section 2.1, we introduce and formally define metric families. In Section 2.2, we define how to construct several metric family members, such as code metrics, commit-by-commit change metrics, and release-by-release change metrics. In Section 2.3, we give several examples of metric families that were defined in this work, and we discuss the strategies that were employed when defining them. In Section 2.4, we turn our attention to the theoretical properties of the metric family members, showing that the change metrics satisfy the mathematical properties of a distance pseudometric in some circumstances. We also theoretically examine an earlier attempt to unify code and change metrics (code delta) and demonstrate that metric families have certain desirable properties which this construct does not. Finally, in Section 2.5, we discuss implementation details of analyzing source code histories, which has implications for the measurements that result from the history analysis.

2.1 Defining families of metrics by labeling abstract syntax trees

We introduce a formalism for defining families of metrics which is based on *labeling* abstract syntax trees (ASTs) in order to capture the phenomenon in the source code which is to be measured. For example, a family of complexity metrics

could be defined by placing labels on loops, decision statements, and the other elements found in source code which contribute to the overall complexity of the program.

The method in which this labeling should be performed for a particular metric family is defined by a *labeling function*, which inputs an abstract syntax tree and outputs a labeled abstract syntax tree. This functional representation of metric families is intended to be expressive enough for a wide variety of metrics, such as coupling metrics, complexity metrics, and other classes of metrics not currently anticipated. In addition, our labeling functions operate on abstract syntax trees (as opposed to simpler representations, such as sequences of tokens) to accommodate metrics that must be sensitive to the context in which tokens appear. For example, a nesting complexity metric could be sensitive to the depth that control structures were nested, while a cohesion or coupling metric may be concerned with the function or class definition that encloses an instance of an external method call.

Although as a general rule, labeling functions only use local information (from the same abstract syntax tree) when labeling, labeling functions may also consult a lookup table of function definitions and function calls, which catalogs the origin and the destination (function name and file name) of all function calls in all files in the product. This is so metrics that deal with references between files (such as coupling and cohesion metrics) can be computed. These lookup tables are constructed by the metric computation engine, at the time that metrics are computed (or potentially beforehand), by tracing function or method calls to their corresponding definitions, in an implementation-dependent manner. Our implementation of the construction of these tables is described in more detail in Chapter 5.

Labeling functions may also consult the name of the file currently being labeled, in order to determine which entries in the lookup table refer to the current file (as opposed to a different file). For example, function calls described in the lookup

table which originate from the same file can be handled differently from function calls originating elsewhere.

Definition 2.1 (Labeling function) *A labeling function is defined as follows:*

$$l\left(T, \text{lookup}\left(\mathbf{R}\right), f\right) \subseteq T \times \mathbf{L_T}$$

where T is the abstract syntax tree to be labeled (or the set of nodes in this tree), $\text{lookup}\left(\mathbf{R}\right)$ *is the lookup table for release* $\mathbf{R}$*, f is the name of the current file to cross-reference against the lookup table, and* $\mathbf{L}$ *is the range of the labeling function when applied to tree T.*

Intuitively, a labeling function takes an AST and returns a set of tuples associating members of a vocabulary of labels with nodes that were in the AST. Although this functional representation gives wide latitude to perform a variety of computations as part of a metric definition, these functions do not directly compute numerical measurements. This is because various members of the family of metrics may be constructed in ways that result in very different measurements. For example, a change metric might only take into account modifications made to the code (relative to a previous version of the artifact), while a static code metric measuring an entire file may take into account anything contained within the file. Hence, rather than defining just one metric, a labeling function defines an entire *family* of metrics, the members of which include multiple varieties of code metrics (computed on only one revision of a file) and change metrics (computed on pairs of revisions).

2.2 Building and computing metric family members

In this section, we describe in more detail how software products and source code are represented, such that they can be labeled by metric families. Next, we

will define how the labeling functions are used in order to compute numerical code and change measurements.

2.2.1 Source code definition model

We define a model for software products and source code artifacts which is based on each file being parsed into an AST. This model can represent code written in any ordinary programming language, assuming that source code artifacts are broken up into files or file-equivalent units. The lookup table of function definitions and calls described in Section 2.1 presumes that source files contain functions, and that the interrelationships between these functions are of interest. With minor modifications, this model could be adapted to other C-like languages and scripting languages (including C, C++, and Java) where certain nodes in a source file (such as function or method references) refer to entities defined in other source files (such as class, function, or method definitions).

The following entities are defined in this model of source code artifacts:

- **Product**: A software product is composed of a sequence of *revisions*, each revision itself being composed of a set of source code *files*. As the product evolves from revision to revision, files are added, removed, and modified, causing the contents of each revision to be different. Relationships between files are not part of the model's definition of a product, although files may refer to each other (typically by including the file name of an import or reference).

- **File:** A file is represented as an AST with an associated pathname (including the file name and the folder where the file exists). The exact definition and format of an AST node is language and implementation-dependent, but the node will typically have properties indicating the type of the node, the method names, variable names, or constants associated with the node, and the line

number of the source file from where the node was extracted.

- **Revisions:** A revision (also known as a commit) consists of a set of *files* for a particular *product* at a particular time. In many source control systems, a revision is committed once a developer has completed a set of modifications, additions, and deletions to the files in a *product*. Some source control systems may separate the concepts of changesets and revisions, allowing for changesets to be applied to a product once they are tested and approved. Regardless of the source control system used, our model assumes that revisions form a tree, such that the parent of the node is the previous revision of a revision, the children of a node are successor revisions of a revision, and the timestamp of each revision is greater than the timestamp of its parent. Aside from a set of *files*, a revision includes metadata describing how and when the revision was made. Along with the aforementioned timestamp, this metadata includes a commit comment (a description of the changes made as part of this revision), any available information on the purpose of the revision (such as if the revision fixed a bug, added a feature, or refactored code), and information on files that were renamed as part of the revision (allowing for file additions and deletions to be distinguished from renames).

- **Releases:** This source code definition model allows for a product to have regular releases. A release is represented as a *revision* with an indicator that the revision was released, typically indicating that the revision was used operationally. Because revisions are related with a tree with ascending timestamps for paths originating at the root of the tree, releases can similarly be organized into such a tree. (If the additional constraint is imposed the the history of revisions should be linear, then the lineage of releases will also be linear). All revisions (or commits) between two release revisions are considered to be

development revisions (or commits) that represent the work done to bring the software from one release to another. (When computing metric variants at the release level, the release revision and all of its development revisions are grouped, such that the entire sequence of revisions up to the final release are grouped into exactly one release. When performing such grouping, the change between two releases is determined by comparing the revision of the release and the revision of the release's parent release.)

2.2.2 Comparing revisions

Static code metrics can be computed by examining a single revision in isolation – comparing two revisions, or determining what has changed between the two revisions, is not necessary. However, most change metrics do require the differences between two revisions to be determined. In this section, we introduce the notation that will be used to measure such differences, and we discuss how the manner in which these differences are determined affects the properties of the change metrics that result.

The process of comparing two revisions of a product entails defining a *mapping* between the two revisions, such that files or AST nodes that persisted unmodified from one revision to another are mapped between the two revisions. Although our mapping-based definition of revision comparisons does not explicitly deal with edit scripts (sequences of operations transforming one revision into another), mappings and edit scripts are closely related when comparing two ASTs [18]. Files (or nodes) that do not participate in a mapping are the ones that are added or deleted in order to move from one revision to another.

Although revision comparisons will most often involve two revisions that are adjacent in the product's revision tree, nothing here precludes comparison of non-adjacent revisions. (In the proofs of Section 2.4, we will consider the case where

non-adjacent revisions are compared more explicitly.)

Comparing two revisions of a product is a two-step process. First, files are mapped between the two revisions, such that added, deleted, and renamed files are identified, and that renamed files and files that persists without being renamed are reflected in this mapping, which defines which files in one revision have counterparts in the other revision. Unmapped files are considered to have been created or deleted between the two revisions. Next, between each pair of mapped files, AST nodes are mapped between the two files, such that added and deleted nodes are identified, and nodes that persist between the two revisions are mapped as being equivalent.

In addition, there may also be a *value mappings* between the labeling function vocabularies associated with two revisions of a file. A value mapping compensates for properties of AST nodes that may spuriously change from revision to revision, even though the source code has not meaningfully changed. The most straightforward example of this is when mapping line numbers from one file to the next. Consider the case where a line break is inserted in the center of a line when a revision is made to a file. Above the line break insertion, the line number of each line will remain the same; however, below the point where the line break is inserted, the line numbers will all increment.

A change metric that uses the line numbers of AST nodes to compute how many distinct lines contain changes may report that many lines have changed in this situation, because each line has lost its old line number and gotten a new one. To resolve this, a value mapping can be established between the line numbers of the two revisions, allowing for the measurement algorithms to correctly identify lines that have not changed or been split. In the example above, the value mapping would map the line numbers of all unchanged lines to each other. One of the split lines in the newer revision would be mapped to the line number of the original line that was split, while the other split line would be mapped to a new line number. In other

words, the value mapping captures the fact that some larger line numbers in the newer revision are the "same" as smaller line numbers in the older one.

Aside from line numbers, we use value mappings for other properties, such as function names or variables, to ensure that a function or variable is not considered to be completely new when it was just renamed between revisions. Algorithms for establishing value mappings will be discussed in more detail in Section 2.5.3.

When implementing a revision differencing algorithm, all of these steps are somewhat interrelated, as file renames may be detected by finding the files that appear the most similar per a file differencing algorithm. However, these implementation details are not significant in our theoretical discussion of revision comparisons. In the next sections, we formalize the file and AST node mapping steps and the properties that must apply to a valid revision differencing algorithm when performing each of these steps.

We provide an illustrated example to demonstrate how two revisions of a file are parsed into ASTs and compared with a node mapping algorithm. Figure 2.1 depicts two revisions of a simple program, and Figure 2.2 depicts the ASTs for each revision. The dotted lines connecting the nodes of the two ASTs represent the nodes which were mapped by a node mapping (e.g. tree differencing) algorithm. All mapped pairs of nodes are equivalent – in other words, both nodes must be of the same type and contain the same value. In the following sections, we describe in more detail the process and requirements associated with comparing two releases of a product.

2.2.2.1 Mapping files across revisions

When comparing two revisions, some files may be present in both revisions (admitting the possibility that some may be modified from one to the next), while other files may be present in one revision but not the other. Refactoring and re-

Figure 2.1: Two versions of a simple PHP script.

```
$a = 3;
if ($b == 5) {
  $a = 7;
}
```

```
$a = 3;
if ($b == 5) {
  $b = process($b);
  $a = 1;
}
```

naming operations may cause files with different names to be mapped between the two revisions. In addition, two files that are mapped to each other need not have identical ASTs.

Definition 2.2 (Mappings between files) *Let* $\mathbf{R}_1$ *and* $\mathbf{R}_2$ *be two releases of the same product. The mapping of files between releases is defined as*

$$\mathit{filemap}(\mathbf{R}_1 \rightarrow \mathbf{R}_2) \subseteq \mathbf{R}_1 \times \mathbf{R}_2$$

where $\mathbf{R}_1 \times \mathbf{R}_2$ *represents all pairings of files between the two releases.*

A valid file mapping between two releases must be one-to-one:

Property 2.2.1 (One-to-one file mapping propery) *Let* $\mathbf{R}_1$ *and* $\mathbf{R}_2$ *be two releases of the same product. For any two mappings* (a, b) *and* (c, d) *in* $\mathit{filemap}(\mathbf{R}_1 \rightarrow \mathbf{R}_2)$

$$a = c \Leftrightarrow b = d$$

A valid file mapping must map all files to themselves when a release is compared against itself:

Property 2.2.2 (File mapping identity propery) *For a release* $\mathbf{R}_1$

$$\mathit{filemap}(\mathbf{R}_1 \rightarrow \mathbf{R}_1) = \{(f, f) \mid f \in \mathbf{R}_1\}$$

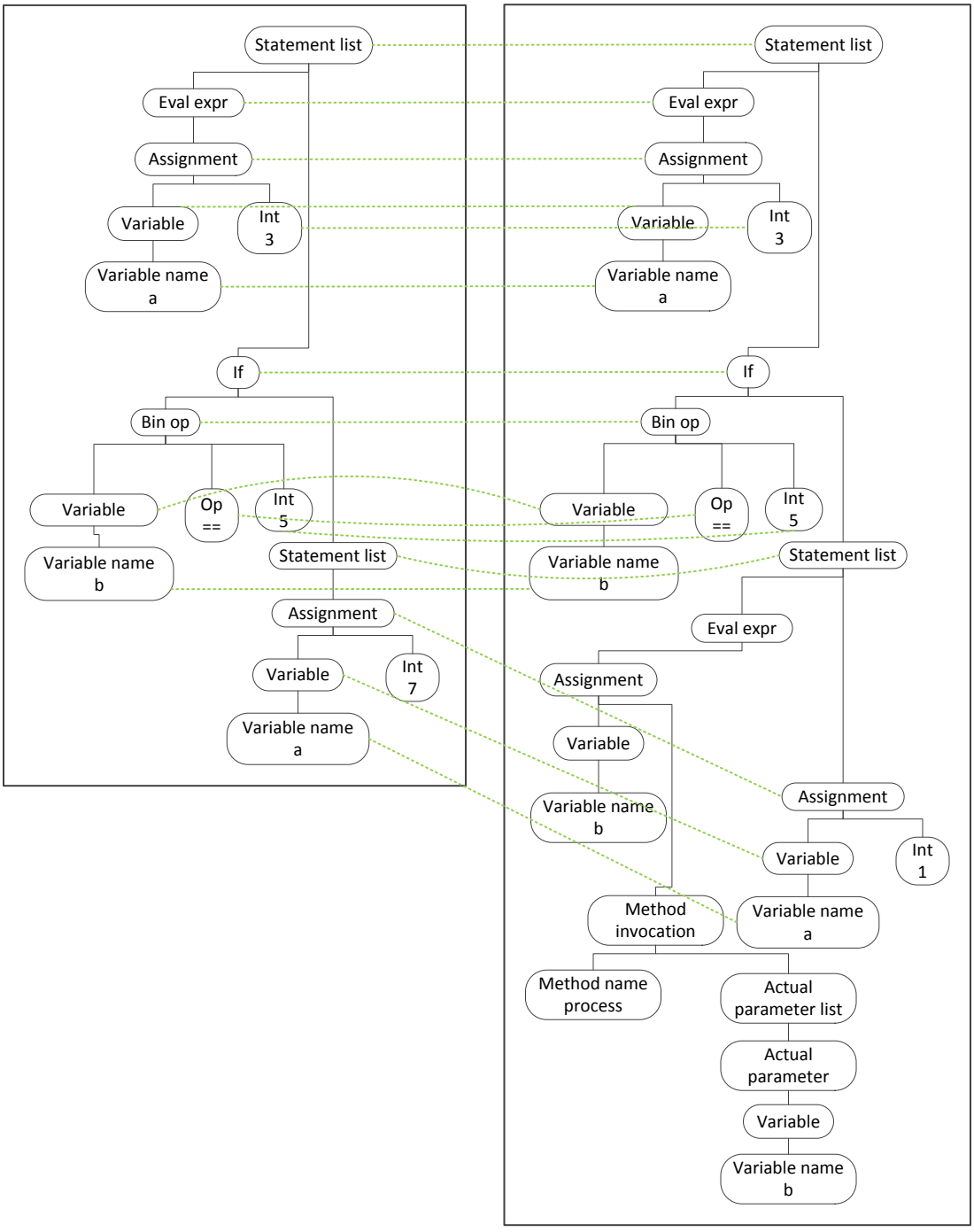


Figure 2.2: Abstract syntax trees for two versions of a PHP script. Dotted lines represent the node mapping.

A file mapping may be *symmetric*, such that reversing the order of two releases under comparison results in the mappings between files to be inverted.

Property 2.2.3 (File mapping symmetry propery) *Let* $\mathbf{R}_1$ *and* $\mathbf{R}_2$ *be two releases of the same product. For any two mappings* (a, b) *and* (c, d) *in filemap*$(\mathbf{R}_1 \rightarrow \mathbf{R}_2)$

$$a = c \Leftrightarrow b = d$$

Finally, a file mapping may be *transitive*, such that a file retains the same identity when being mapped through several individual revisions as it does when directly mapped between the first and last revision in question.

Property 2.2.4 (File mapping transitivity property) *Let* $\mathbf{R}_1$*,* $\mathbf{R}_2$*, and* $\mathbf{R}_3$ *be three releases of the same product. With* f_i *being a file in release* $\mathbf{R}_i$

$$\forall f_1, f_2, f_3 \, (f_1, f_2) \in \mathit{filemap}(\mathbf{R}_1 \rightarrow \mathbf{R}_2) \wedge (f_2, f_3) \in \mathit{filemap}(\mathbf{R}_2 \rightarrow \mathbf{R}_3) \Rightarrow (f_1, f_3) \in \mathit{filemap}(\mathbf{R}_1 \rightarrow \mathbf{R}_3)$$

It is not necessary for Properties 2.2.3 and 2.2.4 to be satisfied in order to use this system to construct code or change metrics. However, if it is desired that change metrics in the metric family satisfy the properties of a mathematical distance pseudometric, the additional requirement of these two properties is imposed.

If all of the above properties are satisfied, the file mappings become an equivalence relation over the set of all files in all releases. Files can then be grouped into equivalence classes, where any given revision of a product will contain no more than one member of any given equivalence class. By convention, in Section 2.4, the variables T and U will be used when referring to two members of such an equivalence class (the same file at two different revisions), and the variables X, Y, and Z will be used when referring to three members of such a class.

2.2.2.2 Mapping AST nodes across files

Recall that when constructing a mapping between the files of two revisions of the same product, it is permissible to map two files to each other even if their their contents (i.e. their ASTs) differ. Accordingly, the differences between the contents of pairs of mapped files must also be defined. We define differences between files in a similar way that we define differences between releases, comparing AST nodes within a file instead of files within a release.

Definition 2.3 (Mapping between nodes) *Let T and U be two ASTs. The mapping of nodes between trees is defined as*

$$nodemap(T \to U) \subseteq T \times U$$

where $T \times U$ represents all pairings of nodes between the two trees.

A valid node mapping between two trees must be one-to-one:

Property 2.3.1 (One-to-one node mapping propery) *Let T and U be two ASTs. For any two mappings (a, b) and (c, d) in $nodemap(T \to U)$*

$$a = c \Leftrightarrow b = d$$

A valid node mapping must map all nodes to themselves when a tree is compared to itself:

Property 2.3.2 (Node mapping identity propery) *For an AST T*

$$nodemap(T \to T) = \{(t, t) \mid t \in T\}$$

Additionally, a valid file differencing algorithm should only map nodes which are equivalent, such that if the *nodemap* function maps two nodes, those nodes

should be *equivalent*. Two nodes are equivalent if all properties of the nodes are equal once the value mapping between the files maps values from one file's labeling function vocabulary to the other file's vocabulary. Only the labeling functions access the node properties; hence, we decline to formalize these properties in our source code model, and we leave this notion of equivalence undefined as well. The form and content of these properties will change from language to language and from parser to parser.

A node mapping may be *symmetric*, such that reversing the order of two releases under comparison results in the mappings between nodes to be inverted. This property will become necessary to satisfy the symmetry condition when defining a pseudometric space of releases.

Property 2.3.3 (Node mapping symmetry propery) *Let T and U be two ASTs. For any two mappings (a, b) and (c, d) in $nodemap(T \to U)$*

$$a = c \Leftrightarrow b = d$$

Finally, a node mapping may be *transitive*, such that a node retains the same identity when being mapped through several individual revisions as it does when directly mapped between the first and last revision in question. This property will become necessary to satisfy the triangle inequality when defining a pseudometric space of releases.

Property 2.3.4 (Node mapping transitivity property) *Let T, U, and V be three ASTs*

$$\forall t, u, v (t, u) \in nodemap(T \to U) \wedge (u, v) \in nodemap(U \to V) \Rightarrow$$
$$(t, v) \in nodemap(T \to V)$$

2.2.2.3 Mapping values between tree nodes

Finally, recall that when two ASTs are mapped, it may be desirable to map certain properties of the tree's nodes to de-emphasize non-significant changes when constructing a metric family. Revisiting an example given in Section 2.2.2, if a single line of code is inserted near the top of a file, every line number below that line changes, even though the corresponding lines of code (and AST nodes) remain unchanged, and the unchanged AST nodes remain mapped to each other.

We formalize this notion of mapping with value mappings, which are defined as bijections between the possible labels generated by the labeling functions for two ASTs. Recall that the range of a labeling function for AST T is $T \times \mathbf{L_T}$. We define a value mapping as follows:

Definition 2.4 (Mapping between values) *Let T and U be two ASTs. For a given labeling function with vocabularies of labels $\mathbf{L_T}$ and $\mathbf{L_U}$ for these two trees, the mapping of values is defined as*

$$valuemap(T \to U) \subseteq \mathbf{L_T} \times \mathbf{L_U}$$

such that valuemap is a bijection between $\mathbf{L_T}$ and $\mathbf{L_U}$

Because the range of the labels generated by a given labeling function is defined by the tree being labeled, the value mapping from a tree to itself must be the identity function.

Property 2.4.1 (Value mapping identity property) *For tree T and value $\lambda \in \mathbf{L_T}$*

$$\lambda = valuemap(T \to T, \lambda)$$

As with file and node mappings, the value mappings must also be symmetric and transitive (with the additional constraint that mappings are also bijective):

Property 2.4.2 (Value mapping symmetry propery) *For trees T,U and value $\lambda \in \mathbf{L_T}$*

$$\lambda = valuemap\,(U \to T, valuemap\,(T \to U, l))$$

Property 2.4.3 (Value mapping transitivity propery) *For trees X,Y,Z and value $\lambda \in \mathbf{L_T}$*

$$valuemap\,(Y \to Z, valuemap\,(X \to Y, \lambda)) = valuemap\,(X \to Z, \lambda)$$

Because the value mapping function is both one-to-one and a bijection, the pairwise value mapping functions for pairs of trees collectively define *equivalence classes* of values across all trees, such that the range of a labeling function for any tree contains one element from each equivalence class. Not all labels will be present in all trees – for example, the line numbers for a file will not go beyond the number of lines in the file. For equivalence classes which may not naturally fall within the range of a labeling function for a given tree, dummy elements may be added to the $\mathbf{L}$ set for a labeling function in order to satisfy the above properties for a value mapping. The exact way that these dummy elements are mapped to elements for other trees is inconsequential, as none of the dummy elements will actually appear in a label so they will not factor into any measurement.

In many cases, it will be necessary to use a matching algorithm to build a value mapping (one is described in Section 2.5.3). However, in some cases, a simpler value mapping method may better align with the semantics of the metric that is being built. For example, a metric family that measures changes to numeric or string literals would use a simpler value mapping method that only maps equal literals, which essentially nullifies the value matching concept because it is not appropriate for that family. If a matching algorithm were used here, changes in the literals would not be registered in the measurements.

2.2.2.4 Notes on revision differencing algorithms

We impose no further requirements on revision differencing algorithms other than the ones described above. These are many different revisions differencing methods that would satisfy these properties, and the characteristics of the algorithm may have implications on the intuitiveness of a change metric (or the degree that a change metric reflects the quantity that its designer intended). For example, a trivial revision differencing algorithm could simply return an empty mapping between the files of any two revisions. Such an algorithm would result in a valid metric space of revisions (as all distances would be zero) but such a metric would not be useful for practical purposes. A trivial value mapping algorithm could simply return a mapping where arbitrary values were mapped, as long as the mapping was done consistently in order for the value mapping properties to be satisfied. In other words, the properties we described above are necessary for the metrics to meet certain theoretical guarantees, but are not sufficient for the metrics to be practically usable. Strategies for implementing these algorithms in a non-trivial way will later be discussed in Section 2.5.

2.2.3 Definitions of predicates used to compute code and change metrics

As described earlier in this section, a metric family is defined by a labeling function, and source code to be measured is modeled by releases and abstract syntax trees plus a revision comparison algorithm. Next, we will define how static code metrics and change metrics are actually constructed. In order to simplify this definition, we first define several sets and predicates that will be used throughout the definitions and proofs that follow.

Definition 2.5 (Labeled element sets) *Let T and U be two trees in revisions* $\mathbf{R}_1$

and $\mathbf{R}_2$ *respectively which were mapped to each other by the file mapping algorithm. Let* $\lambda \in \mathbf{L_T}$ *be a member of the mapping function's range for* T*. We first define sets* A *and* B *to be sets of AST nodes which were labeled with label* λ*:*

$$A_T^\lambda = \{t \mid (t, \lambda) \in l\left(T, \text{lookup}\left(\mathbf{R}\right), f\right)\}$$

$$B_{T\to U}^\lambda = \left\{t \mid \underset{u}{\exists}\,(t,u) \in \mathit{nodemap}\left(T \to U\right) \wedge u \in A_U^{\mathit{valuemap}(T\to U,\lambda)}\right\}$$

The A set simply contains nodes in revision T which were labeled with the given label, while B contains nodes in T which were also present in U and labeled in U after the label has been mapped from T to U. We will define predicates and metrics with respect to these two sets, defining static code metrics with set A alone, and defining change metrics with both set A and set B, as the combination of these two sets allows for the identification of labels that appeared or disappeared when moving between two revisions.

We will next define several predicates that will be used to build numerical measurements.

Definition 2.6 (Metric predicates) *Let* T *and* U *be two trees in two different revisions which were mapped by the file mapping algorithm. These predicates are then defined for* $T \to U$*, or measurements of the change that occurred while transitioning from tree* T *to tree* U*.*

$$\text{PRESENT}_T^\lambda = A_T^\lambda \neq \emptyset$$

$$\text{ADDONLY}_{T\to U}^\lambda \equiv A_T^\lambda \cap B_{T\to U}^\lambda = \emptyset \wedge A_T^\lambda = \emptyset \wedge A_U^{\mathit{valuemap}(T\to U,\lambda)} \neq \emptyset$$

$$\text{DELONLY}_{T\to U}^\lambda \equiv A_T^\lambda \cap B_{T\to U}^\lambda = \emptyset \wedge A_T^\lambda \neq \emptyset \wedge A_U^{\mathit{valuemap}(T\to U,\lambda)} = \emptyset$$

$$\text{ADDDEL}_{T\to U}^\lambda \equiv A_T^\lambda \cap B_{T\to U}^\lambda = \emptyset \wedge A_T^\lambda \neq \emptyset \wedge B_{T\to U}^\lambda \neq \emptyset$$

$$\text{MOD}^{\lambda}_{T\to U} \equiv A^{\lambda}_{T} \cap B^{\lambda}_{T\to U} \neq \emptyset \wedge \left(A^{\lambda}_{T} \neq B^{\lambda}_{T\to U} \vee A^{valuemap(T\to U,\lambda)}_{U} \neq B^{valuemap(T\to U,\lambda)}_{U\to T} \right)$$

$$\text{NONE}^{\lambda}_{T\to U} \equiv A^{\lambda}_{T} = B^{\lambda}_{T\to U} \wedge A^{valuemap(T\to U,\lambda)}_{U} = B^{valuemap(T\to U,\lambda)}_{U\to T}$$

The first predicate, PRESENT, indicates if the given label was present at a single revision with tree T. The next predicates, ADDONLY and DELONLY, indicate if the given label λ was added or deleted in the transition from T to U. The label is deemed to be added if no nodes have the label before the transition, but at least one node is labeled with that label after the transition. The label is deemed to be deleted in the opposite case – if λ is present before but not after the transition. Next, with ADDDEL, a label can be both added and deleted across a transition if λ exists before and after the transition, but none of the nodes with the label persist across the transition. For example, if a labeling function labels nodes with the name of the function definition enclosing a node, the label is added and deleted at the same time if the function is completely rewritten between T and U. These predicates for addition and deletion will be used to account for this case when designing metrics. The next predicate, MOD indicates if the set of nodes labeled with λ was modified in the transition from T to U without adding or deleting the label (in other words, at least one labeled node persists across the transition). The final predicate, NONE, indicates that the set of nodes with label λ persisted across the transition from T to U unchanged.

2.2.4 Computing code and change metrics

We can now build the members of a metric family, static code and change metrics, by defining them in reference to these metric predicates. Each such metric defined here is defined at the intersection of a file and one (or two) releases. In other words, every file at the point of every release has its own measurement. In Chapter 3, we will introduce aggregation methods that generate just one measurement for an

entire release, or one measurement summarizing the changes made to an individual file across all releases.

Definition 2.7 (Static source code metric) *For file T, if $\mathbf{L_T}$ is the domain of the labeling function for the metric family, the static source code metric is defined as:*

$$\sum_{\lambda \in \mathbf{L_T}} \begin{cases} 1 & \textit{if } \mathrm{PRESENT}_T^{\lambda} \\ 0 & \textit{otherwise} \end{cases}$$

We will define two kinds of change metrics that can become members of a metric family. The first change metric, the code delta metric [56], is simply derived from the difference of the code metrics of the file at each release. The second change metric, the difference-based change metric, utilizes the revision differencing process described previously. We will later discuss the differences between the these two kinds of change metrics in Section 2.4.2.

As described in more detail in Section 2.2.2.1, we must first stipulate that the trees being compared are different revisions of the same file. Let T be the tree for a file at revision $\mathbf{R}_1$. Next, let U be the tree for the same file at a different revision $\mathbf{R}_2$:

$$U \equiv \{U \mid (T, U) \in \textit{filemap}\,(\mathbf{R}_1 \rightarrow \mathbf{R}_2)\}$$

Definition 2.8 (Code delta change metric) *For trees T and U and labeling function vocabularies $\mathbf{L_T}$ and $\mathbf{L_U}$, the code delta change metric is defined as:*

$$\mathrm{codedelta}(T, U) \equiv \left| \sum_{\lambda \in \mathbf{L_U}} \begin{cases} 1 & \textit{if } \mathrm{PRESENT}_T^{\lambda} \\ 0 & \textit{otherwise} \end{cases} - \begin{cases} 1 & \textit{if } \mathrm{PRESENT}_U^{\lambda} \\ 0 & \textit{otherwise} \end{cases} \right|$$

Definition 2.9 (Difference-based change metric) *For trees T U and domain $\mathbf{L_T}$ for tree T, and for coefficients p_1, p_2, p_3, and p_4, the difference-based change metric is defined as:*

$$\text{changemetric}(T,U) \equiv \sum_{\lambda \in \mathbf{L_T}} \begin{cases} p_1 & \textit{if } \text{ADDONLY}^{\lambda}_{T \to U} \\ p_2 & \textit{if } \text{DELONLY}^{\lambda}_{T \to U} \\ p_3 & \textit{if } \text{ADDDEL}^{\lambda}_{T \to U} \\ p_4 & \textit{if } \text{MOD}^{\lambda}_{T \to U} \\ 0 & \textit{otherwise} \end{cases}$$

Although the domain $\mathbf{L_U}$ does not explicitly appear in this definitions, the definitions of the predicates perform a value mapping from $\mathbf{L_T}$ to $\mathbf{L_U}$. Because the value mapping is a bijection between $\mathbf{L_T}$ and $\mathbf{L_U}$, the entire domain $\mathbf{L_U}$ is considered.

By varying the p coefficients, the characteristics of the difference-based change metric can be manipulated. For example, a metric family measuring function definitions could count only the ADDONLY or DELONLY predicates to result in a metric measuring the number of functions added or deleted in a release. In addition, if the resulting metric is to qualify as a distance metric (in the mathematical sense), certain constraints will be imposed on the p coefficients as described in Section 2.4.

2.3 Examples and guidance for developing metric families

In the preceding sections, we introduced a functional representation for defining metric families, and we described how the metric family members (i.e. metrics) for a particular metric family could be computed. In this section, we work through some concrete examples of several metric families that we developed. We also describe *design patterns*, or general strategies which we used when designing metric

families. Finally, we present a summary of all the metrics that were implemented as part of this vulnerability prediction study.

2.3.1 Examples

Throughout this chapter, we have introduced a procedure for defining metric families, processing source code, and computing code and change metrics. Finally, we will present several illustrated examples of how metric family members are computed.

Each example starts with two versions of a PHP script (an older version and a new version), such as in Figure 2.1. Parsing each version of the script yields the two ASTs shown in Figure 2.2. The next step is to compute a tree node mapping from the older to the newer version, yielding the dotted lines shown on the same figure.

The first metrics that we will compute are a code and change metric for the *lines of code* metric family. The labeling function which defines this metric family labels each AST node with the source line number where the corresponding source code construct occurred. These labeled trees are depicted in Figure 2.3.

First, we compute the static code metric for this metric family. In this case, the static code metric is a simple lines-of-code count, which is computed by counting the unique labels on each tree (i.e. the number of unique line numbers in the file). In this example, the first script has a measurement of 3 lines of code, while the second script has a measurement of 4 lines of code.

Next, we take the first steps toward computing difference-based change metrics for this metric family. First, the value mapping between the labels of each AST must also be computed. This value mapping is shown in Figure 2.4.

The top half of the figure depicts the bipartite graph weighted by the mapped label values (based on the labeled nodes of the first tree which were mapped to

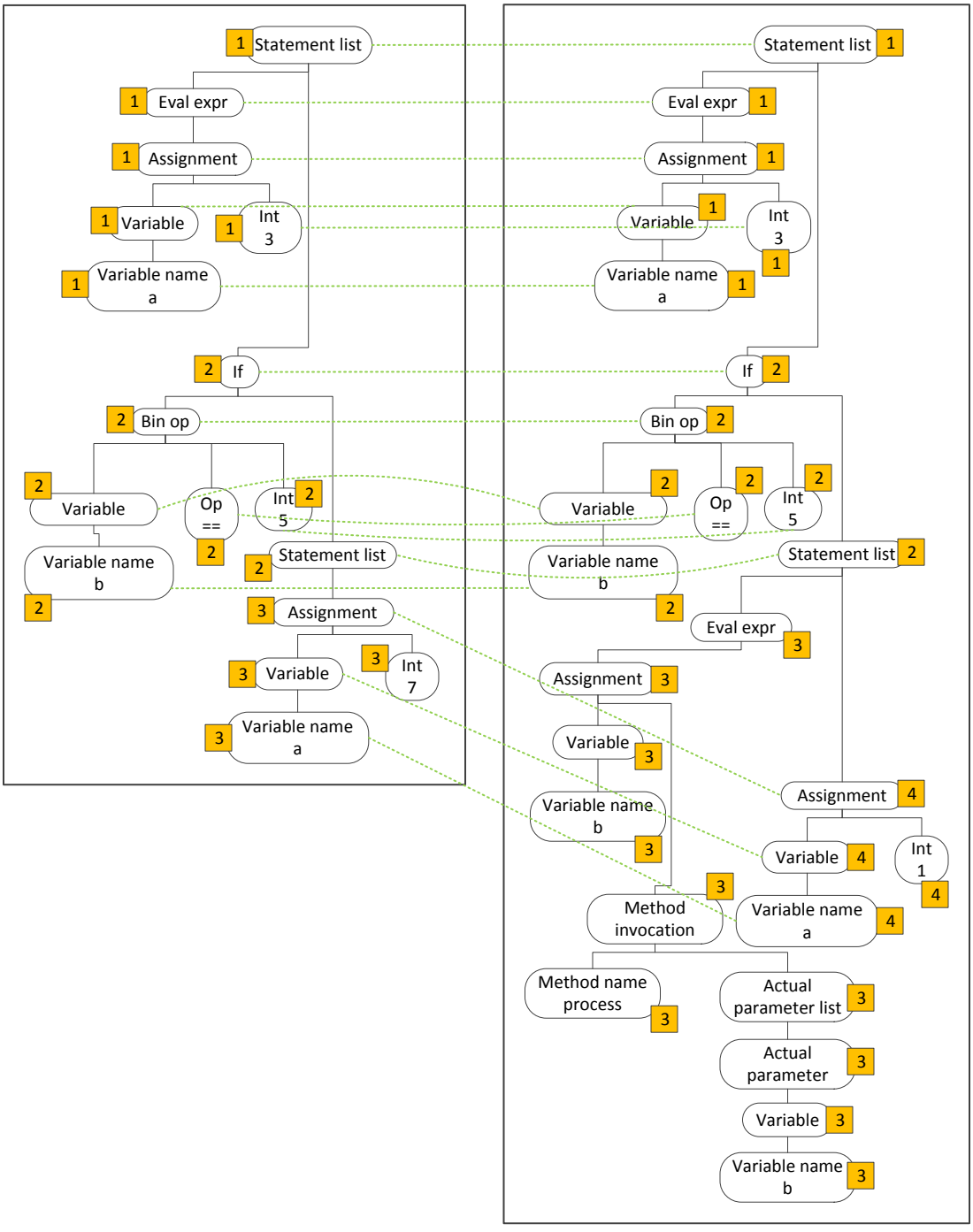
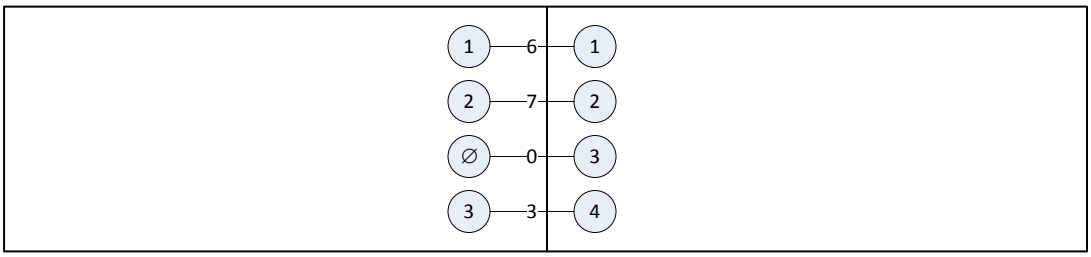
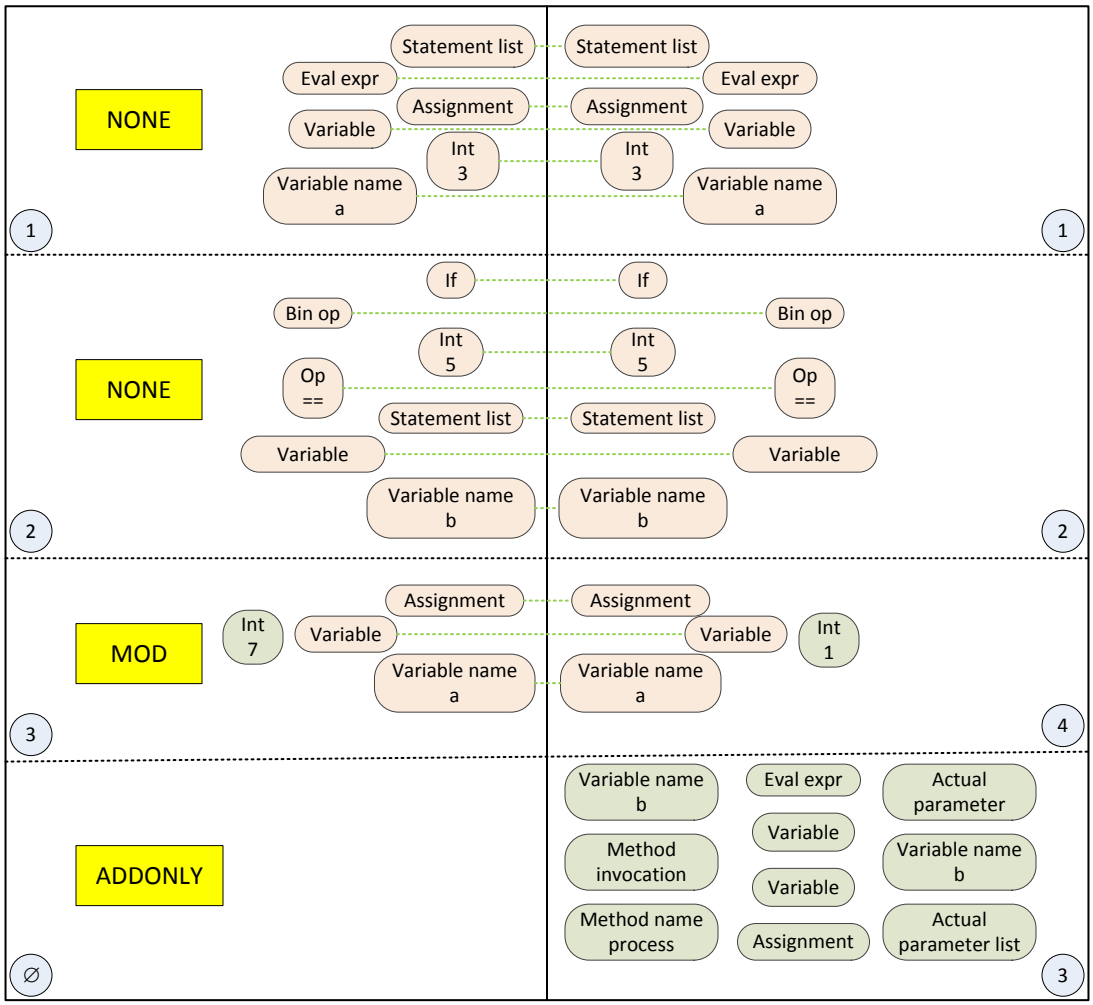


Figure 2.3: Abstract syntax trees for two versions of a PHP application labeled with a line number labeling function



Value mapping



Labeled element sets

Legend: In set A | In set A and set B | Final predicate

Figure 2.4: Value mappings and labeled element sets for a change between two versions of a simple PHP script

labeled nodes of the second tree). From this graph, we can see that the value mapping is simple and ambiguous, meaning that there is no need to find the optimal matching with the Hungarian algorithm. Each line number in the first revision can only be matched to one line number in the second revision – for example, the old line 3 becomes the new line 4. Line 3 in the second revision is new, and is mapped to a dummy node in the first revision with a zero weight.

Now that the node and value mappings have been computed, the labeled element sets from Definition 2.5 can be computed. In the bottom half of Figure 2.4, nodes labeled with each of the four possible labels (matched line numbers) are depicted in separate sections of the diagram. Each one of these sections corresponds to the nodes in the A^{λ} set for that revision, where λ is the line number in question. The color of the node indicates if the node is also present in the B^{λ} set for that revision, which indicates that the node was matched to another node with an identically matched label.

On the left-hand side of the diagram, the metric predicates for each λ, as defined in Definition 2.6, are shown. Any difference-based change metric in the metric family can now be computed by summing each predicate in accordance with the p coefficient weights. For example, if p_1 is 1 and p_4 is 1, then the change metric between the two revisions is 2 because one line was added and another line was modified. This would correspond to an "added and modified lines of code" change metric, a variation of classical code churn.

By following this process – mapping nodes between trees, labeling trees with a labeling function, mapping label values, and satisfying predicates by matching the mapped nodes and values – a variety of code and change metrics can be computed. Notably, every metric family constructed with this method will contain the same kinds of code and change metrics, allowing for a fair comparison between code and change-based features when performing defect and vulnerability prediction experi-

ments.

We now present illustrated examples of two metric families which act as variations of cyclomatic complexity. Figure 2.5 depicts the labeling for a *shallow* cyclomatic complexity metric, while Figure 2.6 depicts a *deep* variation of cyclomatic complexity metric. In both variations, decision structures (structures resulting in branches in the program's control flow path) are labeled, as well as the root node of the AST (representing the single path present in any program). The difference between the two variations is that the deep variation is sensitive to changes made anywhere within the labeled control structure, while the shallow version is only sensitive to changes made to the condition expression. Note that this variation only affects distance-based change metrics – the elementary source code metrics (and hence the code delta change metrics also) will be identical for both.

2.3.2 Single-node metric family design patterns

We now discuss two metric family *design patterns* that we used to implement the remainder of the 49 metric families used in this study. As with general software design patterns, these patterns are not hard-and-fast rules that are forced by the system; rather, we found them to be useful guidelines which we applied throughout the metric development process.

The first pattern, the single-node pattern, applies to metrics focusing on entities that can easily be localized to a single AST node. For example, a metric that was concerned with calls between files would locate AST nodes corresponding to method invocations where the metric being invoked was part of a different file. Such a metric family can function by labeling single AST nodes, because each method invocation is associated with a single `Method invocation` node.[1] In con-

[1] Although some AST representations may, for example, store the name of the method being called in a child AST node of the AST node representing the method invocation, labeling functions can trivially consolidate this information into the label of the single node.

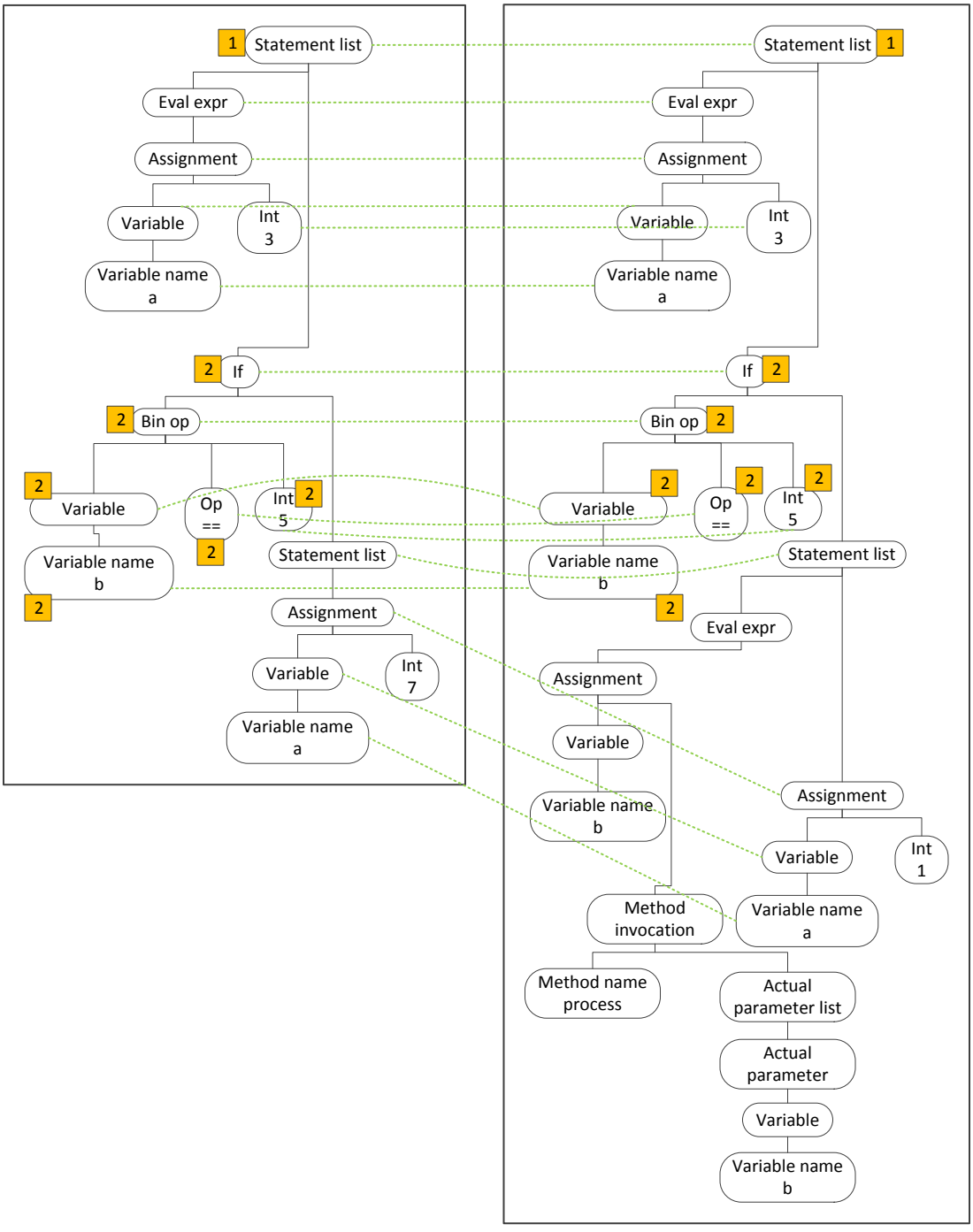


Figure 2.5: Abstract syntax trees for two versions of a PHP application labeled with a shallow cyclomatic complexity (measuring the condition expression) labeling function

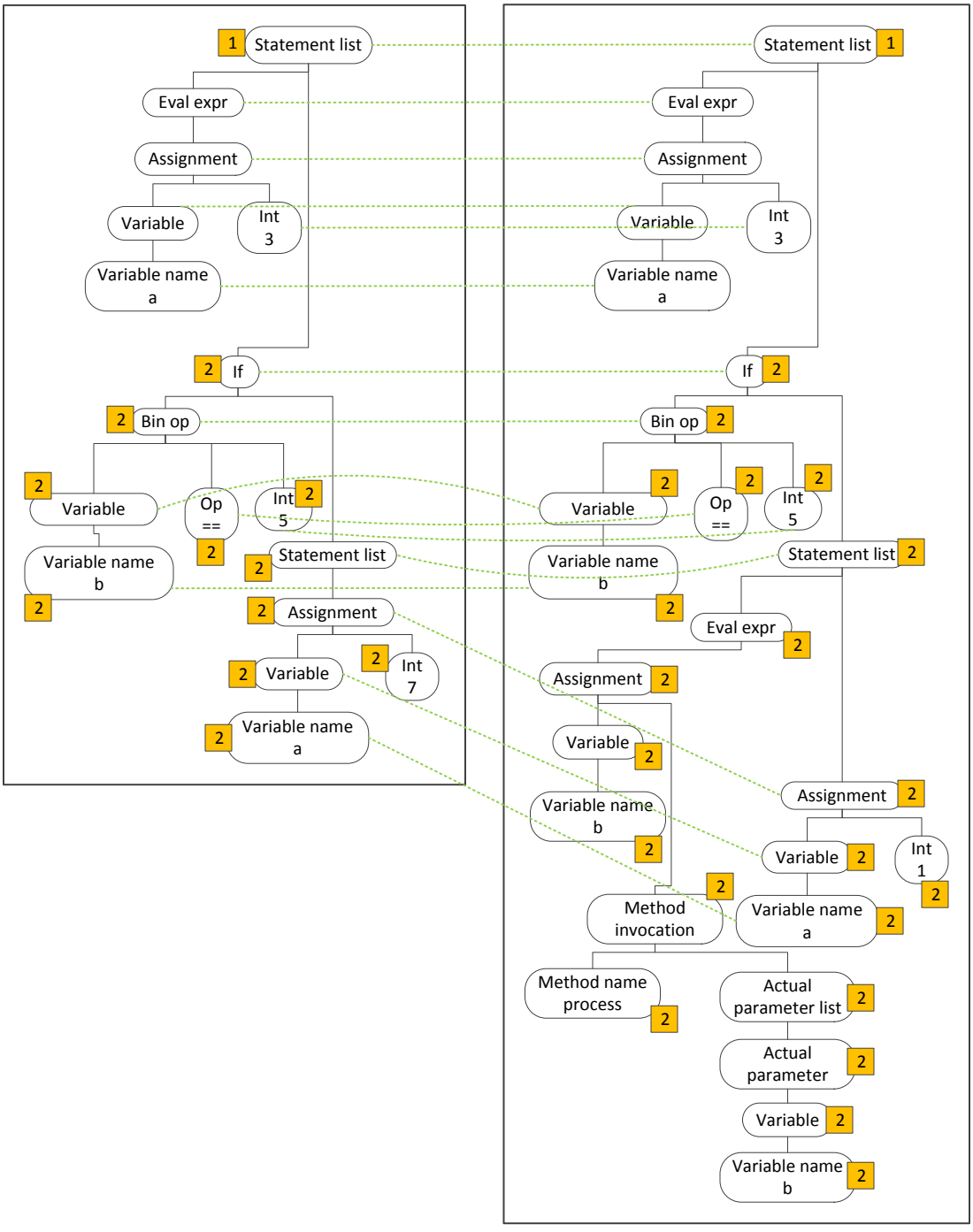


Figure 2.6: Abstract syntax trees for two versions of a PHP application labeled with a deep cyclomatic complexity (measuring the entire structure) labeling function

trast, a metric that was concerned with method definitions may take into account the contents of (or changes to) the method's entire body, meaning that the entity of interest is not localized to a single node.

Some examples of metrics that have been constructed with this pattern include:

- Coupling metrics which measure *outgoing* method invocations (as each invocation is performed at a single AST node)
- Metrics of total or unique variable names, operators, or operands (such as the components of the Halstead metrics)
- Metrics counting a particular type of token, such as class modifiers or conditions

Several patterns can be applied to develop a tree labeling function, choose operators, and develop metrics for these kinds of metrics:

Distinct variants of entities: If a metric is concerned with the number of distinct variants of a type of entity that appears in source code (such as the number of distinct method names that are called, or the number of distinct variable names that appear in a file), the labeling function can return a single label for each AST node of the desired type, where the label's value encodes the name of this distinct entity (such as a method name).

If a metric is concerned with both the distinct variants of an entity and the file that an entity is related to, the labeling function can return a tuple (v, f) where v is a value corresponding to the entity in question (such as the name of a method being called) and f contains the name of the file, such as the file where the called method was defined (as determined by the lookup table). This yields a metric that counts the number of unique methods invoked across all files of the program. The labeling function can further be refined by having it filter the tuples against the parameter

that indicates the current file; for example, by only labeling calls to methods defined in external files.

In either of the cases above, the metric will have an aspect of totaling the number of *unique* variants of an entity that exist. Some examples of metric family members that could be constructed in this way:

- Static source code metric: Number of distinct variants present in the file (for example, the number of different functions called within a file)
- Code delta change metric: Difference between the number of distinct variants present in the file in this revision and the number present in the previous revision (for example, the number of distinct functions called in this revision of a file as compared to the previous revision)
- Difference-based change metric, p_1 positive, other coefficients zero: The number of distinct variants newly added in this revision (for example, the number of newly called functions in this revision)
- Difference-based change metric, p_1, p_2, p_3, p_4 positive: Churn of distinct variants – number of distinct variants that were added, deleted, or modified from the previous revision to this one (for example, the number of distinct functions for which function calls were added, deleted, or modified in this revision)
- Difference-based change metric, p_1 and p_2 positive, other coefficients zero: The number of distinct variants newly added or deleted in this revision (for example, the number of functions were called in the previous revision of a file but not in this revision, or vice versa)

The cases above relate to metrics which are concerned with totaling the *unique* variants of an entity; however, other metrics may be concerned with the *total* number of occurrences of an entity, rather than the number of *unique variants*. For example,

one metric may count the total number of external method calls (method calls to a different file) made from within a file, counting multiple calls to the same method multiple times.

For these cases, instead of basing the labels on some attribute of the entity (such as the name of the method being called), the labeling function can return some kind of serial number, such as an AST node unique identifier. The way that this identifier is constructed is not significant; it is only necessary that the label attached to each AST node will have a different identifier, ensuring that larger numbers of labels will results in larger measurements.

Note that it is not necessary to ensure that the serial numbers across files match or correspond in any way. Instead, because the label vocabularies between files will be compared with a value matching algorithm (described in Section 2.5.3), the value matching algorithm will match the serial numbers between files on its own, assuming that the implementation of the value matching algorithm is adequate.

Note that the labeling function can still internally filter its labels on some attribute that is not present in the label itself – for example, the labeling function could only choose to label external method calls (calls to other files). Some examples of metric family members that could be constructed in this way:

- Static source code metric: Total number of entities present in the file (for example, the number of function calls present in the file)
- Code delta change metric: Difference between the number of entities present in the file in this revision and the number present in the previous revision (for example, the number of function calls in this revision of a file as compared to the previous revision)
- Difference-based change metric, p_1 positive, other coefficients zero: The number of entities newly added in this revision (for example, the number of function

calls added in this revision)

- Difference-based change metric, p_2 positive, other coefficients zero: The number of entities deleted in this revision (for example, the number of function calls deleted in this revision)

- Difference-based change metric, p_1, p_2 positive: Churn of distinct variants – number of distinct variants that were added or deleted from the previous revision to this one (for example, the number of function calls added or deleted.)

Note that p_3 and p_4 are not used when the labeling function produces serial numbers; this is due to the uniqueness of the labels. The MOD predicate is only relevant when multiple instances of the same label may exist simultaneously. The ADDDEL predicate would only be relevant if the single instance of the label moved from one node to another, which would not normally occur as the implementation of the label matching algorithm should prevent this (see Section 2.5.3).

2.3.3 Tree-subset metric family design patterns

Other metrics are not only concerned with entities localizable to a single AST node, but are also concerned with the entire *subsets* of the AST that relate to this entities. For example, one metric may measure the number of modified methods at a particular revision. This requires a metric that can consider changes anywhere in the AST subtree which contains the method definition. In another example, measuring a definition such as a field definition in Java falls under this design pattern, as the rvalue of the method definition can encompass an arbitrary number of AST nodes.

For these metrics, we introduce a second design pattern, for metrics that measure the presence (or change) of entities which can be scattered across a source file. Note that this design pattern is concerned with AST *subsets* as opposed to *subtrees*, or children of a particular node. This is because the AST nodes for an entity such

as a line of code can be scattered around an AST without being connected into a single tree (for example, a line of code can contain multiple, independent statements which are siblings in the AST). In addition, a metric may only be concerned with certain portions of an entity (such as the portions of a loop relating to the loop condition).

Three design patterns can be used to develop tree labeling functions:

- For most metrics of this type, each AST node in the tree subset of interest can be labeled with a label containing a unique identifier for the entity in question. For some entities, such as lines of code or method definitions, a unique identifier already exists (the line number or the method name, respectively). For entities where no such identifier already exists (such as `while` loops), unique serial numbers can be generated, with the only constraint that each entity must have a different identifier.

- For metrics concerned with the relationship between a tree subset and some other entity, each AST node in the tree subset can be labeled with an identifier of the other entities related to the AST subset. For example, a coupling metric tracking file-to-file relationships could label each AST node in a method with the names of the files that call the method.

- For metrics which must consider both the entities of interest and relationships between these entities, each AST node in the tree subset of interest can be labeled with a tuple, similarly to how this can be done with single-node metric families. The metrics resulting from this pattern resemble metrics constructed with the first pattern, but are weighted by the number of relationships that each tree subset has.

In some cases, when constructing metrics with these patterns, it will be necessary to choose between unique identifiers already present in the code (such the

name of a method being defined) or generated serial numbers. If value mappings are being generated with a matching algorithm, then this choice has no consequence – the value mapping algorithm will map the labels without regard to how the unique identifier was actually generated. However, if the unique identifier is extracted from the code (such as a method name) and a matching algorithm is not used for value mappings, then the generated metric will be sensitive to renaming the identifier. For example, if a method is simply renamed from one revision of a file to the next, then metrics constructed without the matching algorithm will register a deletion and a creation of two separate methods; in contrast, the metric constructed with the matching algorithm will register no changes at all.

Some examples of metric family members that can be constructed with tree subset design patterns:

- Static code metric: Number of distinct entities present in the file (for example, the number of function definitions present in the file)
- Code delta change metric: Difference between the number of entities present in the file in this revision and the number present in the previous revision (for example, the number of function definitions in this revision of a file as compared to the previous revision)
- Difference-based change metric, p_1 positive, other coefficients zero: The number of entities newly added in this revision (for example, the number of new functions defined in this revision)
- Difference-based change metric, p_2 positive, other coefficients zero: The number of entities deleted in this revision (for example, the number of function definitions deleted in this revision)
- Difference-based change metric, p_1, p_2 positive: Churn of distinct variants – number of distinct variants that were added or deleted from the previous

revision to this one (for example, the number of function definitions added or deleted)

- Difference-based change metric, p_3, p_4 positive: Existing entities modified – number of existing entities that were modified in this revision (including entities that were completely modified; for example, the number of function definitions modified)

- Difference-based change metric, p_1, p_2, p_3, p_4 positive: Churn of entities – number of distinct entities that were added, deleted, or modified from the previous revision to this one (for example, the number of function definitions that were added, deleted, or modified in this revision)

Some examples of metrics that have been constructed with this pattern include:

- Coupling metrics which measure *incoming* method invocations (as each invocation targets an entire method body)

- Metrics counting certain types of blocks, such as decision blocks or loops. One example of such a metric is cyclomatic complexity

- Metrics counting lines of code (as a line of code is spread across multiple AST nodes)

2.3.4 Implemented metric families

In this section, we describe the 49 metric families which we implemented to support the vulnerability prediction experiments which follow later in this work. In the following table, we list and briefly describe each such metric family. In the table, the *group* column refers to a general classification scheme for these metric families which we will describe in more detail after the table. The *family* column provides a name for the family. *Label node type* lists type types of nodes that the labeling

function is capable of labeling, and *label with* lists the labels that will be applied to these node types. Finally, *conditions* lists any conditions that would cause some nodes of the specified type (but not others) to be labeled.

Table 2.1: List of metric families implemented for this study

Group	*Family*	*Label node type*	*Label with*	*Conditions*
Node contents	Node contents	All	Node type + operator name + variable name + method name	
Node contents	Generic node contents	All	Node type + operator name + built-in method name[2]	
Lines of code	Statements	`Eval_expr`	Node ID	
Lines of code	Lines of code	All	Line number	
Lines of code	Executable lines of code	All	Line number	Descendant of a statement
Lines of code	Non-HTML lines of code	All	Line number	Not in an HTML block

[2]Functions built in to PHP contribute to this measurement, but user-defined functions do not

Group	*Family*	*Label node type*	*Label with*	*Conditions*
Lines of code	Executable, non-HTML lines of code	All	Line number	Descendant of a statement, not in an HTML block
Method definitions	Methods	All	Method name	Descendant of a method
Method definitions	Method signatures	All	Method name	Descendant of the method's signature subtree
Method definitions	Non-command method bodies	All	Method name	Descendant of a method that returns a value
Method definitions	Files calling methods	All	Name of each file containing a call to this method	Descendant of a method
Method definitions	Files mutually calling methods	All	Name of each file that contains a call to this method and is also called by this file	Descendant of a method

Group	*Family*	*Label node type*	*Label with*	*Conditions*
Method definitions	Methods calling methods	All	Name of each method containing a call to this method	Descendant of a method
Branches	Branch conditions	All	Node ID of branch node	Descendant of the condition subtree of a `If` or `Switch` branch node
Branches	Branch contents	All	Node ID of branch node	Descendant of a `If`, `Switch`, or `Try` branch node
Branches	Decision points	`If`, `Foreach`, `For`, `While`, `Do`, `Switch`, `Try`	Node ID	
Branches	Decision point conditions	All	Node ID of decision point node	Node with type `If`, `Foreach`, `For`, `While`, `Do`, `Switch`, `Try`, or descendant of the condition subtree thereof

Group	*Family*	*Label node type*	*Label with*	*Conditions*
Branches	Decision point contents	All	Node ID of decision point node	Node with type `If`, `Foreach`, `For`, `While`, `Do`, `Switch`, `Try`, or descendants thereof
Variables	Variables	`Variable`	Variable name	
Variables	Global variables	`Variable` or `STRING` under `GLOBALS` array	Variable name	Variable is global and referenced inside a method
Variables	Global variables by method	`Variable` or `STRING` under `GLOBALS` array	Variable name + Method name	Variable is global and referenced inside a method
Variables	Local variables	`Variable`	Variable name + Method name	Variable is local and not the `GLOBALS` array
Variables	Variable references	`Variable`	Node ID	
Operators	Arithmetic operators	`OP`, `Post_op`, `Pre_op`	Node ID	Operator is arithmetic (operates on numbers)

Group	*Family*	*Label node type*	*Label with*	*Conditions*
Operators	Comparison operators	`OP`	Node ID	Operator is for comparison
Operators	Logical operators	`OP`	Node ID	Operator is logical (e.g. and, xor)
Operators	Logical operators inside conditions	`OP`	Node ID	Operator is logical (e.g. and, xor) and a descendant of the condition subtree of a `If`, `While`, `Do`, `Switch_case`, or `For` node
Operators	Operator names	`OP`, `Post_op`, `Pre_op`	Node value	
Operators	Operator instances	`OP`, `Post_op`, `Pre_op`	Node ID	
Operators	Built-in functions	`Method_invocation`	Method name	Function is built-in (not defined in user code)
Operators	Built-in function references	`Method_invocation`	Node ID	Function is built-in (not defined in user code)
Constructs	Arrays	`Array`	Node ID	

Group	*Family*	*Label node type*	*Label with*	*Conditions*
Constructs	Breaks	`Break`	Node ID	
Constructs	News	`New`	Node ID	
Constructs	Nulls	`NIL`	Node ID	
Constructs	Numeric literals	`INT`, `REAL`	Node ID	
Constructs	String literals	`STRING`	Node ID	
Constructs	This	`Variable`	Node ID	Variable name is `this`
Constructs	Throws	`Throw`	Node ID	
Method calls	Method calls	`Method_invocation`	Node ID	Not calling a built-in function
Method calls	Methods called by name	`Method_invocation`	Method name	Not calling a built-in function
Method calls	External method calls	`Method_invocation`	Node ID	Not calling a built-in function, called file is not the current one
Method calls	External methods called by file	`Method_invocation`	Name of file containing method def	Not calling a built-in function, file is not the current one

Group	*Family*	*Label node type*	*Label with*	*Conditions*
Method calls	External methods called by name and file	`Method_invocation`	Name of file containing method def + Method name	Not calling a built-in function, called file is not the current one
Method calls	External methods called by mutually calling file	`Method_invocation`	Name of file containing method def	Not calling a built-in function, file is not the current one, other file calls at least one method in the current file
Method calls	External methods conditionally called by name and file	`Method_invocation`	Name of file containing method def + Method name	Not calling a built-in function, method defined in different file, call is under an `If` or short-circuiting operator
Method calls	Internal methods called by name	`Method_invocation`	Method name	Not calling a built-in function, method defined in same file

Group	*Family*	*Label node type*	*Label with*	*Conditions*
Method calls	Internal methods called by name and calling method	`Method_invocation`	Method name + name of method enclosing the invocation	Not calling a built-in function, method defined in same file, method invoked from inside a method
Method calls	Internal methods conditionally called by name	`Method_invocation`	Method name	Not calling a built-in function, method defined in same file, call is under an `If` or short-circuiting operator

Most of these implemented metric families were patterned after various size, coupling, cohesion, and complexity metrics which were included in existing metric tools [82] or used in defect prediction studies [159]. Below, we briefly summarize the motivations for implementing each of these groups of metric families:

- *Node contents*: Metric families that separately label the various "payloads" that an AST node can contain. Primarily intended for use with atom-based predictors (see Chapter 6), such that each one of these "payloads" is considered a separate, potential indicator of vulnerabilities.

- *Lines of code*: Metric families related to measuring (e.g. counting) lines of code

or statements. Under traditional metric classification schemes, size metrics would be constructed from these.

- *Method definitions*: Metric families related to method definitions, method implementations, and the in-degree of calls into methods. Size (such as a method count) and coupling metrics would be constructed from these metric families, plus some of the components of computed cohesion metrics.

- *Branches*: Metric families related to different kinds of control structures, or ways that a program's control structure can branch. Many complexity metrics fall in this group.

- *Variables*: Metric families related to global and local variables. These can act as a type of complexity metric or component parts of a data cohesion metric.

- *Operators*: Metric families related to different kinds of operators (including arithmetic operators).

- *Constructs*: Metric families measuring specific, individual language constructs. These metric families parallel the fine-grained software change predictors used in other defect prediction studies [51].

- *Method calls*: Metric families related to invocation of functions or methods (both within the same file and into a different file), except for those built into the PHP language. These are similar to traditional coupling metrics, and act as component parts of cohesion metrics as well.

Aside from the metric families described in the above table, we also implemented several *derived metrics*, which are compound metrics implemented using formulas that combine several of the metrics discussed above. We implement these derived metrics because some metrics that have commonly been used for defect prediction in the past cannot be implemented by a single metric family member alone,

due to the constraints imposed by the structured metric family definition process. By computing these additional metrics, we ensure that several very common prediction features used in past work are fairly represented.

We first compute the common Halstead volume metric [62], which is commonly used for defect prediction and is widely implemented by measurement tools [82]:

$$N = \text{Methods called} + \text{Built-in function references} + \text{Operator instances} + \text{Variable references}$$

$$n = \text{Methods called by name} + \text{Built-in functions} + \text{Operator names} + \text{Variables}$$

$$\text{Halstead Volume} = N \cdot \log(n)$$

We next implement a file cohesion metric patterned after a class cohesion metric [82]:

$$\text{Cohesion} = \frac{\text{Global variables by method}}{\text{Global variables} \cdot \text{Methods}}$$

Finally, we implement a metric measuring lack of cohesion in methods (LCOM5) [111]:

$$\text{LCOM5} = \frac{\text{Methods} - \frac{\text{Global variables by method}}{\text{Global variables}}}{\text{Methods}}$$

In the machine learning evaluations later in this work, we will evaluate if adding these derived metrics to a predictive model results in an improvement in prediction performance, relative to a model which only uses the "raw" metrics (which include constituent parts of these derived metrics).

2.4 Distance measurement with families of metrics

Previously in this chapter, we introduced a system for constructing families of code and change metrics, and we gave several examples of metrics we implemented

in this system. Next, we turn our attention to the theoretical properties of these metric families; more specifically, the way they relate to the mathematical notion of a metric space.

The mathematical significance of the word *metric* is quite different from the way that the word is used in software measurement. In the mathematical sense, a metric is a function over some set that yields *distance measurements* over pairs of elements from the set. In order to distinguish the two meanings of the word, we will refer to this kind of metric in this paper as a *distance metric*.

Much past research in software metrics revolves around defining axioms [170] or properties [24] that software metrics should hold. These axioms are typically defined with the goal of validating prospective software metrics by ensuring that they are "well-behaved" in some specific sense. However, it has been criticized that axioms and properties of software metrics have largely been used to invalidate or criticize existing metrics, rather than to construct better metrics with the properties that researchers and practitioners want [98].

Past work in distance measurement and software metrics [120] has suggested that the properties of distance metrics can serve as initial axioms for software metrics, and that these axioms can be used in a *constructive* manner, allowing for abstract definitions of software characteristics to yield usable software metrics that also qualify as distance metrics in the mathematical sense of the word. In this section, we generalize this concept of software distance measurement (to a wide variety of metric families) and adapt it to measuring the change between two revisions of the same file.

2.4.1 Proof of distance pseudometrics in metric families

Earlier in this chapter, we introduced the concept of metric families – which can characterize a diverse variety of software properties through the definition of

labeling functions – and showed how every metric family includes both source code metrics and software change metrics, unifying code and change metrics under one construct. We next turn our attention to distance metrics, and demonstrate that the difference-based change metric defined in Section 2.2.4 also qualifies as a distance metric for *any* metric family, as long as the coefficients p are chosen properly.

Definition 2.10 (Distance pseudometric) *Let $d(x, y)$ be a distance function between x and y, where x and y are members of a set (in our case, the set of all revisions of a particular source code file). Function d is a pseudometric if it satisfies the following four properties for all x and y:*

$$\textit{Non-negativity: } d(x, y) \geq 0 \tag{2.1a}$$

$$\textit{Identity: } x = y \Rightarrow d(x, y) = 0 \tag{2.1b}$$

$$\textit{Symmetry: } d(x, y) = d(y, x) \tag{2.1c}$$

$$\textit{Triangle inequality: } d(x, z) \leq d(x, y) + d(y, z) \tag{2.1d}$$

Our difference-based change metrics will only satisfy the properties of a *pseudometric* rather than satisfying all the properties of a metric. In order to qualify as a full distance metric, the distance function must satisfy a fifth property $d(x, y) = 0 \Rightarrow x = y$. Because software metrics are typically only concerned with certain characteristics of a revision of a source code file (such as the file's method calls or variable definitions), there will be many different source code files that should be considered equivalent with respect to these characteristics, and have a zero distance between them. (This principle is similar to one of Weyuker's complexity metric axioms [170], which asserted that it must be possible for two non-identical programs to have the same complexity measurement.) Therefore, it is not always appropriate for software metrics to be full distance metrics, and we only consider distance pseudometrics in this work.

The four properties of distance pseudometrics above correspond with characteristics that software change metrics should intuitively possess. Measurements of software change should not be negative, and a change metric should register no change when comparing a revision of a file against itself. Software change metrics should register additions and deletions in a consistent way such that it is possible to construct a symmetric metric. (Although many software metric family members, such as separate metrics that only measure additions or deletions, may be non-symmetric, the existence of these symmetric metric family members assures that the addition-only and deletion-only metrics were constructed consistently.) Finally, comparing two software revisions which are separated by several versions should not register more change than the aggregate changes summed over the intermediate versions. In fact, over the course of analyzing software metric families and distance metrics, the author revised the construction process for software metric families several times – in other words, aligning software change metrics with the mathematical notion of a distance pseudometric made the definitions of the software metrics in this work more complete and consistent.

Proposition 2.1 (Software metrics as distance pseudometrics) *For any given metric family, the difference-based change metric* changemetric(T, U) *as defined in Definition 2.9 is a pseudometric if the following constraints on coefficients are sat-*

isfied:

$$p_1, p_2, p_3, p_4 \geq 0 \tag{2.2a}$$

$$p_1 = p_2 \tag{2.2b}$$

$$2 \cdot p_1 \geq p_4 \tag{2.2c}$$

$$2 \cdot p_1 \geq p_3 \tag{2.2d}$$

$$2 \cdot p_4 \geq p_3 \tag{2.2e}$$

$$2 \cdot p_3 \geq p_4 \tag{2.2f}$$

Note that changemetric is computed by summing over $\lambda \in \mathbf{L_T}$. It suffices to show that the quantity inside the summation is a pseudometric $\forall \lambda \in \mathbf{L_T}$, as the sum of a set of pseudometrics itself qualifies as a pseudometric. Throughout the proofs that follow, we will employ this strategy, leaving λ as a free variable.

In order to prove Proposition 2.1, we will first state several lemmas which will be helpful in multiple stages of the proof. The first two lemmas relate to characteristics of sets A and B as defined in Definition 2.5:

Lemma 2.1 (Symmetry of label sets) The first lemma relates to the effects of the node mapping and value mapping symmetry properties. Roughly speaking, if the direction in which two trees are compared is swapped, both trees will still share the same number of labels in common after labeling their trees with the metric family's labeling function:

$$\left|A_T^\lambda \cap B_{T \to U}^\lambda\right| = \left|A_U^{valuemap(T \to U, \lambda)} \cap B_{U \to T}^{valuemap(T \to U, \lambda)}\right| \tag{2.3}$$

Proof: First apply the definition of B to the above proposition, applying Property 2.4.2 to consolidate nested value mappings:

$$\left|A_T^\lambda \cup \left\{t \mid \exists u\,(t,u) \in nodemap(T \to U) \wedge u \in A_U^{valuemap(T\to U,\lambda)}\right\}\right| = \left|A_U^{valuemap(T\to U,\lambda)} \cup \left\{u \mid \exists t\,(u,t) \in nodemap(U \to T) \wedge t \in A_U^\lambda\right\}\right| \quad (2.4)$$

Consolidate the two sets:

$$\left|\left\{t \mid \exists u\,(t,u) \in nodemap(T \to U) \wedge t \in A_T^\lambda \wedge u \in A_U^{valuemap(T\to U,\lambda)}\right\}\right| = \left|\left\{u \mid \exists t\,(u,t) \in nodemap(U \to T) \wedge t \in A_T^\lambda \wedge u \in A_U^{valuemap(T\to U,\lambda)}\right\}\right| \quad (2.5)$$

Apply Property 2.3.3:

$$\left|\left\{t \mid \exists u\,(t,u) \in nodemap(T \to U) \wedge t \in A_T^\lambda \wedge u \in A_U^{valuemap(T\to U,\lambda)}\right\}\right| = \left|\left\{u \mid \exists t\,(t,u) \in nodemap(T \to U) \wedge t \in A_T^\lambda \wedge u \in A_U^{valuemap(T\to U,\lambda)}\right\}\right| \quad (2.6)$$

Note that per Property 2.3.1, only one t will satisfy Equation 2.6 for any u, and vice versa. Therefore, the two sets are the same size. □

Lemma 2.2 (Transitivity of label sets) This lemma relates to the effect of the node mapping and value mapping transitivity properties. If a succession of pairs of trees has identical labels for some λ, then the first and last trees in the succession also have identical labels:

$$A_Y^{valuemap(X\to Y,\lambda)} = B_{Y\to Z}^{valuemap(X\to Y,\lambda)} \wedge A_Z^{valuemap(X\to Z,\lambda)} = B_{Z\to Y}^{valuemap(X\to Y,\lambda)} \Rightarrow B_{X\to Y}^\lambda = B_{X\to Z}^\lambda \quad (2.7)$$

Proof: We will start by proving one half (the subset in one direction) of the implied set equality of Equation 2.7:

$$A_Y^{valuemap(X\to Y,\lambda)} = B_{Y\to Z}^{valuemap(X\to Y,\lambda)} \wedge A_Z^{valuemap(X\to Z,\lambda)} = B_{Z\to Y}^{valuemap(X\to Y,\lambda)} \Rightarrow B_{X\to Y}^{\lambda} \subseteq B_{X\to Z}^{\lambda} \quad (2.8)$$

To prove Equation 2.8, we begin with membership in $B_{X\to Y}^{\lambda}$ and expand the definition of B:

$$\left\{x \mid \exists y \in A_Y^{valuemap(X\to Y,\lambda)} (x,y) \in nodemap(X \to Y)\right\} \quad (2.9)$$

Per the left-hand-side of Equation 2.8, $A_Y^{valuemap(X\to Y,\lambda)} = B_{Y\to Z}^{valuemap(X\to Y,\lambda)}$. Therefore:

$$\forall y \in A_Y^{valuemap(X\to Y,\lambda)} \exists z\, (y,z) \in nodemap(Y \to Z) \wedge z \in A_Z^{valuemap(X\to Z,\lambda)} \quad (2.10)$$

Combining Equations 2.9 and 2.10:

$$\left\{x \mid \exists y \in A_Y^{valuemap(X\to Y,\lambda)}, \exists z \in A_Z^{valuemap(X\to Z)} (x,y) \in nodemap(X \to Y) \wedge (y,z) \in nodemap(Y \to Z)\right\} \quad (2.11)$$

Applying Property 2.3.4

$$\left\{x \mid \exists y \in A_Y^{valuemap(X\to Y,\lambda)}, \exists z \in A_Z^{valuemap(X\to Z)} (x,z) \in nodemap(X \to Z)\right\} \quad (2.12)$$

Simplifying and applying the definition of B, we arrive at $B^{\lambda}_{X\to Z}$, which proves Equation 2.8. Now we must prove:

$$A_Y^{valuemap(X\to Y,\lambda)} = B_{Y\to Z}^{valuemap(X\to Y,\lambda)} \wedge A_Z^{valuemap(X\to Z,\lambda)} = B_{Z\to Y}^{valuemap(X\to Y,\lambda)} \Rightarrow B^{\lambda}_{X\to Z} \subseteq B^{\lambda}_{X\to Y} \quad (2.13)$$

The proof of Equation 2.13 is the same as the proof of Equation 2.8 with Y and Z swapped. Therefore Equation 2.7 is proven. □

Lemma 2.3 (Mutual exclusion of metric predicates) Note that five predicates were defined in Definition 2.6. We now must show that these five predicates (ADDONLY, DELONLY, ADDDEL, MOD, NONE) are mutually exclusive and cover all cases – in other words, that for any T and U, exactly one of the five predicates is true.

Proof: It is trivial that ADDONLY and DELONLY are mutually exclusive and also exclude ADDDEL and MOD, because these predicates disagree as to if $A^{\lambda}_T = \emptyset$ and/or $A^{\lambda}_U = \emptyset$. It is also trivial that ADDDEL and MOD are mutually exclusive, because they disagree as to if $A^{\lambda}_T \cap B^{\lambda}_{T\to U} = \emptyset$.

To prove the lemma, we must now show that NONE holds iff the four other predicates do not hold.

$$\text{NONE}^{\lambda}_{T\to U} \Leftrightarrow \neg\text{ADDONLY}^{\lambda}_{T\to U} \wedge \neg\text{DELONLY}^{\lambda}_{T\to U} \wedge \neg\text{ADDDEL}^{\lambda}_{T\to U} \wedge \neg\text{MOD}^{\lambda}_{T\to U} \quad (2.14)$$

First, we replace predicates with their definitions and simplify, ensuring that none of the predicates on the right-hand-side can be satisfied:

$$A_T^\lambda = B_{T\to U}^\lambda \wedge A_U^{valuemap(T\to U,\lambda)} = B_{U\to T}^{valuemap(T\to U,\lambda)} \Leftrightarrow$$
$$\Big(A_T^\lambda \cap B_{T\to U}^\lambda = \emptyset \Rightarrow A_T^\lambda = A_U^{valuemap(T\to U,\lambda)} = \emptyset \wedge$$
$$A_T^\lambda \cap B_{T\to U}^\lambda \neq \emptyset \Rightarrow A_T^\lambda = B_{T\to U}^\lambda \wedge A_U^{valuemap(T\to U,\lambda)} = B_{U\to T}^{valuemap(T\to U,\lambda)}\Big) \quad (2.15)$$

Simplifying further:

$$A_T^\lambda = B_{T\to U}^\lambda \wedge A_U^{valuemap(T\to U,\lambda)} = B_{U\to T}^{valuemap(T\to U,\lambda)} \Leftrightarrow$$
$$\Big(A_T^\lambda \cap B_{T\to U}^\lambda = \emptyset \wedge A_T^\lambda = A_U^{valuemap(T\to U,\lambda)} = \emptyset\Big) \vee$$
$$\Big(A_T^\lambda \cap B_{T\to U}^\lambda \neq \emptyset \wedge A_T^\lambda = B_{T\to U}^\lambda \wedge A_U^{valuemap(T\to U,\lambda)} = B_{U\to T}^{valuemap(T\to U,\lambda)}\Big) \quad (2.16)$$

$$A_T^\lambda = B_{T\to U}^\lambda \wedge A_U^{valuemap(T\to U,\lambda)} = B_{U\to T}^{valuemap(T\to U,\lambda)} \Leftrightarrow$$
$$\Big(A_T^\lambda = A_U^{valuemap(T\to U,\lambda)} = \emptyset\Big) \vee$$
$$\Big(A_T^\lambda \neq \emptyset \wedge A_T^\lambda = B_{T\to U}^\lambda \wedge A_U^{valuemap(T\to U,\lambda)} = B_{U\to T}^{valuemap(T\to U,\lambda)}\Big) \quad (2.17)$$

This can be rewritten as

$$A_T^\lambda = B_{T\to U}^\lambda \wedge A_U^{valuemap(T\to U,\lambda)} = B_{U\to T}^{valuemap(T\to U,\lambda)} \Leftrightarrow$$
$$\Big(A_T^\lambda = A_U^{valuemap(T\to U,\lambda)} = \emptyset \vee A_T^\lambda \neq \emptyset\Big) \wedge A_T^\lambda = B_{T\to U}^\lambda \wedge A_U^{valuemap(T\to U,\lambda)} = B_{U\to T}^{valuemap(T\to U,\lambda)} \quad (2.18)$$

From Lemma 2.1 we can conclude that

$$A_T^\lambda = B_{T\to U}^\lambda \wedge A_U^{valuemap(T\to U,\lambda)} = B_{U\to T}^{valuemap(T\to U,\lambda)} \wedge A_T^\lambda = \emptyset \Rightarrow A_U^{valuemap(T\to U,\lambda)} = \emptyset \tag{2.19}$$

because $\left|\lambda_T \cap B_{T\to U}^\lambda\right| = 0$ and hence $\left|valuemap\left(T \to U, \lambda\right)_U \cap B_{U\to T}^{valuemap(T\to U,\lambda)}\right| = 0$. So Equation 2.18 can be rewritten as:

$$\begin{aligned} &A_T^\lambda = B_{T\to U}^\lambda \wedge A_U^{valuemap(T\to U,\lambda)} = B_{U\to T}^{valuemap(T\to U,\lambda)} \Leftrightarrow \\ &\left(A_T^\lambda = \emptyset \vee A_T^\lambda \neq \emptyset\right) \wedge A_T^\lambda = B_{T\to U}^\lambda \wedge A_U^{valuemap(T\to U,\lambda)} = B_{U\to T}^{valuemap(T\to U,\lambda)} \end{aligned} \tag{2.20}$$

□

Lemma 2.4 (Symmetric metric predicates) The following equations relate to the symmetry of the predicates from Definition 2.6. Symmetry relates to the change in a set of labels when comparing one tree to another (for example, when an element is added when going from tree T to tree U) versus the reverse (for example, when the same element is removed when going from tree U to tree T).

$$\text{ADDONLY}_{T\to U}^\lambda \Leftrightarrow \text{DELONLY}_{U\to T}^{valuemap(T\to U,\lambda)} \tag{2.21}$$

$$\text{ADDDEL}_{T\to U}^\lambda \Leftrightarrow \text{ADDDEL}_{U\to T}^{valuemap(T\to U,\lambda)} \tag{2.22}$$

$$\text{NONE}_{T\to U}^\lambda \Leftrightarrow \text{NONE}_{U\to T}^{valuemap(T\to U,\lambda)} \tag{2.23}$$

$$\text{MOD}_{T\to U}^\lambda \Leftrightarrow \text{MOD}_{U\to T}^{valuemap(T\to U,\lambda)} \tag{2.24}$$

Proof of Equation 2.21: Replacing predicates with their definitions and applying Property 2.4.2, we arrive at:

$$A_T^\lambda \cap B_{T\to U}^\lambda = \emptyset \wedge A_T^\lambda = \emptyset \wedge A_U^{valuemap(T\to U,\lambda)} \neq \emptyset \Leftrightarrow$$
$$A_U^{valuemap(T\to U,\lambda)} \cap B_{U\to T}^{valuemap(T\to U,\lambda)} = \emptyset \wedge A_T^\lambda = \emptyset \wedge A_U^{valuemap(T\to U,\lambda)} \neq \emptyset \quad (2.25)$$

As a consequence of Lemma 2.1, $A_T^\lambda \cap B_{T\to U}^\lambda = \emptyset \Leftrightarrow A_U^{valuemap(T\to U,\lambda)} \cap B_{U\to T}^{valuemap(T\to U,\lambda)} = \emptyset$, which proves Equation 2.25 □

Proof of Equation 2.22: Replacing predicates and applying Property 2.4.2, as it is applied when proving Equation 2.25:

$$A_T^\lambda \cap B_{T\to U}^\lambda = \emptyset \wedge A_T^\lambda = \emptyset \wedge A_U^{valuemap(T\to U,\lambda)} \neq \emptyset \wedge Q_T^\lambda \neq \emptyset \Leftrightarrow$$
$$A_U^{valuemap(T\to U,\lambda)} \cap B_{U\to T}^{valuemap(T\to U,\lambda)} = \emptyset \wedge A_T^\lambda = \emptyset \wedge A_U^{valuemap(T\to U,\lambda)} \neq \emptyset \wedge Q_T^\lambda \neq \emptyset \quad (2.26)$$

This can be proven with Lemma 2.1 as in the previous case. □

Proof of Equation 2.23: Replacing predicates and applying Property 2.4.2. we get the tautology:

$$A_T^\lambda = B_{T\to U}^\lambda \wedge A_U^{valuemap(T\to U,\lambda)} = B_{U\to T}^{valuemap(T\to U,\lambda)} \Leftrightarrow$$
$$A_T^\lambda = B_{T\to U}^\lambda \wedge A_U^{valuemap(T\to U,\lambda)} = B_{U\to T}^{valuemap(T\to U,\lambda)} \quad (2.27)$$

□

Proof of Equation 2.24: This is trivial given the previous three symmetry equations and Lemma 2.3. □

Lemma 2.5 (Transitivity of unchanged labels) Here, we prove that if a set of labels remained unchanged when moving across two pairs of trees, then the set will also remain unchanged when moving across both at once – in other words, that a lack of change between trees is transitive.

$$\text{NONE}^{\lambda}_{X\to Y} \wedge \text{NONE}^{valuemap(X\to Y,\lambda)}_{Y\to Z} \Rightarrow \text{NONE}^{\lambda}_{X\to Z} \tag{2.28}$$

Applying the definition of NONE and Property 2.4.2 to Equation 2.28:

$$\begin{aligned} A^{\lambda}_{X} = B^{\lambda}_{X\to Y} \wedge A^{valuemap(X\to Y,\lambda)}_{Y} = B^{valuemap(X\to Y,\lambda)}_{Y\to X} = B^{valuemap(X\to Y,\lambda)}_{Y\to Z} \wedge \\ A^{valuemap(X\to Z,\lambda)}_{Z} = B^{valuemap(X\to Z,\lambda)}_{Z\to Y} \Rightarrow \\ A^{\lambda}_{X} = B^{\lambda}_{X\to Z} \wedge A^{valuemap(X\to Z,\lambda)}_{Z} = B^{valuemap(X\to Z,\lambda)}_{Z\to X} \end{aligned} \tag{2.29}$$

We will begin by proving w.l.o.g:

$$\begin{aligned} A^{\lambda}_{X} = B^{\lambda}_{X\to Y} \wedge A^{valuemap(X\to Y,\lambda)}_{Y} = B^{valuemap(X\to Y,\lambda)}_{Y\to Z} \wedge \\ A^{valuemap(X\to Z,\lambda)}_{Z} = B^{valuemap(X\to Z,\lambda)}_{Z\to Y} \Rightarrow A^{\lambda}_{X} = B^{\lambda}_{X\to Z} \end{aligned} \tag{2.30}$$

Applying Lemma 2.2 to Equation 2.30, we conclude that:

$$B^{\lambda}_{X\to Y} = B^{\lambda}_{X\to Z} \tag{2.31}$$

So $A^{\lambda}_{X} = B^{\lambda}_{X\to Z}$ which proves Equation 2.30. To prove Equation 2.29, we now must prove:

$$A_X^\lambda = B_{X\to Y}^\lambda \wedge A_Y^{valuemap(X\to Y,\lambda)} = B_{Y\to X}^{valuemap(X\to Y,\lambda)} \wedge$$
$$A_Z^{valuemap(X\to Z,\lambda)} = B_{Z\to Y}^{valuemap(X\to Z,\lambda)} \Rightarrow A_Z^{valuemap(X\to Z,\lambda)} = B_{Z\to X}^{valuemap(X\to Z,\lambda)} \quad (2.32)$$

In Equation 2.32, swap variables X and Z. Then replace λ with $valuemap\,(X \to Z, \lambda)$ and apply Properties 2.4.2 and 2.4.3. This results in Equation 2.30 which has already been proven. □

Lemma 2.6 (Transitivity of modification) In our final lemma, we will show that if a set of labels is modified across one pair of trees and then remains unchanged across another pair of trees, then the set of labels will still be modified when going across both pairs of trees at once:

$$\text{MOD}_{X\to Y}^\lambda \wedge \text{NONE}_{Y\to Z}^\lambda \Rightarrow \text{MOD}_{X\to Z}^\lambda \vee \text{NONE}_{X\to Z}^\lambda \quad (2.33)$$

A stronger version of this lemma could be proven (in particular, $\text{NONE}_{X\to Z}^\lambda$ could be ruled out), but this weaker version is sufficient for this distance pseudometric proof. To prove Equation 2.33, it suffices to show that:

$$A_X^\lambda \cap B_{X\to Y}^\lambda \neq \emptyset \wedge A_Y^{valuemap(X\to Y,\lambda)} = B_{Y\to Z}^{valuemap(X\to Y,\lambda)} \wedge$$
$$A_Z^{valuemap(Y\to Z,valuemap(X\to Y,\lambda))} = B_{Y\to Z}^{valuemap(Y\to Z,valuemap(X\to Y,\lambda))} \Rightarrow A_X^\lambda \cap B_{X\to Z}^\lambda \neq \emptyset \quad (2.34)$$

Apply Property 2.4.3 and Lemma 2.2. We can conclude that:

$$B_{X\to Y}^\lambda = B_{X\to Z}^\lambda \quad (2.35)$$

Combining Equation 2.34 with Equation 2.35:

$$A_X^\lambda \cap B_{X \to Z}^\lambda \neq \emptyset \tag{2.36}$$

□

Theorem 2.1 (Non-negatitivity of metric family members) We can now prove Equation 2.1a, the first property that changemetric(T, U) must satisfy to qualify as a distance pseudometric.

$$\text{changemetric}(T, U) \geq 0 \tag{2.37}$$

As discussed previously, we must only prove that each property is satisfied for any given T, U, and λ. Equation 2.37 is satisfied by the constraint in Equation 2.2a. □

Theorem 2.2 (Identity of metric family members) Equation 2.1b stipulates that identical revisions will have zero distance between them:

$$T = U \Rightarrow \text{changemetric}(T, U) = 0 \tag{2.38}$$

To prove that this is satisfied, we must prove $\forall\lambda$:

$$\text{NONE}_{T \to T}^\lambda \tag{2.39}$$

It suffices to show that:

$$A_T^\lambda = B_{T \to T}^\lambda \tag{2.40}$$

Replacing B with its definition:

$$A_T^\lambda = \left\{ t \mid \exists u \, (t, u) \in nodemap \, (T \to T) \wedge u \in A_T^{valuemap(T \to T, \lambda)} \right\} \tag{2.41}$$

Applying Properties 2.3.2 and 2.4.1:

$$A_T^\lambda = \left\{ t \mid \exists u \, t = u \wedge t \in T \wedge t \in A_T^\lambda \right\} \tag{2.42}$$

Which is satisfied because $A_T^\lambda \subseteq T$. $\square$

Recall from Definition 2.9 that T and U are the trees corresponding to two different revisions of the same file. This property establishes that any given revision of any given file has a zero distance to itself. Significantly, this property does *not* establish that two identical abstract syntax trees have zero distances to each other. Recall that the distance between two trees is determined by both the trees' labels and the trees themselves, and that the labels can depend on information not found within the tree (such as a file name f or a lookup table lookup.

Theorem 2.3 (Symmetry of metric family members) To satisfy the property described by Equation 2.1c, we must prove that for any T and U:

$$\text{changemetric}(T, U) = \text{changemetric}(U, T) \tag{2.43}$$

Per Lemma 2.3, only one of the predicates of Definition 2.6 can be satisfied at once. This means that Equation 2.43 can be proven be considering each predicate in turn:

- ADDONLY$^\lambda_{T \to U}$: Per Equation 2.21 of Lemma 2.4, DELONLY$^\lambda_{U \to T}$. The condition in Equation 2.2b satisfies Equation 2.43 here.
- DELONLY$^\lambda_{T \to U}$: This is a mirror image of the previous case.

- $\text{ADDDEL}^{\lambda}_{T \to U}$: This case is satisfied by Equation 2.22 of Lemma 2.4.
- $\text{MOD}^{\lambda}_{T \to U}$: This case is satisfied by Equation 2.24 of Lemma 2.4.
- $\text{NONE}^{\lambda}_{T \to U}$: Per Lemma 2.3, by exclusion, $\text{NONE}^{\lambda}_{U \to T}$ must be true, because otherwise a different predicate would have to satisfy $T \to U$.

  These five cases collectively satisfy Equation 2.43 □

Theorem 2.4 (Triangle inquality of metric family members) The property described by Equation 2.1d specifies that the distance across two pairs of trees cannot be less than the distance across the whole:

$$\text{changemetric}(X, Z) \leq \text{changemetric}(X, Y) + \text{changemetric}(Y, Z) \tag{2.44}$$

As in Theorem 2.3, we will prove this by considering each case individually. Because Definition 2.6 defines 5 predicates, there are $5 \cdot 5 \cdot 5 = 125$ possible cases. We must show that each case is either:

- **Infeasible**: These three predicates cannot occur together for any X,Y,Z, or
- **Satisfied**: If these three predicates occur together, the triangle inequality is satisfied by the constraints in Proposition 2.1.

One example of an infeasible case is $\text{NONE}^{\lambda}_{X \to Y} \wedge \text{NONE}^{valuemap(X \to Y, \lambda)}_{Y \to Z} \wedge \text{ADDONLY}^{\lambda}_{X \to Z}$ – combining two changes that do not add a value cannot result in a value being added across the two changes. An example of a satisfied case is $\text{ADDONLY}^{\lambda}_{X \to Y} \wedge \text{ADDDEL}^{valuemap(X \to Y, \lambda)}_{Y \to Z} \wedge \text{ADDONLY}^{\lambda}_{X \to Z}$, which is satisfied by Equation 2.2a (ADDDEL cannot reduce the total distance).

Predicate clause set: We will show that many cases are infeasible because they are contradicted by a predicate clause set. A predicate clause set consists of

constraints imposed by previous lemmas. The first predicate clause set is due to Lemma 2.5:

$$\neg\text{NONE}^{\lambda}_{X\to Y} \vee \neg\text{NONE}^{valuemap(X\to Y,\lambda)}_{Y\to Z} \vee \text{NONE}^{\lambda}_{X\to Z}$$

The second is due to Lemma 2.6:

$$\neg\text{MOD}^{\lambda}_{X\to Y} \vee \neg\text{NONE}^{valuemap(X\to Y,\lambda)}_{Y\to Z} \vee \text{MOD}^{\lambda}_{X\to Z} \vee \text{NONE}^{\lambda}_{X\to Z}$$

Closure of predicate clause set: We can compute the closure of the above clause set by applying two operations:

- **Swapping variables**: Variables X, Y, or Z can be swapped wherever they appear in a clause.
- **Reversing predicates**: The initial clause set only contains predicates NONE and MOD. It has been shown that both of these predicates are symmetric – $\text{MOD}^{\lambda}_{T\to U} \Leftrightarrow \text{MOD}^{valuemap(T\to U,\lambda)}_{U\to T}$ and $\text{NONE}^{\lambda}_{T\to U} \Leftrightarrow \text{NONE}^{valuemap(T\to U,\lambda)}_{U\to T}$ – in Lemma 2.4 and Theorem 2.3. Therefore, any given predicate in a clause set can be reversed without swapping variables in other predicates.

By computing the closure of the predicate clause set under these two operations, we find that many cases are infeasible because they are contradicted by some member of this set.

Cases with trivially contradictory predicates: For any predicate, it is apparent from the predicate's definition if either of the trees' A sets are empty:

- $\text{NONE}^{\lambda}_{T\to U}$: $A^{\lambda}_{T} = \emptyset \Leftrightarrow A^{valuemap(T\to U,\lambda)}_{U} = \emptyset$ – this follows from Definition 2.5.
- $\text{ADDONLY}^{\lambda}_{T\to U}$: $A^{\lambda}_{T} = \emptyset \wedge A^{valuemap(T\to U,\lambda)}_{U} \neq \emptyset$

Table 2.3: Feasible cases for proving the triangle inequality of the distance-based change metric

$X \to Y$	$Y \to Z$	$X \to Z$	*Satisfying constraint(s)*
ADDONLY	ADDDEL	ADDONLY	2.2a
ADDONLY	DELONLY	NONE	2.2a
ADDONLY	MOD	ADDONLY	2.2a
ADDONLY	NONE	ADDONLY	2.2a
ADDDEL	ADDDEL	ADDDEL	2.2a
ADDDEL	ADDDEL	MOD	2.2f
ADDDEL	ADDDEL	NONE	2.2a
ADDDEL	DELONLY	DELONLY	2.2a
ADDDEL	MOD	ADDDEL	2.2a
ADDDEL	MOD	MOD	2.2a
ADDDEL	NONE	ADDDEL	2.2a
DELONLY	ADDONLY	ADDDEL	2.2b, 2.2d
DELONLY	ADDONLY	MOD	2.2b, 2.2c
DELONLY	NONE	DELONLY	2.2a
MOD	DELONLY	DELONLY	2.2a
MOD	MOD	ADDDEL	2.2e
MOD	MOD	MOD	2.2a
MOD	MOD	NONE	2.2a
MOD	NONE	MOD	2.2a
NONE	NONE	NONE	Def 2.9

- $\text{DELONLY}^{\lambda}_{T \to U}$: $A^{\lambda}_{T} \neq \emptyset \wedge A_{U}^{valuemap(T \to U, \lambda)} = \emptyset$
- $\text{ADDDEL}^{\lambda}_{T \to U}$: $A^{\lambda}_{T} \neq \emptyset \wedge A_{U}^{valuemap(T \to U, \lambda)} \neq \emptyset$
- $\text{MOD}^{\lambda}_{T \to U}$: $A^{\lambda}_{T} \neq \emptyset \wedge A_{U}^{valuemap(T \to U, \lambda)} \neq \emptyset$

This allows for some combinations of predicates to be trivially ruled out because they are incompatible. For example, $\text{ADDONLY}^{\lambda}_{X \to Y} \wedge \text{ADDONLY}^{\lambda}_{Y \to Z}$ is a contradiction because it contradicts if $A_{Y}^{valuemap(X \to Y, \lambda)} = \emptyset$. Hence, we determine that some cases are infeasible because their predicates are trivially contradictory (can be shown as being contradictory without recourse to any of the lemmas or theorems which we have described).

Equivalent cases: If it is shown that the triangle inequality is satisfied in any given case, then the triangle inequality must also be satisfied in the case that

is produced by swapping X and Y, as this case only differs in that the two distance measurements are added in a different order (as the arithmetic sum operator is symmetric). Therefore, we do not need to prove that both equivalent cases are satisfied – only one or the other.

With the aid of a computer program, we evaluated all 125 cases and eliminated cases that were contradicted by the closure of the predicate clause set, trivially contradictory, or equivalent to another case which was not eliminated. The cases which remain (which represent cases for which the triangle inequality must hold) are shown in Table 2.3. Next to each case, the constraint(s) which cause the case to be satisfied are shown. Because all cases are satisfied, the theorem is proven and the distance-based change metric is shown to be a distance pseudometric. □

2.4.2 Code delta metrics and distance-based change metrics

In Section 2.2.4, we introduced a method for constructing software change metrics to quantify the change of some quality of a source code file between one revision and another. Notably, any quantity that could be represented with a labeling function could be used to build both a static code metric and a family of change metrics, and many members of the family of change metrics also qualified as distance metrics in the mathematical sense (or, more specifically, pseudometrics).

A code delta metric was defined in Definition 2.8. It is easy to show that the code delta metric qualifies as a distance pseudometric, and it is much simpler to reason about it than the difference-based change metric, which the preceding theorems and lemmas analyzed. The code delta metric has been used in the past [56] to characterize various aspects of software change.

The drawback of the code delta metric is that is incapable of characterizing a particular, intuitive notion of software change that a change metric should satisfy. Consider a change metric that quantified addition and deletion of functions in a file.

The code-delta-based metric would measure the absolute difference in the number of functions between two revisions of a file. However, if all of the functions were removed from a file in one revision and replaced with the same number of (completely different) functions in the next, the code delta metric would indicate that the file had not changed at all.

Here, we show that a simple code churn metric cannot be replaced by any code delta metric, regardless of what static code metrics were used to build the code delta metric, even for very small programs, and even if the code delta metric is generalized to a higher-dimensionality Euclidean space.

Definition 2.11 (Generalized code delta metric) *Let $m_1(T), m_2(T), \ldots m_i(T)$ be a set of source code metrics that will be used to build a generalized code delta metric (the functions m can be defined in any manner, including with the* PRESENT *as described in Definition 2.8, or in another way – there is no need to use the system that was proposed in this work). The code delta metric for two revisions of a file T and U is then defined as:*

$$\delta(T,U) \equiv \sqrt{\left(m_1\left(T\right) - m_1\left(U\right)\right)^2 + \left(m_2\left(T\right) - m_2\left(U\right)\right)^2 + \cdots + \left(m_i\left(T\right) - m_i\left(U\right)\right)^2}$$

The traditional code delta metric is a special case of this generalized code delta metric with $i = 1$. The function defined in Definition 2.11 is the Euclidean norm. Therefore, the vector of source code metrics m are a metric embedding of source code trees into an i-dimensional Euclidean space where each tree has coordinates $(m_1(T), m_2(T), \ldots m_i(T))$.

Consider a simple code churn metric, which measures the difference between two files as number of lines that would have to be added, removed, or modified to transform one file into another. (For the sake of simplicity, this example will only

Table 2.4: Four versions of a simple program for a code churn example

a	b	b	d
`var a = 1;`	`var a = 5;`	`var a = 5;`	`var a = 5;`
`var b = 2;`	`var b = 2;`	`var b = 6;`	`var b = 2;`
`var c = 3;`	`var c = 3;`	`var c = 3;`	`var c = 7;`

Table 2.5: Distances between versions of the program shown in Table 2.4

dist	A	B	C	D
A	0	1	2	2
B	1	0	1	1
C	2	1	0	2
D	2	1	2	0

involve modified lines). Table 2.4 depicts 4 versions of a simple 3-line program that will be used to illustrate this example.

The pairwise distances between the 4 versions of the program can be measured by counting the number of lines that are different between each pair of versions. A matrix of these measurements is shown in Table 2.5. We investigate if $\exists i, m_1, m_2, \ldots, m_i$ such that the generalized code delta metric δ can match these distances. We will prove by contradiction that no such collection of static code metrics exists.

Proof: Let δ be the code delta metric that replicates the desired code churn distances. Consider that in a Euclidean space, three points x, y, z are collinear iff $d(x, y) + d(y, z) = d(x, z)$. Therefore, embedded into the Euclidean space, A, B, C are collinear because $\delta(A, B) = 1$, $\delta(B, C) = 1$, $\delta(A, C) = 2$. For the same reason, A, B, D are collinear. Both C and D are collinear with $A$and B. So, A, B, C, D must be collinear. However, B, C, D are not collinear because $\delta(B, C) = 1$, $\delta(C, D) = 2$, $\delta(B, D) = 1$ which is a contradiction. □

The significance of this proof is that no generalized code delta metric can replicate a simple code churn metric, even when the size of the program is bounded and i is arbitrary large. The system of metric families that we defined in this

chapter is able to replicate such metrics, demonstrating that our distance-based change metrics align more closely with the intuitive notion of change, while still allowing for a wide variety of source code phenomena to be measured as change metrics, and ensuring that the resulting change metrics are also distance metrics in the mathematical sense.

2.5 Implementation details

Earlier in this chapter, we presented a system for defining families of code and change metrics. We then listed a number of such metric families which we implemented for the purposes of this study, and we then explored the theoretical properties of these metric families from the standpoint of distance measurement. In this final section, we turn our attention to the implementation details, or algorithms, required to complete these metrics efficiently.

Recall that constructing difference-based change metrics using our metric family construction system entails comparing two versions of a product with each other. Such a comparison will yield three mappings:

1. **File mappings**: Mappings between files in two revisions (Section 2.2.2.1)
2. **Node mappings**: Mappings between AST nodes in two files (Section 2.2.2.2)
3. **Value mappings**: Mappings between the vocabularies of AST labeling functions as applied to two files (Section 2.2.2.3)

Collectively, these three mappings define how the changes made between two revisions are characterized, which influences the difference-based change metrics that result. A number of properties have been described in the previous sections which place some constraints on how the mappings are performed. However, it is possible

to trivially satisfy these properties without performing a meaningful mapping from a measurement standpoint.

For example, a trivial node mapping can be defined which yields an empty mapping (unless a tree is being compared with itself). Such a mapping would satisfy all of the properties required from a node mapping, and the difference-based change metric that resulted would qualify as a distance metric; however, the change metrics would treat all code as having been deleted and re-created in every revision.

Another possible node mapping that would trivially satisfy all our properties involves imposing a total ordering on each tree, and then mapping the first nodes to each other, the second nodes to each other, and so forth. This mapping would falsely report that large segments of trees had changed when they had not actually changed (because the mapped sequences of nodes would become misaligned at some point).

In the following sections, we discuss practical considerations of choosing mapping algorithms that satisfy the required properties and behave in an expected manner.

2.5.1 File mapping algorithms

A file mapping algorithm serves the purpose of determining which files in two different revisions of the same product are "the same", even when files are added, removed, or renamed. An optimal file mapping between two revisions could, for example, minimize the aggregate edit distance between all pairs of matched trees plus the number of nodes in trees that were unmatched. Two practical problems may arise when building file mappings: (1) The node mapping algorithm relies on the file mapping already having been performed; jointly optimizing the node and file mappings would come at the cost of increased computational complexity. (2) The file mapping must still satisfy all the properties in Section 2.2.2.1, which is not

always guaranteed if the file mapping algorithm only uses local (to the two revisions being compared) information.

The `git` version control tool contains a file mapping algorithm which detects when a file in the repository has been renamed. When matching files between two revisions, the algorithm first looks for filenames which are part of one revision but not another. The similarity of these files is then compared; a potential rename is detected if the similarity (proportion of the original file that remains) rises above a predetermined threshold. Although the `git` rename detection functionality is based on measuring the proportion of changed text (rather then the proportion of changed tree nodes), it is not necessary to parse the files before determining how they are mapped – doing so does not help satisfy any of the required properties in Section 2.2.2.1.

Another problem relates to satisfying the symmetry (2.2.3) and transitivity (2.2.4) properties for file mappings – there is no guarantee that the `git` rename detection will behave in such a way that these properties will automatically be satisfied. This problem can be addressed with a *wrapper* approach, where any file mapping algorithm can be invoked by a wrapper algorithm that ensures these properties are satisfied.

For the wrapper algorithm that we describe here, we assume that revisions form a tree with a single root node (which must be the earliest known revision of the program). If this is not the case, such a tree can easily be constructed with a spanning tree over the revision graph.

In the case where a revision is compared to itself, return an identity mapping:

$$\mathit{filemap}(\mathbf{R}_1 \to \mathbf{R}_1) = \{(f, f) \mid f \in \mathbf{R}_1\}$$

Let $\mathit{origmap}(\mathbf{R}_1 \to \mathbf{R}_2)$ be the original file mapping algorithm that is to be wrapped. Let $\mathit{parent}(\mathbf{R}_1)$ be the parent of a revision and $\mathit{ancestors}(\mathbf{R}_1)$ be the set

of all ancestor revisions of a revision. We next consider the case where a revision is being compared to its parent (moving "forward" through the revision history). In this case, the wrapped mapping function is simply called.

$$\mathit{filemap}(\mathit{parent}(\mathbf{R}_1) \rightarrow \mathbf{R}_1) = \mathit{origmap}(\mathit{parent}(\mathbf{R}_1) \rightarrow \mathbf{R}_1)$$

In the case where a revision is being compared to its descendant, the direction of the comparison and the results are reversed. This guarantees Property 2.2.3:

$$\mathit{filemap}(\mathbf{R}_1 \rightarrow \mathbf{R}_2 \in \mathit{ancestors}(\mathbf{R}_1)) = \{(x, y) \mid (y, x) \in \mathit{filemap}\,(\mathbf{R}_2 \rightarrow \mathbf{R}_1)\}$$

In the cases that remain, either the revision is being compared to an ancestor other than its parent, or there exists a common ancestor with the compared revision as its descendant. In either case, two comparisons are transitively combined and mappings are preserved across all three of the revisions in question, recursively comparing revisions until the desired starting and ending points are reached.

$$\begin{aligned}\mathit{filemap}(\mathbf{R}_1 \rightarrow \mathbf{R}_2) = \big\{(x, z) \mid (y, z) \in \mathit{origmap}(\mathit{parent}(\mathbf{R}_2) \rightarrow \mathbf{R}_2) \wedge \\ (x, y) \in \mathit{filemap}(\mathbf{R}_1 \rightarrow \mathit{parent}(\mathbf{R}_2))\big\}\end{aligned}$$

This wrapper algorithm ensures that any file mapping algorithm will satisfy all the properties described in Section 2.2.2.1, assuming that Property 2.2.1 is directly satisfied by the algorithm. In our file mapping implementation for the experiments described in this work, we effectively wrap Git's rename detection algorithm in such a manner.

2.5.2 Tree node mapping algorithms

The problem of comparing ordered trees has been thoroughly studied [18, 150, 176]. Algorithms for comparing two trees generally produce an *edit script* or an alignment between the trees [18]. An edit script is a sequence of node addition, removal, and relabeling operations that transforms the first tree into the second. An alignment between two trees inserts special "space" nodes into both trees such that the trees are structurally isomorphic, and the distance between two trees is determined by comparing pairs of nodes which are in the same positions. Both edit scripts and alignments can be used to produce a mapping between the two trees, or tuples similar to this described in Section 2.2.2.2 which link nodes that the two trees have in common. Although a full exploration of tree comparison algorithms is beyond the scope of this work, such algorithms are clearly applicable for comparing two revisions of a source code file.

Earlier in this section, we discussed the importance of ensuring that (beyond satisfying the properties of Section 2.2.2.2) tree mappings are "well-behaved", in the sense that there should be some criteria for deciding that a node in one tree is eligible to be mapped to a node in another. Tai Mappings [150] are mappings between trees that satisfy three properties:

1. Mappings between trees are one-to-one
2. Sibling order is preserved (one node is to the left of another iff the same is true in the mapped tree)
3. Ancestry is preserved (one node is an ancestor of another iff the same is true in the mapped tree)

If relabeling operations in edit scripts are disallowed (a restriction of the tree edit distance problem which can be accomplished by ensuring that relabeling is no less expensive than deletion and insertion [158]), a fourth property is satisfied – that

mapped nodes have identical properties. Note that "relabeling" here is not related to the tree node labeling functions which we had previously discussed – this is a different use of the word "label". For the purposes of comparing trees, the "label" of a node should be built from properties taken from the source code. Such properties may include the abstract syntax tree node type, the name of an operator, or the contents of a string literal. In general, all properties should be included in a node's label; however, if a property will participate in a value mapping (as described in Section 2.2.2), it should not be included in a tree comparison label, as a value mapping will serve no purpose if nodes with different labels cannot be mapped to each other.

Collectively, the four properties above define one sense of how tree mappings may be "well-behaved". However, Tai Mappings are not guaranteed to satisfy the properties that we describe in Section 2.2.2.2, which are required to ensure that the software change metrics also qualify as distance metrics. We describe two approaches for addressing this:

2.5.2.1 Edit-union tree mapping method

Torsello and Hancock [158] introduce an algorithm for computing the *edit-union* of a set of trees. An edit-union is a directed acyclic graph which is constructed such that all trees in the set are substructures of the edit-union structure. Note that any number of trees can be combined in this way, rather than just two. The edit-union is used to quickly compute the similarity between one tree and a large set of trees by minimizing the edit distance between the tree being compared and all trees in the set.

The same edit-union and matching method can be used to compare multiple revisions of a source code file while satisfying the properties of Section 2.2.2.2. First, construct an edit-union structure which includes all revisions of the file in question.

Each node of the edit-union represents an equivalence class, where no more than one tree node from each revision (the tree node that maps to the given node of the edit-union) is a member of the class. When comparing two revisions of a file, the node mapping *nodemap* is constructed from pairs of nodes which are members of the same equivalence class. This ensures that all properties of Section 2.2.2.2 are satisfied and the tree mapping is also well-behaved. However, the edit-union structure is not guaranteed to be optimal (neither globally – the total size of the edit-union structure is not minimized – nor locally – the tree mappings produced in such a way are not guaranteed to map as many pairs of nodes as possible).

2.5.2.2 Wrapper tree mapping method

As an alternative to using a specialized edit-union structure, a well-behaved tree differencing algorithm which compares pairs of trees can be wrapped so it satisfies all properties described in Section 2.2.2.2. Because the wrapping algorithm for tree nodes is identical to the wrapping algorithm for files, we refer the reader to Section 2.5.1 for more information on this.

The tree mappings that result from this method will be as well-behaved as the tree differencing algorithm makes them. They are also guaranteed to satisfy the properties of Section 2.2.2.2, regardless of how the underlying tree differencing algorithm works. The disadvantage of taking this approach is that the tree differences will become increasingly suboptimal (in the sense that fewer nodes are mapped than would otherwise be possible) as the revisions get farther apart in the revision tree, due to the large number of tree comparisons that are chained together. However, because the vulnerability predication experiments in this work only take measurements of adjacent revisions, we use this wrapping approach when producing our measurements.

A number of algorithms have been proposed for comparing ordered trees in

polynomial time. These algorithms typically apply dynamic programming [18]. One such algorithm can find the edit distance between two trees (and a corresponding edit script) in $O(|T_1|\,|T_2| \min\{D_1, L_1\} \min\{D_2, L_2\})$ time, where $|T_i|$ is the number of nodes in tree i, D_i is the depth of tree i, and L_i is the number of leaves on tree i [18].

2.5.2.3 Direct measurement of tree differences

Typical tree edit distance algorithms allow for a cost function $\gamma(l_1, l_2)$ to be freely defined, where l_1 and l_2 are labels on tree nodes. If the cost function satisfies the properties of a distance metric over the set of labels, then the tree edit distance algorithm will satisfy the properties of a distance metric over the set of trees. Note that "distance" here is measured as the sum of the costs of the edits that were made. Accordingly, distance is minimized by minimizing this sum, rather than maximizing the cardinality of the mapping.

This property of tree edit distance algorithms inspires the possibility of building a change metric that encodes the quality of software to be measured in the cost function γ, omitting all of the machinery described in Section 2.2. However, as we will demonstrate, this is not as good of an idea as it may originally seem. Start by labeling trees with the metric family's labeling function. Construct a cost function γ that returns a distance of 2 between two different labels, 1 between a label and an unlabeled node (or a "space") and 0 otherwise. As long as the metric family's labels are unique (no two nodes in any given tree can have the same label), the tree edit distances that result will qualify as a difference-based change metric with $p_1 = 1$, $p_2 = 1$, $p_3 = 2$, and $p_4 = 2$ [3].

Although such a software change metric would be simple to compute, this approach suffers from the problem of not allowing the tree differencing algorithm

[3]Note that any value can be assigned to p_4 because, given that the labels are unique, the MOD predicate will never hold.

to consider all the properties of a node, resulting in a tree-to-tree comparison that differs from what would be expected. The issue stems from the zero cost of adding and removing tree nodes which are unlabeled.

Consider two programs that declare a single variable x but are otherwise completely different. Next, consider a metric family that measures variable declarations, with a labeling function that labels variable declaration statements with the name of the variable being declared. If the direct measurement approach that we just proposed was applied in this example, the tree differencing algorithm would produce an edit script that rewrites the entire program around the unchanged variable. The resulting distance of zero between the two programs is correct in a mathematical sense, but does not capture the intent of the software change metric that was developed – which should capture the fact that one declaration of variable x is "different" from another if it appears in a completely different context.

To avoid this issue, edit scripts (or mappings) between two revisions of the same file must be computed on unlabeled trees (before applying the metric family's labeling function) and should be based on properties directly taken from the original source code.

2.5.3 Value mapping algorithms

As discussed in Section 2.2.2.3, the domain of a labeling function may differ from file to file and revision to revision. This accommodates the fact that there may not be a "natural" vocabulary of labels which can remain consistent across files. For example, if a labeling function targeted `for` loops, the label placed on the loops may be an automatically generated unique identifier. This problem is also unavoidable when constructing metrics that label tree nodes with line numbers – a trivial modification to a program may cause all program lines to be renumbered, necessitating that the *valuemap* builds a mapping of lines from one revision to the

other. In these situations, a value *matching* algorithm is necessary to ensure that changes between revisions of a file are accounted properly.

Even when a labeling function can draw from a natural vocabulary, using a matching algorithm may still be desirable if the metric should be insensitive to renaming the labels. For example, if a labeling function labeled the entire body of a method with the method's name, and a revision of a file simply renamed the method, then the old name of the method should be matched to the new name of the method.

The optimal matching between two sets of labels maximizes the number of abstract syntax tree nodes which are mapped (by a node mapping) to nodes with identical labels, maximizing the following quantity when moving from tree revision T to tree revision U with labeling function domains $\mathbf{L_T}$ and $\mathbf{L_U}$. In other words, we choose *valuemap* such that the following sum is maximized:

$$\sum_{(t,u)\in nodemap(T\to U)} \sum_{\lambda\in\mathbf{L_T}} \begin{cases} 1 & \text{if } t \in A_T^{\lambda} \wedge u \in A_U^{valuemap(T\to U,\lambda)} \\ 0 & \text{otherwise} \end{cases}$$

To find the optimal matching, we first construct a weighted bipartite graph, where each node of the graph is a label (either in $\mathbf{L_T}$ or $\mathbf{L_U}$) and the edges represent the number of nodes mapped between T and U that have that pair of labels:

$$V \equiv \mathbf{L_T} \cup \mathbf{L_U}$$

$$E \equiv \mathbf{L_T} \times \mathbf{L_U}$$

$$w\left(\lambda_t, \lambda_u\right) \equiv \left|\left\{(t,u) \in nodemap\left(T \to U\right) \mid t \in A_T^{\lambda_t} \wedge u \in A_U^{\lambda_u}\right\}\right|$$

We then solve the *linear assignment problem* on the graph (modified to maximize the edge weights and equalize the graph partitions – discussed below). This

will maximize the sum of the weights of the edges, such that each node is covered by exactly one edge. The *valuemap* can then be extracted by simply reading the set of selected edges (and stripping their weights, which are no longer used). One algorithm to solve the linear assignment problem (used in the experiments performed in this work) is the Hungarian Algorithm, which solves the problem in $O(n^3)$ time, where $n = \max(|\mathbf{L_T}|, |\mathbf{L_U}|)$.

There are two additional considerations when converting the labels to a graph for the linear assignment problem, both pertaining to the presence of labels in either set which will not be matched to any non-dummy labels:

Differing numbers of labels: The linear assignment problem requires for both halves of the bipartite graph to have the same number of nodes. If there are differing numbers of labels in the two sets, then dummy labels can be added to one value set so they have the same number. Note that as discussed in Section 2.2.2.3, these dummy labels are required anyway to ensure that each revision's range of labels has one member of each equivalence class.

Unwanted zero-weight matchings: Some labels will be matched to another label such that the weight of the edge between the labels is zero, because the two labels had no tree nodes in common. When these matches occur between dummy labels, or between a dummy and a non-dummy label, this is not problematic. However, zero-weight matches between two non-dummy labels are not desirable, because they will result in matches between entities in source code that have no relationship with each other. For example, if a variable is deleted in one part of a program and an unrelated variable is added elsewhere in a program, this should be accounted for as the addition of one variable (ADDONLY) and the deletion of another variable (DELONLY), rather than the movement of a single variable (ADDDEL).

This can be addressed by manipulating the weights of the graph that is used to solve the linear assignment problem. First, add a sufficient number of dummy

labels such that all non-dummy labels can be mapped to a dummy label if needed. Next, set the weights of all edges between two non-dummy labels to -1 when it would otherwise be 0. This results in non-dummy labels being matched to dummy labels unless a non-dummy match has at least one AST node in common.

2.6 Summary

In this chapter, we presented the following

- A system for constructing *families* of metrics, which allow for the construction of code and change metrics that characterize the same aspects of the source code
- A proof that certain change metrics in each metric family will satisfy the axioms for a mathematical distance pseudometric, validating that these metrics can measure change in a "well-behaved" way
- A proof that the distance-based change metrics from metric families characterize differences in the same way that traditional code churn metrics do, while the alternative code delta change metrics do not
- A source code definition model for building metric families, based on abstract syntax trees
- Properties that must be satisfied by an implementation of the definition model in order for the above proofs to hold
- Examples of metric families defined for this study
- Algorithms for efficiently computing metric families in a way that satisfies all of the properties mentioned above

This concludes our discussion on how code and change metrics were computed for this study. Later, in Chapter 5, we will discuss the implementations of the tools that computed these metrics in more detail. In the next chapter, we will turn our attention to how code and change metrics can be processed and transformed into *features*, or representations of the attributes of software artifacts that can be directly supplied to machine learning algorithms.

Chapter 3

Constructing features for software defect and vulnerability prediction

Software defect prediction (also known as software fault prediction) is the practice of statistically associating some characteristic of a software product with defects (or bugs) present in the same product. In the previous chapter, we introduced a system for defining and computing a wide variety of static code metrics and software change metrics. In this chapter, we will take a broader look at the characteristics of a software product that could be used for prediction – including, but not limited to, software metrics. In order to do this, we will first present a survey of past studies in defect and vulnerability prediction with a focus on ways that these studies extract characteristics (or features) from software products. Next, we will select a subset of features and methods from this taxonomy and provide formal definitions for them, in order to relate them to the metric computation methods and source code model defined in the previous chapter. These formal definitions will then serve as specifications for the experiments that will be performed in the following chapters.

There is no single standard approach for defect and vulnerability prediction – indeed, there are almost as many approaches as there are studies. Many of the approaches seem similar on the surface, because the studies tend to use similar kinds of metrics or other independent variables – for example, complexity metrics, coupling metrics, or developer experience scores. However, we argue that a less obvious aspect of designing a study – the *experimental setup* (i.e. the way that these independent variables are used) – has a far greater practical impact on how the study may be applicable to in practice. For example, several common experimental setups carry their own unique advantages and disadvantages:

- Many experiments build models that statistically associate characteristics of each source code file with the presence of defects in the same files. Such a model is then used for a static code inspection task, applying the model to select files likely to contain defects and then scrutinizing these files to find those defects. However, the model in such a case is only effective when it is applied at a single point in time, as repeated application of the model would likely result in a near-identical set of files being inspected over and over again. (In other words, the model could only be used once.) This experimental setup also precludes the use of metrics that measure characteristics of one file that may affect the defect-proneness of another file (such as if the increased attack surface of one module reduced the security of a system as a whole [83, 146]).

- Other experiments build more complex features by measuring characteristics of changes or revisions (such as the author of a change or the size of a commit) and then aggregating them, such that each file ends up with a single set of features that, in some way, characterizes the file's development history. Unlike the previous experimental setup, this experimental setup allows for the history of a file (rather than its current state) to be considered, increasing the likelihood that certain historical associations will be found. However, this change history data is much more difficult to collect than static source code metrics, and this experimental setup still has the aforementioned disadvantage of tying features and defects to individual files.

- Some experiments, predicting defects at the level of commits (rather than the level of files), use characteristics of a commit (such as the amount of code changed in the commit or the volume of recent code changes in files affected by the commit [90]) to predict if a defect was introduced during that commit. Because this experimental setup is not bound to individual files, it is capable

of capturing relationships between defects and characteristics of other files in the application. In addition, a commit-level predictor can be used repeatedly throughout the lifetime of the product, as opposed to file-level predictors, which can only effectively be used once. However, developing features at a commit-level granularity may prove problematic, because developers may split functionality into multiple commits – splitting defects from code that enabled them, and frustrating efforts to associate the independent variables with the defects that they're ultimately related to. In addition, many independent variables (such as metrics related to file coupling or shared dependencies) are metrics which have only traditionally been defined at the granularity of whole files (although in Chapter 2, we introduced a way to adapt these metrics to measure change instead).

Empirically studying the practical impact of experimental setup on vulnerability prediction is a primary objective of this work. To better understand how defect and vulnerability prediction experiments were set up in the past, we present a literature review of past studies with an emphasis on the setups of these studies. In particular, we focus on how change data (data on the software's evolution over time, or data on the activities performed over individual commits or windows of time) is utilized, as working with change-based features is less straightforward than working with static code features, and there is more variation in how these features are handled between studies.

3.1 Static code features for defect prediction

We begin by surveying approaches based on *static* code features. We define a static code feature to be any characteristic of the code that can be used to derive an independent variable without consulting the past history of the code (without con-

sulting any revision of the code but the current). The experimental setup of a study which only uses static code features is simple – the static code features for each unit of source code are used to compute independent variables for the same source code units, and the predictive process is executed such that the static code features predict the presence of a defect (or the number of defects, or the number of defects that have occurred in the past, or the number of recent defects or fixes [15, 156]) in the source code unit. Although the experiment may be performed at multiple granularities, with methods, classes, files, packages, or even entire applications serving as source code units, the basic experimental setup remains constant.

In the following paragraphs, we discuss the various static code features that prediction experiments have used as independent variables. Many experimental setups incorporate both static code features and change-based features, aggregating change-based features up to one point in time so they can be combined with static code features. The change-related aspects of these studies will be described in more detail in Section 3.2.

Prediction experiments use a variety of methods to compute the *dependent* variable, or number of defects in a class. Such methods may include counting the number of defects that had been discovered in a source code unit over a certain time period, or counting the number of known defects actually in a source code unit at the point in time when the static code features were collected. In addition, as an alternative to cross-validation, some experiments used next-release validation [139] to validate a predictive model, training a model on one release of a product and using that model to find defects in the next release of the same product. Although this validation method involves the use of multiple releases, we do not consider this to constitute a use of software evolution for defect prediction, and such a study would still utilize static code features (unless a different aspect of the same study is related to software evolution or change metrics).

AST features: Source code modules [72] or files can be parsed into *abstract syntax trees*, such that each tree node is of a particular type. For any given type, the number of nodes of that type can be counted, such that each node type yields one static code feature. AST node features are similar to text features (to be described later in this list) in the sense that, unlike with software metrics, no *a priori* definition of the features need be crafted – the set of features is derived from the source code itself. A source file will generally yield fewer AST node features than text features, because there will be more distinct tokens than AST node types, making AST node features somewhat more generic with a lower dimensionality feature vector. The specific AST features which are used in any given study will depend on the programming language; for example, possible AST features for PHP may include the number of `if` statements or the number of plus signs in a file.

Code style: Code style features are a variety of code metrics that specifically estimate the degree that poor coding practices were employed when developing a source code unit. Several defect prediction studies have used code style as a static code feature, due to the hypothesis that poor style is associated with defective code. Style violations include the use of language constructs that have been deemed unsafe [22] or language usage issues such as missing braces or empty `catch` blocks [78].

Execution metrics: Execution metrics are collected during the execution of a test suite, measuring the behavior of a source code unit at runtime (rather than simply analyzing the code's structure). Some prediction experiments [141] incorporate dynamic software measures [79], which include metrics that are similar to classical source code metrics but are based on traces of the program's execution. For example, a dynamic size measure is computed as the number of executed instructions in the source code unit. The level of code coverage of a source code unit [103] is another example of an execution metric. Although code coverage-based features may be similar to dynamic software measures, the former emphasize the adequacy of

the program's test suite, while the latter emphasize the program itself as it typically executes.

File type: The type of a file (such as the language of the code contained within the file) can act as a simple static code feature, capturing the possibility that different types of files contain defects at different rates.

Graph metrics: Many defect prediction studies incorporate graph metrics, which were typically developed for network or social media analysis. Such metrics are constructed by building a graph from the program's source code units and then measuring the interrelationships between the nodes and edges of the graph. Examples include centrality measures [19], dataflow measures [106], and distances between different layers of the system [179].

Security resources: Several studies [37, 164, 166] associated the presence or absence of security resources on a product's web site, such as security documentation or lists of vulnerabilities, with the number of vulnerabilities found in the product. (Although, strictly speaking this is not an attribute of the program itself, the information is a static feature in the sense that the security information is acquired once at the same time the code is downloaded, and from the same location.)

Security specific: Some static code features were designed to be *security-specific*, in the sense that they were crafted to specifically indicate potential vulnerabilities (i.e. security defects) or kinds of code that are likely to contain vulnerabilities. One feature [145] is linked to SQL-related code, which is potentially more likely to contain vulnerabilities of all kinds. Another set of features [135] relates to code performing sanitization and accessing persistent data, operations which may be related to vulnerabilities.

Specific references: Some studies link specific dependencies or the use of specific functions to the presence of defects. These models are built by generating one independent variable for any such dependency or function that was referenced

from a code unit. This category is distinguished from the *security specific* category by the presence of *all* possible dependencies (which may vary from project to project) in the feature vector rather than a set of security-related dependencies determined in advance.

Static analysis: Static analysis tools produce automatically generated alerts, pointing to the presence of a potential defect or vulnerability in a specific source code unit. Although such tools sometimes yield a high false-alarm rate [10], the output of static analysis tools has been used as another feature for predictive models, essentially combining the number of alerts with other features extracted from the source code to generate a final prediction. Static analysis tools for finding security defects [163] have been used for this purpose, as well as tools for finding defects in general [48].

Static code metrics: Software metrics are very commonly used as features for defect predictors. *Static code metrics*, or metrics computed on a single version of a program, can act as static code features. For the purposes of this taxonomy, we define software metrics as numerical attributes that are computed predominately with information extracted from one specific source code unit, without executing or performing complex program analysis on the code. Software metrics are often used for defect prediction because they can easily be computed with readily available tools, they can encapsulate a variety of aspects of a source file with a relatively low feature vector dimensionality, and they can encapsulate formulas and relationships that other kinds of static code features may not be able to represent. For the purpose of building static code predictors, we disregard the question of whether the metric faithfully represents an underlying attribute [89]; rather, it is sufficient that the metric simply maps source code characteristics to independent variables in a predictable way. A metric measuring one source code unit may be influenced by other source code units; for example, a coupling metric may depend on the number

of times that the file is referenced from a different file. In Chapter 2, we introduced a methodology for formalizing and computing a wide variety of code metrics.

Text features: Text features are similar to *AST features* in the sense that every element in a source file is counted and aggregated to form a high-dimensionality feature vector. In contrast to AST features, text features are extracted from a tokenized source file, encapsulating the token used without regard to its role in the language's grammar.

3.2 Change-based features for defect prediction

Change-based features, in contrast to static code features, consider the past development and release history of the source code, as opposed to only considering the code's current state. Change-based features are used in prediction studies for two primary reasons. First, it is often hypothesized that considering the development history of a file will provide useful information for prediction beyond what's in the file at the current time. Second, it is sometimes desirable to predict the *time* (in addition to or instead of the location) when a vulnerability will be introduced into (or discovered in) the source code. Time-based prediction studies like these naturally have a need for information on the change in the code over time.

The experimental setup for studies with change-based features tends to be more complex and less standardized than it is for studies that only use static code features. This is often due to a fundamental "independence mismatch" between the dependent variable and the independent variables (the change-based features). Although it is easy to localize defects to a particular source file or function, determining the time when the defect was introduced into the file is trickier. Manually compiling this data is time-consuming, and heuristics for automatically collecting this information [144] proved inaccurate in the context of security vulnerabilities, as we will discuss in Chapter 5. Hence, it is much easier to statistically associate

change metrics (which have both a time and a position dimension) with the locations of defects (only having a position dimension), and the need to do this has led to a variety of aggregation techniques, which, to date, have not been explicitly explored in existing surveys of software defect prediction literature. Indeed, the fact that a defect prediction study describes itself as using "change metrics" does not imply that the study uses any particular kind of dependent variable or experimental setup.

We observe that when using change-based features, the experimental setup typically touches upon two orthogonal concerns. *Change measurements* are the quantities that the change-based features ultimately measure, without concern as to how they are aggregated to the level of the source code units under prediction. *Change aggregators* define the way that these quantities are mapped to particular source code units (such as files, commits, or releases). In addition, we describe several *special change-related features* which cannot easily be described in terms of change measurements and change aggregators.

3.2.1 Change measurements

Change entropy: Software changes are often spread across multiple files. This change measurement is based on Shannon's entropy, which measures if a change predominately affects code in one particular file, or if multiple files are equally affected as part of a particular change. Entropy has been computed on the level of time slices [61], measuring if development in a particular time period is concentrated into a small portion of the codebase. Entropy has also been computed on the level of individual commits [46].

Change type: Changes can be classified into types, based on metadata present with the change or manual classification by the researchers. Change types considered in past defect prediction studies include:

- If the change fixes a fault

- If the change is due to a change in requirements [8]
- If the change is a refactoring [33]
- The number of lines modified in order to make changes of the above types [15]

Prediction studies frequently include a feature based on the number of past bug fixes in a source code unit. Such a feature can be reduced to a change type measurement restricted to bug fixes, and we record studies utilizing such features in this way.

Churn: Code churn is very commonly used as a change measurement. Classical code churn measurements incorporate the number of lines of code added, removed, or deleted as part of a change. This category of change measurements incorporates all of these code churn metrics. This category also includes cases where a study incorporates a Boolean indicator variable on each file indicating if it was modified during a particular change, as this can be viewed as a very simple indicator of code churn. Another variant of code churn included in this category is the number of chunks (or sections) modified by a change to a particular file [137].

Churned text: Churned text can be used as part of a high-dimensionality feature vector that includes one feature for every token in the project, similar to the token static code feature. The change measurement is computed as the number of instances of the token that were added or deleted as part of a change [74].

Commit data: Version control systems maintain a variety of metadata on commits, which is often used in prediction studies due to the ease of obtaining it. Such metadata (which we define as data other than the actual patches being applied to the source files) includes:

- The number of files or subsystems changed in a commit [46]
- The commit, change, or issue identifier (used to count the number of unique commits or issue fixes affecting a file) [8, 127]

- The number of developers involved in a change [8]
- Components of the commit time (such as the hour of day or day of week) [137]
- The types of the files modified in the commit [137]
- Text features extracted from the commit comments [74]
- If the change resulted in an entry in the product's change log [90]
- If a particular file was newly created as part of the change [23, 113, 127]

If multiple commits can be grouped together into a single logical change (such as groups of commits related to the same change request) and selected elements of this commit data can also be extracted at the level of these groups [8].

Current metrics: The static code metrics of a product (or of the files affected by a change) at the time of a change can act as simple change measurements.

Current text: As with the current metrics, the text features of a product at the time of a change can act as change measurements.

Operation type: By comparing the source code of a particular class or file before and after a change, fine-grained operation types can be identified and counted. Examples of such operation types include adding and deleting functions [151], adding and deleting else-parts in a method body [49], and adding and deleting statements.

3.2.2 Change aggregators

The change measurements listed above are not always defined at the level of the source code units being used for prediction. For example, a code churn feature defined at the level of individual changes cannot directly be used to predict a dependent variable only defined on individual files. Change aggregators are used to bridge this gap, so the independent and dependent variables are defined for the same entities.

Any study employing change measurements must also employ one or more change aggregators. Depending on the circumstances, using a change aggregator will not necessarily entail transforming the data – for example, no transformation is necessary when file code churn is used as a feature and prediction is being done on the level of bug-introducing changes.

Bug-fixing changes: With this change aggregator, change measurements may directly be used as independent variables in order to predict which changes will fix bugs. For measurements defined at the level of both files and changes (such as code churn), the file measurements may be added together to yield just one feature per change. This aggregator may be used in conjunction with other aggregators (such as the cumulative aggregator); in this case, the application of the other aggregator is repeated multiple times throughout the product's history, such that the "recent history" of the product at the time of each change is incorporated into the features for that change. (See Definition 3.12 for an example of this.)

Bug-introducing changes: With this change aggregator, change measurements may directly be used as independent variables in order to predict which changes will introduce bugs. The additional considerations for bug-fixing changes as described above also apply to bug-introducing changes.

Cumulative: Aggregating changes in a *cumulative* way entails building the change history of each file (or other source code unit) and combining the historical measurements for a file into a single set of features with no time dimension. This allows for change metrics to be used in predictors when the defects were localized to individual files, but not to any particular point in time. The aggregation can be done with a variety of operators. Operators used in past defect prediction studies include sum, minimum, maximum, average, and distinct (for counting distinct values of the measurement in a file's history). The change measurements being combined can extend back to the beginning of the product's history, or they can go back a finite

amount of time, such as with a sliding window up to the time of the change where the aggregation is occurring [152] (as in Definition 3.12).

Decaying cumulative: One variety of the *cumulative* change aggregator adds a decay factor, such that recent changes are weighted more heavily in the aggregated feature than distant changes. Linear, logarithmic, and exponential decay factors have been used [38], as well as a more specialized defect prediction technique based on causality testing [28].

Distribute by co-commit: It is sometimes desirable for changes made in one file (or a similar source code unit) to be reflected in the features for other, related files. This can be done by distributing a change measurement across multiple files, basing the distribution on the presence of co-commits with the changed file (commits where multiple files were changed at the same time). Once way to accomplish this is to copy a file's change measurement across all files involved in a commit [60]. Another way is to measure the co-commit frequency between files within periodic time slices, and then distribute change measurements across files within the timeslice proportionally to the co-commit frequency [124].

Distribute by modification: Some change measurements (for example, change entropy [59]) are defined at the level of individual changes but not at the level of individual files. If prediction is to be done at the file level, this results in a mismatch between the definition of the dependent and independent variables. This change aggregator converts change-level measurements into file-level measurements by measuring the frequency that files are modified within time slices, and then distributing the change-level measurements within each time slice to each file, with weights proportional to the frequencies of file modification during that time slice.

Gini: One study [50] used a Gini coefficient-based change aggregator as a feature. The Gini coefficient (as applied to discrete distributions) measures the extent that the possible values of a variable are observed at unequal frequencies.

This can be used to aggregate changes by applying it to a change measurement which takes on discrete values, measuring the extent that the values observed over time for a file (or other source code unit) occur unequally.

Metric difference: A number of prediction studies incorporate code metrics as a feature while localizing vulnerabilities to the level of changes or commits. Because code metrics are defined at the file level (rather than the change level), they must be adapted in order to use them as independent variables. One common way to do this is to simply compute the difference (or the absolute difference) between the metric values before and after the change. This aggregator can be combined with other aggregators (such as the cumulative aggregator) if prediction is being done at the file level but change history is to be reflected in the features [8].

Periodic metric difference: This change aggregator is a variation of the *metric difference* aggregator. Instead of computing the metric difference resulting from every change, this aggregator samples the metric periodically, takes differences between consecutive intervals, computes the aggregate change on a file level using another aggregator (such as *cumulative*).

Periodic metric entropy: This feature is similar to the *change entropy* measurement, except instead of measuring the degree that the churned code is spread across multiple files, it measures how the accumulated *metric difference* within a periodic time slice differs between files. As with the *periodic metric difference* aggregator, this can be used in conjunction with another aggregator such as *cumulative*.

3.2.3 Special change-related features

Most change-related features can be described in terms of a *change measurement* being transformed into features with a *change aggregator*. However, a smaller class of change-related features are too complex or atypical to be defined in this way.

Age: When predicting at the level of files or classes, one common change-related feature is age (the amount of time that has passed between the creation of the file or class and the time of prediction).

Cache: Bugcache [75] is a unique, adaptive predictor which operates over a long period of time, incorporating recent mispredictions and the recent history of files into its decisions. Its methodology is quite different from a typical machine learning prediction experiment, which can yield a full set of prediction results following one run of the algorithm.

Change scattering: One class of features relates to the *scattering* of code modifications across various regions of a file over a given period of time [124]. Because this feature requires an entire interval of changes to be checked for scattering together, it cannot be expressed with the change aggregators that we described above.

Consecutive edits: A simple file-level change-related feature is the number of occurrences where the file was edited in two consecutive revisions or changes.

Developer characteristics: A wide variety of features have been proposed that relate to the characteristics of developers who edited a unit of code. We consider these to be change-related features because they typically take into account the developer's history of past activities on the project. These features include the level of individual and organizational ownership of a source code unit [179], the number of new developers who worked on a source code unit during a specific time period [171], and if a file was changed by developers who also changed many other files [139].

Developer discussion: One study [12] associated the frequency that a class was mentioned on a development e-mail list with the discovery of defects in that class.

Developer switches: This feature is similar to the *consecutive edits* feature in that it relates to how a sequence of changes to the product affects a particular

file or source code unit. Developer switches occur when the developer who makes a change (affecting a particular source code unit) is different from the developer who last changed the same source code unit.

Frequent itemsets: One study [85] used the concept of *frequent itemsets* to build a change-related feature. A frequent itemset is a set of files which are frequently committed together, and the feature measures the number of frequent itemsets that a particular file is in.

Inter-change interval: As with *consecutive edits* and *developer switches*, this feature relates to the sequence of edits made to a file or other source code unit. The inter-change interval measures the amount of time that elapsed between two consecutive changes affecting the source code unit.

Process compliance: Process compliance, or the degree that documented development processes were followed during the development of a given component, was used as a defect-predicting feature in one study [81].

Time checked out: If a project was developed with a version control system that records details of when files are checked out and checked in, the cumulative length of time that a file has been checked out can be used as a file-level defect prediction feature.

3.3 Surveying past defect prediction studies

Several reviews of software defect prediction techniques have previously been published [26, 123, 136]. One systematic literature review [123] categorizes articles by the software metrics that were used, characteristics of the data sets used for the study, the machine learning technique used, and the granularity (e.g. method, class, file, or package) at which the defects were localized. A meta-analysis of defect prediction studies [136] considered the dataset, machine learning method, metrics, and the researcher group who performed each study. Two systematic studies [38,

93] took a different approach for a similar purpose, performing extensive arrays of experiments to compare different types of dependent variables on a single defect dataset.

We performed our own survey of the literature because none of the existing literature reviews or comparative studies explicitly cataloged the various ways that experiments could be set up for a defect prediction study. In particular, we were concerned with the various ways that change data or historical data on the evolution of the software could be transformed into independent variables for multiple experimental setups. Ultimately, the motivation for our survey was to place our study of multiple experimental setups for vulnerability prediction in context with the previous work, and to ensure that our experiments encompassed and were in line with as many of the known methodologies for deriving change metrics as possible.

An initial set of surveyed papers was selected by multiple means, including Google Scholar searches for `code churn`, `change metrics`, and other terms likely to be related to the analysis of code changes or software evolution for defect prediction. Additional papers were selected by following references which related to these subjects. In addition, a special effort was made to capture as many studies related to vulnerability prediction (rather than defect prediction in general) as possible. While the 105 papers surveyed by no means constitute a complete set of publications related to defect prediction to date, the experimental setups and methodologies for defect prediction (including methodologies for building change metrics) are well-covered. We omit any papers which did not provide sufficient detail to determine how the features in the study were built. To ensure completeness, no effort was made to prune multiple variations (e.g. conference and journal versions) of the same study – all surveyed papers are enumerated in this chapter.

The following table lists the surveyed papers, along with the features from Sections 3.1 and 3.2 that each study incorporated:

Table 3.1: Survey of features used in related defect-prediction studies

Study	*Static features*	*Change measurements*	*Change aggregators*	*Special change-related features*
[72]	AST features, code metrics			
[179]	Code metrics, execution metrics, graph metrics, specific references	Churn	Cumulative	Consecutive edits, developer characteristics
[141]	Code metrics, execution metrics, graph metrics			
[79]	Code metrics, execution metrics			
[171]	Code metrics, file type	Change type, churn, commit data	Cumulative	Age, developer characteristics
[113]	Code metrics, file type	Commit data	Cumulative	
[103]	Code metrics, graph metrics, execution metrics	Churn	Cumulative	Consecutive edits, developer characteristics

Study	*Static features*	*Change measurements*	*Change aggregators*	*Special change-related features*
[151]	Code metrics, graph metrics	Change type, churn, commit data, operation type	Cumulative	Developer characteristics
[19]	Code metrics, graph metrics			Developer characteristics
[106, 116, 157, 178]	Code metrics, graph metrics			
[37, 164, 166]	Code metrics, security resources			
[145]	Code metrics, security specific			
[48]	Code metrics, static analysis	Churn	Cumulative	
[96, 165]	Code metrics, text features			

Study	*Static features*	*Change measurements*	*Change aggregators*	*Special change-related features*
[38, 87]	Code metrics	Change entropy, change type, churn, commit data	Cumulative, decaying cumulative, distribute by modification, periodic metric difference, periodic metric entropy	
[33]	Code metrics	Change entropy, change type, churn, commit data	Cumulative, decaying cumulative, distribute by modification, periodic metric difference	
[8]	Code metrics	Change type, churn, commit data	Cumulative, metric difference	
[100]	Code metrics	Change type, churn, commit data	Cumulative, decaying cumulative	Age
[15]	Code metrics	Change type, churn, commit data	Cumulative	Age

Study	*Static features*	*Change measurements*	*Change aggregators*	*Special change-related features*
[53]	Code metrics	Change type, churn, commit data	Decaying cumulative	Age
[121, 138, 142, 175]	Code metrics	Change type, churn	Cumulative	
[152]	Code metrics	Churn, commit data, current metrics	Bug-introducing changes, cumulative	
[49]	Code metrics	Churn, commit data, operation type	Cumulative	
[124]	Code metrics	Churn, commit data	Cumulative, distribute by co-commit	Change scattering, developer characteristics
[12]	Code metrics	Churn, commit data	Cumulative	Age, developer discussion
[139]	Code metrics	Churn, commit data	Cumulative	Developer characteristics
[74]	Code metrics	Churned text, commit data, current text	Bug-introducing changes, metric difference	
[20]	Code metrics	Churn	Cumulative	Developer characteristics

Study	*Static features*	*Change measurements*	*Change aggregators*	*Special change-related features*
[101]	Code metrics	Churn	Cumulative	Time checked out
[154, 155, 156]	Code metrics	Churn	Cumulative	
[122]	Code metrics		Bug-fixing changes, metric difference	
[39]	Code metrics		Bug-introducing changes, metric difference	
[28]	Code metrics		Decaying cumulative, metric difference	
[32, 107, 108]	Code metrics		Metric difference	
[81]	Code metrics			Process compliance

Study	*Static features*	*Change measurements*	*Change aggregators*	*Special change-related features*
[13, 27, 54, 65, 68, 71, 73, 76, 80, 88, 91, 92, 93, 102, 118, 119, 131, 140, 159, 167, 169, 177]	Code metrics			
[22, 78]	Code style			
[35, 133, 134, 135]	Security specific			
[104]	Specific references			
[163]	Static analysis			
[64, 129]	Text features			
[69]		Change entropy, change type, churn, commit data, current metrics	Bug-introducing changes, cumulative	Developer characteristics, inter-change interval
[46]		Change entropy, change type, churn, commit data	Bug-introducing changes	Developer characteristics, inter-change interval

Study	*Static features*	*Change measurements*	*Change aggregators*	*Special change-related features*
[61]		Change entropy, change type, churn, commit data	Cumulative, distribute by modification	Developer characteristics, inter-change interval
[59]		Change entropy	Cumulative, decaying cumulative, distribute by modification	
[90]		Change type, churn, commit data	Bug-introducing changes, cumulative	Developer characteristics
[60]		Change type, churn, commit data, operation type	Cumulative, distribute by co-commit	Age, inter-change interval
[137]		Change type, churn, commit data	Bug-introducing changes, cumulative	Developer characteristics
[97]		Change type, churn, commit data	Bug-introducing changes	Developer characteristics
[99]		Change type, churn, commit data	Cumulative, decaying cumulative	Age

Study	*Static features*	*Change measurements*	*Change aggregators*	*Special change-related features*
[126]		Change type, churn, commit data	Cumulative	
[43, 44]		Change type, operation type	Bug-introducing changes, cumulative	
[47]		Change type	Cumulative	
[50]		Churn, commit data, operation type	Gini	
[23]		Churn, commit data	Bug-introducing changes	Developer characteristics
[9, 144]		Churn, commit data	Bug-introducing changes	
[127]		Churn, commit data	Cumulative	Age, developer switches, inter-change interval
[51]		Churn, operation type	Cumulative	
[153]		Churn	Bug-fixing changes, change scattering	
[41]		Commit data	Bug-introducing changes	Developer characteristics

Study	*Static features*	*Change measurements*	*Change aggregators*	*Special change-related features*
[75]				Cache
[85]				Frequent itemsets

From this table, it is clear that a wide range of techniques and experimental setups (i.e. change aggregators) have been employed when constructing features for defect predictors. Accounting for this diversity of methodologies has historically been difficult; a meta-analysis [136] of defect prediction studies concluded that the identity of the group of researchers who performed the study was the strongest predictor of the reported experimental performance. However, it is likely that the identity of the research group ultimately served as a proxy for additional factors not included in their meta-analysis, such as the details on change aggregation and change measurements that we enumerated in the table above.

3.4 Composable data operators for constructing feature variants

In the previous sections, we surveyed a diverse array of techniques for constructing features for defect prediction studies. The features used in many studies could be decomposed into static code features, change measurements, and change aggregators, suggesting that each of these aspects could be defined and evaluated separately. In this section, we formally define a subset of these aspects and techniques, with the goal of building a library of operators that we can draw from when performing the experiments later in this work. More specifically, we formalize these techniques by defining *composable* data operators which, when used in conjunction with each other, can aggregate and transform defect data and features (such as

software metrics) to meet the needs of a given experiment.

The operators described in this section was designed to be used in conjunction with the *metric family* framework described in Chapter 2, although other kinds of metrics and features (such as any static code feature described in Section 3.1 or any change measurement from Section 3.2) could also be used.

The overall process involves computing some set of metrics or features for an application and then applying the operators to the resulting structure to transform it into an *example set*, a structure directly usable as input for a machine learning tool. In the discussion that follows, each operator is briefly discussed and traced back to the taxonomy of static features, change measurements, and change aggregators outlined earlier in this chapter.

We first define the concept of a *data cube*, which is a structure representing three-dimensional feature (i.e. metric) data on a product. In the formalization of the operators which we present here, a cube is represented by a set of tuples, each of which assigns a real value to a particular feature (also called a variable) for a particular file at a particular release of the product. For change-based features, by convention, a measurement value assigned to a file at a particular release represents a measurement of the change in that file from the preceding release to the release which appears in the tuple.

Definition 3.1 (File/release/feature cube) *For a set of variables* $\mathbf{V}$ *and sets of files* $\mathbf{F}$ *and releases* $\mathbf{R}$ *which are part of a particular product, a software feature cube* C *is defined as:*

$$C \subseteq \mathbf{F} \times \mathbf{V} \times \mathbf{R} \times \mathbb{R}$$

We next define *matrices* which are two-dimensional because they omit either the file or the release dimension from the cube. These matrices are defined in order to accommodate experimental setups that do not accommodate all three dimensions,

or for encoding features that do not have a three-dimensional representation. Aside from the omitted dimension, all of the above details regarding the definition of the data cube also apply to the matrices.

Definition 3.2 (File/feature matrix) *For a set of variables* $\mathbf{V}$ *and set of files* $\mathbf{F}$ *which are part of a particular product, a file/feature matrix* F *is defined as:*

$$F \subseteq \mathbf{F} \times \mathbf{V} \times \mathbb{R}$$

Definition 3.3 (Release/feature matrix) *For a set of variables* $\mathbf{V}$ *and set of releases* $\mathbf{R}$ *which are part of a particular product, a release/feature matrix* R *is defined as:*

$$R \subseteq \mathbf{V} \times \mathbf{R} \times \mathbb{R}$$

We now define an operator to add measurements for a given static code feature (such as a static source code metric) to the cube. The cube that results from applying the operator will be the union of the original cube and the tuples assigning the values to the new variable, allowing for multiple features to be present in the same cube. This operator primarily corresponds to the **code metrics** static feature from the surveyed papers, although other static features could be added similarly.

Definition 3.4 (Add static measurement) *Let* C *be a file/release/feature cube and* m *be a function which computes a static code measurement for a specified file at a specified release. Let* $\text{name}(m)$ *be an identifier for the measurement and* $\text{inrelease}(f, r)$ *be a function indicating if a file is present in the specified release. The measurement can be added as a feature to the cube with the following function:*

$$\text{addstaticmeasurement}(C, m) = C \cup$$
$$\{(f \in \mathbf{F}, \text{name}(m), r \in \mathbf{R}, x) \mid x = m(f, r) \wedge \text{inrelease}(f, r)\}$$

The operator to add change measurements is similar, except no measurement is made for the first release in the dataset. This corresponds to the **churn** change measurement and values produced by the **metric difference** aggregator, as well as the novel change metrics that we introduced in Chapter 2.

Definition 3.5 (Add change measurement) *Let C be a file/release/feature cube and m be a function which computes a change measurement between two specified files at a specified release. Let* $\text{firstrelease}(r)$ *be a function indicating if a release is the first and* $\text{parent}(r)$ *be the parent of release r. The measurement can be added as a feature to the cube with the following operator:*

$$\text{addchangemeasurement}(C, m) = C \cup$$
$$\{(f \in \mathbf{F}, \text{name}(m), r \in \mathbf{R}, x) \mid x = m(f, \text{parent}(r), r) \wedge \neg\text{firstrelease}(r) \wedge \text{inrelease}(f, r)\}$$

The next two operators add a feature to the cube indicating the number of revisions or days that have passed since a file was last changed. This corresponds to the **inter-change interval** special change feature.

Definition 3.6 (Add last-revision-changed) *Let* $\text{revdiff}(r', r)$ *be the number of revisions between r' and r, such that* $\text{revdiff}(r, r) = 0$. *Let* $\text{filechanged}(f, r)$ *indicate if file f was changed in release r. The operator to add the number of releases since a file's last change to file/release/feature cube C is:*

$$\text{addlastchangedrev}(C) = C \cup \big\{(f \in \mathbf{F}, \text{"lastchangerevs"}, r \in \mathbf{R}, x) \mid$$
$$x \geq 0 \wedge x = \min\{\text{revdiff}(r', r) \mid r' \in \mathbf{R} \wedge \text{filechanged}(f, r')\}\big\}$$

Definition 3.7 (Add last-date-changed) *Let* $\text{date}(r)$ *be a serialized release date (in days) for release r. The operator to add the number of releases since a file's last*

change to file/release/feature cube C is:

$$\text{addlastchangeddate}(C) = C \cup \big\{(f \in \mathbf{F}, \text{"lastchangedays"}, r \in \mathbf{R}, x) \mid$$
$$x = \min\{\text{date}(r) - \text{date}(r' \in R) \mid \text{revdiff}(r', r) > 0 \wedge \text{filechanged}(f, r')\}\big\}$$

The next operator adds a feature to a release/feature matrix indicating if a release was a major (first version number component changed), minor (second component changed), or patch (only last component changed) release. (Although other version numbering schemes are possible, the PHP applications in our study used this numbering scheme). Such a feature could be beneficial for prediction if certain kinds of releases are more likely to introduce defects. This feature is related to the **commit data** change measurement (adapting it from commits to whole releases).

Definition 3.8 (Add version jump) *Let R be a release/feature matrix, and* versmajor(r) *and* versminor(r) *be functions that extract the major and minor components from the version number of r. The following operator adds several features to this matrix*

indicating if the release was a major, minor, or patch release:

$$\text{addversjump}(R) = R \cup$$

$$\left\{ \begin{array}{c} (v, r \in \mathbf{R}, x) \mid \exists r' \, \text{revdiff}(r', r) = 1 \\ \wedge \left(\left(v = \text{"versjumpmaj"} \right. \right. \\ \wedge x = \begin{cases} 1 & \textit{if } \text{versmajor}(r) \neq \text{versmajor}(r') \\ 0 & \textit{otherwise} \end{cases} \\ \left. \left. \right) \vee \left(v = \text{"versjumpmin"} \right. \right. \\ \wedge x = \begin{cases} 1 & \textit{if } \text{versmajor}(r) = \text{versmajor}(r') \wedge \text{versminor}(r) \neq \text{versminor}(r') \\ 0 & \textit{otherwise} \end{cases} \\ \left. \left. \right) \vee \left(v = \text{"versjumppat"} \right. \right. \\ \wedge x = \begin{cases} 1 & \textit{if } \text{versmajor}(r) = \text{versmajor}(r') \wedge \text{versminor}(r) = \text{versminor}(r') \\ 0 & \textit{otherwise} \end{cases} \left. \left. \right) \right) \end{array} \right\}$$

The next operator adds a feature to a release/feature matrix indicating the amount of time that passed between the parent release and the release being measured. As with the previous feature, this feature falls under the **commit data** change measurement.

Definition 3.9 (Add date jump) *Let R be a release/feature matrix. The following operator adds a feature to this matrix indicating the amount of time that had*

passed since the previous release:

$$\text{adddatejump}(R) = R \cup \big\{(\text{"datejump"}, r \in \mathbf{R}, x) \mid \exists r' \, \text{revdiff}(r', r) = 1 \wedge x = \text{date}(r) - \text{date}(r')\big\}$$

Another operator, which also adds a feature to a release/feature matrix, indicates the number of files that were changed during a release. This feature also falls under the **commit data** change measurement.

Definition 3.10 (Add changed file count) *Let R be a release/feature matrix. The following operator adds a feature to this matrix indicating the number of files which changed in each release:*

$$\text{addfileschg}(R) = R \cup \big\{(\text{"fileschg"}, r \in \mathbf{R}, x) \mid x = |\{(f, r) \mid \text{inrelease}(f, r) \wedge \text{filechanged}(f, r)\}|\big\}$$

In Chapter 2, we defined several *derived metrics*. These metrics are not part of standalone metric families in the metric family construction system; rather, they must be computed as a function of several other metric family members. The following operators add such derived metrics to a cube or matrix, once the constituent metric family members have been added.

Definition 3.11 (Add derived metric) *Let m be the definition of a derived metric and* $\text{name}(m)$ *be an identifier for that metric. The operators to add a derived metric to a file/release/feature cube C, a file/feature matrix F, or a release/feature*

matrix R are:

$$\text{addderivedcube}(C, m) = C \cup \big\{(f \in \mathbf{F}, \text{name}(m), r \in \mathbf{R}, x) \mid x = m(\{(v, y) \mid (f, v, r, y) \in C\})\big\}$$

$$\text{addderivedfile}(F, m) = F \cup \big\{(f \in \mathbf{F}, \text{name}(m), x) \mid x = m(\{(v, y) \mid (f, v, y) \in F\})\big\}$$

$$\text{addderivedrel}(R, m) = R \cup \big\{(\text{name}(m), r \in \mathbf{R}, x) \mid x = m(\{(v, y) \mid (v, r, y) \in R\})\big\}$$

Several change aggregators incorporate the concept of *sliding windows* to base a defect prediction feature on the recent history of a file. These features are motivated by the premise that the recent history of a file's change and evolution influences the likelihood that the file contains a defect better than a more distant history period. To this end, we introduce a collection of operators that compute a derived feature for each file at each release in a cube, basing the feature on a window of releases preceding that release.

The first functions, designed to be used in conjunction with later operators, compute the window over which later operations will operate. Release windows effectively form a path from the first release toward the root, avoiding releases which are not ancestors of the first release.

Definition 3.12 (Sliding windows) *Let r be a release and l be the length of the desired sliding window in days or revisions (i.e. releases). Let* $\text{ancestors}(r)$ *be the set of ancestors of release r. The following functions build a sliding window of the*

specified length:

$$\text{slidewinddays}(r, l) = \left\{ r' \in \mathbf{R} \mid r' \in \text{ancestors}(r) \wedge \text{date}(r) - \text{date}(r') < l \right\}$$

$$\text{slidewindrevs}(r, l) = \left\{ r' \in \mathbf{R} \mid r' \in \text{ancestors}(r) \wedge \text{revdiff}(r', r) < l \right\}$$

The next functions, also designed to be used in conjunction with later operators, implement various operations computing different kinds of derived features that involve sliding windows. These operators implement the **cumulative** and **decaying cumulative** change aggregators. The first four functions compute the sum, minimum, maximum, or mean of the features of releases within the window. The last function computes a decaying sum, where older releases influence the derived feature less than newer ones.

Definition 3.13 (Sliding window cube operations) *Let V be a set of variable values masked by a sliding window. The following sliding window operations may be passed to the sliding application operator application function:*

$$\text{slidecubesum}(V) = \sum_{(r,y,d) \in V} y$$

$$\text{slidecubemin}(V) = \min_{(r,y,d) \in V} y$$

$$\text{slidecubemax}(V) = \max_{(r,y,d) \in V} y$$

$$\text{slidecubemean}(V) = \text{mean}_{(r,y,d) \in V} y$$

$$\text{slidecubedecaysum}(V) = \frac{\sum_{(r,y,d') \in V} y \cdot \left(d - \min_{(r',y',d') \in V} d' \right)}{\sum_{(r,y,d') \in V} d - \min_{(r',y',d') \in V} d'}$$

Finally, the following operator utilizes the sliding window and sliding window operation functions above, applying a sliding window operation to each revision's window for each file. Note that the sliding window operation is applied to each feature separately, meaning that the number of new derived features will be equal to the number of features originally present in the cube. The name of the sliding window operation is prepended to the name of each individual feature for disambiguation. The length parameter of function w must be fixed to the desired value before passing it to this function.

Definition 3.14 (Sliding window cube operator application) *Let C be a cube, w be a curried sliding window function from Definition 3.12, and o be an operation function from Definition 3.13. Let* $\text{name}(o)$ *be an identifier for that operation. The operator to apply the specified operation over the specified sliding window is as follows:*

$$\text{slidecubeop}(C, w, o) = C \cup \Bigg\{ (f \in \mathbf{F}, \text{name}(o) + v, r \in \mathbf{R}, x) \;\Bigg|\; \big(\exists v_f, v_r, v_x (v_f, v, v_r, v_x) \in C\big) \wedge x = o(\{(r', y, \text{date}(r')) \mid r' \in w(r) \wedge (f, v, r', y) \in C\}) \Bigg\}$$

The next sliding window operator converts a release/feature matrix into a file/release/feature cube by transferring the values of features to files with weights proportional to the file's recent edit frequency. This operation, implementing the **distribute by modification** change aggregator, allows for release-level features to be incorporated into file-level experimental setups.

Definition 3.15 (Distribute feature to files by edit frequency in sliding window) *Let R be a release/feature matrix and w be a curried sliding window function from Definition 3.12. The following operator distributes release/feature values propor-*

tionally to their edit frequency within the sliding window:

$$\text{distributefreqslide}(R) = \left\{ (f \in \mathbf{F}, v, r \in \mathbf{R}, x) \mid \exists y \, (v, r, y) \in R \wedge \text{inrelease}(f, r) \wedge x = y \cdot \frac{\left|\left\{r' \in w(r) \mid \text{inrelease}(f, r') \wedge \text{filechanged}(f, r')\right\}\right|}{\left|\left\{(f', r') \in \mathbf{F} \times w(r) \mid \text{inrelease}(f', r') \wedge \text{filechanged}(f', r')\right\}\right|} \right\}$$

A simpler variant of the edit-frequency-based feature distribution transfers the release-level feature to any files that were modified during that release.

Definition 3.16 (Distribute feature to changed files) *Let R be a release/feature matrix. The following operator converts the matrix to a file/release/feature cube by distributing release/feature values to files which have changed in the respective release:*

$$\text{distributechanged}(R) = \left\{ \begin{array}{c} (f \in \mathbf{F}, v, r \in \mathbf{R}, x) \mid \exists y (v, r, y) \in R \wedge \text{inrelease}(f, r) \wedge \\ \Big((\neg \text{filechanged}(f, r) \wedge x = 0) \vee \\ \Big(\text{filechanged}(f, r) \wedge x = \\ \dfrac{y}{|\{f' \in \mathbf{F} \mid \text{inrelease}(f', r) \wedge \text{filechanged}(f', r)\}|} \Big) \Big) \end{array} \right\}$$

The following operator, which converts a file/release/feature cube into a release/feature matrix, serves the opposite purpose of the above feature-distribution operators. By allowing for file-level features to be incorporated into release-level experimental setups, this operator facilitates (for certain experimental setups) the **bug-fixing changes** or **bug-introducing changes** aggregation patterns.

Definition 3.17 (Combine files) *Let C be a file/release/feature cube and o be an operator function from Definition 3.18. The cube can be transformed into a*

release/feature matrix using the specified operator with the following function:

$$\text{combinefiles}(C, o) = \left\{(v \in \mathbf{V}, r \in \mathbf{R}, y) \;\middle|\; y = o\left(\{(x, f \in \mathbf{F}) \mid (f, v, r, x) \in C\}\right)\right\}$$

As in the case of the sliding window cube operators, there is also a set of file combination operators. The first four operators combine file-level features by taking the sum, maximum, minimum, or mean of each file's value for a release. The last operator, implementing the **change entropy** change measurement, computes the Shannon entropy of the feature's distribution across the files in the release.

Definition 3.18 (Combine file operators) *Let V be a set of file-to-feature values as constructed in Definition 3.17. The following operators can be used when transforming a file/release/feature cube into a release/feature matrix:*

$$\text{combinesum}(V) = \sum_{(f,x)\in V} x$$

$$\text{combinemax}(V) = \max_{(f,x)\in V} x$$

$$\text{combinemin}(V) = \min_{(f,x)\in V} x$$

$$\text{combinemean}(V) = \operatorname*{mean}_{(f,x)\in V} x$$

$$\text{combineentropy}(V) = \exp\left(-\sum_{(f,x)\in V} \frac{x}{\text{combinesum}(V)} \log \frac{x}{\text{combinesum}(V)}\right)$$

In order to transform a file/release/feature cube into a file/feature matrix, the data from one release can simply be selected and other releases can be discarded. This facilitates experimental setups where, although the history of a file may be incorporated into the file's features, the prediction is performed at one particular point in time and only the vulnerabilities present at that time are included in the experiment.

Definition 3.19 (Select release) *Let C be a file/release/feature cube and r be a release. The following operator converts the cube to a file/feature matrix by filtering out all but the selected release:*

$$\text{selectrel}(C, r) = \{(f, v, x) \mid (f, v, r, x) \in C\}$$

We now turn our attention to structuring feature data and binding it to vulnerabilities so it can be used to train (or fit) a machine learning model. This set is referred to as an *example set* and contains one independent variable (a count of defects), one or more dependent variables (features), and an identifier for each example, or object, in the set. To reinforce the fact that the same machine learning model training processes can be used regardless of the experimental setup, we define an example set so the identifiers can be files (one object per file), releases (one object per release), or tuples of files and releases (one object for each instance of a file being included in a release).

Definition 3.20 (Example set) *Let $\mathbf{I}$ be a set of example identifiers (which can be files, releases, or tuples of both). Let $\mathbf{A} = \mathbf{V} \times \mathbb{N}$ be a set of feature assignments (i.e. measurements). Let $\mathbf{D}$ be the set of all defects in the application's defect dataset. An example set E is defined as:*

$$E \subseteq \mathbf{I} \times 2^{\mathbf{A}} \times \mathbf{D}$$

Next, we introduce operators to transform a release/feature matrix or a file/release/feature cube into an example set. This operator fulfills the **bug-introducing changes** aggregation pattern. The independent variable in the example set is the number of defects that were introduced (i.e. that first appeared) during a particular release. Note that if a defect migrates from one file to another between releases, we do not consider the defect to be newly introduced, and in this circumstance, the introduced

function should not indicate this.

Some features can be deemed *required* when building the example set. A required feature must have a non-zero value for an example to be included in the example set. This can be used to exclude unchanged files from an example set by requiring non-zero code churn. This can also be used to exclude examples missing data from an example set. For example, as described in Chapter 5, the source code for some releases in our PHP experiments could not be located. Such missing data will result in missing tuples in the relevant cubes, matrices, or example sets.

Multiple cubes or matrices may be combined when building a single example set. The sets of features in each cube or matrix will be combined into a single set of dependent variables in the example set.

Definition 3.21 (Create example set from release/feature matrix) *Let R^* be a set of release/feature matrices and V_{req} be a set of identifiers of required variables. Let* $\text{introduced}(d, f, r)$ *indicate if defect d was introduced into file f at release r. The following operator creates a prediction example set including one example for every release not missing a required variable:*

$$\text{examplesetrel}(R^*, V_{req}) = \left\{ (r, A, D) \;\middle|\; \begin{array}{c} \forall v_{req} \in V_{req}\, \exists x, R \in R^*\, x > 0 \wedge (v_{req}, r, x) \in R \\ \wedge \big(A = \{(v, x) \mid \exists R \in R^*\, (v, r, x) \in R\}\big) \\ \wedge \big(D = \{d \mid \exists f\, \text{introduced}(d, f, r)\}\big) \end{array} \right\}$$

Definition 3.22 (Create example set from file/release/feature cube) *Let C^* be a set of file/release/feature cubes and V_{req} be a set of identifiers of required variables. The following operator creates a prediction example set including one example*

for every change not missing a required variable:

$$\text{examplesetchg}(C^*, V_{req}) = \left\{ ((f,r), A, D) \;\middle|\; \begin{array}{c} \forall v_{req} \in V_{req}\, \exists x, C \in C^*\, x > 0 \wedge (f, v_{req}, r, x) \in C \\ \wedge\big(A = \{(v,x) \mid \exists C \in C^*\, (f,v,r,x) \in C\}\big) \\ \wedge\big(D = \{d \mid \text{introduced}(d,f,r)\}\big) \end{array} \right\}$$

Finally, the last operator that we define transforms a file/feature matrix into an example set. In the example set and experimental setup that results, the dependent variable will be the number of defects present in the release to which the matrix pertains (supplying the same release that was supplied to the *select release* operator).

Definition 3.23 (Create example set from file/feature matrix) *Let F^* be a set of file/feature matrices and V_{req} be a set of identifiers of required variables. Let r be the release from which the features of F^* are taken, and* $\text{present}(d,f,r)$ *indicate if defect d is present in file f at release r. The following operator creates a prediction example set including one example for every file not missing a required variable at the given release:*

$$\text{examplesetfile}(F^*, V_{req}) = \left\{ (f, A, D) \;\middle|\; \begin{array}{c} \forall v_{req} \in V_{req}\, \exists x, F \in F^*\, x > 0 \wedge (f, v_{req}, x) \in F \\ \wedge\big(A = \{(v,x) \mid \exists F \in F^*\, (f,v,x) \in F\}\big) \\ \wedge\big(D = \{d \mid \text{present}(d,f,r)\}\big) \end{array} \right\}$$

3.5 Summary

In Figure 3.1, we present a graph that depicts the operators that we defined and how they interrelate to one another. An edge is drawn from one operator to another if the output of the first operator is compatible with the input of the second. In this way, each path through the depicted graph represents a sequence of feature transformations that can be performed, ending with an example set which

can be used for training a defect prediction model. This graph complements this chapter's survey of different defect prediction features, showing how various feature construction operators can be mixed and matched within the same experiment.

Together, the experiment construction system defined in this chapter and the metric construction system that was defined in Chapter 2 formally define our method for computing metrics from an application's change history and converting those metrics into machine learning features. This method will be implemented and used throughout the remainder of this work. In the following chapter, we will describe the *machine learning algorithms* that utilize these machine learning features (or example sets) for training a vulnerability prediction model that can help find vulnerabilities during a quality assurance task.

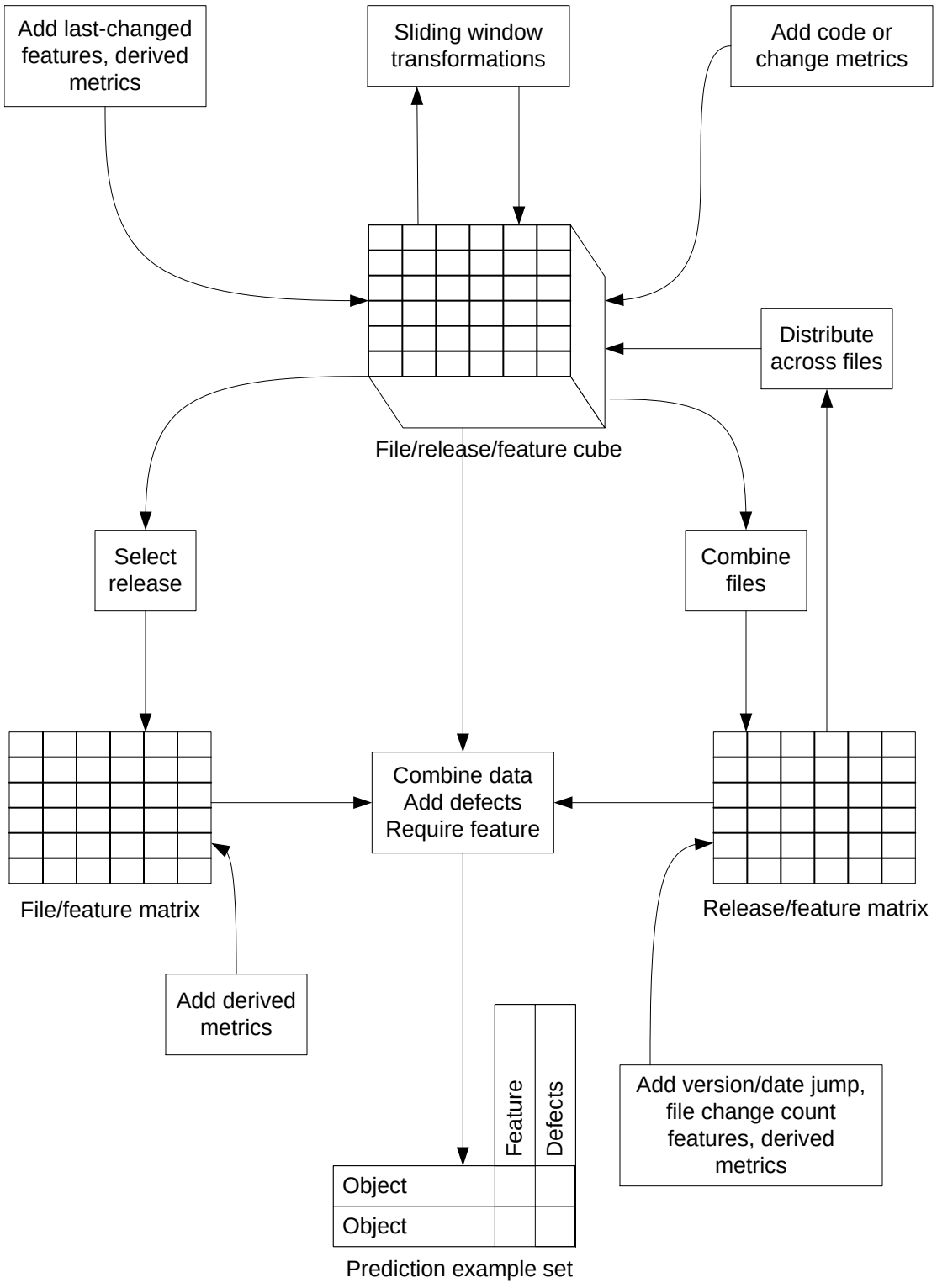


Figure 3.1: Operations for constructing metric variants

Chapter 4

Machine learning algorithms for building vulnerability prediction models

In previous chapters, we described how families of static code metrics and change metrics can be built and then transformed into example sets, which are the data structures that are passed to machine learning algorithms for vulnerability prediction. In this section, we explore the final methodological aspect of vulnerability prediction – the model training process itself.

We used the Weka 3.7.11 [57] software package to perform all machine learning model training and prediction processes, coupled to a customized version of the `RWeka` package to allow for example weighting to be performed from R. This means that we did not attempt to implement the core machine learning algorithms from scratch, because mature implementations of these algorithms were freely available in Weka. However, we did design and implement several *meta-learning* algorithms, which wrap these core machine learning algorithms in order to provide capabilities (such as effort-sensitivity) not present in the core algorithms themselves. We describe both the core machine learning algorithms and the meta-learning algorithms that we designed throughout this chapter.

4.1 Choosing a machine learning algorithm

Many different machine learning algorithms have been developed and made available through free software packages such as Weka [57] (the package used in this study). Choosing an appropriate algorithm is important, as the machine learning method used to build a defect predictor can have a greater impact on effectiveness

than the features, metrics, or other aspects of preparing the data for the example set [92].

In this study, we used four types of machine learning algorithms (i.e. models). We chose these models because they were readily available through Weka with no tuning required, and because they have frequently been used in past defect prediction studies:

- **Random forest**: Random forest models are "forests" (collections) of decision trees which vote on the correct class of each example that is classified. These models are pure classification algorithms – in other words, they do not yield probability estimates, and the example weighting method is used to make this classifier effort-sensitive or cost-sensitive. (Although it is hypothetically possible to use the vote percentages for a random forest as a stand-in for a probability estimate, we are aware of no work establishing the theoretical soundness of such an approach. Furthermore, because this kind of probability estimate substitute could only take on a small number of different values, ties would become a problem when ranking examples by probability.)

- **Naive Bayes**: Naive Bayes models are simple Bayesian probability models that can be fit without supplying any prior information or a structural model describing the interrelationships between features. This type of model is "naive" in the sense that it assumes that the features are independent, which is clearly not the case for most of our metrics; however, practical experience has found Naive Bayes models to be highly effective in many applications even when this assumption clearly does not hold. Because the Naive Bayes model yields a probability estimate for each example, the probability estimate-based method for effort-sensitivity and cost-sensitivity can be used.

- **Logistic Regression**: This is the standard Weka implementation of a multi-

nomial logistic regression classifier. Probability estimates are generated and utilized as they are in the Naive Bayes model.

- **Linear Regression**: The Weka implementation of linear regression utilizes the Akaike Information Criterion for feature selection, meaning that not all features will be included as variables in the resulting linear regression model. Because this is a regression model, it yields vulnerability count estimates rather than vulnerability probability estimates; however, this estimate can be used in the same way to make the linear regression model into a cost-sensitive and effort-sensitive classification model.

4.2 Cross-validation experimental design

The machine learning portion of each experiment consists of a *training* step, which generates (or fits) a machine learning model, and a *prediction* step, which applies the model to the data in order to yield a set of predictions. One source of bias that must be avoided in the experiments is related to using the same data in the training step and the prediction step. This causes the performance indicators to be unduly influenced by the data on currently *known* vulnerabilities (or, rather, vulnerabilities that were "seen" while the model was trained), overestimating the performance at which the model would find *unknown* vulnerabilities at a later date (in this case, vulnerabilities not seen by the model during the training process).

In a real-world scenario, the vulnerabilities being sought by the model would be, by definition, unknown, making this a non-issue. (A related issue – regarding how the model would be acquired if no vulnerabilities are known yet – is discussed in Chapter 7.) However, in our experiments, rather than dividing vulnerabilities into hypothetical "known" and "unknown" sets, which would reduce the effective size of our dataset and hinder our ability to compare features and experimental setups, we

utilize *cross-validation*, a widely used [38, 58, 93, 139] experimental technique for defect prediction.

When performing cross-validation, we randomly split the examples in the example set into a fixed number of partitions (or *folds*), equalizing (to the greatest extent possible) the number of vulnerable examples within each fold. We fix the number of folds to be 10, a common choice in defect prediction experiments [58]. Performing cross-validation while equalizing the number of vulnerabilities per fold (called *stratified* cross-validation) reduces the variance introduced by the random splitting and ensures that a model will never be trained using a subset of the data containing no vulnerabilities.

After splitting the examples, we repeat the training and prediction steps of the model training process for 10 iterations. During each iteration, one fold is left-out during the training step, and only that fold us used in the prediction step. Because each example is in exactly one fold, all examples have exactly one prediction, as in the case where cross validation was not performed at all.

4.3 Evaluating and comparing experimental outcomes

When a machine learning process is executed on an example set, the output of this process is a set of *predictions*. A prediction encodes the subset of examples from the example set which the model deems likely to be vulnerable. For example, when performing an experiment which constructed a file-level example set, the *prediction* is represented by a set of files which the machine learning model judged to have an elevated likelihood of containing vulnerabilities.

To evaluate the performance of an experiment, the predictions generated by the model must be compared with the set of examples that actually contained (or introduced, in the case of change-level or release-level experiments) the vulnerabilities. With the simplest experimental setups, an ideal model would predict all the

examples which actually contained vulnerabilities to be vulnerable, and predict all of the other examples to be non-vulnerable. However, when *effort* is taken into account, the ideal predictions from a model may differ from which files were actually vulnerable. For example, in a file-level example set, if an extremely large file contains a vulnerability, it may be more cost-effective to refrain from inspecting this file, even if the file is certain to be vulnerable. (This is discussed in more detail in Section 4.5.1.)

Ultimately, a comparative evaluation of two models (i.e. two experiments) must be based on the number of vulnerabilities covered by the prediction (vulnerabilities that were "hit" by the predictive model), the number of vulnerabilities not covered by the prediction (vulnerabilities "missed" by the model), and the amount of code contained within the examples that were part of the prediction. Such an evaluation effectively assesses how well a model would improve the efficiency of a notional *code inspection* task. In this code inspection task, a developer or security professional reads source code and attempts to find vulnerabilities present in the code. However, because it is infeasible to inspect all of the source code, the predictions provided by the model are used to select portions of the code to inspect.

In a file-level prediction experiment, specific files in a release are chosen by the model, and the entire selected files are inspected for vulnerabilities. In a release-level prediction experiment, the model is applied to each release, and all changes made during the release are inspected if the model predicts the release to be vulnerable. In a change-level prediction experiment, the model is applied to each file that was changed during the release, and the changed portions of each selected file are inspected. In all three of these experimental setups, the goal is to maximize the number of vulnerabilities covered by the inspected code while minimizing the amount of code that is inspected. In this notional code inspection task, it is assumed that the inspector catches all vulnerabilities in the inspected code. However, if the

inspector was not perfect and vulnerabilities in the inspected code were caught with a fixed probability less than 1, then the relative performance assessment of multiple experiments would be consistent, as all experiments would be affected equally.

4.3.1 Performance indicators

In order to compare the performance of two or more experiments, *performance indicators*, or statistics based on the model's hits and misses, can be computed from the model predictions. The performance indicators must be chosen carefully to match the performance goals. For example, the base release of Moodle has 2922 files, 24 of which contain the 25 vulnerabilities in this release (two are in the same file). Consider a trivial predictive model that simply predicts that all files are non-vulnerable. If *accuracy* (the percentage of correct predictions) was used as the sole performance indicator, these predictions would be 99% ($\frac{2898}{2922}$) accurate, despite the fact that the model is useless. In the case of a different trivial predictive model, which predicts all files to be vulnerable, consider the *recall* performance indicator, which is the percentage of vulnerabilities covered by the predictions. This trivial model would have 100% ($\frac{25}{25}$) recall, despite being as useless as the first one.

The first example illustrated the problem of *class imbalance* in machine learning, which frequently arises in software defect prediction [72, 128, 167]. Because the vast majority of examples fall into the same class (non-vulnerable), accuracy-based performance indicators incentivize predicting that too few examples are defective, or vulnerable. This is a particular problem in vulnerability prediction (compared to standard defect prediction) because security defects are generally far fewer in number than non-security defects. The implications of this on the machine learning process will be discussed in Section 4.5.

The vulnerability dataset is not complete, because it does not contain any of the currently unknown vulnerabilities which are likely to be present in the source

code, as discussed in Chapter 5. Furthermore, even if the dataset contained all known vulnerabilities, it would still have an unknown number of *unknown* vulnerabilities, vulnerabilities which existed (or which still exist) in PHPMyAdmin or Moodle but have never been found or reported. For this reason, the primary performance indicators for comparing experimental results should be insensitive to the total number of vulnerabilities in the dataset, ensuring that an incomplete vulnerability dataset does not skew the performance indicators.

This drives the selection of three primary performance indicators for assessing our experimental results:

- **Inspection (I)**: The amount of code or the number of examples covered by the vulnerability prediction. For a non-effort-sensitive evaluation, this is the raw number of covered examples; for an effort-sensitive evaluation, it is the total effort summed across all covered examples. Effort-sensitive evaluations are further explored in Section 4.5.

- **Inspection ratio (IR)**: The proportion of code inspected, derived by dividing the inspection by the maximum possible inspection, which would be attained by covering all examples in the vulnerability prediction. This is sometimes referred to as the Inspection Rate [139].

- **True Positives (TP)**: The number of vulnerabilities covered by the vulnerability predictions.

- **Recall (R)**: The proportion of the vulnerabilities covered by the vulnerability predictions.

When used together, inspection ratio and recall allow for the performance of an experiment to be fairly assessed while avoiding skewed or biased results due to the factors mentioned previously. An additional advantage of using these two

performance indicators is that the usefulness of a prediction model can be compared to a *null model*, corresponding to a random selection of examples to predict, by simply comparing **IR** and **R**. If the recall is not greater than the inspection ratio ($\mathbf{IR} \geq \mathbf{R}$), then there is no advantage to using machine learning, as random selection of examples to inspect would yield the same or better results. On the other hand, if the recall is greater ($\mathbf{R} > \mathbf{IR}$), then the use of machine learning provided an advantage when performing the vulnerability prediction task.

4.3.2 Targeting performance indicators when comparing experiments

After performing several cross-validation experiments with several different example sets, the relative performance of those several experiments can be compared by comparing the **IR** and **R** performance indicators. However, this only yields a partial ordering of experimental performances, rather than a total ordering, meaning that it cannot always be determined if one performed better than the others. Consider the case where one experiment has both a higher **IR** and a higher **R** than another experiment. This means that one model found more vulnerabilities while inspecting more code, while another model found fewer vulnerabilities while inspecting less code. It cannot be determined if the higher **R** was due to an improvement in the model or the fact that more code was inspected in the second case.

This raises the issue of the *sensitivity* of a prediction model (which is distinct from the concept of effort-sensitivity). When comparing several experiments, the model with the higher **R** can be said to be more "sensitive" because more code is predicted to be vulnerable. This is similar to the metaphor of a detector that can be calibrated to trigger more often, at a higher sensitivity, or less often, at a lower sensitivity. The issue with comparing experimental results arises because the sensitivity of the prediction model and prediction process was uncontrolled. Being able to control the sensitivity of a model would also increase the practical utility

of the prediction process, by allowing for users to specify a code inspection *budget* (measured in a number of lines, files, or changes) and receive a targeted set of predictions that fits within this budget.

For this reason, we implemented a sensitivity parameter in our model training algorithms, allowing for a specific **IR** to be targeted by the model. This sensitivity parameter is implemented by the meta-learning algorithms later described in this chapter. For now, just consider that the training process can be executed in such a way that the resulting model will have a certain value of **IR**.

Because we can control the value of **IR**, we can now impose a total ordering on experimental results by simply comparing the values of **R**, because **IR** is fixed. In order to examine the performance of a model at multiple **IR** values, we instruct the training process to produce multiple sets of predictions at multiple sensitivities, such that at least one set of predictions has an **IR** close to .10, .20, and .40. In addition, we also explore the performance of the model at a wide range of sensitivities, attempting to cover the entire range of **IR** from .01 to .99 such that no value between .01 and .99 is more than .05 away from some **IR** of some set of predictions.

These fixed values of **IR** then allow us to compute additional performance indicators as follows:

- $\mathbf{R_5}$: Recall when **IR** is fixed to .05
- $\mathbf{R_{10}}$: Recall when **IR** is fixed to .10
- $\mathbf{R_{20}}$: Recall when **IR** is fixed to .20
- $\mathbf{R_{40}}$: Recall when **IR** is fixed to .40
- $\mathbf{I_{20}}$: Inspection when **R** is fixed to .20
- $\mathbf{I_{40}}$: Inspection when **R** is fixed to .40
- $\mathbf{I_{80}}$: Inspection when **R** is fixed to .80

- $\mathbf{I_1}$: Inspection when **IR** is fixed to 1 – maximum possible inspection

Any of the above performance indicators can be compared between experiments in order to establish a total ordering over some performance criterion. The latter indicators, which measure **I** instead of **R**, are used when comparing different experimental setups that yield different values of $\mathbf{I_1}$, making the use of **IR** inappropriate. In this case, the sense of the performance indicator is inverted (lower values are better).

4.3.3 Performance evaluation with cost-effectiveness curves

By using the revised set of performance indicators above, we address the problem of being unable to impose a total ordering over the performance of various experiments. However, this introduces the additional consideration of needing to select a specific value of **IR** before doing the performance evaluation. Frequently, one experiment might have, for example, a better $\mathbf{R_5}$, while another has a better $\mathbf{R_{20}}$. In a case like this one, it still might not be clear which is "better". An additional problem is that (as described in more detail in Section 4.5), it might not be possible to fix **IR** at the desired value. The training algorithm might not be able to find a sensitivity parameter that hits the value within an acceptable tolerance, or it might not even be possible to hit the desired **IR** at all (for example, when no combination of files collectively contains precisely 5% of the product's source code). This problem is aggravated when performing release-level prediction, because of the extremely large variance in the sizes of the releases.

The solution to both problems is to base the performance indicators on an *interpolated* performance curve. In the previous section, it was mentioned that the prediction algorithm seeks to cover a range of **IR** from .01 to .99. This is done by computing performance indicators for every set of predictions that was made and drawing a curve through points representing the two performance indicators in

question (e.g. **R** versus **IR**). By doing this, **IR** (or, in some cases, **R**) can be fixed to any value and the performance indicator can be read off of the interpolated curve. The error introduced by the interpolation process is minimized by ensuring that the entire range of **R** is well covered, and that particular values of **R** targeted by specific performance indicators such as $\mathbf{R_{20}}$ are directly computed by the training algorithm as closely as possible.

This performance curve will generally pass through the set of points formed by each set of predictions with the **IR** performance indicator acting as the X-coordinate and the **R** indicator acting as the Y-coordinate. If several points share the same **IR**, they are combined by taking the mean of all the **R**. In this way, the performance curve becomes a function of **IR**. This curve is known as a *cost-effectiveness curve* and has been used in several previous defect prediction studies [7, 93, 124, 125, 126].

Cost-effectiveness curves are similar to ROC (Receiver Operating Characteristic) curves, which have also been used to evaluate defect prediction models [65]. However, there are some significant differences between these two types of curves. ROC curves are only defined in terms of positive and negative classifications. They cannot account for the fact that examples may have larger numbers of vulnerabilities (more than 1) or differing effort, making them potentially inappropriate for defect prediction evaluation [93]. A ROC curve cannot be expressed as a function of the X-coordinate (because several points can have the same X-coordinate). Unlike with cost-effectiveness curves, all points on a ROC curve have a Y-coordinate greater than or equal to that of points with a lesser X-coordinate (the plot of a ROC curve resembles that of an increasing function). With cost-effectiveness curves, it may intuitively seem that a point with a greater **IR** should not have a lesser **R** than a point with a lesser **IR**. However, in practice, this sometimes occurs due to random factors when building a classifier.

We construct interpolated cost-effectiveness curves by connecting a sequence of

points with increasing **IR** with line segments, averaging points with an identical **IR** as described previously. We then compute performance measures fixed to a specific **IR** (such as $\mathbf{R_{20}}$) by simply locating the point on the curve with the appropriate **IR**. To find points fixed to a specific **R** (such as $\mathbf{I_{20}}$), a similar curve is constructed with **R** and **I** on the X-axis and Y-axis respectively.

Using cost effectiveness curves introduces the additional possibility of computing the *area* under the cost-effectiveness curve and using this area as an additional performance indicator. As mentioned previously, one experiment might have, for example, a better $\mathbf{R_5}$ indicator but a worse $\mathbf{R_{20}}$ indicator than another. By integrating the curve, the performance of two experiments can be compared without needing to select a particular **IR**. The indicator that results from this integration is called the area under the cost-effectiveness curve, or **AUCEC** [126].

The **AUCEC** indicator is very similar to the AUC (Area Under the Curve) indicator commonly computed in conjunction with ROC curves. The AUC indicator is typically computed with a trapezoid-based area measurement [42], which is equivalent to our method of integrating under a cost-effectiveness curve which has been interpolated by connecting points with increasing **IR** with line segments. Values of the **AUCEC** indicator range from 0.0 to 1.0, with higher numbers indicating better performance. A null model where $\mathbf{R} = \mathbf{IR}$ at all points will have an **AUCEC** of 0.5, meaning that a model should have an **AUCEC** above 0.5 for it to have demonstrated usefulness for vulnerability prediction. Figure 4.1 depicts a notional cost-effectiveness curve showing how the area of an **AUCEC** is measured.

The **AUCEC** indicator reflects the performance of a model across the entire range of **IR**; however, not all values of **IR** are equally important. For example, using a model to target inspection of 5% of the code could result in significant effort savings in comparison to not using the model at all. On the other hand, using a model to *discard* 5% of the code, leaving 95% of the code to inspect, is unlikely to

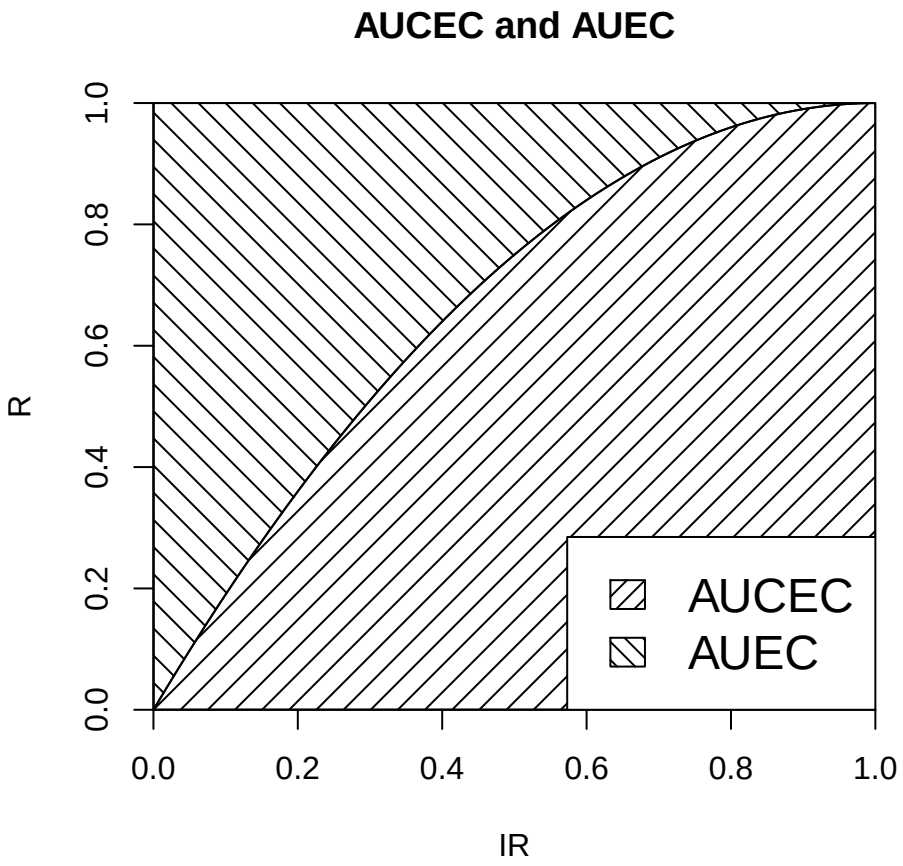


Figure 4.1: Notional cost-effectiveness curve depicting the areas which are integrated when computing an AUCEC or AUEC performance indicator

be worth the trouble. Clearly, the performance of a model at **IR** = .05 is much more important than the performance at **IR** = .95. For this reason, we define an additional indicator, $\mathbf{AUCEC_{50}}$ similar to one proposed by [7] that only integrates the curve where **IR** < .50. To make the range of possible values extend from 0 to 1, $\mathbf{AUCEC_{50}}$ is then computed as double this area. The $\mathbf{AUCEC_{50}}$ of the null model will be 0.25, rather than 0.5 as is the case with the basic **AUCEC** indicator.

Although the **AUCEC** indicator allows for the performance of several experiments to easily be compared across an entire range of **IR**, note that as discussed previously, comparing **IR**-based indicators is inappropriate when multiple experimental setups with different $\mathbf{I_1}$ are being compared. In these cases, the performance indicator must be based on the raw effort **I**.

To provide an **AUCEC**-like performance indicator for comparing multiple experimental setups, we introduce a novel **AUEC** indicator, or area under the effort curve. Note that the **AUCEC** indicator essentially represents an "mean" recall across all inspection ratios, because **IR**, and hence the limits of integration, always extend from 0 to 1. In contrast, the **AUEC** indicator represents a "mean" *amount of effort* across all *recalls*. A higher **AUEC** is undesirable in the sense that more effort is required to find the same number of vulnerabilities, which a higher **AUCEC** is desirable in the sense that more vulnerabilities are found with the same amount of effort expended.

To compute the **AUEC** indicator, an effort curve must be drawn with **R** on the X-axis and **I** on the Y-axis. As with the **AUCEC**, the area of this curve must be computed, with the limits of integration extending from 0 to 1. From Figure 4.1, note that **R** is on the Y-axis and **I** is on the X-axis. As seen in this figure, by flipping the X and Y axes, it becomes apparent that the area to be integrated to compute the **AUEC** indicator is the inverse of the area integrated for the **AUCEC** indicator. Because the new Y-axis is **IR** rather than **I**, this can then can be multiplied by $\mathbf{I_1}$

to derive the **AUEC** from the **AUCEC** without computing a second curve. The **AUEC** can be computed from the following formula:

$$\mathbf{AUEC} = \mathbf{I_1} \cdot (1 - \mathbf{AUCEC})$$

In general, when comparing multiple experiments, we will prefer to compare the $\mathbf{AUCEC_{50}}$ indicator when comparing identical experimental setups (as signified by identical $\mathbf{I_1}$) and the **AUEC** measure otherwise.

4.4 Repeating experiments and testing for statistical significance

If the same machine learning process is repeated multiple times on the same example set with the same parameters, different predictions (and hence different performance indicators) will likely be observed between repetitions of the experiment. This is due to two sources of randomness:

- *Cross-validation folds*: The cross-validation learning process entails splitting the data into several partitions (or folds) before building the machine learning models. This splitting is done at random, resulting in the machine learning models being trained with different input data every time the process is repeated.

- *Random classification algorithms*: The machine learning algorithm which we use the most in this study (random forest) utilizes a random number generator, making random decisions during the model training process. This can result in the machine learning algorithm outputting a different model every time the algorithm is run, even on identical input data.

Because of the random factors that cause the performance indicators to vary across multiple repetitions of the same experiment, we repeat each experiment 10

times with a different random seed at each repetition. We then compute a full set of performance indicators for each repetition. Finally, we take the mean performance indicator for each repetition and report this mean throughout the tables of experimental results in this document. Furthermore, when plotting performance curves, we internally compute one curve for each iteration, and then average all the curves together, such that the Y-value for any given X-value is the mean of the corresponding Y-values across all curves. For this reason, many plots and tables of performance indicators will report that the prediction process found a fractional number of vulnerabilities.

Although reporting averages reduces the impact of the variance of the performance indicators, there is the additional possibility that two performance indicators will have too high of a variance for them to be compared and for conclusions to be drawn. In other words, in some experiments, the random variation introduced as part of the prediction process may overwhelm any performance differences due to differences in the actual models.

In order to evaluate if a performance difference between two experiments is an artifact of random variation or reflects true differences between the experimental results, we perform a Wilcoxon rank-sum test at $p = .01$ on each performance indicator, comparing each indicator associated with an experiment with the corresponding indicator associated with a *baseline* experiment which is chosen in advance.

Note that this test does not provide any particular level of assurance that specific experimental errors, such as Type-I or Type-II errors, have been ruled out. Because hundreds of significance tests and performance comparisons are performed throughout this work, the likelihood that at least one performance difference is wrongly reported to be significant (e.g. the experiment-wise error rate) is relatively large. This hypothesis test is simply being used as a rough instrument to assess if the randomness-induced noise is overwhelming the observed performance difference

in any given case. Consider that because this study was only done on two PHP applications, reducing the experimentwise error rate would not necessarily increase the generalizability of the results, because a significant conclusion within one application may not hold across the universe of all applications that can be studied. By preparing additional datasets which consist of automatically extracted vulnerability data from a large number of Linux packages (as proposed in Chapter 7), it may be possible to perform large-scale studies for a greater assurance of generalizability in future work.

4.5 Meta-learning algorithms for effort-sensitive classification

Most of the machine learning algorithms examined in this study are known as *classification algorithms*. Classification algorithms input a training example set and output a model, which can predict that individual examples (including examples that were not part of the training data) are vulnerable or non-vulnerable. The use of vulnerability prediction in a quality improvement activity ultimately boils down to building (or acquiring) such a model and then utilizing it to find "suspicious" code that merits further investigation.

In the previous section, we observed that accuracy (the percentage of correct predictions against known data) is an unsuitable performance indicator for evaluating vulnerability prediction models. However, the issue with accuracy measurement has broader implications – classification algorithms typically train models to maximize accuracy, presenting additional problems for vulnerability prediction:

1. Due to the class imbalance inherent in vulnerability data, the classifier is incentivized to build a trivial model that predicts all files to be non-vulnerable in order to maximize the classifier's likely accuracy in the face of unknown data. Simply adopting more suitable performance indicators does not alter

this behavior – it only exposes the problematic behavior of the classification algorithm.

2. The classification algorithm does not take effort into account, so it is not tuned to choose the smallest amount of code likely to contain the largest number of vulnerabilities. To the extent that larger examples (such as larger files) will tend to contain more vulnerabilities, this incentivizes the algorithm to build a trivial model which simply predicts that the largest examples are vulnerable. Such a model may not improve the efficiency of quality improvement tasks over using no model at all.

3. As discussed in the previous section, it is desirable to tune the sensitivity of a model so it tends to select a larger or a smaller amount of code to inspect, in accordance with the user's budget. In particular, such tuning is essential when performing experiments for comparative performance evaluation.

Many defect prediction studies address at least one of these problems to some extent in order to achieve reasonable performance. One study [93] developed a meta-learning framework to preferentially search for models that took effort into account, seeking to solve the second problem. Several other studies [100, 128, 167] used a variety of techniques, such as undersampling and cost-sensitive classification, to correct for class imbalance, addressing the first problem. The cost-effectiveness curves of [126] solve the first and third problems by using probability-based classifiers to produce a continuum of prediction results across the entire curve. They also partially address the second problem of effort sensitivity by selecting smaller examples before larger ones. For probability-based classifiers and regression algorithms, another approach [88] further optimizes this case by preferentially selecting examples with the highest relative risk.

Because of our emphasis on comparing different experimental setups, which

can only be done with effort-sensitive evaluation, making classifiers effort-sensitive is especially important. However, none of the above techniques enable effort-sensitive classification with the random forest algorithm, which has proven to be an effective classification technique for a variety of vulnerability prediction problems. In this section, we introduce a *meta-learning* algorithm, or an algorithm that wraps and enhances existing machine learning algorithms, based on *cost-sensitive classification* [36] and *cost-proportionate example weighting* [174]. This algorithm can optimize a variety of classification algorithms for effort-awareness while simultaneously accounting for class imbalance and allowing for the sensitivity of the classifiers to be adjusted.

The concept of *cost-sensitivity* is related, but distinct, from the concept of *effort-sensitivity*. Our ultimate goal is to build effort-sensitive classifiers for vulnerability prediction (classifiers that prefer smaller examples while staying within a total effort budget), while the lower level machine learning concept of cost-sensitivity is the tool we use to accomplish this.

4.5.1 Class labels, example sets, and cost-sensitive classification

Traditional classification algorithms operate by assigning class *labels* to examples and then building models that are optimized to assign the correct labels to the classified examples (including examples that weren't part of the training set) as often as possible.

Unfortunately, when dealing with effort-sensitive vulnerability prediction, predicting the "correct" label for an example may be contrary to the goals of the problem that is actually being solved. Consider the case depicted in Table 4.1. In this case, an algorithm has estimated the probability that a vulnerability exists in each of four examples. (We will assume the probability estimates to be reasonable, and ignore the possibility that multiple vulnerabilities could be in the same exam-

Table 4.1: Illustration of the effects of effort-sensitivity when buliding vulnerability predictors

Probability of vulnerability	.99	0.4	0.32	0.4
Effort	1000	100	100	150
Expected effort/vulnerability	1010	250	313	375
Optimal labels	FALSE	TRUE	TRUE	TRUE

ple. Also, classifiers do not necessarily produce good accurate probability estimates, thus meta-learners shouldn't assume that such estimates are always available [36].)

When classification algorithms maintain internal probability estimates, they typically set a threshold of 0.5 and will choose all examples with at least that probability of being vulnerable. Such a classifier would choose the first of the four examples, which is extremely likely to be vulnerable, and leave out the other three, which are not. However, when taking the effort required to inspect each file into account, it would have been optimal to inspect the last three files and ignore the first, even though a vulnerability is almost certainly present in the first file, because it is so large that inspecting a combination of other files would likely find more vulnerabilities while reading less code.

If the classification algorithm used in the experiment supplies probability estimates like these, then it is a simple matter to select examples in ascending order based on the expected cost per vulnerability (described in Section 4.5.3). However, if the classification algorithm does not give such probability estimates, then the user of the algorithm has no choice but to take the recommendations supplied by the classifier as-is. Although the user of the algorithm does know the effort associated with each example, this information cannot be meaningfully used without the probabilities. The implication is that because the classifier must directly generate effort-sensitive predictions, the labels on the training data must reflect the desired effort-sensitivity strategy. *This means that in the training data, some examples must deliberately be labeled with the "wrong" class so the training algorithm yields models*

which reflect the desired effort-sensitivity strategy.

This class flipping concept (also noted by [36]), when combined with the technique of *example weighting*, motivates the training strategy which we describe below.

4.5.2 Effort-sensitive classification through class flipping and example weighting

We present a method for training vulnerability predictors by applying cost-sensitive example weighting, in order to address the three issues with conventional training algorithms which we mentioned above (class imbalance, effort-sensitivity, and sensitivity adjustments). Example weighting is the practice of allowing for the training process to treat certain examples as being more "important" than others. The model training algorithm will then build a model that places a high priority on classifying examples similar to the highly weighted ones correctly, even at the expense of classifying other examples incorrectly.

Cost-sensitive classifiers take into account the notion that a higher cost is incurred when misclassifying some examples than when misclassifying others. Rather than attempting to maximize the accuracy of the resulting model, cost-sensitive training algorithms attempt to minimize the total misclassification cost, measured as the sum of the misclassification costs of each misclassified example. Example weighting can be used to implement a cost-sensitive classifier [174]. Because of the support for direct example weighting in Weka 3.7.11, which was used to execute the machine learning portions of the experiments in this study, we use example weighting (rather than, for instance, a resampling method) as a mechanism for implementing cost-sensitivity.

We observe that two kinds of costs are associated with both correct and incorrect predictions: (1) the cost of inspecting an example, whether vulnerabilities were found in the example or not, and (2) the cost of failing to inspect an example which

contains vulnerabilities. Let c_{true} be the cost of a positive classification (inspecting an example) and c_{false} be the cost of a negative classification (not inspecting an example). Let e be the effort associated with the example and v be the number of vulnerabilities in the example. The example weights are then set as follows:

$$c_{\text{true}} = e \tag{4.1}$$

$$c_{\text{false}} = v \cdot m \tag{4.2}$$

where m is a constant which influences the sensitivity of the classifier.

The example label and weight are then computed as:

$$\text{weight} = |c_{\text{true}} - c_{\text{false}}| \tag{4.3}$$

$$\text{label} = \begin{cases} \text{true} & \text{if } c_{\text{true}} < c_{\text{false}} \\ \text{false} & \text{if } c_{\text{false}} < c_{\text{true}} \end{cases} \tag{4.4}$$

Note that in the case where $c_{\text{false}} = c_{\text{true}}$, the weight of the example will be zero – regardless of the label, the example will have no influence in the classification process. By weighting examples according to the formula above and flipping labels when necessary (when the cost of inspecting the example outweighs the cost of missing the vulnerabilities in it), we infuse cost-sensitivity, and by extension effort-sensitivity, into a training algorithm (such as random forest) which ordinarily would not support it. This also has the side effect of allowing for classification algorithms (as opposed to just regression algorithms) to incorporate the number of vulnerabilities in an example, rather than just a flag indicating the presence of vulnerabilities, in a theoretically sound way.

Table 4.2: Cost-sensitive example weighting with two sensitivity constants

$m = 1100$				
# vuls	1	2	0	1
e	1000	1500	1000	1500
c_{true}	1000	1500	1000	1500
c_{false}	1100	2200	0	1100
weight	100	700	1000	400
label	TRUE	TRUE	FALSE	FALSE

$m = 2000$				
# vuls	1	2	0	1
e	1000	1500	1000	1500
c_{true}	1000	1500	1000	1500
c_{false}	2000	4000	0	2000
weight	1000	2500	1000	500
label	TRUE	TRUE	FALSE	TRUE

Table 4.2 depicts the weights and labels for four examples for two different values of m. Note that at $m = 1100$, the fourth example is labeled as being non-vulnerable even though a vulnerability is actually present in the example. At $m = 2000$, the label of the fourth example flips to being vulnerable, and the weights of the vulnerable examples increase while the weight of the non-vulnerable example stays the same. This is because at higher values of m, failing to inspect a vulnerability is considered to be more undesirable than at lower values of m. Hence, examples with vulnerabilities are weighed higher accordingly.

For a given example set, there is a close relationship between the value of m and the number of lines of code inspected (total effort) selected by the predictions. Higher values of m will generally result in more code being inspected, although random effects in the classifier and the indirect nature of this parameter mean that the total effort will not strictly increase as m increases. However, the ability to target a desired total effort allows us to satisfy the third goal of cost-sensitivity (computing a performance curve through a range of inspection ratios). The search algorithm which we use to accomplish this is detailed in Section 4.6.

4.5.3 Effort-sensitive classification for probability-based classifiers and regression models

The generalized method for making effort-sensitive classifiers that we described above has the advantage of working with any classification algorithm that supports example weighting, even if the models fitted by the algorithm return no information other than the predicted class of an example. As the random forest classification algorithm, which falls into this category, is effective in many cases, we will utilize this method extensively throughout the experiments in this work. However, this method has the disadvantage of being computationally expensive, because the process of searching for the right m may cause many models to be trained until the desired **IR** has been attained.

We utilize an alternative, simpler method when working with regression algorithms (such as linear regression) or classification algorithms that yield true probability estimates (Naive Bayes or logistic regression). This leverages the fact that when supplied with an input example, these models not only output a predicted class, but also output some kind of numerical prediction or expectation. This numerical prediction can then be adjusted to account for the effort of each example, and the examples can then be sorted by their predicted relative risk [88].

The entire process is as follows:

1. If the machine learning algorithm is a classification algorithm, as opposed to a regression algorithm, the number of vulnerabilities in each example is converted to a boolean, which indicates if the file has at least one vulnerability.

2. The model is trained as usual, without any attempts to make it effort-sensitive. Cross-validation is performed here, assigning one numerical prediction to each example.

3. The numerical predictions or probabilities for each example are divided by the

effort of the example, yielding the predicted (or expected) number of vulnerabilities per line of code.

4. The examples are sorted by this predicted number of vulnerabilities in descending order.

5. Examples are selected in order until the effort budget has been expended.

Because the effort budget is not needed until the end, there is no need to repeat the training process for various values of m, which was done in the previous method.

4.6 A meta-learning algorithm for building cost-effectiveness curves

In the previous section, we introduced a meta-learning algorithm for effort-sensitive classification, which wrapped existing machine learning algorithms so they would build models that prefer low-effort predictions over high-effort ones. This meta-learning algorithm required that a tuning parameter m be set to establish how much a model would be penalized if it selected high-effort examples for inspection. We now present a second meta-learning algorithm that determines this value of m while accomplishing two additional goals – searching for specific values of **IR** and building a cost-effectiveness curve. As specific in Section 4.3.2, the entire learning and meta-learning process is repeated until a wide range of **IR** values is covered without leaving any large gaps.

The mechanism for achieving control over **IR** was alluded to in Section 4.5.2, which demonstrated how adjusting the value of m affects the **IR** performance indicator of the final results. However, this control is indirect – working backwards from the desired **IR** and attempting to compute the corresponding m would not be reliable due to the random nature of the classification process. We instead implement a *search algorithm* which adaptively builds models with various values of m,

effectively building a cost-effectiveness curve while also hitting the targeted values of **IR** as closely as possible. This search algorithm consists of the following steps:

Choose a tolerance value: Because specific values of **IR** can only be targeted indirectly by adjusting m, it is not always possible to target a particular **IR** when performing an experiment and hit the targeted value exactly. Even if it were possible to directly target an **IR**, note that **I** does not increase continuously, but instead increases in a stepwise manner as an increasing number of examples are inspected, with the sizes of the steps determined by the effort required to inspect each example. For these reasons, a tolerance is built into the search algorithm so the search terminates when it finds a prediction which is close enough to the desired **IR**. We initially assign the tolerance to be the greater of .01 or $2 \cdot \frac{\text{median}(e)}{\Sigma e}$ where e is the effort for all examples.

Choose initial bounds for m: **IR** usually increases as m increases, because a higher m indicates a higher penalty for missing vulnerabilities, so more examples are inspected as a result. The general strategy of the search algorithm is to choose an m that yields predictions close to the desired **IR** by iteratively narrowing down the range of m where the desired **IR** may fall. To begin this process, initial values of m must be chosen. We start with a lower bound of $m_{\text{lower}} = \min_x \frac{e_x}{v_x}$, where e_x is the effort for example x and v_x is the number of vulnerabilities. We then choose an upper bound of $m_{\text{upper}} = m_{\text{lower}} \cdot \exp(10)$.

Expand the initial bounds for m: Next, we train two models, one at each bound of m, and verify that all targeted values of **IR** fall between the two values of **IR** that result from this. In some experiments, the upper bound of m was not high enough to cover the maximum desired **IR**. In these cases, we squared the upper bound and repeated this step until it was high enough.

Locate uncovered IR targets: The next step is to compile a list of target values of **IR** that have not yet been covered by any model trained during a previous

iteration. These include the **IR** targets enumerated in Section 4.3.2. In addition, if there are any intervals with a size greater than .05 which do not contain the **IR** of any trained model, the midpoint of each such interval is added as another target value. A target or interval is deemed to have been covered if the **IR** of at least one trained model is within the current tolerance value of that point or interval.

Mark intervals of m to explore: We next sort the list of all values of m that have been explored in ascending order. Adjacent pairs of values in this list will then form intervals which contain the unexplored values of m. Each explored value of m will also have an **IR** that resulted from its exploration. Find all intervals of m where at least one uncovered **IR** target falls between the interval endpoints' values of **IR**. These intervals will be marked for exploration, because some m within the interval will likely yield one of the values of **IR** that is being sought.

Choose points to explore within marked intervals of m: At this point, at least one interval of m should be marked. If no such intervals are marked, then all targeted values of **IR** must have been found, and the process is finished. Otherwise, points within the marked intervals must be explored in an attempt to find the targeted values of **IR**. Such points are explored in batches of 21. All points in a batch are evenly distributed across all the marked intervals on a log scale, meaning that within each interval, the points to explore will appear to be evenly distributed between the interval's endpoints when taking the logarithm of both the points to explore and the endpoints. The logarithmic scale is used here so smaller values of m are covered more densely than larger ones.

Explore points and iterate: Finally, the search algorithm has selected some values of m that should be explored, in order to accomplish the goal of finding the m that will result in a trained model having the desired value of **IR**. To explore these points, models are trained and evaluated (as described earlier in this chapter) for each such value of m. The performance indicators from the training process are

then fed back into the next iteration of this algorithm, as they will determine which values of m should be tried in the next iteration.

Increase the tolerance value if necessary: As mentioned previously, the tolerance value determines how close to a target **IR** the algorithm must get before the target is deemed to have been covered. Sometimes, a targeted value of **IR** cannot be covered under the current tolerance value and the tolerance must be increased. The tolerance value is increase by multiplying it by 1.1 whenever exploring additional values of m has been become fruitless. The exploration is deemed to have become fruitless when m is changing by no more than $\frac{1}{100 \cdot \text{mean}(e)}$, where e is the effort for the examples. This "rule of thumb" was derived from our observations that changes in m smaller than this very rarely resulted in the target **IR** being hit.

4.7 Summary

In this chapter, we described the process for training machine learning models from example sets, as well as the performance indicators which can be used to determine how well an individual model served the purpose of making vulnerabilities easier to find. Although this training process is independent of any particular experimental setup, we use it in conjunction with the three setups – file-based, change-based, and release-based – which we defined in the previous chapter. In this chapter, we also described the cross-validation process that allows for an entire experiment to be run on a single example set – training the model and computing performance indicators on the same data without biasing the results. Central to our machine learning algorithms and performance indicators was the concept of *effort*, or the notion that the models will be used to guide a code inspection task, and that the cost of performing such a task will be proportional to the amount of code to be inspected.

This effort-centric learning process will be crucial when we present our ex-

perimental results, as dissimilar experimental setups can only be comparatively evaluated in relation to a unifying indicator such as effort. In the chapters that follow, both our meta-learning algorithms and our performance indicators will be fully utilized, allowing for us to systematically explore each aspect of fine-grained vulnerability predictive models while minimizing factors which would confound a fair comparison of the results.

Chapter 5

Tools and datasets for evaluating vulnerability prediction techniques

In order to evaluate our methodology for constructing families of metrics, transforming those metrics into machine learning features, and building vulnerability predictive models, we first needed to collect a historical vulnerability *dataset* that would support this analysis. A vulnerability dataset, consisting of a source code history and historical information on vulnerabilities that were present in that codebase throughout a given time period, will facilitate a *retrospective* evaluation of how well our predictive techniques would have worked in a real-world scenario.

Security vulnerabilities originate by inadvertently being *introduced* into a codebase. Several months or years later, the vulnerabilities are often *discovered* (sometimes serendipitously) and fixed. Between the introduction and discovery dates, a vulnerability is considered to be *latent*, because it persists in the codebase, presumably without the developers' or users' knowledge. Our retrospective evaluation leverages this latency period to gauge how predictive models could help developers in practice, by applying the models to source code artifacts and noting if the models initiate the inspection of artifacts with latent vulnerabilities, which would hypothetically result in the vulnerabilities being discovered and remediated. The dataset in this chapter was specifically collected at a granularity that would enable the tracking of latent vulnerabilities to perform this retrospective analysis. This dataset is described in the first half of this chapter, in Section 5.1.

In the second half of this chapter, we will discuss the tools and scripts that extract metrics and other data from the vulnerability dataset. To perform this extraction process, the code in the repository must be *ingested* and parsed into abstract syntax trees. Next, the *differences* between consecutive releases must be

computed, in order to support the computation of change metrics that characterize the change between releases. Next, some mechanism for *developing metrics* must provided for the practitioner or experimenter to easily define families of metrics (or, more specifically, to build labeling functions). Finally, a metric *computation engine* is needed to apply the constructed metrics against the ingested trees and difference maps, yielding tables of code and change measurements for each file at each release. These tools are discussed in Sections 5.2 and 5.3.

5.1 A vulnerability dataset for PHP web applications

Over time, the research community has released public datasets to support research in the software engineering and modeling communities. For example, the PROMISE repository [94] has many freely available defect prediction datasets, which usually incorporate information on the locations of software defects in a product, source code metrics for each module in the product, and sometimes the source code for these modules as well. However, none of the publicly available datasets contained enough detail on vulnerabilities to support the experiments in this work. (In fact, none of the datasets had vulnerabilities at all – all of them covered software defects in general – although even if we had switched our focus to general defects, they still wouldn't have had the necessary detail.)

Some existing defect and vulnerability datasets included:

- *Software metric and defect count datasets* [38, 94, 177]: These datasets typically contain module metric and defect count data as described above. However, they are unsuitable for this study because they typically don't cover multiple versions of each product, or if they do, they localize defects to a single point in time and don't track the evolution of these defects.

- *Exploit traffic data* [70, 109]: Although these datasets cover vulnerabilities

(rather than defects in general), they only provide traces or simulations of exploits without localizing the vulnerabilities to particular source code files. For this reason, they are not suitable for this study.

- *Static analysis tool benchmarks* [77]: Some benchmarks do provide vulnerable source code files and localize the vulnerabilities to their respective locations in these files. However, they do not feature multiple versions of the same program, and they are often artificially limited to make the static analysis task more tractable.

- *Vulnerability databases* [110, 112, 114]: Public vulnerability databases compile information on known vulnerabilities from a wide range of products. This makes them useful starting points for our own data collection; however, because the databases do not specify which parts of the source code are affected by the vulnerability, they are not directly usable for our purposes. In addition, as we will discuss later in this chapter, their information on the range of versions affected by the vulnerability is often wrong.

Because none of the existing databases were suitable for our purposes, we collected a new vulnerability dataset as part of this research. This dataset covers two open-source PHP applications and consists of:

- Source code for each version of the application, which was extracted from Git
- A catalog of vulnerabilities that affect each of the versions covered in the dataset
- Vulnerability localization information for each version (which vulnerabilities were in which files at each point in time)
- Commit identifiers and other metadata on the vulnerabilities

As an aid to replication and to facilitate future work in this field, we also released this dataset to the public as part of our previously published work [165]. The following sections describe how the data was collected and how the quality of the data was assured. Also, we present some statistics related to the vulnerabilities in this dataset.

5.1.1 Choosing applications for the dataset

We limited this study to applications written in the PHP language, as the domain of web applications for vulnerability prediction was relatively unexplored, and because so many PHP applications are afflicted by vulnerabilities. We selected two open-source PHP applications: PHPMyAdmin and Moodle.

We selected open-source applications to ensure that a source code repository featuring the history of the codebases and vulnerabilities would be available. In order to select the two specific applications that we studied, we consulted the National Vulnerability Database [110] and selected the two applications with the largest number of vulnerability entries that were not *plugin-dominated*. A plugin-dominated applications is one that had many vulnerabilities in optional plugins rather than the main codebase; because of the importance of tracking the history of a single codebase over a long period of time, we excluded these applications. One application in our dataset – PHPMyAdmin – is a web-based SQL database management tool. The other – Moodle – is an online learning course management system.

5.1.2 Processing Git repositories

Next, we extracted the source code for each release of both of these projects from their *Git* repositories. Git is a distributed version-control system that allows for users to download the entire history of a project and consult it "offline", without having to request specific versions from a centralized server. The presence of Git

repositories for both products made it much easier to obtain the source code for each release, and finding the vulnerability-introducing commits (described later) was much quicker that it otherwise would have been. In order to eliminate potential confounding effects caused by packaging errors or differences, we obtained all source code from the Git repository, rather than downloading and extracting redistributable packages for the products.

Both products generally had a standardized, three-part version number, with a major, minor, and patch component. For example, version 2.5.3 of a product would have a major release of 2, a minor release of 5, and a patch release of 3. In some cases, more granular versions of the products or hotfixes were also produced. We did not obtain source code for these granular versions and limited ourselves to versions with the aforementioned form.

To begin the Git processing step, we mapped each release to a particular Git *commit*. By supplying the Git toolchain with a commit identifier, it is possible to extract a snapshot of the product's source code as it was after said commit had been applied to the repository. For Moodle, the process of mapping release identifiers to commits was easy – Moodle reliably tagged the commit for each release with the release's version number. In contrast, although PHPMyAdmin often tagged releases with their version numbers, we found that the tags were sometimes missing or incorrect, mostly due to errors introduced as repositories were migrated from one system to another over the project's decade-long history (issues similar to those encountered by others [143]). In these cases, we had no choice but to carefully review commit comments, changelogs, and publicly available information such as release announcements in order to determine which historical commits corresponded to which releases.

In the end, we obtained 100 PHPMyAdmin and 95 Moodle releases. All Moodle releases back to the first could be obtained; however, no PHPMyAdmin releases

before 2.2.0 were available because the history did not extend beyond that point. In addition, 4 PHPMyAdmin releases could not be obtained because no information on a particular release was available or because commits were missing from the Git repository. These missing releases presented a particular problem throughout this work, because they often impeded us from obtained an accurate view of the software's evolution. For example, if a vulnerability is first observed in a release after a missing release, the release where the vulnerability was introduced becomes unclear, and the vulnerability cannot be used in any prediction experiment involving the evolution of the software over time. This issue will be discussed in more detail in Section 6.1.

Although the repository contains files other than PHP files, including source code in languages such as JavaScript, these files are not used in this PHP analysis study and are excluded from the dataset. In addition, we manually identified files containing test code or libraries maintained by third parties, because neither was capable of containing the kind of vulnerabilities we analyzed for this study. These files were excluded from the dataset as well, both to reduce the volume of data to process and to prevent them from skewing the prediction results which are to follow.

Aside from knowing which commits are associated with each release, it is also important to be able to identify the *parent* release associated with each release. This is because change is measured by comparing two releases; hence, it is important to know in which "direction" to go in order to trace forwards and backwards through the release history. Identifying the parent release is not always a simple matter of lexicographically sorting version numbers. For example, PHPMyAdmin 3.0.0 is the direct successor to version 2.11.9, even though versions 2.11.10 and 2.11.11 also exist. This happens when development diverges into multiple branches, with new features being added to one branch and bug fixes being applied to another.

To determine the parent release for any given release, we compare the similar-

ity between pairs of releases by using Git's rename-tracking diff functionality and measuring the number of lines in the resulting diff output. Pairs of releases with smaller diffs were considered to be the most similar. Then, whenever there was a question as to which release was the parent of another, the feasible parent with the smallest difference was used. For example, it is a given that version 2.11.11 has the parent 2.11.10, so no diff is performed here. Version 2.11.9 is a feasible parent of version 3.0.0, but version 1.0.0 would not be, because too many version numbers would be skipped.

5.1.3 Finding security advisories

After obtaining the source code histories of two PHP applications, the next step was to acquire a set of *known* vulnerabilities in each application. Limiting the dataset to known vulnerabilities means that we did not attempt to discover any vulnerabilities in the products which had not yet been patched by the developers. Instead, we began the data collection process for each application by obtaining information on the complete set of released security patches for each, and then working backwards from the patches to identify the nature and origin of each vulnerability. This ensured that we did not introduce an extra source of bias into the data by searching for vulnerabilities ourselves, and that the experimental results reflected how well the models would perform when faced with issues that the products actually experienced.

In order to mine these vulnerabilities and incorporate them into our dataset, we first compiled a master list of security vulnerabilities in each application which would serve as a basis for our data collection. These master lists considered of all of the security advisories that had been issued for PHPMyAdmin and Moodle. These security advisories almost always coincide with the release of a patch or patch release, providing both a description of the vulnerability and a hint as to how the vulner-

ability was manifested on a source code level. For Moodle, we used the National Vulnerability Database (NVD) [110], while for PHPMyAdmin, we PHPMyAdmin's internal security advisory database.

Because of the large number of available security advisories, we took a sample of security advisories and mined vulnerabilities from these. As we will discuss shortly, our definition of a "vulnerability" makes it so one advisory could map to multiple vulnerabilities, which may have different types. Therefore, the end result of the collection process yielded more vulnerabilities than security advisories that were sampled.

For several reasons, some security advisories did not yield any vulnerabilities. These reasons include:

- The available information was not sufficient for us to find the vulnerability in question. (affecting 4 PHPMyAdmin and 4 Moodle advisories)
- Although a putative vulnerability was announced and patched, we could not find a means to actually exploit the vulnerability. (2 PHPMyAdmin and 2 Moodle)
- The issue being announced affected the entire program, and there was no specific change or file which caused the vulnerability. For example, in one case, cross-frame attack protection was omitted from the entire application. (4 PHPMyAdmin and 1 Moodle)
- The vulnerability was caused by third-party libraries or components. We omitted these vulnerabilities because the components were not developed through the application's repository, and hence their change history was not visible. (1 PHPMyAdmin and 8 Moodle)

5.1.4 The vulnerability data collection process

After compiling a list of candidate security advisories, the next step was to collect the information that we needed for each vulnerability. The required information included:

- The commit where the vulnerability was introduced
- The first version that incorporated the vulnerable commit
- The commit where the vulnerability was fixed
- The version that incorporated the fix
- The file that contained the vulnerability (which may migrate from file to file) at each version

Previously, we noted that one security advisory could map to multiple "vulnerabilities". Here, we more formally define what we consider a "vulnerability" for the purposes of this work. A vulnerability is an unintentional software defect (in contrast to an intentional backdoor) which allows for a malicious user (authenticated or unauthenticated) to compromise a security property of the application or system (such as integrity, confidentiality, or availability) in a way that affects the system's owner or other users. The defect that constitutes the vulnerability may span multiple lines of code. Because we don't restrict vulnerabilities to a single line of code, we then must consider when multiple defective lines of code constitute a single vulnerability, or when they constitute multiple vulnerabilities.

In the accounting scheme we used for this work, a security advisory can yield multiple vulnerabilities, establishing a one-to-many relationship between them. A separate vulnerability is counted for each file and each vulnerability-introducing commit, with the limitation that only one vulnerability will cover each distinct

exploit that is possible. For example, if a single commit introduced vulnerable code into two separate files enabling two separate exploits, this is counted as two vulnerabilities. However, if all the vulnerable code was introduced into the same, file, then this is only counted as one vulnerability, unless it was introduced during two separate commits enabling two separate exploits.

Vulnerabilities may *evolve* over time, in the sense that the vulnerable code is modified without being repaired. In some cases, files are substantially refactored while preserving the vulnerability, changing the characteristics of the exploit but preserving the vulnerability. In some extreme cases, code is moved from one file to another while preserving the vulnerable functionality, or code is completely rewritten while faithfully replicating the vulnerability that was in the old code. *In our dataset, all of these situations are considered to be cases where a single vulnerability evolves and moves, rather than cases where one vulnerability is removed and an identical one is put in its place.* This was motivated by a desire to model activities that *worsen* the security posture of a system, rather than activities which simply relocate security problems that already exist.

Because our accounting of vulnerabilities requires us to locate vulnerability-introducing commits, most of the vulnerability data collection process revolves around this activity. The process typically starts by viewing the patch issued at the same time as the security advisory and informally determining how the unpatched code would have been exploited, based on our knowledge of PHP exploit patterns. In some cases, consulting exploit code available on public websites [112] was useful. For some vulnerabilities (especially for Moodle), this process was complicated by the fact that security advisories were issued at the same time as major releases, obscuring the change that fixed the vulnerability. We were generally able to find the vulnerabilities in these cases anyway, although it was much more difficult to do so. (In some extreme cases, a vulnerability fix had apparently been intentionally

concealed among other changes with a misleading commit comment.)

Once a vulnerability fix was located, we then traced backwards through the application's revision history until the vulnerability-introducing commit was located, keeping in mind that (as described above) refactorings of the vulnerable code must be followed until the vulnerability's inception is found. This was a largely manual process, although Git's "blame" command, which shows the commit where a specific line of code was introduced, was sometimes helpful.

Many past defect prediction studies attempted to automate the above step by automatically "blaming" the origin of code modified by defect-fixing changes, and then considering those commits to be defect-introducing commits. This approach [144], often referred to as the *SZZ algorithm*, has the advantage of being completely automated. However, we found that such an approach would have been ineffective for our purposes, because vulnerable code was nearly always modified multiple times (and even extensively refactored) before being fixed. The ineffectiveness of the SZZ algorithm for vulnerabilities may be due to the fact that vulnerabilities are long-lived, while defects are potentially located more quickly (before the defective code is modified again) because they more obviously impact user functionality.

After locating the vulnerability-introducing commit and file, the next step was to trace the vulnerability through every vulnerable revision of the software. In other words, for each vulnerability, at every susceptible version, exactly one file must be vulnerable. We consider a version to be susceptible if it is released before the patch that fixes the vulnerability, as all later versions are presumed to have been patched. We refer to this process as *tracing* the vulnerability through the release history.

Unfortunately, the process of tracing the vulnerability (or even finding it in the first release where it existed) was not straightforward. Even if it is determined that vulnerable code was committed at a certain time, the next release of the product

will not necessarily contain that vulnerable code (as the code may only exist in an unreleased, long-lived development branch). Although an intact Git version history would have made it possible to trace a commit to a release, the Git history was not always intact – in some cases, commit parents were not recorded properly due to conversions from other version control systems. Even if the repository was intact, this process still would not have been completely reliable because developers do not always record metadata on merges in Git properly.

As an alternate approach, we extracted short strings from the vulnerability-introducing commits, and then searched all releases in the datasets for the first release where the string appears. In some cases, the vulnerable code was rewritten before being released for the first time; when this happened, we tried a different string, extracted from the rewritten code, until the vulnerability was found.

After identifying the first file and release where the vulnerability was present, the next step was to identify all of the files which contained the vulnerability until the point where it was fixed. Note that a vulnerability can appear to move from file to file throughout its lifetime because (1) the file itself is renamed, or (2) the code is refactored and moved to another file. The former case is dealt with by the file mapping process (discussed in Chapter 2). To detect the latter case, we look for cases where the original file of the vulnerability differs from the file that was patched when the vulnerability was fixed. When this occurs, we repeat the string matching process described in the previous paragraph (with alternative strings, if necessary) until the vulnerability has been tracked from its origin to the location where it was fixed.

5.1.5 Vulnerability data collection results

This vulnerability collecting process yielded 75 vulnerabilities for PHPMyAdmin and 51 vulnerabilities for Moodle. In Table 5.1, we categorize these vulner-

Table 5.1: Classes of vulnerabilities in each application

	Moodle	PHPMyAdmin
Code Injection	7	10
CSRF	3	1
XSS	9	45
Path Disclosure	2	12
Authorization issues	28	6
Other	2	1

abilities with a simple classification scheme.

These vulnerability categories serve as a simple way to summarize the kinds of issues that are represented in our dataset. They will also become part of a statistical analysis in Chapter 6 regarding the correlation between vulnerability categories and vulnerability discoverability. A brief description of each category is below:

- **Code Injection**: These vulnerabilities allow for attackers to subvert the normal control flow of a program in arbitrary ways, usually by inducing the execution of PHP, SQL, or native code of the attacker's choosing. This category also includes the ability to set arbitrary variables on the server, resulting in a similar effect.

- **CSRF**: CSRF (cross-site request forgery) vulnerabilities allow for malicious, third-party websites to induce a user's browser to perform an action on the target website. This allows for a malicious actor to perform an an action under the victim's identity.

- **XSS**: XSS (cross-site scripting) vulnerabilities allow for a malicious user to plant harmful Javascript onto the target website. Victims who later visit the target website execute the Javascript, possibly allowing for actions to be performed under their own identifies as with CSRF vulnerabilities.

- **Path Disclosure**: These vulnerabilities allow for the full pathname of the application's PHP directory to be obtained by an attacker. This information is sometimes needed to launch a subsequent attack.

- **Authorization issues**: This category encompasses defects allowing an attacker to use an application in an unauthorized way (or violate one of its security properties) not falling under another category. Vulnerabilities in this category include: Forgotten/incorrectly implemented authentication or authorization, information disclosure vulnerabilities, and improperly implemented encryption.

- **Other**: All other vulnerabilities fall into this category, including defects enabling phishing or man-in-the-middle attacks.

The vulnerability categories were distributed differently across both applications. PHPMyAdmin had more cross-site scripting vulnerabilities, while Moodle had more authorization issues. This is possibly because authorization mechanisms are more central to Moodle (due to its use as a multi-user system deployed on the internet), introducing more opportunities for this kind of vulnerability to occur. Despite this, though, our categorization scheme was appropriate for both applications. However, a different categorization scheme may be required for domains other than web applications written in PHP, as the other domains may not be susceptible to vulnerabilities such as cross-site scripting.

Some vulnerabilities were identified but could not be included in the dataset for several reasons. 5 PHPMyAdmin and 2 Moodle vulnerabilities could not be included because they affected Javascript or other non-PHP code. 1 PHPMyAdmin and 3 Moodle vulnerabilities were excluded because the vulnerability's introduction and fix occurred in different files, preventing the vulnerability from being traced through the history of the product. (For example, one Moodle vulnerability was

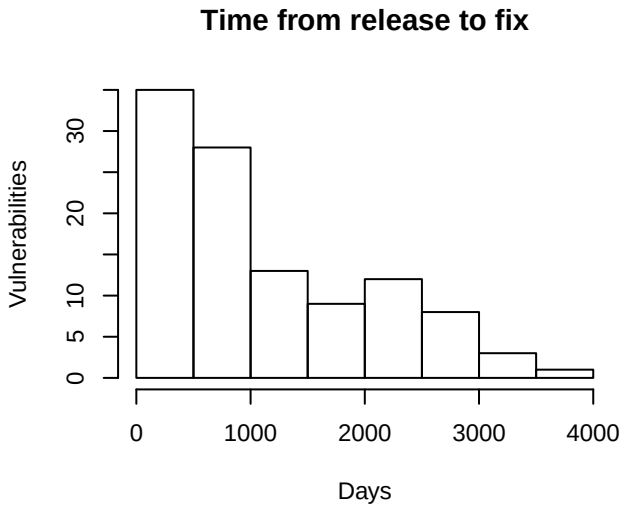


Figure 5.1: Histogram depicting the number of days between the first release of a vulnerability and the vulnerability being fixed in the version control repository

introduced into one file but fixed by making a compensatory change in a different file.) Finally, 3 Moodle vulnerabilities were not included because they were all copies (clones) of another vulnerability (which was included in the dataset).

5.1.6 Vulnerability lifetimes

Now that we have collected a comprehensive dataset on vulnerabilities in PHP web applications, we can accurately measure the *lifetimes* of vulnerabilities, or the amount of time that passed between the day when the vulnerable code was released to the general public and the day when the vulnerability was fixed. A histogram with these vulnerabilities is depicted in Figure 5.1.

The results show that many vulnerabilities were extremely long-lived. The median vulnerability lifetime was 871 days, and one vulnerability was not fixed until 3993 days after it was introduced. The slow discovery of vulnerabilities provides additional evidence that the SZZ approach for defect localization is inappropriate for vulnerabilities (as discussed earlier in this chapter). In addition, these long

vulnerability lifetimes represent a large window of opportunity that attackers have to discover previously unknown vulnerabilities and compromise a system, underscoring the need for better approaches to avoid introducing the vulnerabilities in the first place.

5.1.7 Data on vulnerable application versions

One side-effect of building our vulnerability dataset was that we were able to independently determine which versions of each project were susceptible to any given vulnerability. This information, which we will refer to as *vulnerable-versions data* has often been found to be inaccurate in past research [105]. To further explore this question, we compare the vulnerable-versions data that we collected against the vulnerable versions reported by public data sources (namely, the vendors' own security advisories or the NVD).

When assessing the accuracy of this vulnerable-versions data, we verify the accuracy of the *first* version in the range – it is much less likely for the last version to be wrong, as there is never any question as to when the vulnerability's patch was released. We present this comparison between the "correct" first version and the first version reported by the security advisories in Table 5.2. With vulnerabilities designated as *advisory earlier*, the security advisory indicated an earlier version than what we found manually; for *inspection earlier*, the reverse was true. (Vulnerabilities where we could not reliably determine the earliest version were omitted from this table.)

As we report in the table, the version information in the security advisories was usually not correct for either application. Note that vendors generally do not have an incentive for determining which old versions of a product are vulnerable, as it is generally expected that users will upgrade when able. It was common to find security vulnerabilities that asserted that all versions of a product were vulnerable, when

Table 5.2: Comparing earliest vulnerable versions in vulnerability advisories with those found by code inspection

	PMA (Vendor)	PMA (NVD)	Moodle (NVD)
Advisory earlier	26	29	18
Inspection earlier	28	25	20
Versions match	14	14	13

we discovered that the vulnerability was actually introduced early in the product's development history but not at the first version. Similarly, many advisories indicated that just a few recent versions were vulnerable, when the vulnerability was actually much more long-lived. In general, the inaccuracy of this vulnerable-versions data underscored the importance of manually tracing vulnerabilities through source code histories to their origins.

5.1.8 Ensuring data quality

Because collecting this software evolution and vulnerability data was largely a manual process, there were many opportunities for errors to be introduced. For this reason, we took measures to minimize these errors when practical.

To ensure that a fair sample of vulnerabilities was obtained, we avoided excluding any vulnerability from the dataset because it were difficult to localize, or due to the perceived unimportance of the vulnerability. Although (as detailed earlier in this chapter) we were occasionally unable to find a vulnerability despite our best efforts, this was rare. To maximize the likelihood that the vulnerabilities that we did find were interpreted correctly, we consulted additional information on individual security advisories when it was available. For example, scripts available from third-party exploit databases [112] were sometimes helpful for determining the exact attack vector for a vulnerability.

Manually mapping releases to commits was another potential source of error, especially due to the complex repository branching structure of PHPMyAdmin. In order to determine that the release mapping was sensible, we screened our dataset for suspicious patterns of change (such as files appearing and disappearing multiple times, which would indicate that the releases were out of chronological order).

Finally, to screen the vulnerability commit and release mappings for errors, we enforced a strict chronology on this data in order to check it. Dates of the vulnerability-introducing commits and first vulnerable versions were compared, and a warning was generated if the dates were too far apart (or if a vulnerability appeared to have been released before the commit was made, which would indicate an error in determining which release was first vulnerable). Similarly, the lifetime of a vulnerability must span at least one release, and when tracing a vulnerability through the source code, the location of the vulnerability immediately before the fix must match the file affected by the fixing commit.

5.2 A tool for ingesting PHP datasets to enable metric computation

For our metric family construction system, the first step in computing metrics for a codebase is to parse each file into an abstract syntax tree (or AST). Because some experiments in this study entail examining individual changes to predict the introduction of vulnerabilities over time, it is necessary to parse each version of each source code file. The determination of which code in a file changed from one release to the next need not be made at this point – the comparison and differencing will occur after the files have all been parsed. Previously, we extracted each version of each PHP application from a Git repository into a separate directory and tracked file copies and renames from one release to the next. From these directories of extracted source files and rename tracking information, it becomes a simple matter to parse each file and assign each AST a *canonical file identifier*, such that two ASTs with

the same file identifier represent two different versions of the same file, even if the file had been periodically renamed.

Although our methodology for processing source repositories and building metrics is applicable to numerous programming languages, we are only concerned with parsing PHP because the applications in our dataset (PHPMyAdmin and Moodle) are primarily written in PHP (the remainder of the code being client-side HTML and Javascript), and nearly all vulnerabilities are found within the PHP portions of the code. Only 5 of the PHPMyAdmin vulnerabilities and 2 of the Moodle vulnerabilities were found in non-PHP code, and these vulnerabilities were excluded from the study.

Because the PHP language was developed without a formal specification or grammar [34], different PHP parsers are based on different grammars. As a result, different parser implementations may yield different kinds of abstract syntax tree nodes, and the metric definitions will be dependent on a specific PHP parser as a result. Because it was one of the few parsers available at the time of this study, we used the PHP compiler front-end of de Vries and Gilbert [34] as built into the compiler developed by Biggar [17].

This parser ingests individual PHP source files and outputs an AST in an XML format. The tools that follow the parser in our pipeline then parse the XML files and build a tree data structure in memory. Figures 2.1 and 2.2 depicts a sample fragment of PHP script and the AST that is generated by the parsing process.

Not all PHP source files were successfully parsed into ASTs. Source file parsing failed when: (1) the source file contained text in a character set not supported by the parser; (2) the source file contained PHP language constructs which did not exist when the parser was implemented; or (3) the source file contained comments and whitespace but no executable statements, preventing a tree from being constructed. These files were excluded from the dataset. In the PHP projects in our dataset, the

issue with unsupported character sets primarily occurred with language translation string files, none of which were responsible for any vulnerabilities. The cases of the files with no source code were apparently caused by unfinished, accidentally released files which never had any content, making it safe to exclude these files as well.

The case of files with constructs not supported by the parser is potentially more problematic. Although the parser doesn't support certain object-oriented PHP constructs which did not exist at the time the parser was written, these constructs were rarely used except in very recent versions of the products. By cutting off the Moodle analysis at Version 2.0.9, limiting the analysis to any releases which were at this version or before, the occurrence of this issue was minimized. In total, only 0.1% file versions for PHPMyAdmin and 0.9% file versions for Moodle were unparsable. Unparsable files are treated as if they do not exist, and no vulnerabilities were within an unparsable file at any version.

As described in Section 5.1.2, four releases of PHPMyAdmin could not be located due to missing downloadable packages and missing tags in the version control system. The presence of missing releases complicated the analysis of PHPMyAdmin for two reasons. First, there is no way to distinguish a vulnerability that is introduced during a missing release from a vulnerability that was introduced in the release after a missing release. Second, change metrics cannot be computed for the release after a missing release due to the lack of a baseline to measure the change against. Because we could not distinguish vulnerabilities introduced after missing releases from vulnerabilities introduced during missing releases, any vulnerabilities that were first seen in the release after a missing release were excluded from the change-level prediction experiments (as well as any file-level prediction experiments where a performance comparison with change-level prediction experiments was done). Any vulnerabilities which were apparently introduced in the first available release of a product were excluded in a similar way. Finally, any releases where

change metrics could not be computed (including the first release of a product) were excluded from change-level prediction experiments.

5.2.1 Computing tree differences

After building ASTs for each revision of each PHP file, the next step is to build tree *node maps* that characterize the differences between two versions of the same file. By mapping the nodes that remain unchanged between one version and another, the added and deleted nodes in each tree can easily be determined [150], giving a full picture of how a source file changed between the two versions.

In Chapter 2, we presented a theoretical discussion of mapping tree nodes between multiple versions of source files. Here, we discuss how we implemented this mapping for the purposes of this study. Because we are only concerned with computing the difference from an older to a newer adjacent revision (i.e. the differences between a release and its parent release), the wrapper tree mapping approach mentioned in the same chapter can be trivially implemented by simply ensuring that the one-to-one mapping property is satisfied. Both tree mapping algorithms which we discuss in this section will satisfy this property.

Because computing the tree node map between two trees is, by far, the most computationally intensive step when computing changes, and because the definition of the particular metric being computed has no bearing on how the node map is constructed, we pre-compute the node maps for pairs of files across all adjacent revisions and save them in the same location where the ASTs are stored.

Early experiments revealed that the process of comparing two ASTs would be memory-bound. Because tree differencing algorithms have polynomial space requirements with respect to the sizes of the trees, comparing the largest source files failed even when allocating a maximum amount of Java memory (10500MB) to the differencing tool. Due to the need to minimize the amount of memory used, we

incorporated the RTED algorithm and reference implementation [117], configuring the implementation so RTED would compute the optimal tree differencing strategy. RTED ensure that all trees can be compared in $O(n^2)$ space with a worst-case time complexity of $O(n^3)$.

Despite this, though, some trees could still not be compared within the available memory. In these cases, where the tree difference could not be computed with RTED, we computed a less constrained tree node mapping by building a preorder traversal for each tree and then using Hirschberg's algorithm to find the longest common subsequence between the two traversals in $O(n)$ space. Note that a mapping made with this algorithm is not guaranteed to satisfy all three properties of a Tai mapping as discussed in Chapter 2. However, because the one-to-one mapping property is still satisfied, such a mapping can still be used to construct valid change and distance metrics.

In Table 5.3, we list the largest typical memory requirements for computing differences between trees of various problem sizes, where the problem size of a comparison is measured as the product of the sizes of each tree. When allocating the specified amount of memory and suppressing comparison of any problems greater than a specified size, comparison of all releases except for one Moodle release succeeded. (Because our implementation stored all of the input ASTs and output data in RAM as well, the effective memory requirements for releases with a very large amount of code are higher. When such problems were encountered for individual releases, differences for these releases were re-computed with a slightly smaller maximum problem size.)

Very large trees with a problem size between 8E7 and 2E9 could not be computed with the RTED algorithm, even when allocated a maximum amount of memory. Hirschberg's algorithm was used to compute node maps in these cases. However, the large majority of trees differences were below this threshold and could be

Table 5.3: Typical memory requirements for RTED tree difference problem sizes

Problem size	Memory
2E7	5000MB
5E7	7700MB
8E7	10500MB

computed with the RTED algorithm.

5.3 A tool for designing families of metrics

In Chapter 2, we introduced a methodology for constructing *metric families* that allowed for both code and change (distance) metrics to be generated from a single metric family definition. As described earlier, such a family of metrics can be defined by writing a *labeling function*, which takes an AST and affixes labels onto selected tree nodes, which characterize the quality of the software which is to be measured. In this section, we present a tool which allows for users to develop and test these labeling functions.

In our tool for designing and computing families of metrics, labeling functions are implemented in the JavaScript programming language. We selected JavaScript because it is an easily embeddable scripting language, allowing for a development environment where a user can design and test a metric family definition without recompiling. Metric family definitions are saved in an internal database, which then provides the metrics used for vulnerability prediction experiments.

Aside from acting as a user interface to develop JavaScript definitions of metrics and save them in a database, the tool also provides feedback in the form of a *scatterplot* interface that allows for the behavior of the metric to be visualized, with respect to other metrics and the vulnerabilities in the dataset. Comparing the values of two metrics provides a quick "sanity check" to ensure that a file metric is functioning as intended. For example, if one metric is designed to be more restrictive

than another, then its values must be less than the values of the other metric if the code is working as designed. If it is desirable to visualize how the values of a metric relate to the vulnerabilities in the dataset, then this can also be examined through the scatterplot interface (for example – to determine if the values for vulnerable files tend to be distributed differently than non-vulnerable files). However, we did not consult this vulnerability distribution information when performing this study – in other words, the metrics were not tailored to find any particular class of vulnerability – because doing so would have corrupted the cross-validation experimental design by introducing foreknowledge of the vulnerabilities' locations.

5.3.1 Developing families of metrics in JavaScript

In Figure 5.2, we depict the user interface for constructing Javascript labeling functions. The `match` dropdown is used to specify if the Hungarian algorithm (as described in Chapter 2) is to be used for value matching when computing change metrics. The available settings are:

- `yes`: For when value matching must always be performed – for example, when labeling line numbers
- `no`: For when value matching should not be performed, such as when labeling mathematical operators
- `either` When a default supplied by the experimental setup should be used because either approach is sensible – for example, when labeling method names

When a metric is to be computed based on the metric family definition, the computation engine evaluates (executes) the JavaScript in the definition. So the JavaScript function can label the tree, two objects are made available within the context of the executed JavaScript:

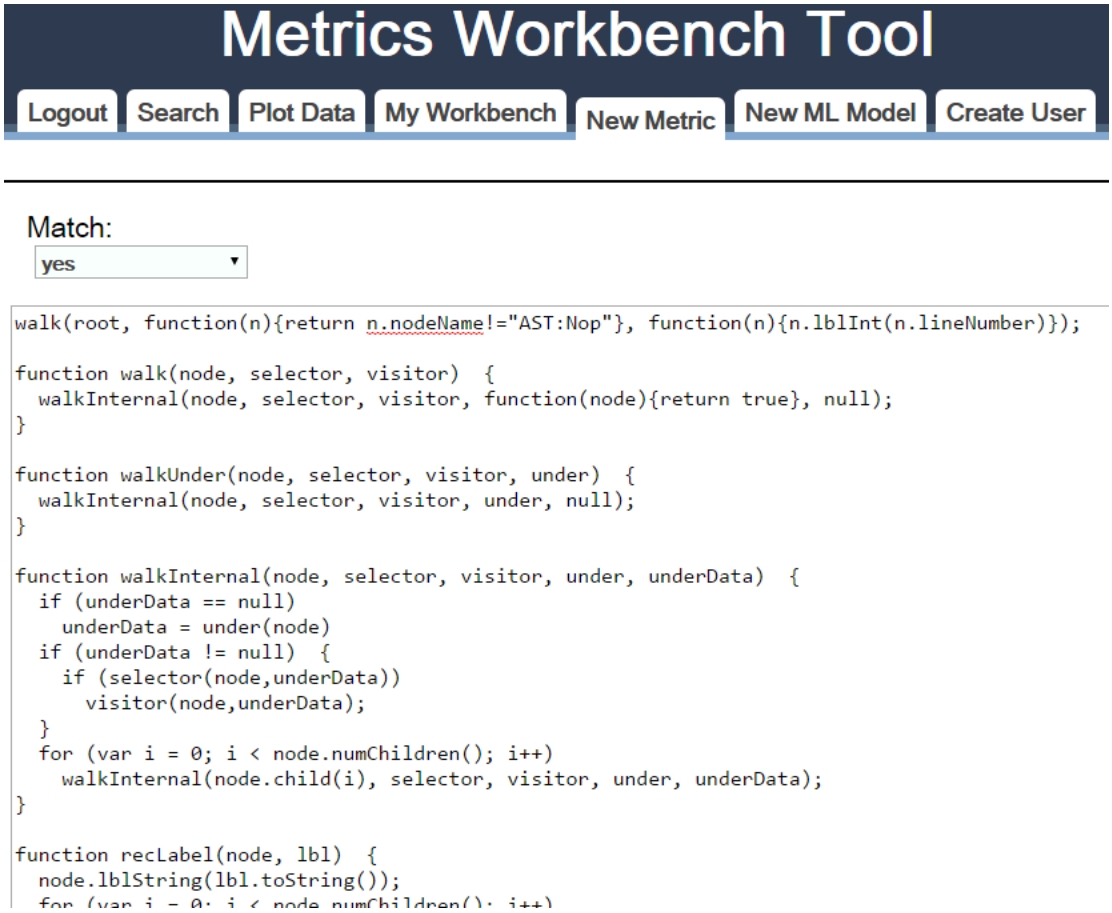


Figure 5.2: Developing a labeling function in Javascript with the metrics development tool

- `root`: The root node of the tree to be labeled. The tree can be traversed (by retrieving the root's children) and labeled through properties and methods of the root.

- `tbls`: The method call and definition lookup tables described in Section 2.1. These lookup tables allow for the potential targets of a function call, or the potential callees of a function definition, to be enumerated. This information is important for constructing coupling and similar metrics. Because functions in PHP are generally placed within a flat, global namespace, the tables are constructed conservatively so a callee or caller is always recorded in the relevant table as long as the name of the function being called matches the name of the function definition in question.

In the remainder of this section, we will discuss the properties and methods defined on tree nodes and tables that can be used to traverse the tree, consult the lookup tables, and place labels on individual tree nodes or entire subtrees. By using these methods and implementing any additional needed functionality by writing extra JavaScript functions within the metric definition itself, a wide variety of metric families can be implemented.

Tree node properties and methods: Various properties and methods of AST nodes allow for the characteristics of the nodes to be introspected and labels to be applied. Some of these properties and methods are essential, while others were implemented for convenience, to more quickly perform common tasks that we encountered when developing labeling functions.

- *Get number of children*: Returns the number of children of this node.

- *Get specific child*: Returns the child node with the specified index.

- *Get parent node*: Returns the parent of this node (unless this node is the root node).

- *Get node values*: Returns all values attached to this node. Note that node values come from the source code, and are not the same as labels. For example, the value of a binary operator AST node type is the symbol of the exact operator being used, because the same node type is used for every type of binary operator.

- *Check for node value*: Checks if the node contains the given value.

- *Label node with string*: Attach a new label to the node using the given string value.

- *Label node with strings*: Attach one or more new labels to the node using the given string values.

- *Label node with integer*: Attach a new label to the node using the given integer value.

- *Label node with ID*: Attach a new label to the node using the node's unique identifier as a value. (In other words, no other node on this AST will have the same label as this one.)

- *Label node with own values*: Attach one or more new labels to the node using the node's values.

Lookup table properties and methods: Similar to the tree node properties and methods, the lookup tables also have properties and methods defined for the convenience of the metric developer. Tables are implemented as ordered maps of string keys (method names) to arrays of string value (the names of the files defining or calling the method with the name in question).

- *Get number of keys*: Returns the number of string keys in the lookup table.

- *Check for key by string*: Checks if the table contains the specified key.

- *Get key by index*: Returns the index of the string in question in the sequence of keys.
- *Get value count by key string*: Returns the number of values under the specified key.
- *Get value count by key index*: Returns the number of values under the key with the specified index.
- *Check for value by key string*: Checks if the specified string value exists under the specified key.
- *Check for value by key index*: Checks if the specified string value exists under the key with the specified index.
- *Get indexed value by key string*: Returns the value with the specified index under the specified key.
- *Get indexed value by key index*: Returns the value with the specified index under the key with the specified index.

5.3.2 Visualizing metrics with animated scatterplots

Animated scatterplot visualizations, which can be displayed through the metric design tool, can depict the both the relationship between two metrics and the relationships between a metric and the presence of known vulnerabilities in a file. The animated nature of the visualization allows for the change, or evolution, of the measurements to effectively be tracked from one release to the next.

Figure 5.3 depicts the scatterplot interface, which shows the values of two metrics (a variable count metric on the X axis and a cyclomatic complexity metric on the Y axis) for all files in a product at the point of a particular release. Mousing over a file's point (pictured) causes a crosshair to appear, allowing for the point's

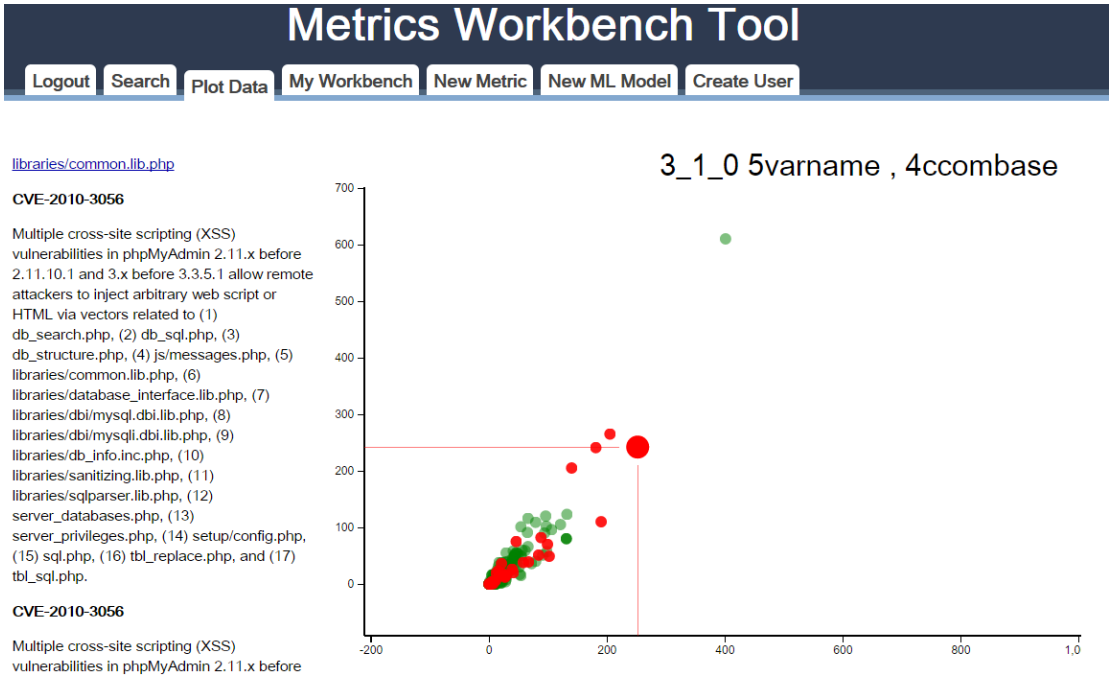


Figure 5.3: User interface for visualizing metrics with the metrics development tool

position to be more accurately determined. This scatterplot can be *animated* by stepping forward or backward through the succession of releases; this movement is under control of the user. When a file has been modified between releases, that file's point smoothly transitions to its new location. Files that were added or removed during a release fade in or out. Because identities of files are tracked across renames, a renamed file's point moves to its new location rather than disappearing and appearing again. The position of a point on the X and Y axes allows for the relative values of two metrics across an entire range of files to be compared visually, verifying that the relative values of a newly developed metric fall within an expected range. For example, a metric measuring internal method calls should not be greater than a metric measuring all method calls in an unrestricted manner, so all points should fall within a certain range on the visualization when plotting these two metrics. We found this to be a useful way to quickly verify that newly developed metrics were behaving in an expected manner.

The color of a scatterplot point indicates if the corresponding file contained a vulnerability during the release in question. Green points represent files that do not contain vulnerabilities at the current release, while red points contain vulnerabilities. The tool may be configured so files with vulnerabilities are colored based on the vulnerability's type – for example, in one configuration, SQL injection vulnerabilities were colored blue and cross-site scripting vulnerabilities were colored orange. This allowed for differing relationships between metrics and vulnerabilities of different types to be visualized. When stepping from one release to the next, the colors of points frequently change as vulnerabilities are introduced and fixed. To allow the user to anticipate when a vulnerability is about to be introduced, files which are about to become vulnerable in the next release of the product are colored yellow. Yellow points stay yellow for one release and change to another color (such as red, blue, or orange) in the next release, when the file becomes vulnerable. Clicking on a file (pictured in Figure 5.3) causes information on the vulnerabilities in the file at the selected release to be displayed.

Although other software evolution visualizations have also depicted relationships between files (such as shared packages or method call relationships), we omit this information from our visualization, as overlaying this information on the graph would likely make the graph unreadable. The lack of a need to spatially arrange the files according to these relationships allows us to show metrics by simply positioning the point. Past visualizations which also incorporated relationship data frequently resorted to strategies such as complex three-dimensional shapes or textures to accomplish this [40].

When vulnerabilities are introduced into files, the simultaneous change in the point's position and color is of special interest, as this movement illustrates the nature of the source code change that caused the vulnerability's appearance. Closely examining these changes allows for the relationship between metrics and vulnera-

bilities to be examined in a more direct way, highlighting two types of associations: *static* associations between the absolute value of a metric and the presence of vulnerabilities, and *dynamic* associations between the change of a metric value and the simultaneous appearance or disappearance of vulnerabilities.

It has been observed that consumers of animated visualizations must be given the ability to independently identify interesting points or events in the midst of large numbers of datapoints making complex movements [2, 161]. To assist users in rapidly identifying vulnerability introduction events, we overlay the plot with two additional visual cues, or features, which highlight these cases. This ensures that points can easily be followed when vulnerabilities are introduced, and that vulnerability introductions are not missed or obscured.

- **Color change features:** As described previously, in the release immediately preceding the introduction of a vulnerability, a file's point on the scatterplot is turned yellow and moved in front of any overlapping points. This alerts the user to the change which is about to take place in the next frame.

- **Cycling animated bubbles:** In a release where new vulnerabilities are introduced, a cycling animated bubble reprises the point movement associated with each vulnerability introduction, replaying the change that just took place and eliminating the need to the need to manually jog the screen back-and-forth to view the movement again. Vulnerabilities introduced upon the file's creation are indicated by a stationary, expanding bubble with a different color. We note that moving bubbles were used by Beyer and Hassan [16] for a similar purpose, highlighting large structural software changes on a source code dependency graph.

Figure 5.4 demonstrates how these two visual cues facilitate easy identification of vulnerability-introducing changes. During the step before the introduction

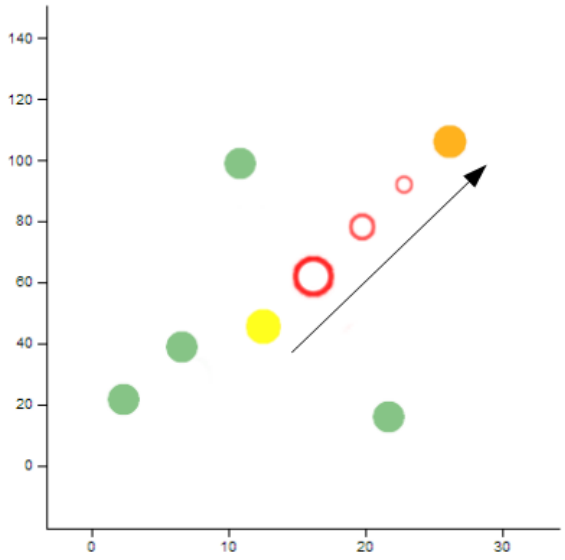


Figure 5.4: Scatterplot animation of a change in a file which caused the introduction of a vulnerability. (The arrow is included for illustrative purposes and is not present in the actual visualization.)

(representing the previous version of the software), the file with the impeding vulnerability turns yellow. In the animation step where the vulnerability is introduced, the point moves to its new location, representing the change in metrics caused by the modification. A cycling, animated bubble then moves between the old and new locations of the point, allowing for the direction and magnitude of this change to be closely examined.

5.4 Summary

In this chapter, we identified that existing defect and vulnerability datasets didn't have the kind of data needed to meet the needs of this study. We then introduced a fine-grained vulnerability dataset consisting of two open-source PHP applications, with information on the introduction, discovery, and evolution of a subset of vulnerabilities in both of these applications.

We then presented several tools which preprocessed and computed metrics

on this dataset. One key preprocessing step involved computing the abstract syntax tree (AST) differences between each successive revision of a file, which is used for computing change metrics when using the metric computation system that we described in Chapter 2. The graphical metric development tool then allowed for metrics to be developed (by writing labeling functions in Javascript) and visualized in order to debug them.

The end result of applying both of these tools is a compilation of code metric, change metric, and vulnerability evolution data for each version of each application in our dataset. In the next chapter, we will use this data to empirically evaluate and compare each of the vulnerability prediction methodologies and setups proposed earlier in this work.

Chapter 6

Experiments and results

In the preceding chapters, we defined and formalized a method for computing code and change metrics, building predictive models, and using these models to guide a code inspection task to regions of the program most likely to contain vulnerabilities. This method was motivated by a structured survey of techniques for predicting defects and vulnerabilities and the identification of gaps that made it more difficult to apply these models in practice. In addition, we collected a vulnerability dataset for two open-source PHP applications, which contained the fine-grained vulnerability data which was necessary for building the predictive models in accordance with the methodology that we defined.

In this chapter, we perform a series of experiments, utilizing the data that we collected and the methodology that we defined, in order to evaluate how our proposed vulnerability prediction methods work in practice. Because we defined many alternative methods for setting up experiments, building features, and training models, the central organizing principle for these experiments is to perform a *consistent and comparative* evaluation of each of these alternatives in practice. Three notable aspects of our experimental setup facilitate this consistent evaluation: (1) ensuring that the same families of metrics are used as predictive features for each experimental setup, (2) evaluating each experimental setup with the same vulnerabilities, and (3) using consistent lines-of-code effort-based evaluation criteria for each experiment.

6.1 General setup of experiments

The process to set up an experiment begins by constructing an *example set* which contains the independent variables (metrics and other features) and dependent variables (vulnerability counts) for the experiment in question. A machine learning training algorithm is then run to build a predictive model. This algorithm attempts to predict the number of vulnerabilities that are present, basing the prediction on the values of the dependent variables, which can be determined without foreknowledge of where the vulnerabilities were. These predictions can then be used as input for quality assurance processes such as code inspections, based on the predictions that some regions of code (or some changes) are more likely to contain vulnerabilities than others.

In Chapter 3, we introduced a system for constructing a wide variety of features and integrating them into example sets. We briefly summarize the three kinds of example sets which our system can construct:

- **File-level**: The features of a file at a particular release predict if the same file at the same release contains vulnerabilities. This prediction model could be practically utilized by applying it to every file in a release, selecting individual files flagged by the model for further scrutiny. This example set is built with the examplesetfile function defined in Section 3.4, as well as the selectrel function to select the base release for prediction.

- **Release-level**: The features of a particular release predict if a new vulnerability was introduced in that release. This prediction model could be utilized by applying it to every release, subjecting the release (the new and modified code in the release) to additional scrutiny if the model predicts that vulnerabilities were introduced. This example set is built with the examplesetrel function defined in the same section.

- **Change-level**: The features of a particular file at a particular release predict if a new vulnerability was introduced into that file at that release. The model is not trained or evaluated to detect the case where an existing vulnerability migrated from one file to another at a release. If the model does happen to predict the vulnerability migration anyway, for the sake of consistency of performance evaluation between experimental setups, the model is not "credited" for discovering the migration. This prediction model could be utilized by applying it to every changed file at every release, subjecting the file's changes (the new and modified code in the file) to additional scrutiny if the model predicts that vulnerabilities were introduced. This example set is built with the examplesetchg function.

Recall that the functions defined in Section 3.4 for constructing example sets take a parameter V_{req} indicating which variables (features) are required. A file, release, or change will not be included in the example set unless each required feature has a non-zero value. When constructing the experiments in this section, we typically declare the metric `Lines of code` to be required. If this metric is missing, then the example is not included at all. This occurs when defining release and change metrics for a release where the source code for the release's parent is missing, preventing change metrics from being computed. If this metric is present but has a zero value, then there is no code to measure (because the file is empty, or unchanged in the case of change metrics), and there is no reason to perform machine learning prediction on the example in question.

6.1.1 Vulnerabilities used in experiments

Table 6.1 presents a summary of the vulnerabilities in PHPMyAdmin and Moodle, including the reasons why certain vulnerabilities were excluded from some experiments. The first section of this table lists the releases that were chosen for

Table 6.1: Version numbers and vulnerability counts for PHPMyAdmin and Moodle

	PHPMyAdmin	Moodle
First release	2.2.0	1.0.0
Base release	3.1.0	2.0.0
Last release	4.0.9	2.0.9
Non-missing releases first–last	100	95
Vuls in file experiments	35	25
Vuls in matched file experiments	31	21
Vuls in change experiments	61	37
Vuls in matched change experiments	46	37
Vuls introduced first commit	3	4
Vuls introduced first release	4	1
Vuls multiple/ambiguous introductions	7	3
Other vuls introduced through base rel	46	37
Other vuls introduced after base rel	15	6
Total vuls	75	51

experiments for each product. The *first release* of a product represents the first release of the product for which source code is available in our dataset. There may have been releases of the product that predated our first release, but these releases were not stored in a version control system and were hence unavailable. The *base release* of a product is a version with a maximal number of latent vulnerabilities (vulnerabilities that have been introduced but not yet fixed). This is release is used for all file-level prediction experiments (because maximizing the number of vulnerabilities will yield more stable prediction results) and any *matched* release-level or change-level experiments (experiments that will be combined with or compared to the file-level experiment, and therefore must end at the same version).

The next section of the table lists the number of vulnerabilities that were included in each class of experiment (i.e. the sum of the independent variable across all examples of the example set). *File experiments* refers to the file-level experiment type described above, and *change experiments* refers to either the release-level or the change-level experiment type, as both of these experiment types can use the same

vulnerabilities to compute the dependent variable. The *matched* file and change experiments require that a file-level experiment and a change-level experiment are performed with the same vulnerabilities. To create this matched set, we first take the intersection of the vulnerabilities that would have been included in the file-level and change-level experiments. Next, we also add any vulnerabilities that would have been included in the file-level experiment had they not been fixed before the base release. This addition is designed to penalize file-level prediction tasks that wait too long before attempting to find the vulnerabilities, losing the opportunity to find them before they would have been found by users outside the development organization.

The last sections of the table give a summary of the number of vulnerabilities in the dataset (not limited to vulnerabilities included in experiments). All vulnerabilities in the dataset fall into one of five categories:

- **Introduced at first commit**: Some vulnerabilities were already present in the first commit into the version control system. Such vulnerabilities cannot be included in change-level experiments, because their introduction does not correspond to any specific version. However, they can be included in file-level experiments if they are not fixed before the base release.

- **Introduced at first release**: Some vulnerabilities were introduced into the codebase after the first commit, but before the first release. These vulnerabilities cannot be included in change-level experiments, because the first release lacks a baseline and has no change metrics. However, they can be included in file-level experiments if not fixed before the base release.

- **Multiple/ambiguous introductions**: Some vulnerabilities were either introduced after a missing release (preventing the actual release of introduction from being properly determined) or were introduced in multiple feature and

maintenance branches, complicating the accounting of the vulnerability's introduction. Again, these vulnerabilities can be included in file-level but not change-level experiments.

- **Introduced through base release**: These vulnerabilities were introduced at or before the base release. They are included in change-level experiments and can be included in file-level experiments if not fixed.

- **Introduced after base release**: These vulnerabilities were introduced after the base release. They are included in change-level experiments, as long as they were introduced at or before the last release. (Some additional Moodle releases exist after the last experimental release, which were excluded from experiments for reasons discussed previously.)

6.1.2 Metrics and features used in experiments

We computed each of the code and change metrics for all files (excepting excluded files as mentioned in Chapter 5) and all analyzed versions of each PHP application. Computing static code metrics for each file was straightforward – after executing the labeling function, the number of distinct labels in each AST was computed (i.e. the number of non-empty A sets as defined in Section 2.2.3). This distinct label count is the static source code metric of Definition 2.7.

The code delta change metric of Definition 2.8 is also straightforward to compute. This metric is computed as the difference between the static code metric of a file and the metric of the file's parent release. (As with all other change metrics that we compute, all change is measured with respect to a release's parent, as defined in Chapter 5.)

Two difference-based change metrics as defined in Definition 2.9 were implemented:

- Add-only change metric: Applying the formula of Definition 2.9 with constants $p_1 = 1$, $p_2 = 0$, $p_3 = 1$, $p_4 = 1$
- Distance change metric: Applying the formula of Definition 2.9 with constants $p_1 = 1$, $p_2 = 1$, $p_3 = 1$, $p_4 = 2$

The major difference between these two metrics is that the add-only change metric only takes *added* code into account, allowing for code to be deleted without any metric registering a change. In contrast, the distance change metric registers both *added* and *deleted* code, allowing for the deletion of code within an individual change to influence the vulnerability prediction.

The A and B sets as defined in Section 2.2.3 were computed for each label. This allowed for the change in each label between the two versions to be characterized – for example, it can be determined if all of the values for a particular label were added (or deleted) from one version to the next. In order to compute the metric, these characterizations are then summed per Definition 2.9.

Most of the complexity of computing change metrics stems from the need to perform *value matching* as originally discussed in Section 2.2.2.3. Informally, the value of a label may change from one version of a file to the next, even when the new label should be considered to be the "same label" as the old one. For example, if a blank line is added to a source file, causing the former line 12 to become line 13 in the next version, the metric should not register the addition of one line and the removal of another, as the same line exists in both. A similar situation occurs when methods are renamed but not modified between two versions (for example, to comply with a new naming convention for the project).

Recall that when designing a particular metric, the tool allows for value matching to be enabled or disabled. If value matching is disabled, then the change metrics are computed with a trivial value map (only labels with identical strings or integers are mapped between versions). If value mapping is enabled, then the Hungarian

algorithm approach outlined in Section 2.5.3 is used to determine which labels are the "same" when comparing two versions. All details discussed in Section 2.5.3, including the prevention of zero-weight matchings, were implemented. For additional performance optimization, trivial matches (values that have exactly one match between the two releases) are removed from the graph before executing the Hungarian algorithm.

Finally, we implemented an additional *atom-based* predictor. The atom-based predictor, which is similar to the token-based predictors which have previously been used in vulnerability prediction [129], leverages the fact that each tree may be labeled with many different values when computing a metric. Instead of utilizing a feature vector of different metrics when performing vulnerability prediction, it is possible to instead utilize a feature vector computed from one single metric, such that each possible label is a separate feature and the number of instances of that label on the AST is the value.

The atom-based predictor was evaluated in this study because it potentially allows for effective vulnerability prediction to be performed while defining just one metric, rather than defining a large collection of metrics. The similar token-based predictor was found to perform favorably in comparison with software metric predictors [165].

Because each value of a label becomes a separate feature, the atom-based predictor is computed differently from standard metrics. The ADDDEL and MOD cases (as described in Definition 2.6) cannot occur, because each instance of a value for a label is counted separately. For example, if one atom-based feature measures the number of plus signs in a source file, it makes little sense to say that an individual plus sign was "edited". The predictor is "atomic" in the sense that each labeled node in the AST is counted separately, while with standard metrics, nodes (for example, all nodes in a single method's definition) were logically grouped by their label's

values. For similar reasons, value mapping (or matching) cannot be performed either; hence, no atom-based predictor can be defined for metric families that require value matching.

For the purposes of this study, we computed one atom-based predictor, based on the `Generic node contents` metric. Because this label for this metric incorporates a variety of AST node types and AST node values, all of which are part of the PHP language itself rather than being tied to any individual file, it is a good candidate for atom-based prediction.

6.1.3 Common conventions for experimental setups and reporting of experimental results

In the sections that follow, we will report the results of vulnerability prediction experiments that evaluate the effects of different classifiers, classifier configurations, experimental setups, and features. Throughout these sections, we will employ a number of common experimental techniques and conventions. We describe these techniques and conventions here. All conventions apply to all experiments unless otherwise specified in the experiment itself.

- Experiments are performed on both PHPMyAdmin and Moodle. The results of the experiments for each product are reported separately and are not mixed or compared.

- Experiments are file-level, release-level, or change-level experiments. File-level experiments use the examplesetfile construct (as defined in Chapter 3) with the releases specified in Table 6.1 selected with the selectrel construct. Release-level experiments use the examplesetrel construct and change-level experiments use the examplesetchg construct.

- Experiments with metric-based predictors use the entire set of metric families listed in Chapter 2, but not the derived metrics unless otherwise specified.
- Experiments with atom-based predictors use atom-based features as described previously, deriving features from the labels for the `Generic node contents` metric family. Due to the larger size of Moodle, atom-based features are only explored for PHPMyAdmin as these features are much greater in number than metrics and could not be feasibly computed for all of the Moodle files and releases.
- Experiments use effort-sensitive machine learning algorithms and effort-sensitive performance indicators. Effort is measured by the effort column `Lines of code`.
- Accordingly, the effort column `Lines of code` is required in each example set.
- File-level experiments get metrics with the addstaticmeasurement construct. Change-level experiments and release-level experiments get metrics with the addchangemeasurement construct.
- Change-level experiments use the add-only change metric from Section 6.1.2, while release-level experiments use the distance-based change metric. Release-level experiments also incorporate the current metrics of each release, added with the addstaticmeasurement construct.
- Release-level experiments aggregate both the change measurements and the static measurements using the combinefiles construct with the combinesum operator.
- Experiments done on Moodle use the random forest classification algorithm. File-level and change-level experiments done on PHPMyAdmin use the random forest algorithm, while release-level experiments use the Naive Bayes

algorithm. In addition, all atom-based features use the random forest algorithm.

When reporting performance indicators and graphs for a set of experiments, the following conventions are used:

- In a set of experiments, each experiment is done on a different example set (or, in some cases, with different machine learning parameters) in order to explore the effects of different features or processes on the performance of the resulting predictive model.

- The results of sets of experiments are reported in tables, such that each experiment in the tables is part of the same set. One experiment is designated as the baseline experiment and *shaded*. For the other experiments, any performance indicators that significantly differ from the baseline are *bolded*. Comparisons are only made within projects (PHPMyAdmin to PHPMyAdmin and Moodle to Moodle).

- When comparing multiple types of experimental setup, or when explicitly assessing the effects of effort, inspection-based performance indicators such as i_{20}, i_{40}, and i_1 are reported.

- When comparing compatible experimental setups (such that the examples for each experiment have identical effort), recall-based performance indicators such as $aucec$, $aucec_{50}$, r_{10}, r_{20}, and r_{40} are reported.

- In addition to the tables, cost-effectiveness curves are plotted for selected experiments. In these plots, the recall performance indicator R is on the y-axis and the inspection ratio IR, measured in lines of code, is on the x-axis.

6.2 Evaluating effort-sensitive and non-effort-sensitive training of models

Previously, we introduced two methods for making a training algorithm effort-sensitive – an example weighting method used in conjunction with the random forest algorithm, and a probability-based method used with other algorithms. We have also introduced effort-sensitive and non-effort-sensitive performance indicators, because using an appropriate performance indicator allows for a model to be evaluated appropriately. We now perform an empirical evaluation of the effort-sensitive training meta-learning algorithm to determine if model performance is improved when the choice of training method is aligned with the choice of performance indicator. In other words, we examine if effort-sensitive training algorithms work better in contexts where effort-sensitivity matters, and, conversely, if non-effort-sensitive training algorithms are better when the evaluation criteria are not sensitive to effort.

6.2.1 Effort-sensitive training algorithms in an effort-sensitive evaluation context

Using effort-sensitive performance indicators, Table 6.2 compares the performance of effort-sensitive and non-effort-sensitive training algorithms with metric predictors. Table 6.3 presents the same information for atom-based predictors, and Figure 6.1 shows cost-effectiveness curves when using both kinds of predictor. For file-level predictors, using an effort-sensitive learning algorithm resulted in a significant improvement in $aucec_{50}$ in both cases.

The same results for release-level predictors are shown in Tables 6.4 and 6.5 and Figure 6.2. The results for change-level predictors are in Tables 6.6 and 6.7 and Figure 6.3. With the exception of release-level token predictors (which experienced poor performance in general), these results are similarly significant to those with

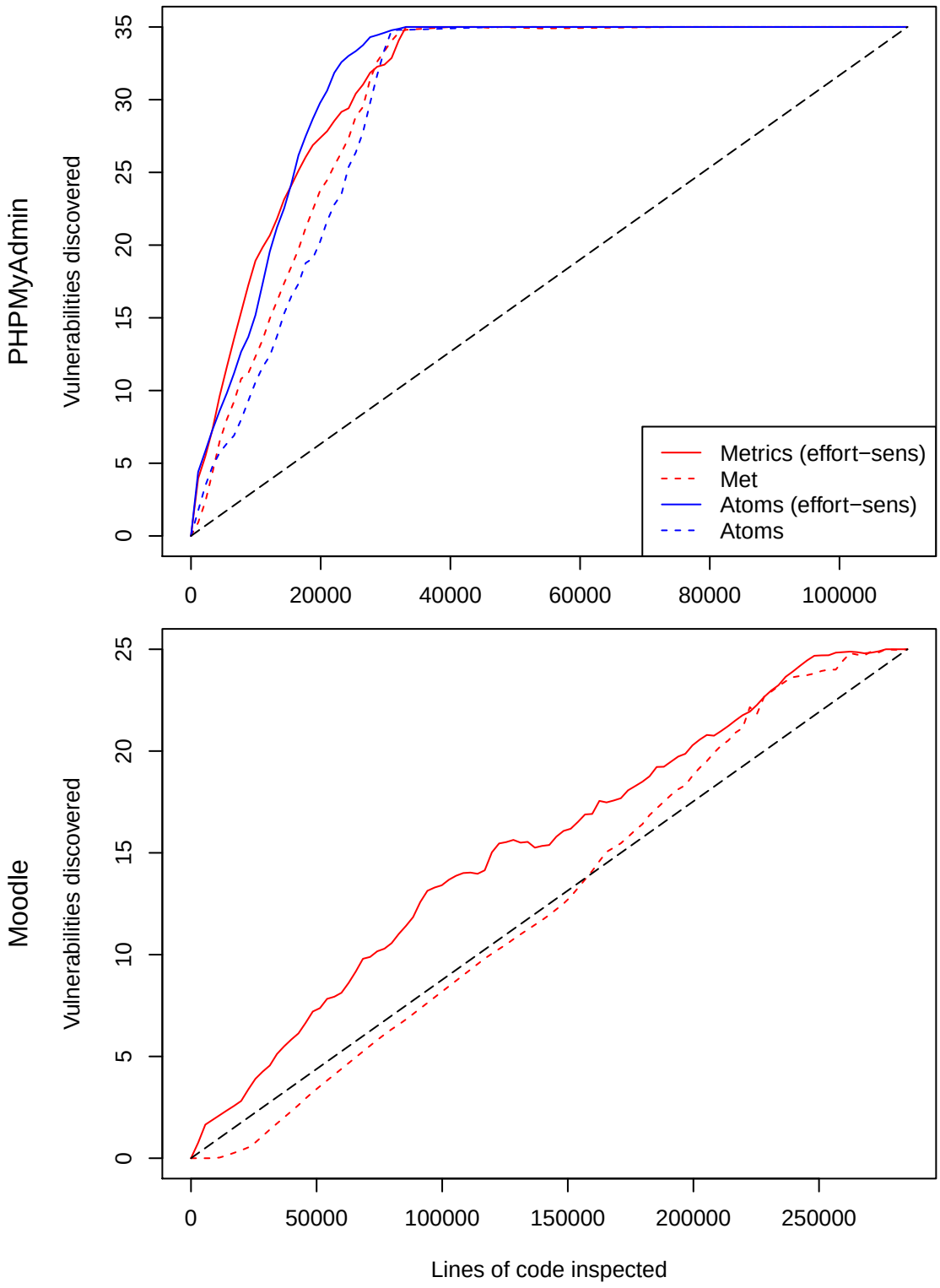


Figure 6.1: Performance of effort-sensitive model training algorithms under effort-sensitive performance criteria for file-level predictors

Table 6.2: Performance of effort-sensitive training algorithms for file metric predictors (effort-sensitive performance evaluation criteria)

Application	Training	r_{20}	r_{40}	i_{20}	i_{40}	$aucec_{50}$
Moodle	Effort	**0.317**	**0.559**	**35598**	**68725**	**0.374**
Moodle	Non-effort	0.164	0.381	66000	120379	0.219
PHPMyAdmin	Effort	**0.815**	1.000	**3198**	**7215**	**0.787**
PHPMyAdmin	Non-effort	0.728	1.000	5393	11251	0.730

Table 6.3: Performance of effort-sensitive training algorithms for file atom predictors (effort-sensitive performance evaluation criteria)

Application	Training	r_{20}	r_{40}	i_{20}	i_{40}	$aucec_{50}$
PHPMyAdmin	Effort	**0.910**	1.000	**3550**	**8848**	**0.795**
PHPMyAdmin	Non-effort	0.651	0.999	6479	13321	0.702

Table 6.4: Performance of effort-sensitive training algorithms for release atom predictors (effort-sensitive performance evaluation criteria)

Application	Training	r_{20}	r_{40}	i_{20}	i_{40}	$aucec_{50}$
Moodle	Effort	**0.337**	0.483	162529	**245757**	**0.338**
Moodle	Non-effort	0.228	0.498	170209	314067	0.301
PHPMyAdmin	Effort	0.238	0.461	83950	148433	0.289
PHPMyAdmin	Non-effort	0.222	0.477	84544	153685	0.288

Table 6.5: Performance of effort-sensitive training algorithms for release atom predictors (effort-sensitive performance evaluation criteria)

Application	Training	r_{20}	r_{40}	i_{20}	i_{40}	$aucec_{50}$
PHPMyAdmin	Effort	0.205	**0.437**	94977	169005	**0.250**
PHPMyAdmin	Non-effort	0.204	0.480	89835	158785	0.277

Table 6.6: Performance of effort-sensitive training algorithms for change metric predictors (effort-sensitive performance evaluation criteria)

Application	Training	r_{20}	r_{40}	i_{20}	i_{40}	$aucec_{50}$
Moodle	Effort	0.445	**0.815**	**71424**	176926	**0.536**
Moodle	Non-effort	0.405	0.685	111583	190532	0.458
PHPMyAdmin	Effort	**0.888**	1.000	13198	35050	**0.791**
PHPMyAdmin	Non-effort	0.657	0.993	12231	34482	0.726

file-level predictors, showing that the effort-sensitive algorithm improves prediction performance as measured by effort-sensitive performance indicators.

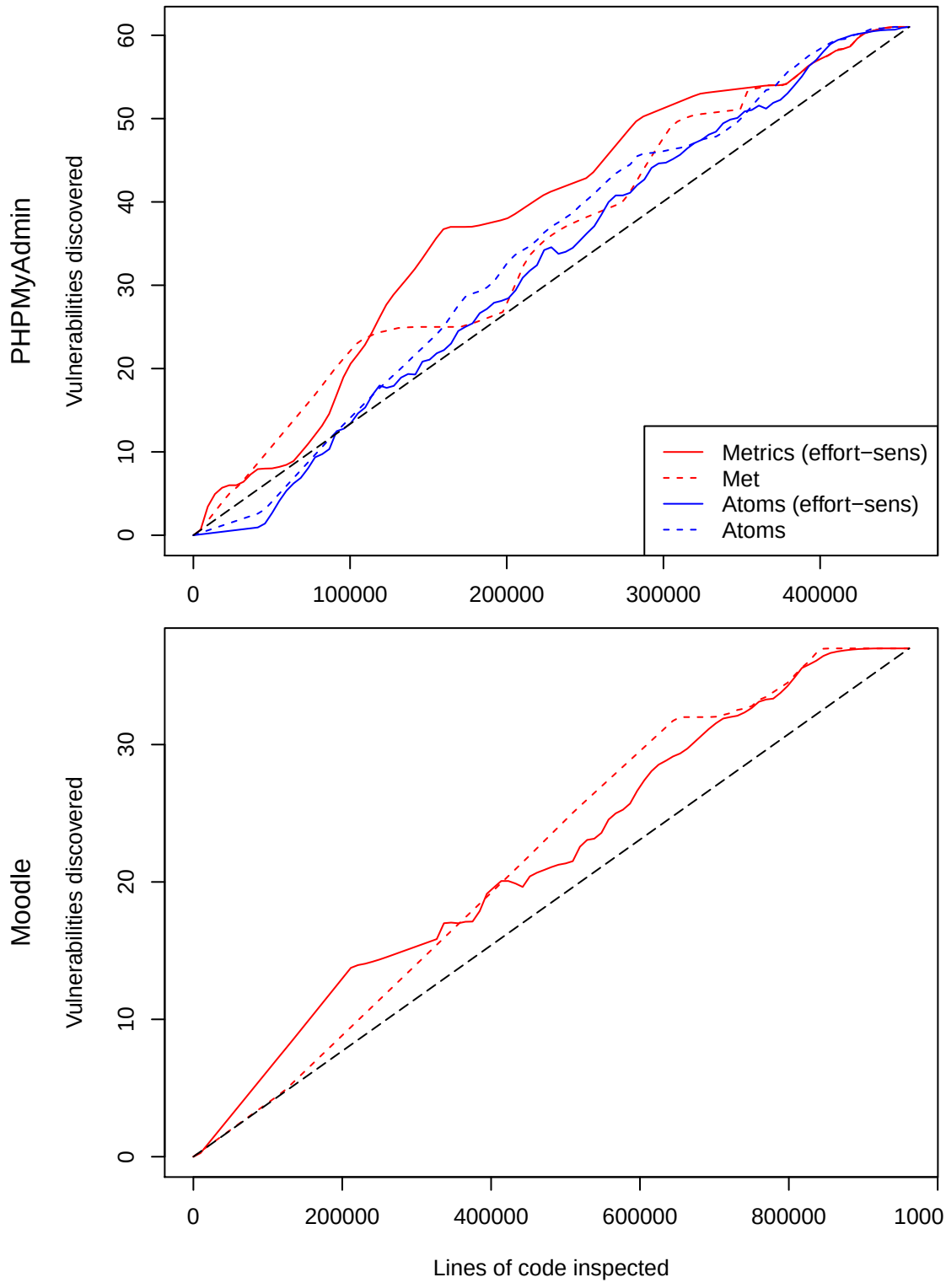


Figure 6.2: Performance of effort-sensitive model training algorithms under effort-sensitive performance criteria for release-level predictors

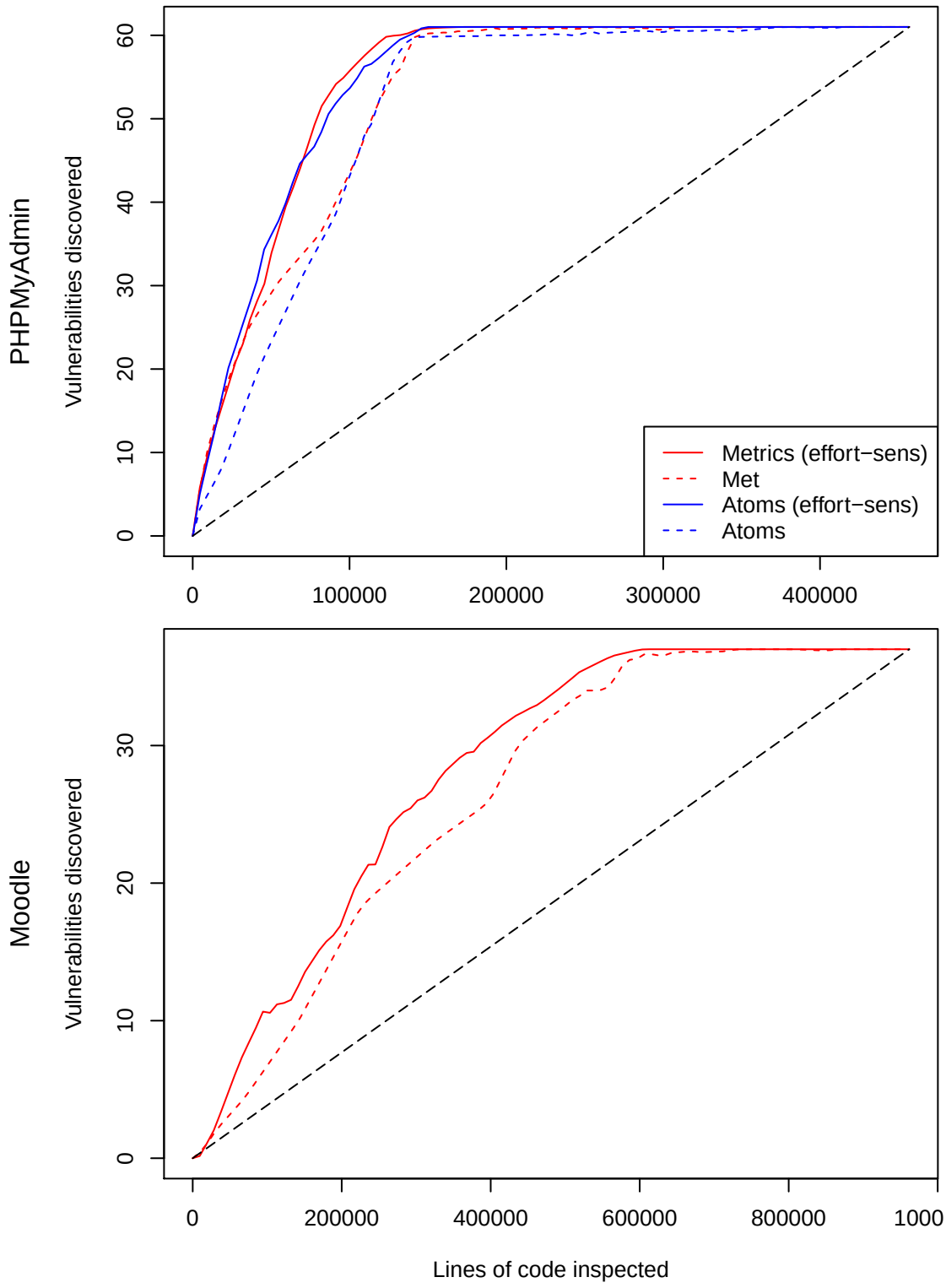


Figure 6.3: Performance of effort-sensitive model training algorithms under effort-sensitive performance criteria for change-level predictors

Table 6.7: Performance of effort-sensitive training algorithms for change atom predictors (effort-sensitive performance evaluation criteria)

Application	Training	r_{20}	r_{40}	i_{20}	i_{40}	$aucec_{50}$
PHPMyAdmin	Effort	**0.849**	**1.000**	**13541**	**30596**	**0.788**
PHPMyAdmin	Non-effort	0.632	0.982	26295	52681	0.690

These results demonstrate that the effort-sensitive training algorithms should be used for all types of predictors, as long as the predictor is going to be used in a context where the effort required to inspect a unit of code is relevant. We will next briefly explore if effort-sensitive training algorithms are appropriate for use in a *non-effort-sensitive* context.

6.2.2 Effort-sensitive training algorithms in an non-effort-sensitive evaluation context

We now explore if the effort-sensitive meta-learning algorithm can be used in all circumstances – whether the performance criteria take effort into account or not – or if effort-sensitive training algorithms can only be used in effort-sensitive situations. In order to perform these experiments, we train models as in the effort-sensitive case, using the effort-sensitive meta-learning algorithm. Then, we produce new cost-effectiveness curves with a dummy effort of 1 assigned to each example.

These results are reported in a similar manner to the results for the effort-sensitive evaluation contexts. Tables 6.8 and 6.9 and Figure 6.4 report the results for file-level predictors. Tables 6.12 and 6.13 and Figure 6.6 show the results for release-level predictors, and Tables 6.10 and 6.11 and Figure 6.5 show the results for change-level predictors. In this non-effort-sensitive evaluation case, the results are dramatically different from the effort-sensitive case – all performance indicators were significantly worse when using an effort-sensitive predictor and non-effort-sensitive performance indicators.

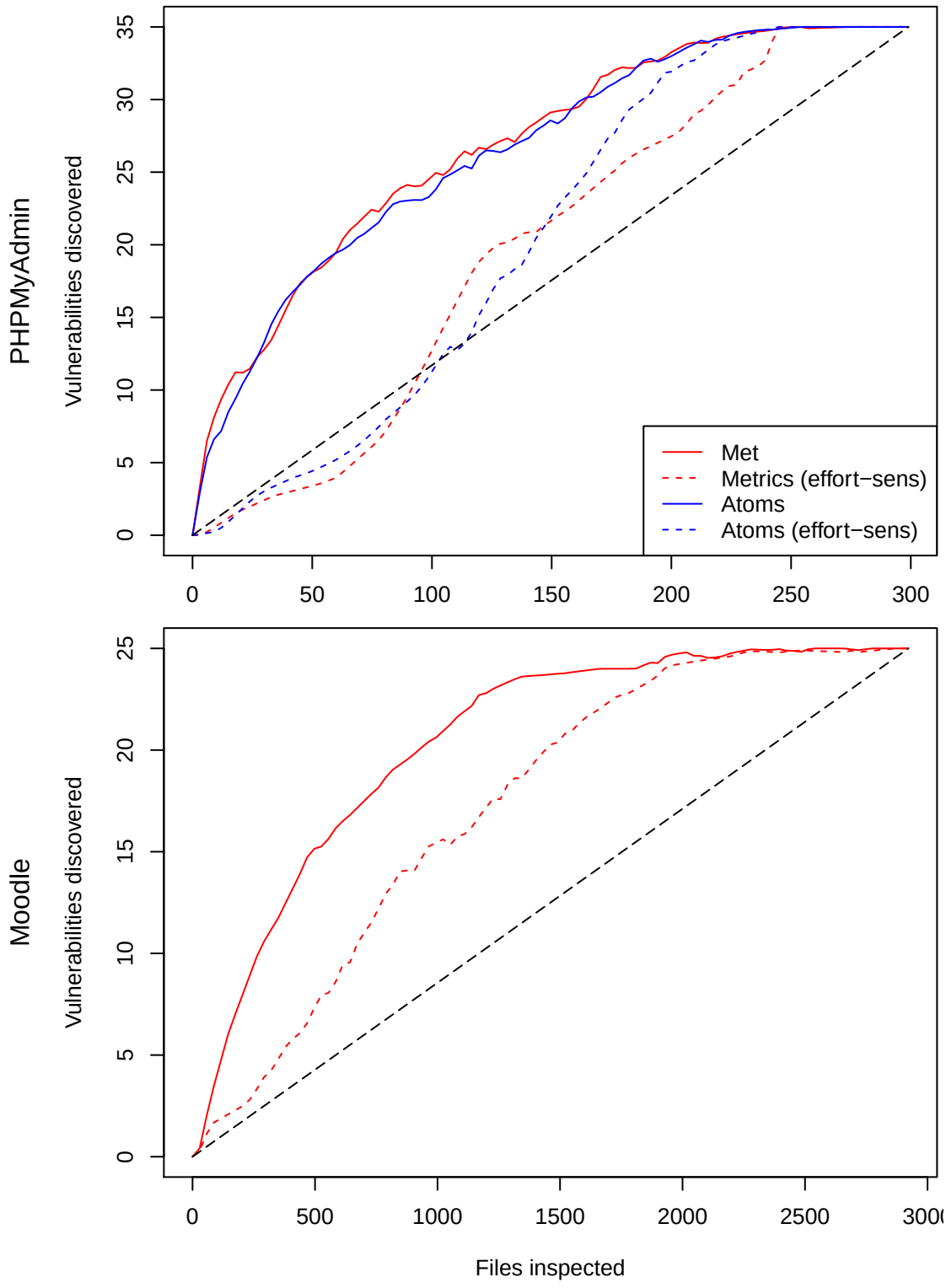


Figure 6.4: Performance of effort-sensitive model training algorithms under non-effort-sensitive performance criteria for file-level predictors

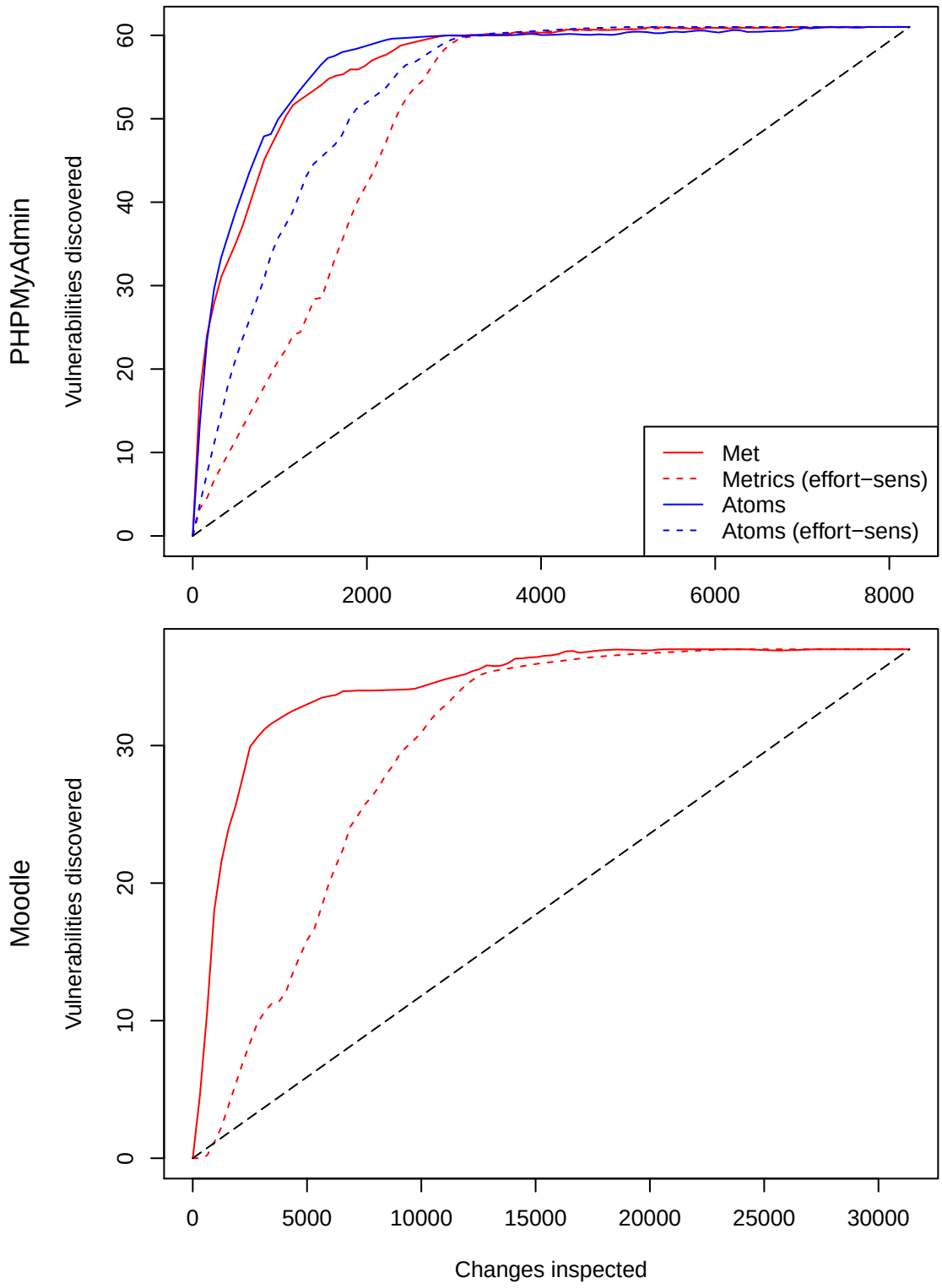


Figure 6.5: Performance of effort-sensitive model training algorithms under non-effort-sensitive performance criteria for change-level predictors

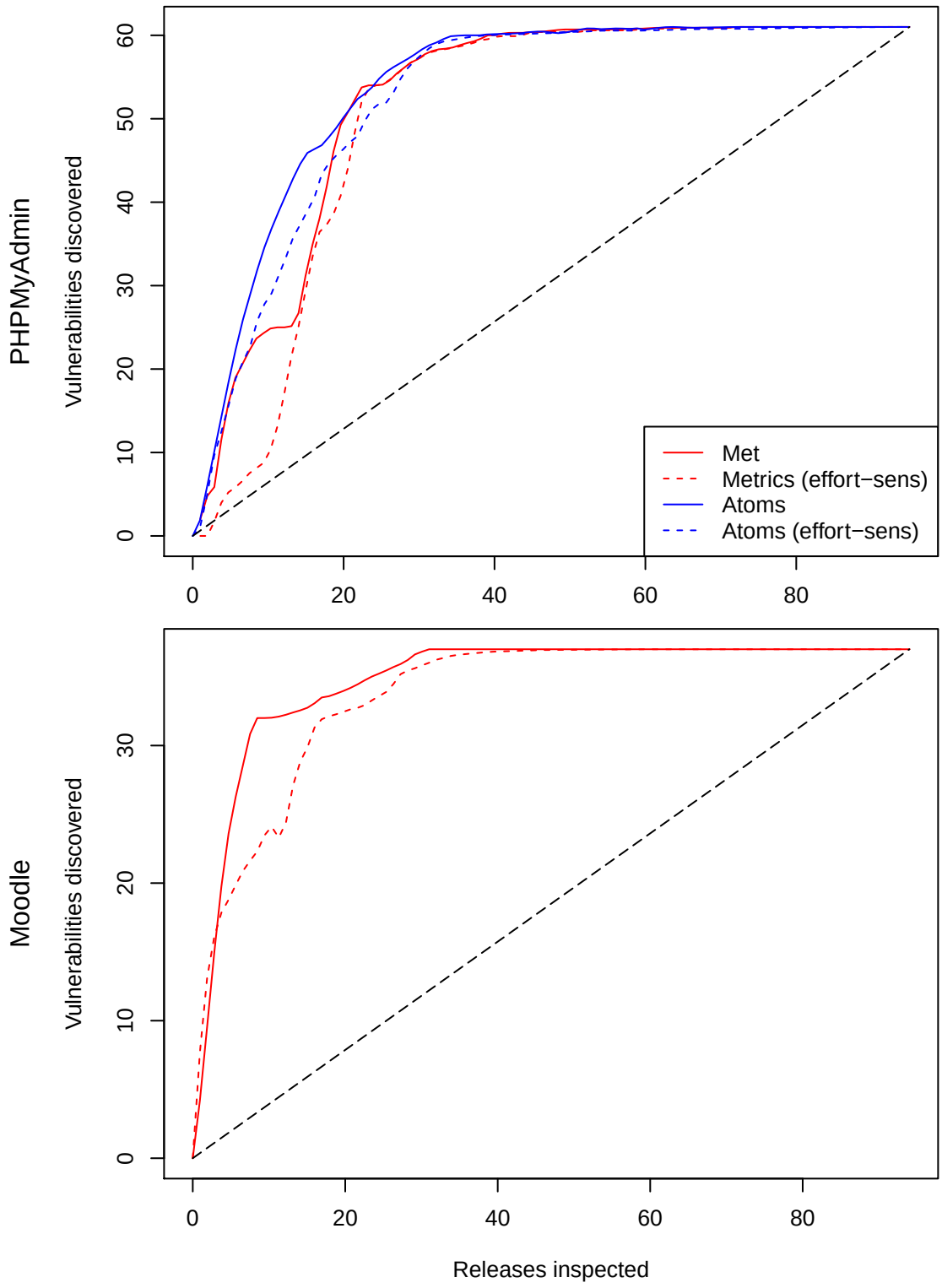


Figure 6.6: Performance of effort-sensitive model training algorithms under non-effort-sensitive performance criteria for release-level predictors

Table 6.8: Performance of effort-sensitive training algorithms for file metric predictors (non-effort-sensitive performance evaluation criteria)

Application	Training	r_{20}	r_{40}	i_{20}	i_{40}	$aucec_{50}$
Moodle	Effort	**0.344**	**0.668**	**377.25**	**620.6**	**0.429**
Moodle	Non-effort	0.646	0.908	121.11	273.5	0.652
PHPMyAdmin	Effort	**0.113**	**0.537**	**79.81**	**107.5**	**0.259**
PHPMyAdmin	Non-effort	0.555	0.763	8.94	33.9	0.575

Table 6.9: Performance of effort-sensitive training algorithms for file atom predictors (non-effort-sensitive performance evaluation criteria)

Application	Training	r_{20}	r_{40}	i_{20}	i_{40}	$aucec_{50}$
PHPMyAdmin	Effort	**0.148**	**0.433**	**76.3**	**116.1**	**0.245**
PHPMyAdmin	Non-effort	0.555	0.747	10.9	32.4	0.559

Table 6.10: Performance of effort-sensitive training algorithms for change metric predictors (non-effort-sensitive performance evaluation criteria)

Application	Training	r_{20}	r_{40}	i_{20}	i_{40}	$aucec_{50}$
Moodle	Effort	**0.575**	0.949	**2425.7**	**4923**	**0.615**
Moodle	Non-effort	0.910	0.961	476.6	803	0.856
PHPMyAdmin	Effort	**0.551**	0.983	**529.9**	**1214**	**0.643**
PHPMyAdmin	Non-effort	0.904	0.984	42.3	169	0.852

Table 6.11: Performance of effort-sensitive training algorithms for change atom predictors (non-effort-sensitive performance evaluation criteria)

Training	r_{20}	r_{40}	i_{20}	i_{40}	$aucec_{50}$
Effort	**0.769**	0.984	**271**	**605**	**0.749**
Non-effort	0.943	0.984	73	175	0.869

Table 6.12: Performance of effort-sensitive training algorithms for release atom predictors (non-effort-sensitive performance evaluation criteria)

Application	Training	r_{20}	r_{40}	i_{20}	i_{40}	$aucec_{50}$
Moodle	Effort	**0.872**	0.993	3.34	3.25	**0.820**
Moodle	Non-effort	0.913	1.000	1.50	2.80	0.877
PHPMyAdmin	Effort	**0.539**	0.979	**4.02**	**11.70**	**0.647**
PHPMyAdmin	Non-effort	0.821	0.984	3.40	6.25	0.768

As evidenced by these results, the benefits of using effort-sensitive meta-learning algorithms is only realized when effort-sensitive evaluation criteria are used.

Table 6.13: Performance of effort-sensitive training algorithms for release atom predictors (non-effort-sensitive performance evaluation criteria)

Application	Training	r_{20}	r_{40}	i_{20}	i_{40}	$aucec_{50}$
PHPMyAdmin	Effort	**0.747**	0.982	4.05	**7.99**	**0.725**
PHPMyAdmin	Non-effort	0.801	0.984	3.35	6.28	0.768

Previous studies in defect prediction have noted that a trivial model may simply select the largest files in a product [88]. Assuming that the number of defects in a file is correlated to the size of a file, this results in an apparently effective, but trivial predictor that is not especially useful in practice. However, by penalizing the inspection of large files with a penalty proportional to the file's size, the effort-sensitive training algorithms in our study correct for the bias toward large files. As seen in the results, though, models trained while correcting for this bias perform very poorly when ordinary, non-effort-sensitive performance indicators are used. Noting that these are the indicators most commonly used in previous work, our results suggest that it was *inevitable* that models in previous studies would have had a bias for large files – if they did not, the performance of their models would have been very poor.

Because this study evaluates vulnerability prediction algorithms in the context of a hypothetical code-inspection task, effort-sensitivity is inherent to our use case, and we will exclusively utilize effort-sensitive training algorithms and effort-sensitive performance indicators throughout the remainder of this work.

6.3 Evaluating and comparing machine learning algorithms

We now examine what kind of model is ideal for each product (PHPMyAdmin or Moodle), experimental setup (file-level, change-level, or release-level) and feature type (metric-based or atom-based). Each variety of model was trained for each product, experimental setup, and feature type. The default Weka settings for each

Table 6.14: Machine learning algorithms for file metric predictors (effort-sensitive performance evaluation criteria)

Application	Algorithm	r_{10}	r_{20}	aucec	$aucec_{50}$	t
Moodle	LgR	**0.0922**	0.333	**0.557**	0.354	**6.92**
Moodle	LR	0.1915	0.364	0.599	0.388	**4.20**
Moodle	NB	**0.0800**	**0.197**	**0.461**	**0.232**	**3.65**
Moodle	RF	0.1701	0.317	0.611	0.374	310.78
PHPMyAdmin	LgR	**0.4746**	**0.674**	**0.664**	**0.585**	**14.09**
PHPMyAdmin	LR	0.5733	**0.888**	**0.867**	0.785	**1.58**
PHPMyAdmin	NB	**0.1354**	**0.509**	**0.813**	**0.625**	**1.54**
PHPMyAdmin	RF	0.5671	0.815	0.893	0.787	257.56

Table 6.15: Machine learning algorithms for file atom predictors (effort-sensitive performance evaluation criteria)

Application	Algorithm	r_{10}	r_{20}	aucec	$aucec_{50}$	t
PHPMyAdmin	LgR	**0.309**	**0.374**	**0.441**	**0.342**	**21.7**
PHPMyAdmin	LR	0.544	**0.771**	**0.868**	**0.757**	**88.6**
PHPMyAdmin	NB	**0.223**	**0.554**	**0.830**	**0.659**	**17.5**
PHPMyAdmin	RF	0.496	0.910	0.898	0.795	1952.6

algorithm were not changed for this experiment (nor were they changed for any other experiment in this work).

We first compare the performance of various machine learning algorithms for file-level prediction. Table 6.14 shows the performance for prediction with metric features, while Table 6.15 shows the performance for atom-based features. In addition, Figure 6.7 shows cost-effectiveness curves for file-level prediction with metric features.

For both metric and atom features, the random forest algorithm had the best overall performance for both applications. The random forest algorithm usually had better performance than the others, and in no case was another algorithm significantly better (as evidenced by the $aucec_{50}$ scores). The cost-effectiveness curves reveal that the random forest algorithm performed better than the null model at all **IR** (which was not true for logistic regression and Naive Bayes), and the random forest algorithm did not experience a performance plateau before discovering all the

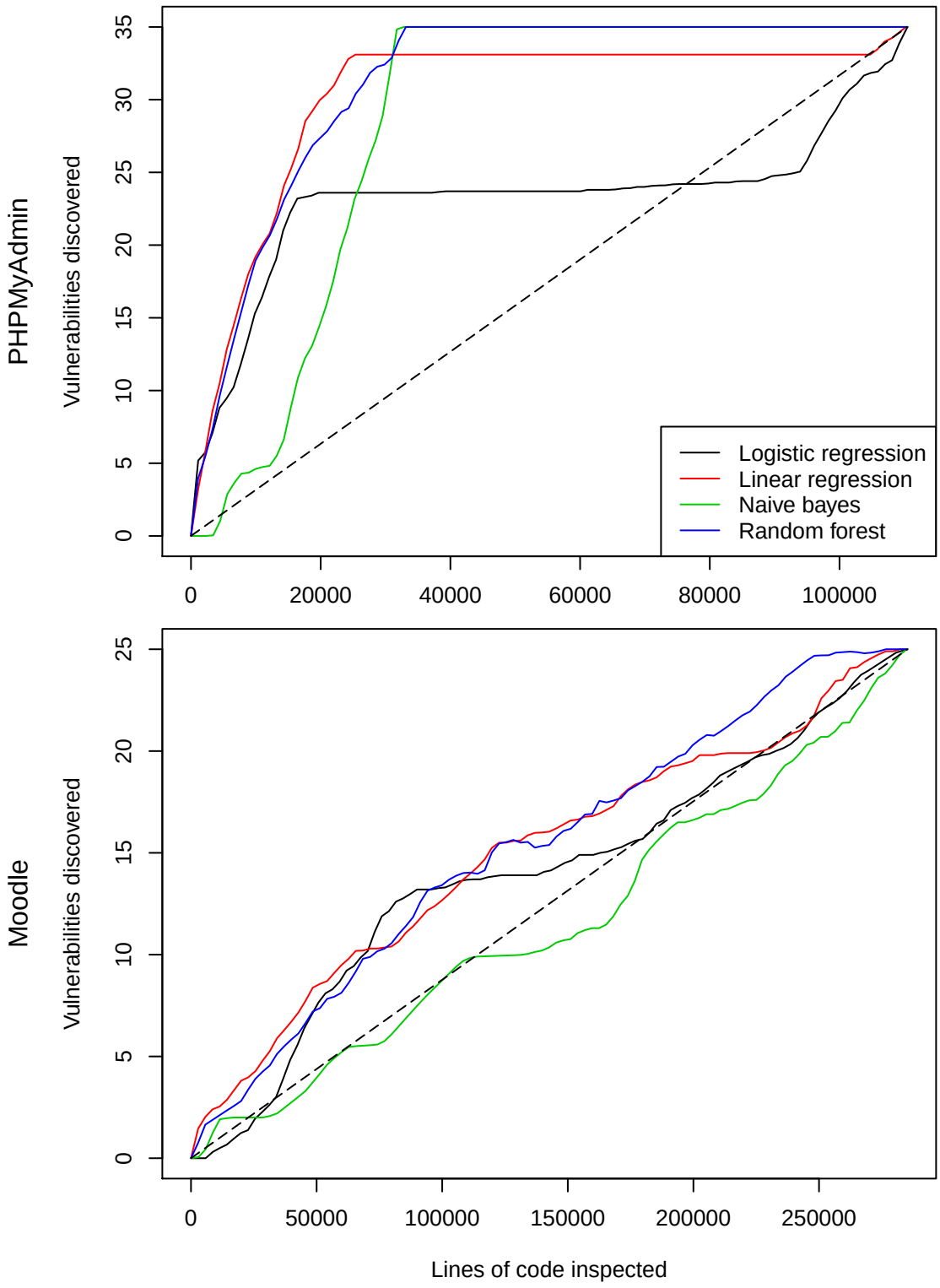


Figure 6.7: Performance of classification algorithms on effort-sensitive file-level prediction

Table 6.16: Machine learning algorithms for change metric predictors (effort-sensitive performance evaluation criteria)

Application	Algorithm	r_{10}	r_{20}	aucec	$aucec_{50}$	t
Moodle	LgR	0.255	0.500	**0.701**	**0.494**	**231.2**
Moodle	LR	0.294	0.445	**0.691**	**0.504**	**189.1**
Moodle	NB	0.260	0.424	**0.645**	**0.420**	**191.8**
Moodle	RF	0.286	0.445	0.764	0.536	1599.5
PHPMyAdmin	LgR	0.470	**0.749**	**0.859**	**0.720**	**24.9**
PHPMyAdmin	LR	0.484	**0.817**	**0.812**	**0.706**	**17.6**
PHPMyAdmin	NB	**0.303**	**0.473**	**0.795**	**0.591**	**18.1**
PHPMyAdmin	RF	0.495	0.888	0.895	0.791	499.3

Table 6.17: Machine learning algorithms for change atom predictors (effort-sensitive performance evaluation criteria)

Application	Algorithm	r_{10}	r_{20}	aucec	$aucec_{50}$	t
PHPMyAdmin	LgR	**0.373**	**0.430**	**0.607**	**0.442**	**325**
PHPMyAdmin	LR	**0.413**	**0.597**	**0.581**	**0.519**	**6911**
PHPMyAdmin	NB	**0.343**	**0.527**	**0.751**	**0.544**	**207**
PHPMyAdmin	RF	0.563	0.849	0.894	0.788	8752

vulnerabilities (as with logistic regression and linear regression for PHPMyAdmin, indicating that some subset of vulnerable files tended to be selected last instead of first for these algorithms).

Next, we compare the performance of machine learning algorithms for change-level prediction. These performance indicators can be found in Table 6.16 (for metric predictors) and Table 6.17 (for atom predictors) with cost-effectiveness curves in Figure 6.8. As with file-level prediction, the random forest algorithm was the best overall algorithm for change-level prediction. Unlike with file-level prediction, the cost-effectiveness curves for PHPMyAdmin and Moodle share similar shapes and characteristics, including generally poor performance of Naive Bayes and an early performance plateau for linear regression.

Finally, we examine machine learning algorithms for release-level prediction. Table 6.18 and Table 6.19 show performance indicators for metric and atom features respectively, and Figure 6.9 depicts the cost-effectiveness curves. Unlike in the

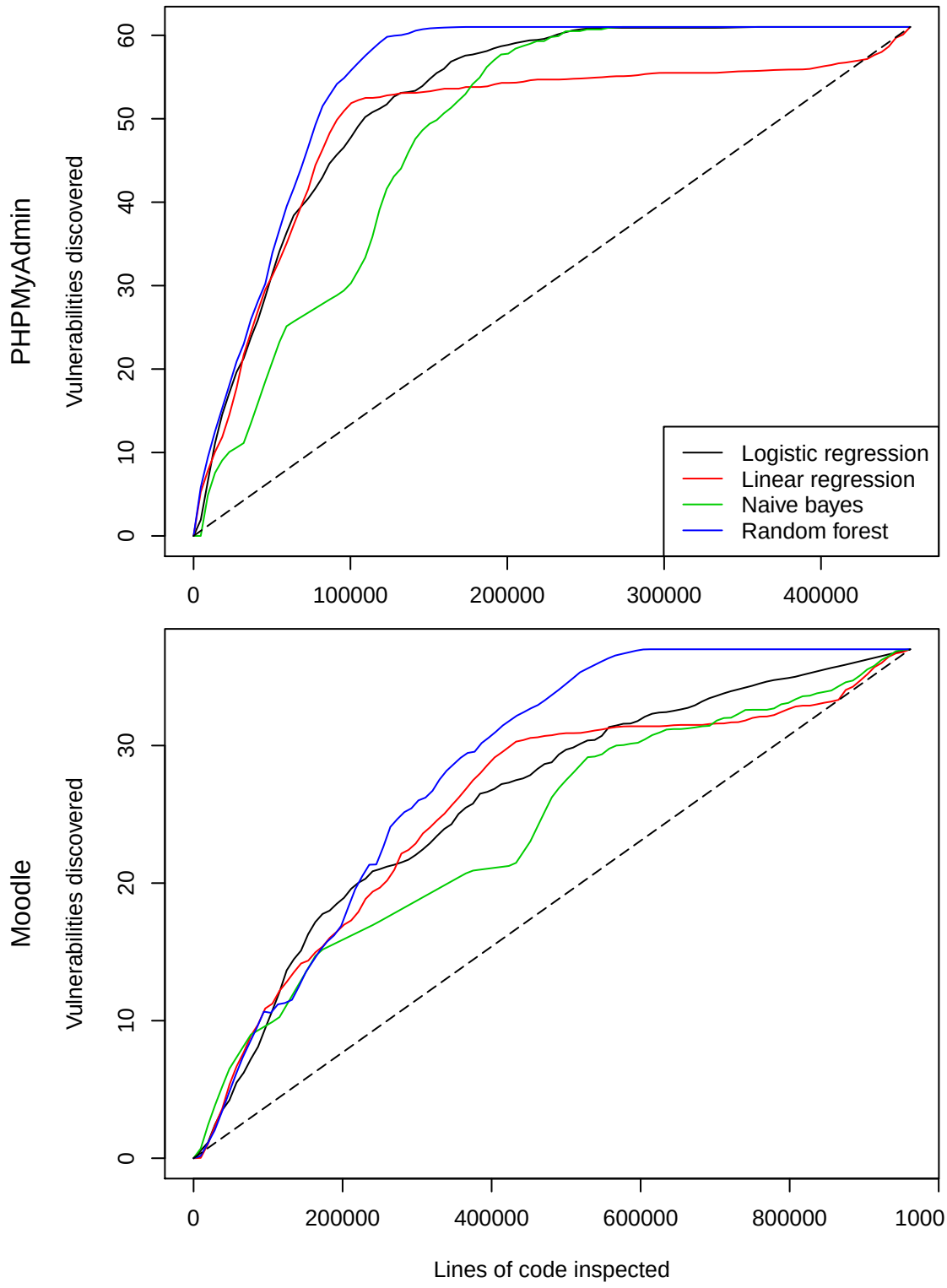


Figure 6.8: Performance of classification algorithms on effort-sensitive change-level prediction

Table 6.18: Machine learning algorithms for release metric predictors (effort-sensitive performance evaluation criteria)

Application	Algorithm	r_{10}	r_{20}	aucec	$aucec_{50}$	t
Moodle	LgR	0.0659	0.224	0.451	0.260	1.129
Moodle	LR	0.1367	**0.287**	0.563	0.334	**1.125**
Moodle	NB	**0.0598**	**0.162**	**0.515**	**0.232**	**0.939**
Moodle	RF	0.1627	0.337	0.588	0.338	223.232
PHPMyAdmin	LgR	**0.0750**	**0.183**	**0.504**	**0.267**	**1.464**
PHPMyAdmin	LR	**0.1751**	0.315	0.584	0.356	**1.600**
PHPMyAdmin	NB	0.1311	0.276	0.621	0.376	1.283
PHPMyAdmin	RF	**0.0297**	**0.238**	**0.556**	**0.289**	**197.245**

Table 6.19: Machine learning algorithms for release atom predictors (effort-sensitive performance evaluation criteria)

Application	Algorithm	r_{10}	r_{20}	aucec	$aucec_{50}$	t
PHPMyAdmin	LgR	0.0903	**0.275**	**0.545**	0.324	**19.4**
PHPMyAdmin	LR	0.1271	**0.188**	**0.567**	**0.292**	**51.0**
PHPMyAdmin	NB	0.1372	0.197	0.616	0.351	12.1
PHPMyAdmin	RF	**0.0229**	0.205	**0.528**	**0.250**	**1561.3**

case of the file-level and change-level predictors, the random forest algorithm is not the best model in every circumstance. For PHPMyAdmin, Naive Bayes was the best machine learning algorithm (for both metric-based and atom-based predictors), while for Moodle, the performance of Naive Bayes was little better than the null model. Meanwhile, the random forest model was the best algorithm for Moodle, while for PHPMyAdmin, it performed significantly worse than Naive Bayes.

6.3.1 Notes on the machine learning algorithm comparison

In summary, when comparing four machine learning algorithms for effort-sensitive prediction, Random Forest was the best algorithm in every case except for PHPMyAdmin release-level prediction, which performed better with Naive Bayes. Although the remainder of the experiments in this work will use effort-sensitive prediction, we briefly present these results for non-effort-sensitive prediction (and

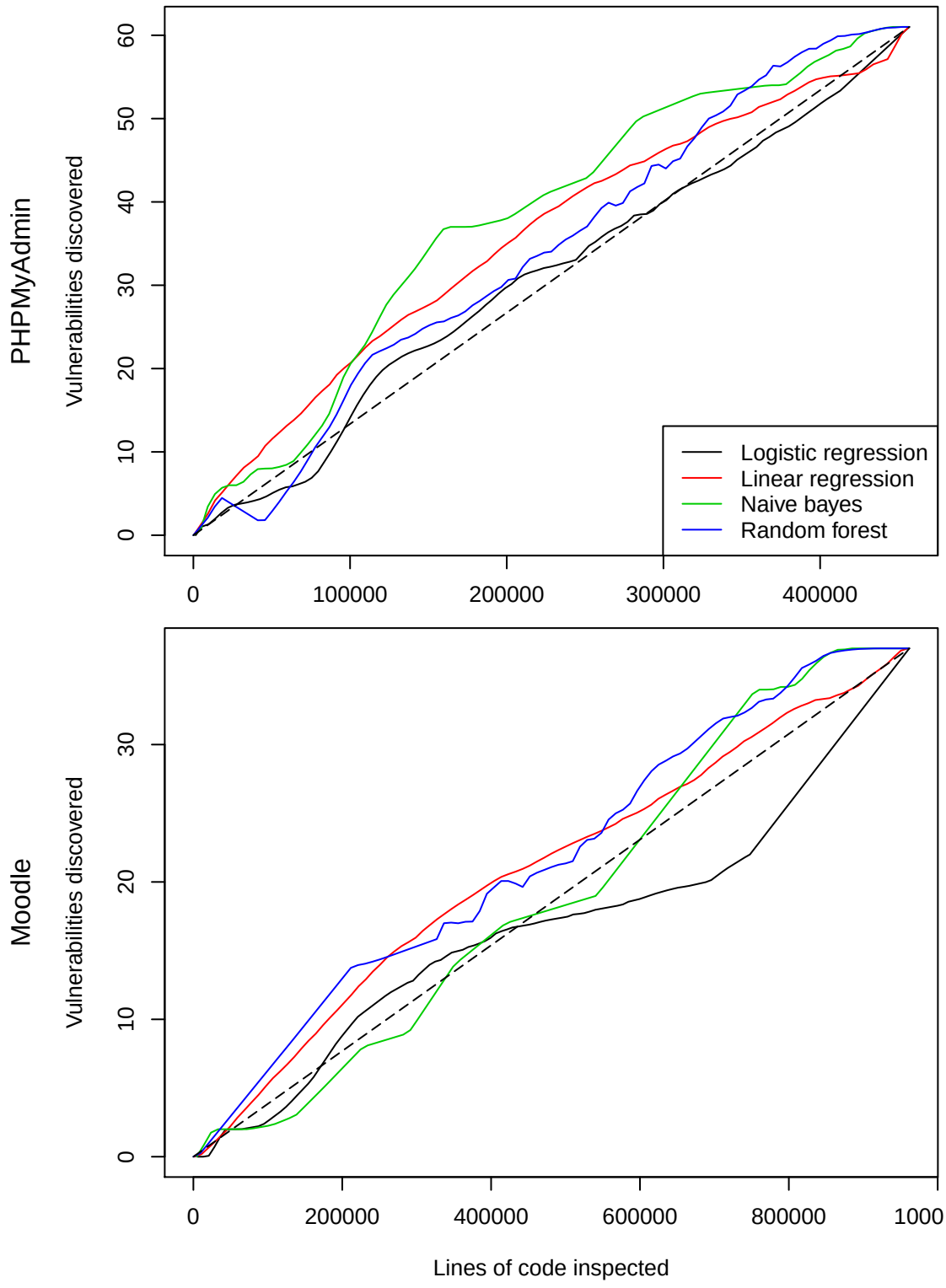


Figure 6.9: Performance of classification algorithms on effort-sensitive release-level prediction

Table 6.20: Machine learning algorithms for file metric predictors (non-effort-sensitive performance evaluation criteria)

Application	Algorithm	r_{10}	r_{20}	aucec	$aucec_{50}$	t
Moodle	LgR	**0.308**	**0.451**	**0.542**	**0.436**	**6.77**
Moodle	LR	0.375	0.588	**0.683**	**0.565**	**4.02**
Moodle	NB	0.442	0.660	**0.763**	**0.606**	**3.59**
Moodle	RF	0.424	0.646	0.818	0.652	223.32
PHPMyAdmin	LgR	0.294	**0.393**	**0.598**	**0.431**	**14.36**
PHPMyAdmin	LR	0.405	0.594	0.757	0.575	**1.23**
PHPMyAdmin	NB	0.343	0.485	**0.717**	**0.520**	**1.47**
PHPMyAdmin	RF	0.365	0.555	0.766	0.575	124.58

Table 6.21: Machine learning algorithms for change metric predictors (non-effort-sensitive performance evaluation criteria)

Application	Algorithm	r_{10}	r_{20}	aucec	$aucec_{50}$	t
Moodle	LgR	**0.643**	**0.716**	**0.778**	**0.695**	**225.7**
Moodle	LR	**0.748**	**0.792**	**0.809**	**0.745**	**197.1**
Moodle	NB	**0.811**	**0.859**	**0.883**	**0.813**	**190.9**
Moodle	RF	0.843	0.910	0.928	0.856	1342.7
PHPMyAdmin	LgR	**0.620**	**0.736**	**0.831**	**0.727**	**24.5**
PHPMyAdmin	LR	**0.660**	**0.823**	**0.838**	**0.755**	**17.6**
PHPMyAdmin	NB	0.707	**0.814**	**0.880**	**0.781**	**17.9**
PHPMyAdmin	RF	0.740	0.904	0.925	0.852	433.1

Table 6.22: Machine learning algorithms for release metric predictors (non-effort-sensitive performance evaluation criteria)

Application	Algorithm	r_{10}	r_{20}	aucec	$aucec_{50}$	t
Moodle	LgR	0.332	0.371	0.430	0.368	1.078
Moodle	LR	0.670	0.732	0.732	0.685	1.101
Moodle	NB	**0.450**	0.924	**0.894**	**0.788**	**0.969**
Moodle	RF	0.865	0.913	0.938	0.877	62.335
PHPMyAdmin	LgR	**0.261**	**0.480**	**0.590**	**0.440**	**1.492**
PHPMyAdmin	LR	**0.480**	**0.700**	**0.776**	0.663	**1.630**
PHPMyAdmin	NB	0.400	0.782	0.855	0.712	1.298
PHPMyAdmin	RF	**0.586**	**0.821**	**0.884**	**0.768**	**88.855**

evaluation) for the sake of comparison. Tables 6.20, 6.21, and 6.22 contain the performance indicators for prediction with metric features, while Tables 6.23, 6.24, and 6.25 contain the results for atom features (with the tables presenting file-level, change-level, and release-level results respectively).

Table 6.23: Machine learning algorithms for file atom predictors (non-effort-sensitive performance evaluation criteria)

Application	Algorithm	r_{10}	r_{20}	aucec	$aucec_{50}$	t
PHPMyAdmin	LgR	**0.171**	**0.307**	**0.459**	**0.297**	**22.4**
PHPMyAdmin	LR	**0.122**	**0.177**	**0.450**	**0.210**	**87.6**
PHPMyAdmin	NB	0.337	**0.470**	**0.718**	**0.508**	**16.8**
PHPMyAdmin	RF	0.379	0.555	0.756	0.559	988.1

Table 6.24: Machine learning algorithms for change atom predictors (non-effort-sensitive performance evaluation criteria)

Application	Algorithm	r_{10}	r_{20}	aucec	$aucec_{50}$	t
PHPMyAdmin	LgR	**0.328**	**0.367**	**0.429**	**0.345**	**328**
PHPMyAdmin	LR	**0.507**	**0.589**	**0.590**	**0.545**	**6881**
PHPMyAdmin	NB	**0.669**	**0.784**	**0.869**	**0.765**	**210**
PHPMyAdmin	RF	0.785	0.943	0.930	0.869	8225

Table 6.25: Machine learning algorithms for release atom predictors (non-effort-sensitive performance evaluation criteria)

Application	Algorithm	r_{10}	r_{20}	aucec	$aucec_{50}$	t
PHPMyAdmin	LgR	**0.411**	**0.646**	**0.692**	**0.593**	**19.6**
PHPMyAdmin	LR	**0.469**	**0.744**	**0.828**	**0.706**	**52.0**
PHPMyAdmin	NB	0.311	0.433	0.812	0.624	12.0
PHPMyAdmin	RF	**0.567**	**0.801**	**0.883**	**0.768**	**602.6**

These results differ from the effort-sensitive results in the sense that the random forest algorithm is the best performing algorithm in every case. For the PHPMyAdmin release-level predictors, unlike with the effort-sensitive predictors, the random forest algorithm performed significantly better than the Naive Bayes algorithm.

We also examine how the choice of experimental setup affects the statistical significance of the performance differences between the various machine learning algorithms. From Section 4.4, recall that, although specifically minimizing Type I or Type II errors is not an explicit goal of performing significance tests in this work. However, these statistical tests can help gauge the *stability* of the relative

performance of two experiments by examining if the results of one experiment are consistently higher than another.

When making effort-sensitive performance comparisons, we note that for the $aucec_{50}$ indicator, the best performing algorithms performed significantly better than all other algorithms for change-level predictors. In contrast, for file-level and release level predictors, approximately only half of the differences were significant.

Finally, we observe the wide variation between machine learning algorithms when it comes to the amount of time required to train a single set of predictors (i.e. compute all the points for a single cost-effectiveness curve). For experiments with metric-based features, the logistic regression, linear regression, and Naive Bayes algorithms generally took similar amounts of time to construct. In contrast, the amount of time to build the random forest predictor was orders of magnitude greater. For example, the Naive Bayes algorithm for PHPMyAdmin file-level predictors took an average of 1.47 seconds to run, while running the random forest algorithm for the same example set took an average of 124.58 seconds. Predictors with atom-based features took longer to construct in general, and for these, the linear regression and random forest algorithms took much longer than the others.

The slow performance of the random forest algorithm is largely due to the number of iterations required to build the cost-effectiveness curve. The search-based method used with this algorithm must train dozens of models with various values of m, while the probability-based method used for the other algorithms only trains one model. In the case of linear regression and the atom-based features, the slow performance was likely due to the need to perform feature selection and the high cardinality of atom-based features, which are much larger in number than standard software metrics. Although classifiers with the random forest algorithm are slow to construct, because this algorithm is very effective for vulnerability prediction tasks, we will continue to use it for the experiments throughout this section.

6.4 Evaluating feature variants

In Section 3.4, we introduced a system of composable operators which allowed for many variants of defect prediction features to be constructed. For example, a sliding window operator such as slidewindrevs can transform a change metric that quantifies the changes made during a particular release into a change metric that quantifies all changes made in the last five releases. As discussed in Section 3.3, many of these operators were derived from features used in past defect prediction studies, which found that these more complex features resulted in better defect prediction performance than simple code or change metrics alone.

However, we note that these feature variants have not yet been explored in the context of vulnerability prediction, nor have they been explored in a context where release-by-release (as opposed to commit-by-commit) change data is available. In particular, the release-by-release change data could affect the way that these features perform in a prediction context because many of them are based on change data. In this section, we systematically examine each of these feature variants in the context of a vulnerability prediction evaluation on our PHP vulnerability datasets.

6.4.1 Computation method of change metrics

Section 2.2.4 describes several choices that can be made when computing a change metric for a given metric family:

- The code delta change metric (based on simple subtraction) or the distance-based change metric (based on a comparison between releases) can be used.
- If using the distance-based change metric, value matching for selected metric families can be enabled or disabled. This allows for insignificant (e.g. renaming methods) and significant (e.g. modifying method bodes) changes to be distinguished.

Table 6.26: Variants of change metrics (effort-sensitive performance evaluation criteria)

Change	Match	Moodle			PHPMyAdmin		
		r_{10}	r_{20}	$aucec_{50}$	r_{10}	r_{20}	$aucec_{50}$
Add	No	**0.288**	0.462	0.535	0.488	**0.888**	**0.791**
Add	Yes	0.286	**0.445**	0.536	0.495	**0.888**	**0.791**
Delta		0.252	0.512	0.554	0.481	0.852	0.771
Dist	No	**0.203**	**0.434**	**0.508**	**0.542**	0.870	**0.788**
Dist	Yes	**0.196**	**0.436**	**0.510**	**0.540**	0.869	**0.788**

- Several coefficients can be varied to control the way that the distance-based change metrics are computed.

Here, we examine the effects of each of these choices on the performance of the final defect predictors. The results are depicted in Table 6.26 for change-level predictors and Tables 6.27 and 6.28 for release-level predictors with metric-based and atom-based features, respectively. The "change" column indicates if the code delta change metric or the distance-based change metric was used, with "add" and "dist" referring to the variants described in Section 6.1.2. The "match" column indicates if value matching was enabled for metrics that allow it. In addition, for release-level predictors, the "current" column indicates if the current static code metrics of a release are included as features (in addition to the change metrics for the release). This allows for the model to base its predictions of vulnerability introduction on both the change made during a release and on the metrics of the entire codebase at the time of the release. Finally, the addversjump feature was included in all models compared here.

For change-level prediction, using the $aucec_{50}$ indicator for comparison, the *add* coefficients in conjunction with the distance-based change metric were the best overall performers. They were significantly better than the code delta metric in the case of PHPMyAdmin; in the case of Moodle, they were slightly worse but not significantly so. Enabling or disabling value matching for selected metrics had little

Table 6.27: Variants of release metrics (effort-sensitive performance evaluation criteria)

Change	Match	Current	Moodle			PHPMyAdmin		
			r_{10}	r_{20}	$aucec_{50}$	r_{10}	r_{20}	$aucec_{50}$
		Yes	**0.0552**	**0.131**	0.305	**0.1148**	**0.337**	**0.358**
Add	No	No	0.1639	0.344	0.342	0.1311	0.304	0.387
Add	No	Yes	**0.0342**	**0.249**	**0.292**	0.1311	0.341	**0.402**
Add	Yes	No	0.1627	0.337	0.338	0.1311	**0.276**	**0.376**
Add	Yes	Yes	**0.0366**	**0.247**	**0.292**	0.1311	0.341	**0.402**
Delta		No	0.1652	0.343	0.344	0.1311	0.292	0.381
Delta		Yes	**0.0663**	**0.287**	**0.303**	0.1311	0.294	0.382
Dist	No	No	0.1623	0.337	0.349	0.1311	**0.351**	**0.406**
Dist	No	Yes	**0.1211**	**0.318**	0.325	0.1311	**0.350**	**0.406**
Dist	Yes	No	0.1588	0.335	0.347	0.1311	**0.339**	**0.397**
Dist	Yes	Yes	**0.1146**	**0.294**	0.318	0.1311	**0.349**	**0.405**

Table 6.28: Variants of release atom features (effort-sensitive performance evaluation criteria)

Change	Current	r_{10}	r_{20}	$aucec_{50}$
	No	**0.148**	0.224	0.375
Add (atoms)	No	**0.159**	**0.208**	0.366
Add (atoms)	Yes	0.137	**0.197**	**0.351**
Delta (atoms)	No	0.135	0.230	0.367
Delta (atoms)	Yes	0.133	0.228	0.365
Dist (atoms)	No	**0.160**	0.249	**0.389**
Dist (atoms)	Yes	0.113	0.239	0.360

impact on performance. However, recall that as described in Section 5.3, metrics can be designed such that value matching must be performed (or must not be performed) whenever the metric is computed. Therefore, this result does not imply that the ability to perform value matching has no impact on performance, because value matching was being performed for many metrics even in the experiments that attempted to disable it.

For release-level prediction for PHPMyAdmin, most variations of the distance-based change metrics significantly outperformed the code delta change metrics. For Moodle, the "add" variant of the distance-based metrics lagged behind the code delta change metric, while the "dist" variant outperformed it. This motivates our

use of the "dist" variant of the distance-based change metric in our other release-level experiments. The same conclusions also hold for the atom-based features for release-level prediction. In addition, for PHPMyAdmin, the current metrics feature (which, as with all the experiments, was combined with the addversjump feature) performed fairly well – with an r_{20} of .337 where .200 would be expected from the null model – although the change metrics performed significantly better.

6.4.2 Categories of metrics and atoms

In the previous set of experiments, we explored if any particular *metric family members* (such as distance-based or non-distance-based change metrics) performed better than others for change measurement. In this set of experiments, we explore if any *metric families* consistently perform better than others.

We divide metrics into *basic* metrics (metrics which primarily measure the characteristics of a single file) and *intermodule* metrics (metrics that primarily measure how a file interacts with other files in the system). In this set of experiments, both sets of metrics together are referred to as *all* metrics. In addition, in some experiments we augment the basic and intermodule metrics with *derived* metrics, which compute compound measurements (such as Halstead's volume) which are combinations of several quantities that must be measured separately. In addition, in this set of experiments, we examine combinations of metric-based and atom-based predictors, to see if the combination can perform better than metrics or atoms alone. Because of the heavy computational requirements for training the atom-based features, we limit this examination to file-level and release-level predictors, which yield far smaller example sets than the change-level predictors do.

The results of these experiments are depicted in Table 6.29 for file-level predictors, Table 6.30 for change-level predictors, and Table 6.31 for release-level predictors. Across both PHPMyAdmin and Moodle, no set of metrics performed con-

Table 6.29: Features for file-level prediction (effort-sensitive performance evaluation criteria)

Metrics	Atoms	Moodle			PHPMyAdmin		
		r_{10}	r_{20}	$aucec_{50}$	r_{10}	r_{20}	$aucec_{50}$
	Yes				**0.496**	**0.910**	**0.795**
All	No	0.168	0.307	0.376	0.595	0.788	0.781
All+Computed	No	0.170	0.317	0.374	0.567	0.815	0.787
All+Computed	Yes				**0.498**	**0.918**	**0.796**
Basic	No	0.177	0.303	0.376	0.543	**0.870**	0.787
Basic	Yes				**0.490**	**0.903**	0.792
Intermodule	No	0.111	0.248	**0.332**	**0.658**	0.805	**0.805**
Intermodule	Yes				**0.492**	**0.907**	0.791
Subset+Computed	No	0.151	0.260	**0.272**	0.623	**0.844**	**0.803**

Table 6.30: Features for change-level prediction (effort-sensitive performance evaluation criteria)

Metrics	Moodle			PHPMyAdmin		
	r_{10}	r_{20}	$aucec_{50}$	r_{10}	r_{20}	$aucec_{50}$
All	0.278	0.428	0.519	0.496	0.888	0.793
All+Computed	0.258	0.451	0.535	0.495	0.888	0.791
Basic	0.256	0.427	0.520	0.491	**0.846**	**0.782**
Intermodule	0.310	**0.545**	**0.571**	0.471	**0.709**	**0.723**
Subset+Computed	0.250	0.461	0.530	0.476	**0.799**	**0.751**

sistently, significantly better than the combination of the two sets (the *all* set). In addition, different sets of metrics appeared to perform better for different kinds of predictors – for example, for Moodle, the intermodule set performed better than the basic set for change-level prediction, but the reverse was true for file-level prediction. Adding the derived metrics to the set of all the basic and intermodule metrics did not significantly affect performance in any case – in other words, the prediction algorithm was able to take advantage of the constituent parts of the derived metrics without them being combined into more complex, composite metrics.

Notably, the performance of the atom-based feature was similar to the performance of the set of all metrics, performing slightly but significantly better for file-level predictors and slightly but significantly worse for release-level predictors.

Table 6.31: Features for release-level prediction (effort-sensitive performance evaluation criteria)

Metrics	Atoms	Moodle			PHPMyAdmin		
		r_{10}	r_{20}	$aucec_{50}$	r_{10}	r_{20}	$aucec_{50}$
	Yes				**0.160**	**0.249**	**0.389**
All	No	0.108	0.292	0.316	0.131	0.349	0.405
All+Computed	No	0.115	0.294	0.318	0.131	0.349	0.405
All+Computed	Yes				**0.158**	**0.261**	0.396
Basic	No	0.123	0.308	0.330	**0.125**	**0.294**	**0.381**
Basic	Yes				**0.160**	**0.256**	0.394
Intermodule	No	0.131	0.314	0.339	**0.209**	**0.412**	**0.426**
Intermodule	Yes				**0.159**	**0.251**	**0.391**
Subset+Computed	No	0.133	0.326	0.328	**0.120**	**0.355**	**0.398**

This reenforces our conclusion [165] that automatically generated features (such as atoms, or in the cited work, text tokens) can take the place of hand-crafted metrics for vulnerability prediction. However, due to the severe performance impact of the atom-based features (described earlier in this chapter), we limit our study to metric-based features for the remainder of this work.

6.4.3 Aggregation methods for release-level prediction

Recall from Figure 3.1 that a cube of features must be transformed into a release/feature matrix before it can be used for release-level prediction. These transformations are performed with the combinefiles operator which we defined in Definition 3.17. For example, when predicting which release of a product will introduce new vulnerabilities, the change metrics of all files changed during the release can be added together to characterize the composite change made over the course of the release.

In this section, we compare the combinesum, combinemax, combinemin, combinemean, and combineentropy operations which were defined in the same section. In each experiment, exactly one of these operations is used in conjunction with the combinefiles

Table 6.32: Aggregation functions for release-level prediction (effort-sensitive performance evaluation criteria)

Aggregation	Moodle			PHPMyAdmin		
	r_{10}	r_{20}	$aucec_{50}$	r_{10}	r_{20}	$aucec_{50}$
Entropy	**0.1582**	**0.355**	**0.388**	0.1311	**0.353**	**0.407**
Max	0.0504	0.211	0.267	0.1351	**0.320**	0.398
Mean	0.0942	0.215	0.329	**0.0820**	**0.300**	**0.357**
Sum	0.1079	0.292	0.316	0.1311	0.349	0.405

operator to perform the transformation.

The results of the comparison are shown in Table 6.32. As shown in this table, the entropy-based aggregation operator performed significantly better than the baseline (the simpler sum operator) for both PHPMyAdmin and Moodle. Note that the entropy-based and sum-based aggregation of change metrics measure change in very different ways – entropy measures the extent that change is spread across multiple files, while the sum measures the amount of change that had been made (but not its distribution) – yet each of these quantities, by themselves, yielded effective vulnerability predictors. Recall that we are employing effort-sensitive evaluation criteria in these experiments, so it is not surprising that measuring the character of the changes made in a release (with the entropy operator) is more effective than measuring the volume of those changes (with the sum operator) – attempting to base a predictor on volume will backfire as a larger release will likely introduce more vulnerabilities but will also take more effort to inspect.

6.4.4 Metadata features for prediction over time

We now consider several *metadata* features that could potentially replace or supplement change metrics when performing change-level or release-level prediction. These metadata features consider characteristics of the change such as the number of files changed during the revision, or the amount of days that had passed since

Table 6.33: Non-metric predictors to augment metrics for change-level prediction (effort-sensitive performance evaluation criteria)

Metrics	Extra	Moodle			PHPMyAdmin		
		r_{10}	r_{20}	$aucec_{50}$	r_{10}	r_{20}	$aucec_{50}$
Yes		0.278	0.428	0.519	0.496	0.888	0.793
Yes	Chg days ago	0.279	0.439	0.524	0.512	0.894	0.791
Yes	Chg revs ago	0.279	0.439	0.524	0.512	0.894	0.791
Yes	Date jump	0.269	0.408	0.513	**0.555**	0.893	**0.807**
Yes	Files changed	0.257	0.434	0.520	**0.556**	0.898	0.803
Yes	Version jump	0.278	0.428	0.519	0.496	0.888	0.793

the prevision revision of the application.

The following metadata features are evaluated here, with the definitions of the features coming from Section 3.4:

- **Change days ago**: The number of days since a file had last been changed using the addlastchangeddate operator.
- **Change revisions ago**: The number of revisions since a file had last been changed using the addlastchangedrev operator.
- **Date jump**: The number of days since the previous *release* of the product, using the adddatejump operator.
- **Version jump**: The magnitude of the current release (major, minor, or patch), computed by comparing this release's version number against the previous one using the addversjump operator.

In Table 6.33, we evaluate if any of these metadata features improve prediction performance for change-level prediction. The results indicate that none of the metadata features resulted in a notable performance improvement for both products.

Next, we evaluate metadata features in the context of release-level prediction. In this case, we evaluate each metadata feature in two contexts – one to explore

Table 6.34: Non-metric predictors to augment metrics for release-level prediction (effort-sensitive performance evaluation criteria)

Metrics	Extra	Moodle			PHPMyAdmin		
		r_{10}	r_{20}	$aucec_{50}$	r_{10}	r_{20}	$aucec_{50}$
Change+Curr		0.1079	0.292	0.316	0.1311	0.349	0.405
Change+Curr	Date jump	0.1164	0.301	0.320	0.1311	0.349	0.405
Change+Curr	Files changed	0.1062	0.293	0.316	0.1311	0.349	0.405
Change+Curr	Version jump	0.1079	0.292	0.316	0.1311	0.349	0.405
Curr	Date jump	0.0607	**0.149**	0.313	**0.1148**	**0.335**	**0.355**
Curr	Files changed	0.0738	**0.132**	0.309	**0.1148**	**0.328**	**0.353**
Curr	Version jump	0.0588	**0.142**	0.310	**0.1148**	**0.337**	**0.358**

if the feature improves prediction performance in conjunction with change metrics, and another to see if the metadata features can actually take the place of change metrics for release-level prediction if they are combined with the static code metrics of the release.

These results for metadata features and release-level prediction are shown in Table 6.34. As in the case of the change-level predictors, no metadata features resulted in a meaningful improvement in performance for both products. Furthermore, when the change metric features were excluded from the release-level predictors, performance dropped notably and significantly (following the $aucec_{50}$ indicator for PHPMyAdmin and the r_{20} indicator for Moodle). In other words, the metadata features were no substitute for change metrics in the context of release-level prediction.

6.4.5 Augmenting metrics with history-based features

In our final set of experiments, we will examine if considering the recent *change history* of a product can improve prediction performance. Change history features are distinct from the change metric features in the sense that change history features consider multiple changes over a window of days or revisions in the past, while change metric features only consider the change made from one revision to the next. As

with previous sets of experiments, features are constructed using the operators we defined in Section 3.4.

Because there are many ways to construct change history features, we explore numerous combinations of the following parameters:

1. **Operator**: Most experiments performing computations over a sliding window will use one of the following operators which were defined previously: slidecubesum, slidecubemin, slidecubemax, slidecubemean, slidecubedecaysum.

2. **Range**: The slidewindrevs operator is used to create a sliding window spanning the specified number of releases. For example, when the slidecubemean operator was used, a moving average over the specified number of past releases is computed.

3. **Mean change interval**: A mean change interval feature can be added (which is associated with the frequency that a file has been changed recently). This feature is computed by adding a last-changed-date feature with the addlastchangeddate operator and then creating a moving average with the slidecubemean operator.

We first examine how change history can be used to augment static code metrics in the context of file-level prediction. Because file-level prediction takes place at one specific revision, this results in the utilization of the recent change history for each file at the time of prediction. The results, depicted in Table 6.35, show no advantage to augmenting static code metrics with change history for PHPMyAdmin, with many such features (including adding the mean change interval) actually reducing performance, sometimes significantly so. In contrast, for Moodle, both the mean change interval and sliding windows with the slidecubemean operator showed a notable performance increase (significant in the latter case).

We next examine the results of replacing static code metrics with change history altogether. We refer to change history features for file-level prediction as

process metrics, and this second set of experiments is indicated by the word "process" in the left-hand column of the table. The results for using process metrics only mirrored the results from the first set of experiments. PHPMyAdmin predictors performed significantly worse when substituting code metrics with process metrics, although performance was still well above what would be expected by the null model. In contrast, for Moodle, which benefited from augmenting code metrics with change history, performance improved when replacing static code metrics with process metrics using the slidecubemean operator.

In the previous set of experiments, we examined if considering the recent change history of a file can affect the performance of file-level vulnerability prediction. Next, we examine how features based on recent change history affect change-level and release-level prediction. Recall that change-level and release-level prediction occurs over a range of revisions, predicting the revisions where vulnerabilities will be introduced into a product (along with, in the case of change-level prediction, the file where the vulnerability is to appear). In this context, change history is computed with a sliding window, looking at the recent history of each file by considering a fixed number of previous revisions at each release.

Table 6.36 shows the results of augmenting change metrics with change history for change-level predictors. Results for change-level prediction were very similar to the results for file-level prediction – there was no consistent benefit to considering change history when analyzing PHPMyAdmin, while for Moodle there was, with the slidecubemean operator conferring the highest benefit.

However, leveraging change history when performing release-level prediction (depicted in Table 6.37) had the opposite effect. Adding change history uniformly improved vulnerability prediction performance for PHPMyAdmin, while for Moodle, the change history features worsened performance in every case. However, as in the previous cases, change history features computed with the slidecubemean operator

Table 6.35: Process metrics to augment file metrics for file-level prediction (effort-sensitive performance evaluation criteria)

Metrics	Op	Range	Chgint	Moodle			PHPMyAdmin		
				r_{10}	r_{20}	$aucec_{50}$	r_{10}	r_{20}	$aucec_{50}$
Code			No	0.168	0.307	0.376	0.595	0.788	0.781
Code		22	Yes	0.158	0.346	0.398	**0.472**	0.771	0.761
Code		45	Yes	0.155	0.311	0.404	**0.487**	0.755	0.758
Code	decay	11	No	0.154	0.298	0.369	0.528	0.787	0.767
Code	decay	22	No	0.183	0.303	0.369	**0.490**	0.762	**0.760**
Code	decay	22	Yes	0.151	0.317	0.370	**0.489**	0.759	**0.762**
Code	decay	33	No	0.161	0.287	0.362	0.549	0.766	0.768
Code	decay	45	No	0.194	0.300	0.353	**0.528**	0.781	0.769
Code	decay	45	Yes	0.138	0.340	0.366	0.562	**0.755**	0.766
Code	max	11	No	0.193	0.288	0.372	0.555	0.795	0.777
Code	max	22	No	0.180	0.315	0.391	**0.465**	0.742	**0.749**
Code	max	22	Yes	0.152	0.350	0.394	**0.481**	**0.733**	**0.750**
Code	max	33	No	0.167	0.318	0.392	**0.493**	0.785	0.766
Code	max	45	No	0.177	0.311	0.389	0.540	0.799	0.773
Code	max	45	Yes	0.146	0.347	0.396	0.542	0.799	0.769
Code	mean	11	No	0.147	0.351	0.405	**0.479**	0.777	0.760
Code	mean	22	No	0.200	0.360	**0.439**	**0.487**	**0.739**	**0.749**
Code	mean	22	Yes	0.193	**0.393**	**0.440**	**0.474**	**0.739**	**0.746**
Code	mean	33	No	0.221	**0.405**	**0.441**	**0.499**	0.747	**0.753**
Code	mean	45	No	0.168	**0.399**	**0.432**	0.554	0.784	0.771
Code	mean	45	Yes	0.186	**0.410**	**0.440**	0.551	0.777	0.768
Code	sum	11	No	0.173	0.319	**0.346**	**0.525**	0.780	0.768
Code	sum	22	No	0.190	0.316	0.371	**0.512**	**0.736**	**0.757**
Code	sum	22	Yes	0.130	0.300	0.370	**0.518**	**0.732**	**0.757**
Code	sum	33	No	0.190	0.309	**0.350**	0.574	0.766	0.773
Code	sum	45	No	0.182	0.315	0.356	0.554	0.825	0.784
Code	sum	45	Yes	0.170	0.324	0.366	0.543	0.812	0.784
Process	decay	22	Yes	0.132	**0.237**	**0.321**	**0.465**	0.756	**0.759**
Process	decay	45	Yes	0.119	0.271	**0.334**	**0.426**	0.770	**0.744**
Process	max	22	Yes	0.120	0.310	0.369	**0.478**	0.742	**0.746**
Process	max	45	Yes	0.129	0.302	0.385	**0.418**	**0.744**	**0.735**
Process	mean	22	Yes	0.179	0.370	**0.422**	**0.403**	0.734	**0.735**
Process	mean	45	Yes	0.210	**0.409**	**0.437**	**0.349**	0.761	**0.726**
Process	sum	22	Yes	0.202	0.310	0.367	**0.513**	**0.713**	**0.749**
Process	sum	45	Yes	0.159	0.327	0.379	0.536	0.820	0.774

were most beneficial.

Table 6.36: Change history to augment metrics for change-level prediction (effort-sensitive performance evaluation criteria)

Op	Range	Moodle			PHPMyAdmin		
		r_{10}	r_{20}	$aucec_{50}$	r_{10}	r_{20}	$aucec_{50}$
		0.278	0.428	0.519	0.496	0.888	0.793
decay	11	0.292	**0.539**	**0.571**	**0.557**	0.872	0.798
decay	22	0.258	**0.470**	0.537	**0.549**	0.885	0.795
decay	33	0.271	**0.490**	**0.544**	**0.550**	**0.865**	0.788
decay	45	0.275	**0.491**	**0.544**	**0.531**	**0.851**	0.783
max	11	0.292	**0.478**	**0.556**	**0.560**	0.896	0.802
max	22	0.297	**0.526**	**0.564**	0.510	0.905	0.792
max	33	**0.321**	**0.534**	**0.570**	0.521	0.888	0.791
max	45	**0.323**	**0.514**	**0.568**	0.506	0.880	**0.782**
mean	11	0.309	**0.595**	**0.608**	**0.564**	0.899	**0.804**
mean	22	0.304	**0.561**	**0.597**	**0.528**	0.896	0.796
mean	33	0.254	**0.552**	**0.587**	0.526	0.888	0.786
mean	45	0.281	**0.567**	**0.594**	0.517	0.881	0.790
sum	11	0.252	**0.492**	**0.549**	**0.544**	0.888	0.797
sum	22	0.256	**0.519**	**0.578**	0.523	0.875	0.786
sum	33	0.275	**0.530**	**0.567**	0.502	0.861	**0.777**
sum	45	0.298	**0.522**	**0.571**	0.496	**0.840**	**0.769**

Table 6.37: Change history to augment metrics for release-level prediction (effort-sensitive performance evaluation criteria)

Metrics	Op	Range	Moodle			PHPMyAdmin		
			r_{10}	r_{20}	$aucec_{50}$	r_{10}	r_{20}	$aucec_{50}$
Change+Curr			0.0366	0.247	0.292	0.1311	0.341	0.402
Change+Curr	decay	22	0.0577	0.188	0.287	0.1303	0.343	0.400
Change+Curr	decay	45	0.0650	0.164	0.290	**0.1293**	0.333	0.395
Change+Curr	max	22	0.0421	0.174	0.261	0.1304	0.345	0.403
Change+Curr	max	45	0.0626	0.223	0.290	**0.1293**	0.335	0.396
Change+Curr	mean	22	0.0531	0.242	0.295	0.1302	0.342	0.401
Change+Curr	mean	45	0.0546	0.235	0.295	0.1302	0.343	0.402
Change+Curr	sum	22	0.0472	0.182	0.274	0.1304	0.345	0.403
Change+Curr	sum	45	0.0667	0.239	0.290	**0.1311**	0.340	0.398

6.5 Discussion of algorithms and feature variants

The results of the experiments involving variations on features was mixed. Some experiments found benefit for either one product or the other when augmenting code and change metrics with more complex features, but a consistent benefit

was rarely found when using any technique. In contrast, the choice of machine learning algorithm (both the effort-sensitivity enhancements and the core algorithm itself) had a largely consistent (and in some cases, dramatic) impact on the performance of the models that resulted. These findings are consistent with previous observations [92] that properly choosing a learning method (or algorithm) is far more important than attempting to find the "best" features for defect prediction problems.

However, our findings do contradict studies such as [124] which found that predictors that consider software change history perform consistently better than other kinds of predictors. One possible reason for the discrepancy is that the change history predictors in this study worked on *releases*, while the history predictors in previous studies worked on the level of *commits* or other granular units. It's possible that the finer-grained commit level change data would have replicated the results of these past studies – that change data is more efficacious than static code metrics.

Our findings confirm the results of a related study [129] in that it is possible to build effective predictors with a non-metric feature based on fine-grained code characteristics (in our case, atoms, in the other case, text tokens). The advantage of using fine-grained code characteristics, rather than traditional software metrics, is that there is no need to manually define a large collection of metrics. However, because there are many more different atoms (or tokens) than there are metrics, processing time and memory requirements for learning with fine-grained characteristics is much slower – sometimes to the point of intractability.

Now that we have studied the effects of using different *learning methods* and *features* for vulnerability prediction, we will compare the performance of different *experimental setups* in the next section. These experimental setups will utilize the learning methods and features described previously to perform vulnerability prediction in different contexts, such as continuously predicting vulnerable changes

over time, or looking for individual vulnerable files at one specific moment in the product's history.

6.6 Comparing and combining different prediction setups

In previous sections, we explored the three setups for prediction – file-level, release-level, and change-level prediction – separately, considering them to be separate problems entirely. In this section, we compare these three prediction setups – first by comparing their performance indicators, as we did in the previous section – and then by running simple simulations to explore the practical impact of each one.

We first explore the possibility of combining multiple prediction setups, such that a total effort budget is allocated between two prediction strategies. Different prediction strategies have different strengths and weaknesses. For example, file-level prediction for PHPMyAdmin is far more efficient in terms of effort, but change-level prediction finds vulnerabilities sooner. We introduce the concept of *tandem* vulnerability prediction to describe how two different kinds of predictors may be used in a single experiment.

6.6.1 Combining prediction setups

We define *tandem vulnerability prediction* as the practice of using two different predictive models as part of the same experiment and measuring which vulnerabilities were found by either of the two models. Recall that a standard performance indicator measures the **R** (recall) of a prediction model that has been tuned to target a predetermined **IR** (inspection ratio). Tandem vulnerability prediction apportions a predetermined **IR** code inspection budget to both models. In order to apportion the effort in this way, it is necessary to choose a value for the *total effort* that is to be distributed, and the *proportion* of the effort that goes to each model.

Note that computing the combined **R** is more complicated than simply summing the **TP** for the two models. Some vulnerabilities may have been found by both models, and they cannot be counted twice. Furthermore, when combining a file-level model and a change-level model with tandem prediction, some vulnerabilities will be unavailable to one model or another. File-level models cannot find vulnerabilities that were fixed before the version where the prediction takes place. Change-level models cannot find vulnerabilities that were introduced before the beginning of the software's history (i.e. before change data for the product was being recorded).

In order to compute the **R** for a point on the tandem cost-effectiveness curve, we first choose the set of vulnerabilities under consideration. In some experiments, referred to as *matched set* experiments, we compute the set of all the vulnerabilities that both models were capable of finding (i.e. the intersection). In other experiments, we simply consider the set of vulnerabilities for one of the two models. For example, later on, we will compare the performance of file-level and change-level models, while basing the computation of **R** on the set of vulnerabilities that the change-level model is capable of predicting. In such an experiment, when the proportion of effort devoted to the file-level model reaches 100%, it becomes impossible for any model (even a perfect one) to yield a **R** of 1.0, because vulnerabilities fixed before the base release count toward the denominator of the proportion but can never be found by the file-level model. Note that this characteristic of the tandem performance indicator is deliberate, in order to capture how it is preferable to find vulnerabilities as they are introduced (rather than performing file-level prediction at a later date), finding them more quickly than they would have been found without the model.

In Figures 6.10 and 6.11, we plot the number of vulnerabilities that would be found with a file-level and change-level model in tandem, across a range of total effort and proportion parameters. Each line on the graph represents a different

Table 6.38: Mixing file and change metrics for vulnerability discovery

Application	$\%_{file}$	r_{20}	i_{20}	i_{40}	i_1	auec	$aucec_{50}$
Moodle	40	0.484	84395	181736	1132890	274271	0.524
Moodle	50	0.491	83099	166337	1132890	276928	0.524
Moodle	70	0.530	81888	154997	1132890	268203	0.539
PHPMyAdmin	40	0.931	8712	18429	423400	32681	0.846
PHPMyAdmin	50	0.888	8293	18585	423400	34455	0.837
PHPMyAdmin	70	0.814	8039	16166	423400	44882	0.788

inspection budget, measured in lines of code, to be distributed between the file-level and change-level predictor. The X-axis represents proportion of the budget to be distributed to the file-level predictor (as opposed to the change-level predictor), and the Y-axis represents the total number of vulnerabilities which would be found when using the two predictors in tandem at that mix. This total is computed from the set of vulnerabilities which the change-level predictor is capable of finding, meaning that vulnerabilities present since the beginning of the history period are not counted. Table 6.38 shows similar information in tabular form, presenting several performance indicators at several different mixtures of file-level and change-level models.

For PHPMyAdmin, the results indicate that the prediction tends to become more efficient as more effort is devoted to file-level prediction, in the sense that more vulnerabilities are discovered while inspecting the same amount of code, even when taking into account the fact that the change-level models have an opportunity to catch vulnerabilities that wouldn't persist long enough for the file-level models to find them. However, it is still desirable to devote as much of the inspection budget to the change-level models as possible, to eliminate the latency period where undiscovered vulnerabilities persist in the system. Note that (at least at higher inspection budgets) these curves generally have a middle region where nearly as many vulnerabilities are found as in the file-level-only region, but with less than 100% of the budget devoted to the file-level predictors. This raises the possibility that a mix of a file-level and change-level models may be a better choice than using

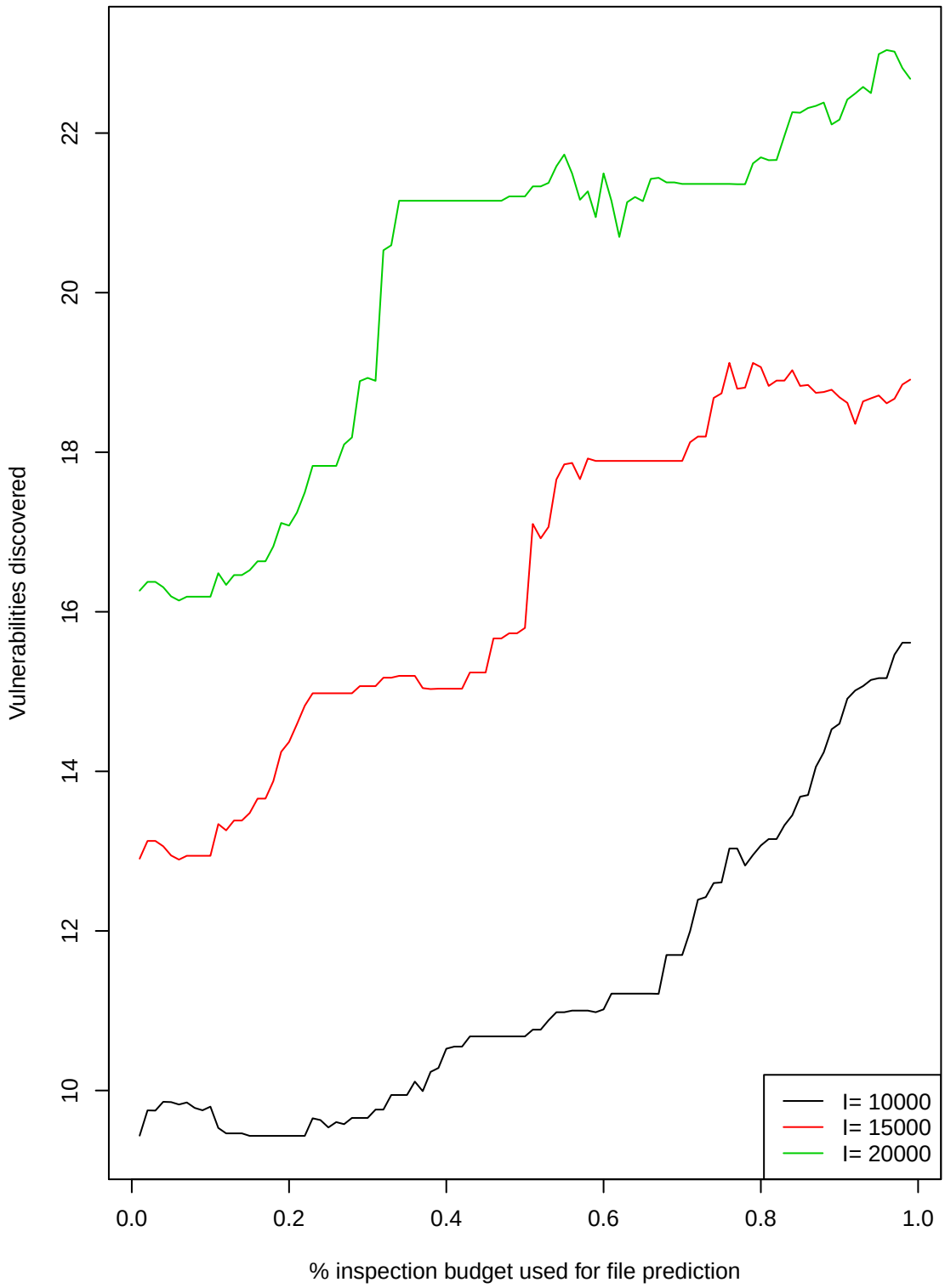


Figure 6.10: Vulnerabilities discovered in PHPMyAdmin when allocating code inspection budget to file-level and change-level predictors at the specified ratio

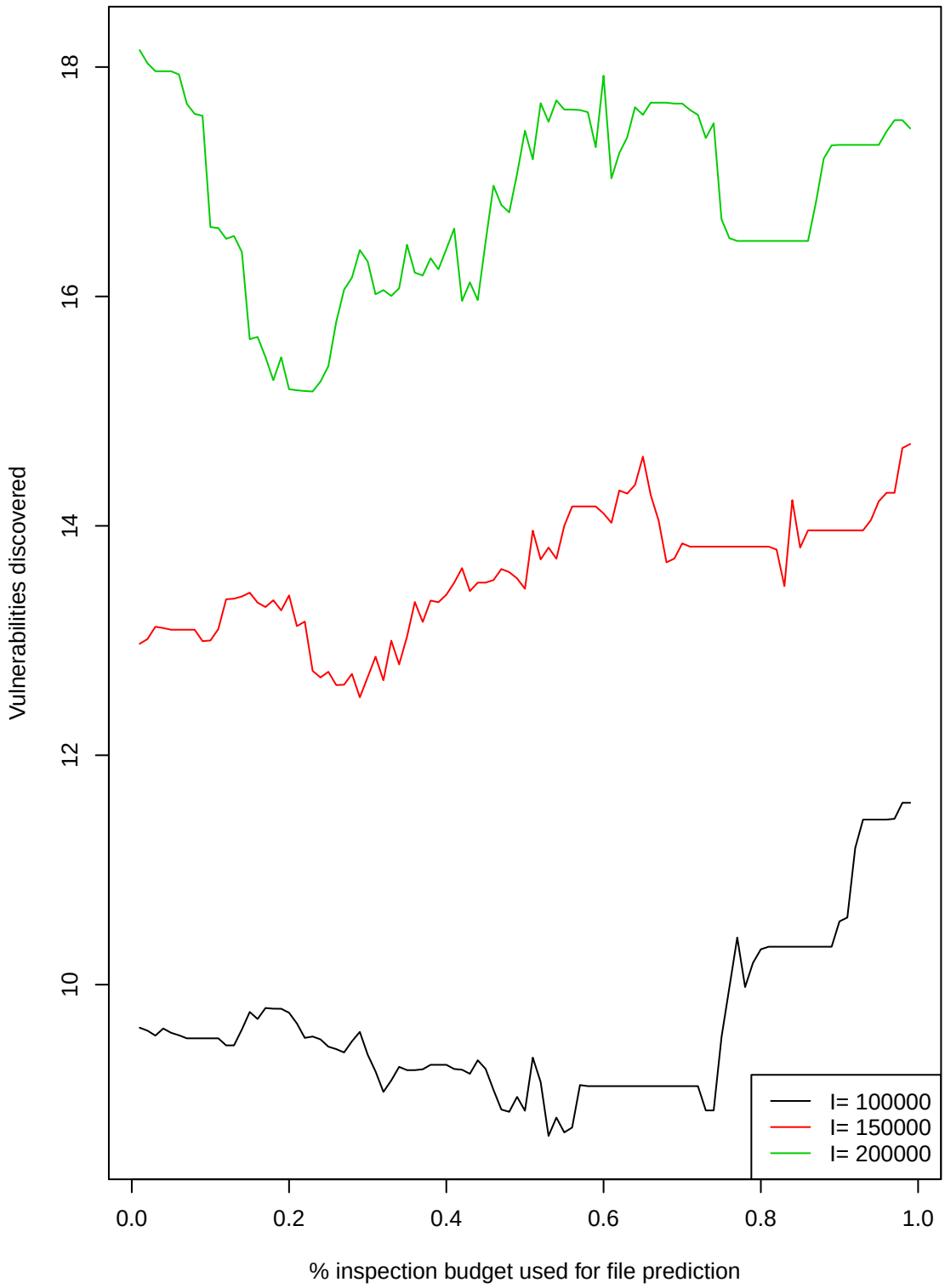


Figure 6.11: Vulnerabilities discovered in Moodle when allocating code inspection budget to file-level and change-level predictors at the specified ratio

a file-level model alone, which we will explore in the following sections.

The results for Moodle are somewhat different. Except at small inspection budgets, the total number of vulnerabilities discovered remains relatively constant across all mixtures of file-level and change-level models. This suggests that for Moodle, a pure change-level model would be ideal, as vulnerabilities could be discovered as soon as they're introduced, before their release to the public.

6.6.2 Simulations to assess the practical impact of prediction setups

We now take an alternate approach to comparing file-level and change-level prediction, by *simulating* how vulnerabilities are introduced (throughout the normal development process) and discovered (with the assistance of predictive models) over time. The intent of these simulations is to further quantify the relative benefits of file-level prediction (finding more vulnerabilities while inspecting fewer lines of code) and change-level prediction (finding the vulnerabilities that *are* found sooner).

These vulnerability simulations overlay the actual history of vulnerabilities being discovered and repaired (based on the commit history of each product) with the theoretical benefit that would have been derived by incorporating vulnerability prediction models into this process. These simulations extend from the first release in our dataset to the "base release", which is the release where file-level prediction is performed in an attempt to find all of the accumulated vulnerabilities. They assume that a certain total effort budget is to be allocated across the entire simulation period – either being spent over time on change-level prediction, being spent all at once on file-level prediction, or a combination of the two when the models are used in tandem. Vulnerability prediction models are then used to select particular changes or files to inspect. We presume that the vulnerabilities in any inspected artifact will be discovered; however, as we discussed in Section 4.3, our evaluation would not be affected even if this did not hold. If a vulnerability is found by a change-level model,

then it is assumed that the vulnerability is fixed immediately, and a file-level model later on would not have the opportunity to "re-discover" it. (In other words, the models can only be credited for finding each vulnerability once.)

In Figures 6.12, 6.13, and 6.14, we graph the results of simulations for PHPMyAdmin and Moodle at three different inspection budgets. Each line color represents a different experimental setup, or strategy (file-level, change-level, or a 50/50 mix of file-level and change-level in tandem). The solid lines of each color represent the *cumulative* number of vulnerabilities that would be found by a certain time, while the dotted lines represent the *current* number of undiscovered vulnerabilities persisting in the software. Note that the solid line includes vulnerabilities that were found naturally (vulnerabilities that were not found by the model in the simulation, but were found at the corresponding time in real life). For this reason, even when only a file-level model is employed, vulnerabilities are still discovered steadily throughout the simulation period. Filled dots indicate the total number of vulnerabilities discovered at the very end of the simulation period, while open dots indicate the number of vulnerabilities that remain undiscovered at the end of the simulation period.

For PHPMyAdmin, the simulation results for each inspection level reflect what would have been expected from the earlier measurements in Figure 6.10. File-level prediction resulted in the discovery of the most vulnerabilities in the long term but the least vulnerabilities in the middle of the simulation period. The same pattern held for Moodle.

We now plot an alternative formulation of the simulation results that separates vulnerabilities that were discovered naturally from vulnerabilities discovered by a model. These new results are depicted in Figures 6.15, 6.16, and 6.17. In these graphs, solid lines (and filled dots) represent vulnerabilities discovered by a model, while dotted lines (and open dots) represent vulnerabilities discovered naturally (i.e.

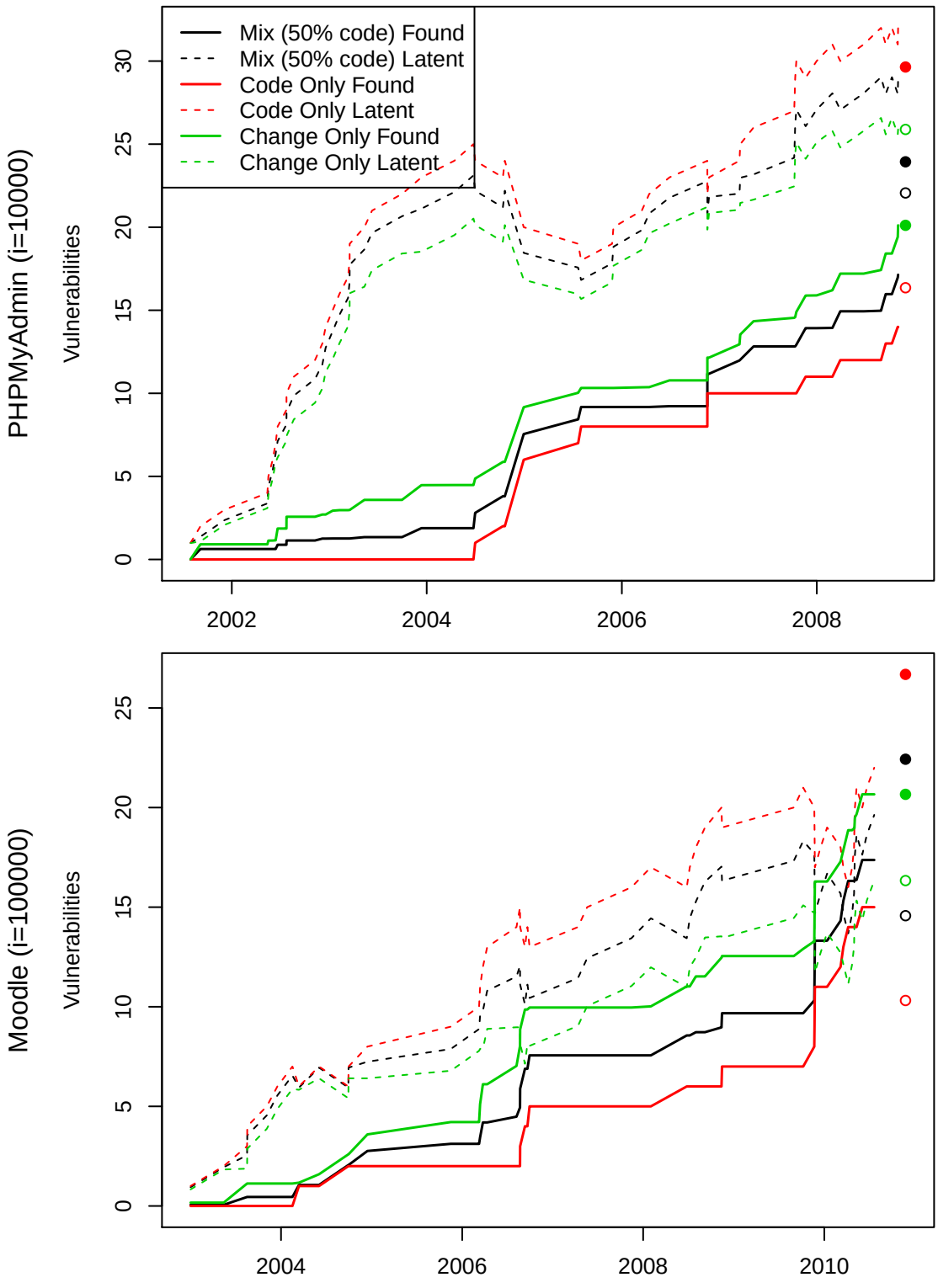


Figure 6.12: Vulnerabilities discovered or undiscovered (latent) over time when inspecting 10000 (PHPMyAdmin) or 100000 (Moodle) lines of code over the experimental period

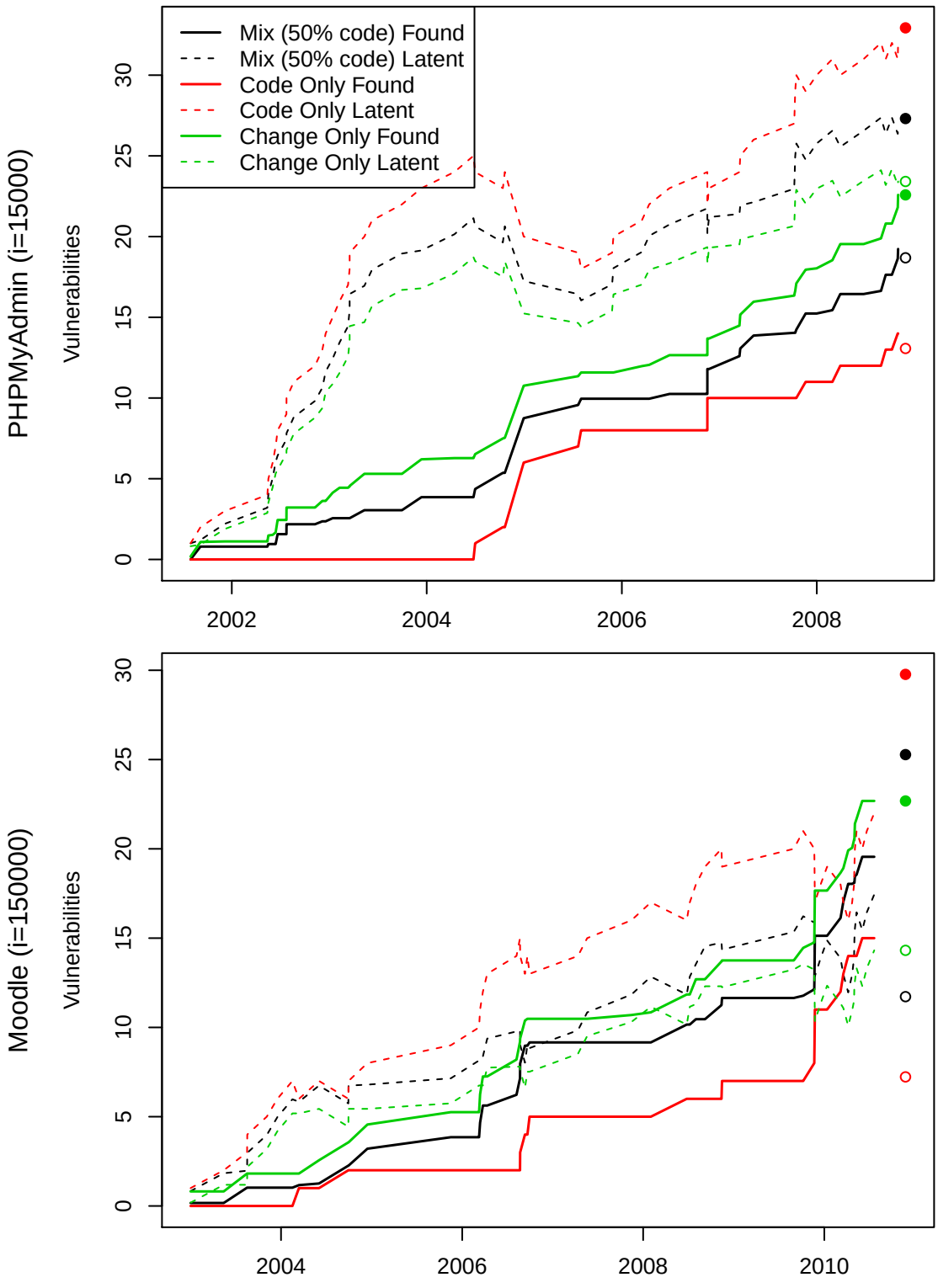


Figure 6.13: Vulnerabilities discovered or undiscovered (latent) over time when inspecting 15000 (PHPMyAdmin) or 150000 (Moodle) lines of code over the experimental period

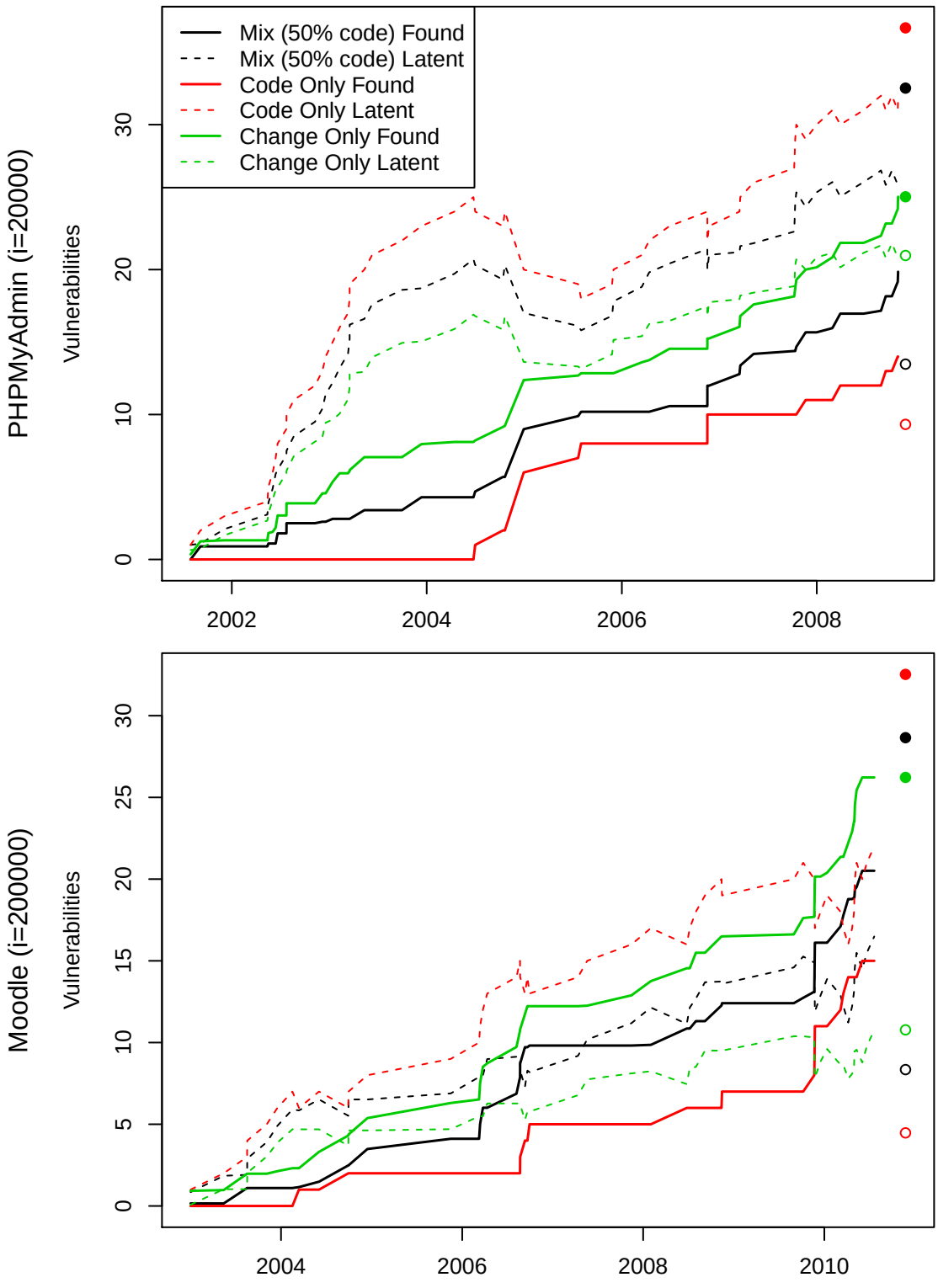


Figure 6.14: Vulnerabilities discovered or undiscovered (latent) over time when inspecting 20000 (PHPMyAdmin) or 200000 (Moodle) lines of code over the experimental period

before the point where the model would have been applied, or if all models failed to find the vulnerability when they had the chance).

Under this alternative evaluation method, the best strategy for finding the most vulnerabilities changes in some cases. For PHPMyAdmin with a total inspection of 20000 lines of code, the number of vulnerabilities found by the 50/50 mix of the two models is very close to the number found by the file-level model alone. In this case, it would be preferable to choose the mixed model because vulnerabilities would be found more quickly. For Moodle, inspecting 150000 or 200000 lines of code, the change-level model finds the most vulnerabilities (or close to the most vulnerabilities) while finding them immediately, without having to wait until the end of the simulation period to search for them.

For both PHPMyAdmin and Moodle, models incorporating change-level prediction fared better under the second evaluation criteria than the first. This is due to the second criteria penalizing strategies that found vulnerabilities too slowly, by not crediting the model for the vulnerability's discovery if it was found naturally before the point where the model was applied.

Therefore, when using this type of simulation to select a model for a particular application, it is important to consider the importance of finding vulnerabilities before they are discovered by others (the case for many "naturally" found vulnerabilities in the dataset). If it is only important that the vulnerabilities be found, regardless of *how* they are found, then the file-level predictors were the most effective, due to their ability to find many vulnerabilities at once with less effort. However, if it is important that an organization internally finds vulnerabilities before they are found by others, change-level or mixed models become more attractive, because the best way to prevent others from finding vulnerabilities is to ensure that they don't make it into a released build in the first place.

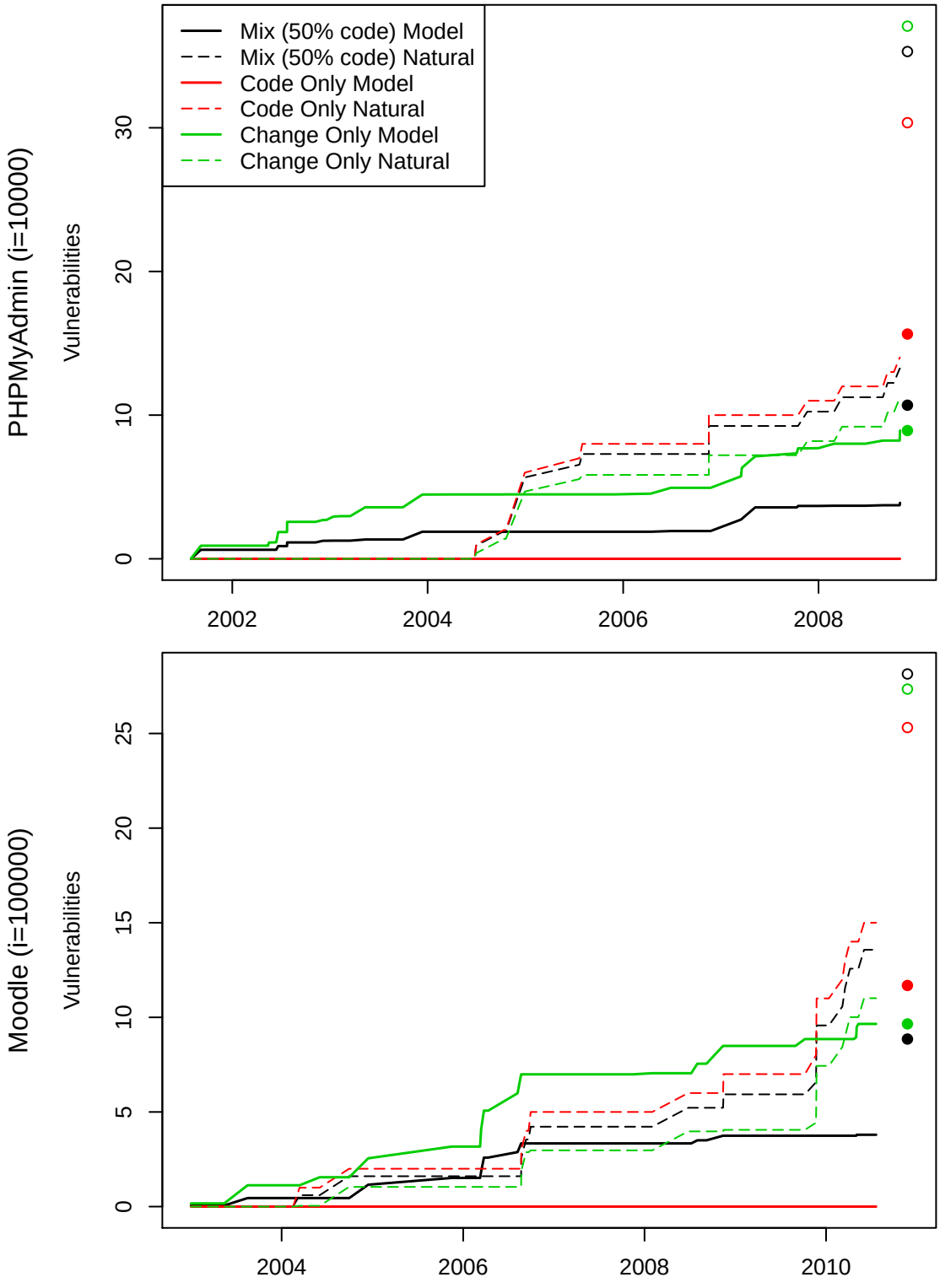


Figure 6.15: Vulnerabilities found by model and naturally (by community) when inspecting 10000 (PHPMyAdmin) or 100000 (Moodle) lines of code over the experimental period

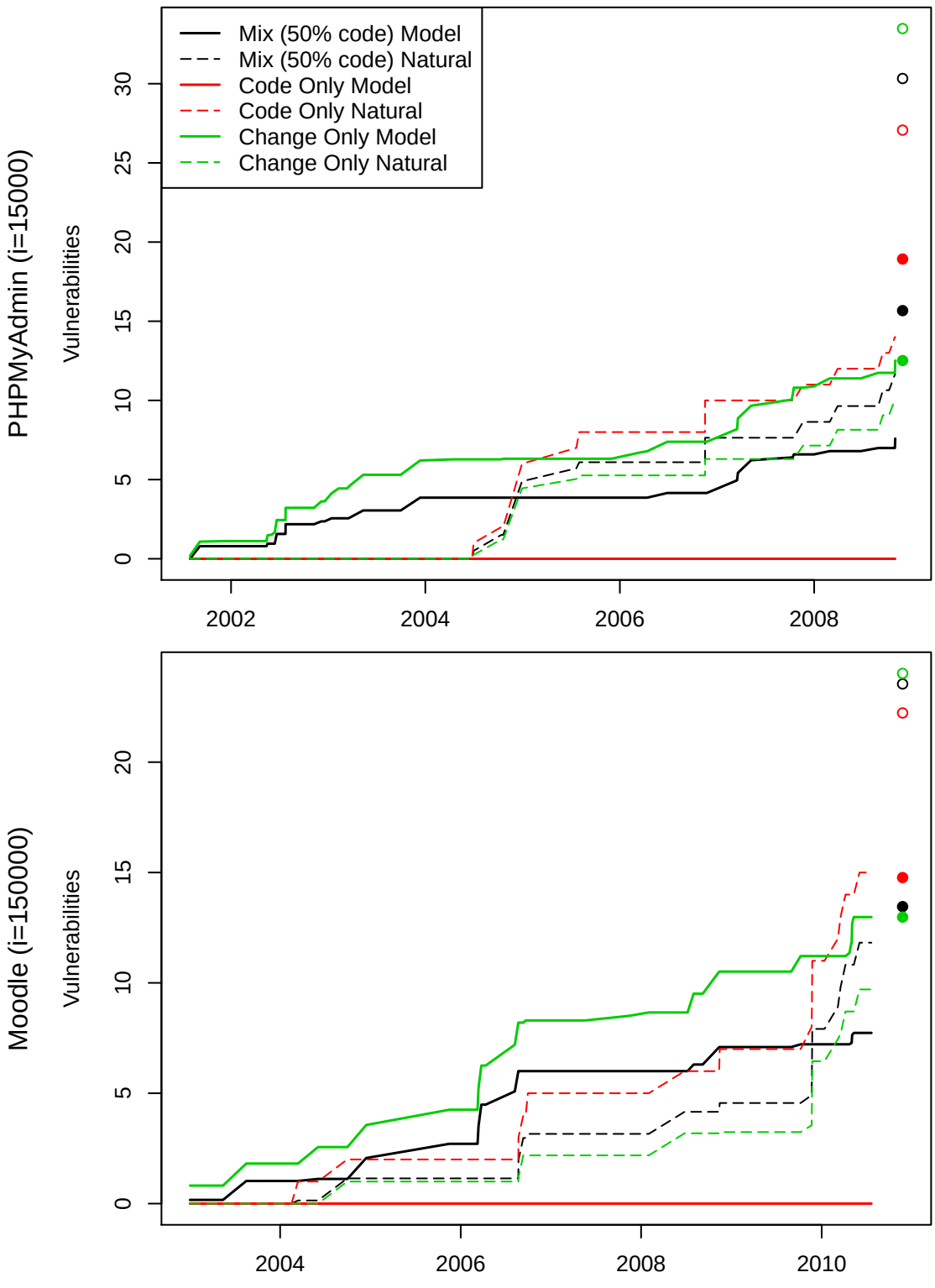


Figure 6.16: Vulnerabilities found by model and naturally (by community) when inspecting 15000 (PHPMyAdmin) or 150000 (Moodle) lines of code over the experimental period

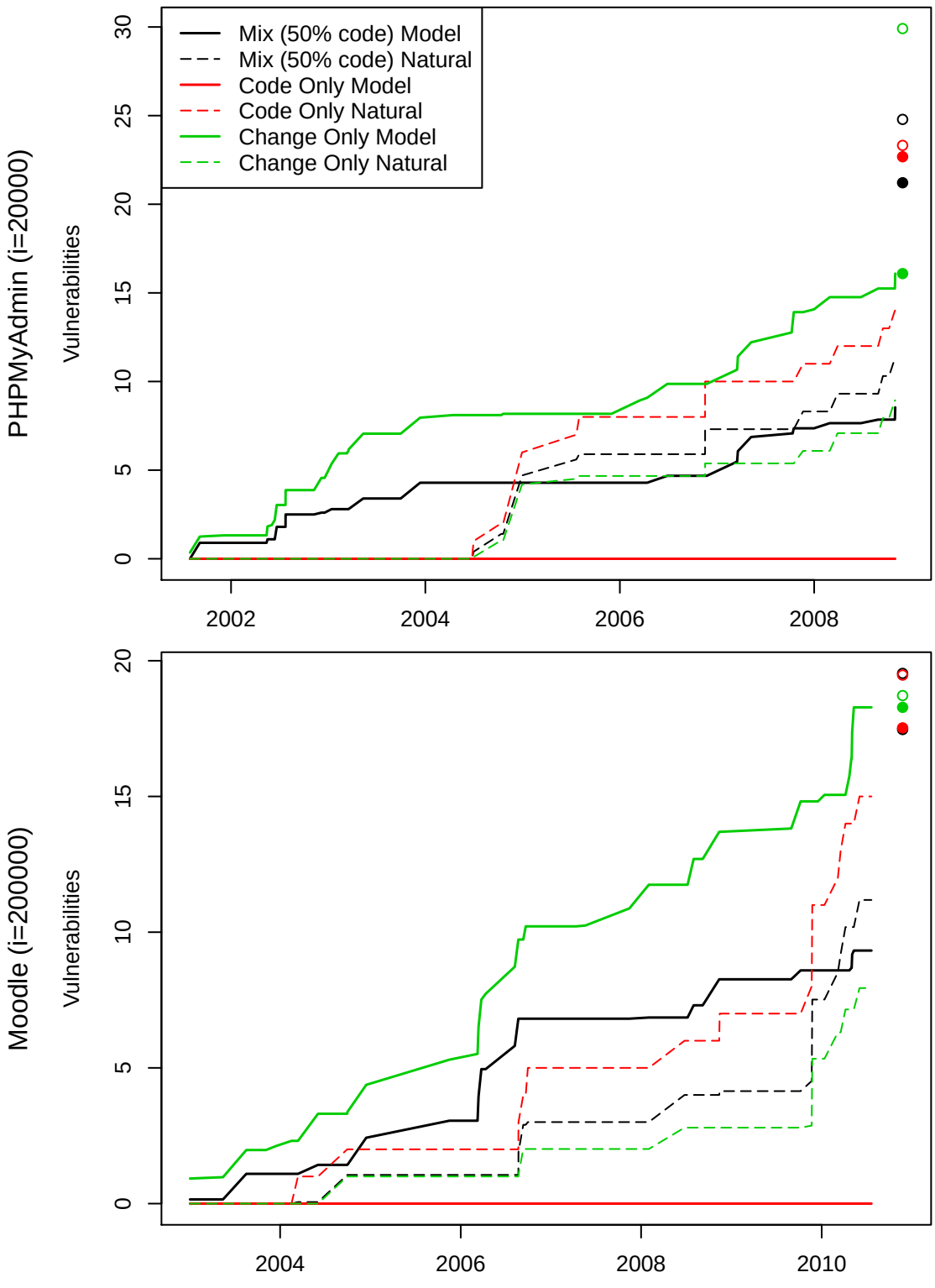


Figure 6.17: Vulnerabilities found by model and naturally (by community) when inspecting 20000 (PHPMyAdmin) or 200000 (Moodle) lines of code over the experimental period

6.6.3 Final comparisons of prediction setups

Recall that the vulnerability datasets used for these experiments do not contain all of the vulnerabilities which were present in either product. Not only were the datasets only sampled from the full population of known vulnerabilities, but it is likely that an unknown number of unknown vulnerabilities (vulnerabilities, patched or unpatched, that were not described by any vendor security advisory) exist in each product. Vulnerabilities omitted from the dataset for either reason fall into three categories:

- Vulnerabilities that were both introduced and patched before or at the base release (the release where file-level prediction is performed).
- Vulnerabilities that were introduced before or at the base release and patched after the base release.
- Vulnerabilities that were introduced after the base release.

Vulnerabilities in the first category impact the apparent performance of file-level, change-level, and release-level predictors alike, because all three predictors would have had the ability to find the omitted vulnerability, but were denied the ability to do so. Hence, any performance indicators reported in this work should be considered a *lower bound* on the *absolute* performance that would be realized from predicting vulnerabilities – in a production situation, more vulnerabilities would have been covered by the predictors while still expending the same amount of effort.

However, vulnerabilities in the second category only impact the apparent performance of change-level and release-level predictors, because the file-level predictors would not have found them. Therefore, any comparison between file and change/release predictors places a *lower bound* on the *relative* performance of the change/release predictors as compared to the file predictors.

With these caveats in mind, we present one final set of tables and figures comparing the performance of each prediction setup that we've described earlier in this chapter.

6.6.4 Absolute and relative comparisons of prediction strategies

In the results, simulations, and comparisons which we presented in previous sections, the tendency was that change-level predictors required the inspection of more code than file-level predictors to find the same number of vulnerabilities. This largely held true even when crediting the change-level predictors for vulnerabilities that the file-level predictors could not have found (vulnerabilities that had already been fixed at the base release). In this section, we examine the relative performance of multiple experimental setups in a systematic way, studying the reasons why this occurs.

In Figure 6.18, we plot the recall (**R**) for file-level, change-level, and release-level predictors as a function of the proportion of the examples that were inspected. Note that this plot was generated from a non-effort-sensitive predictor and is only intended to explore how well the machine learning algorithm can discriminate between vulnerable and non-vulnerable examples in each case. (Also note that **R** is scaled different for each line so each reaches 100% recall at 100% inspection.)

Note that for both PHPMyAdmin and Moodle, the change-level and release-level predictors outperform the file-level predictors when measured under these criteria. In other words, the machine learning algorithm is better able to distinguish vulnerable *changes* than vulnerable *files*. Although release-level prediction performance lags behind change-level prediction performance for PHPMyAdmin, the same does not hold for Moodle, which shows that in some cases changes can be distinguished equally well whether they are aggregated into releases or not.

The plots in Figure 6.19 are similar to the previous, except effort is taken into

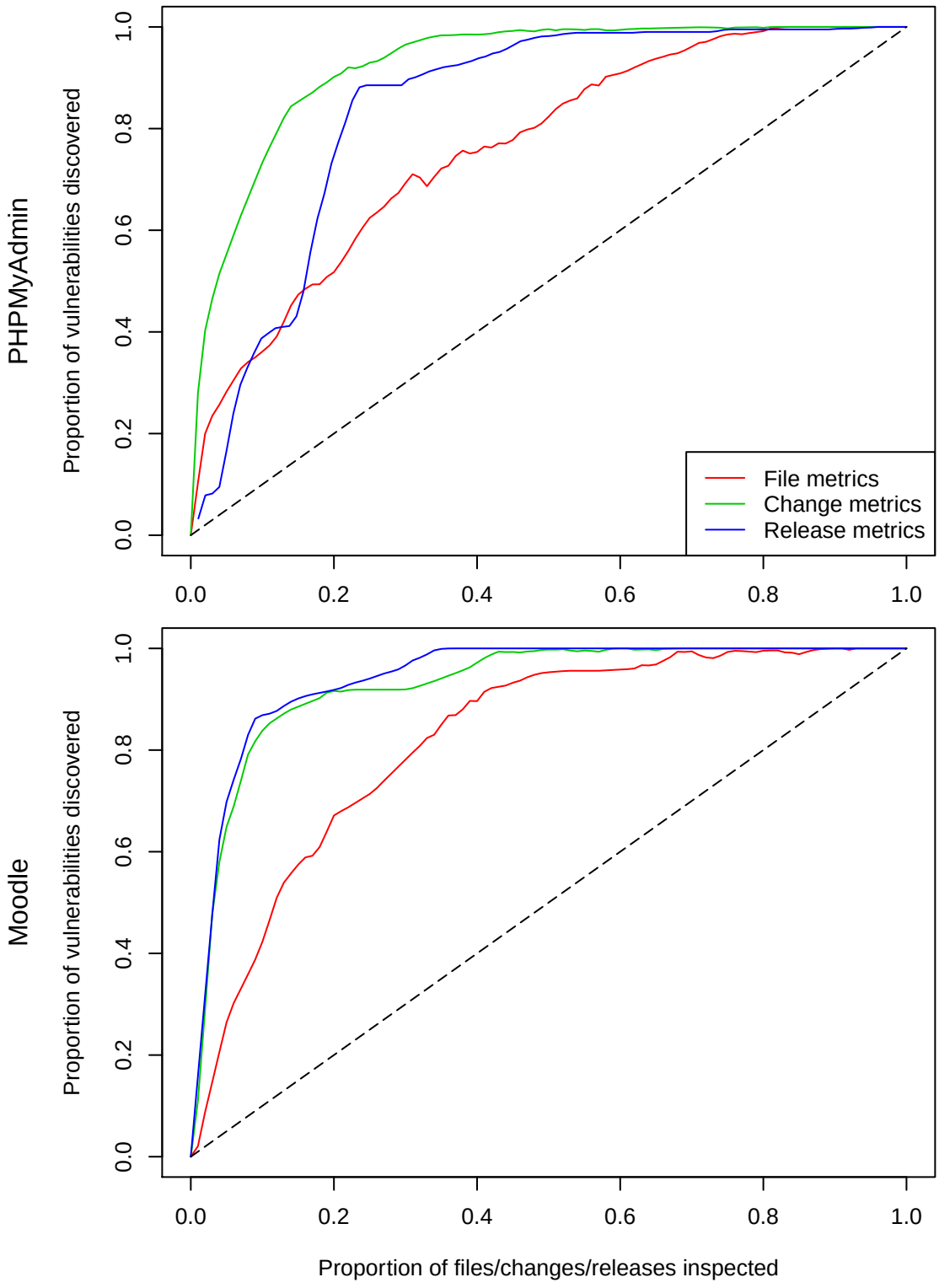


Figure 6.18: Normalized performance comparison of file, change, and release predictors by number of files, changes, or releases inspected

account and effort-sensitive models were trained. In these plots, the X-axis measures the percentage of lines of code inspected, rather than the percentage of examples inspected. In other words, these plots examine the ability of the machine learning algorithm to distinguish vulnerable lines of *code*, rather than vulnerable examples.

For PHPMyAdmin, file-level predictors fare better under the effort-sensitive comparison than they did in the effort-insensitive comparison. In fact, the performance curves for file-level and change-level predictors are remarkably similar – inspecting a similar proportion of code yields a similar proportion of vulnerabilities for both. The results for Moodle differ slightly in that the relative performance of release-level predictors drops dramatically. The similar drop in release-level prediction performance in both applications can be explained by the fact that releases are very large (i.e. they churn large amounts of code) compared to files or individual file changes. This puts a ceiling on the performance of predictors which operate on the release level, because they cannot select vulnerable artifacts at a fine enough granularity.

Finally, we compare the *absolute* performance of file-level, change-level, and release-level predictors, basing the comparison on the total number of vulnerabilities found, and the total amount of code inspected, rather than the proportion. Note that inspecting 100% of the files in the base release requires far less effort than inspecting 100% of the change between the initial release and the base release. This is because of the high degree of *churn*, or code that is introduced into the codebase and subsequently modified or deleted. In other words, when compared with file-level predictors, change-level and release-level predictors inspect much more code at an **IR** of 100% because they can inspect the same lines of code over and over again, every time that code is modified.

The results of the absolute comparison are depicted in Figure 6.20. We first examine the dashed lines, which represent vulnerabilities discovered from a matched

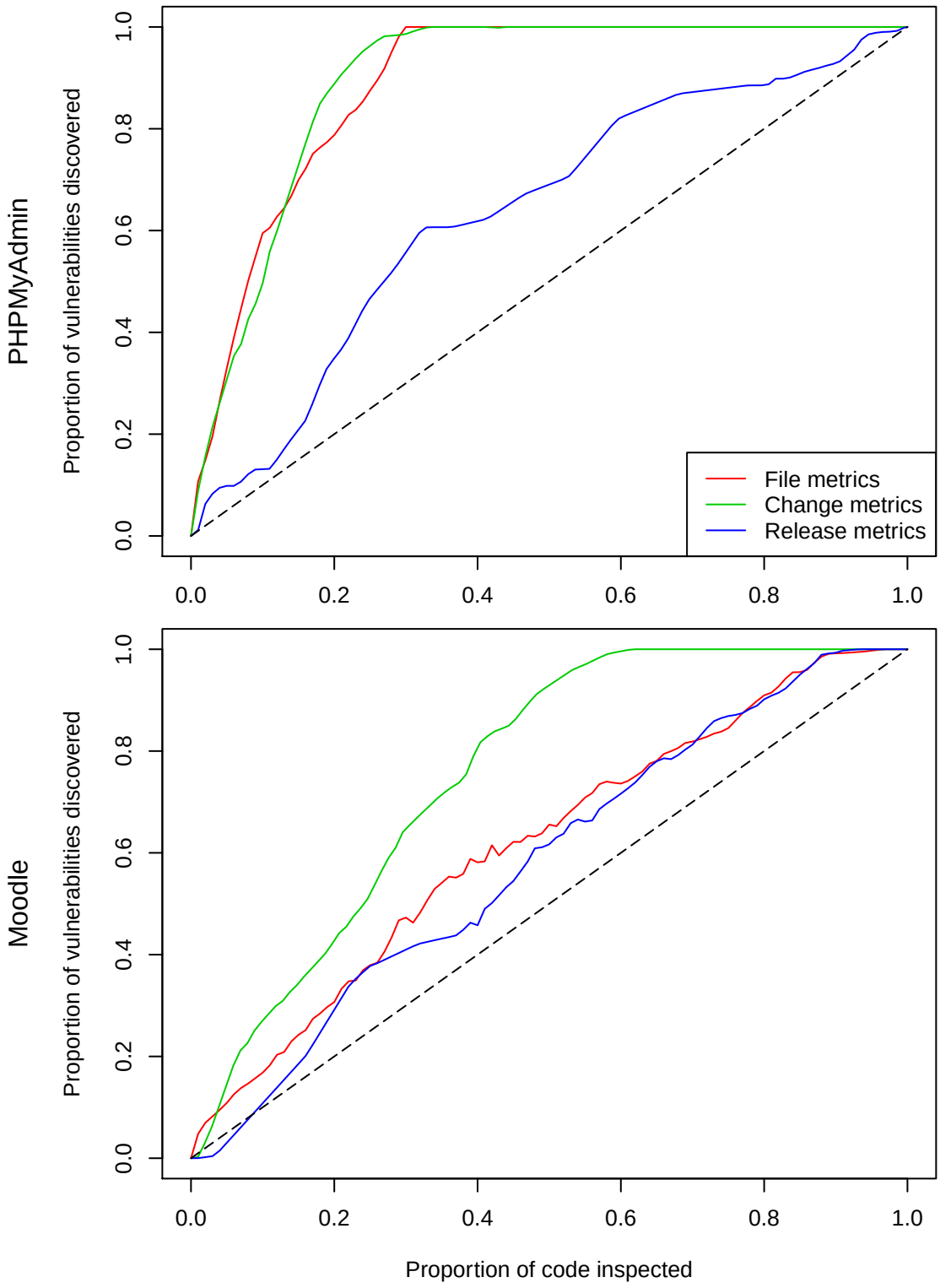


Figure 6.19: Normalized performance comparison of file, change, and release predictors by number of lines of code inspected

set as described in Section 6.6.1. We now see that, contrary to the previous comparison, file-level predictors discover the same number of vulnerabilities with far less effort. Even when examining the solid lines (which represent the total number of vulnerabilities discovered by change-level and release-level predictors, not just those in the matched set), change-based prediction performance slightly lags, or does not exceed, the file-level performance.

In summary:

- When effort is not considered, machine learning models could more readily identify vulnerable changes and releases than vulnerable files.
- When proportionally considering the effort required to inspect each change, release, or file, change-level and file-level predictors performed better than release-level predictors.
- However, when also compensating for the inherent disadvantage suffered by change-level and release-level predictors due to code churn, file-level predictors fared far better than change-level or release-level predictors.

For reference purposes, Tables 6.39 and 6.40 present a consolidated set of performance indicators for a variety of experimental setups (file-level, change-level, release-level), measurement periods (in the case of change-level and release-level predictors, either up to the base release or across the entire history), and feature types (metrics or atoms). We refer the reader to previous sections in this chapter for a more thorough exploration of each of these prediction aspects.

6.7 Correlations of vulnerability discovery

In this final section, we explore the concept of vulnerability *discoverability*, or if certain vulnerabilities in the dataset are discovered by the predictive models more easily than others. The questions that we explore here include:

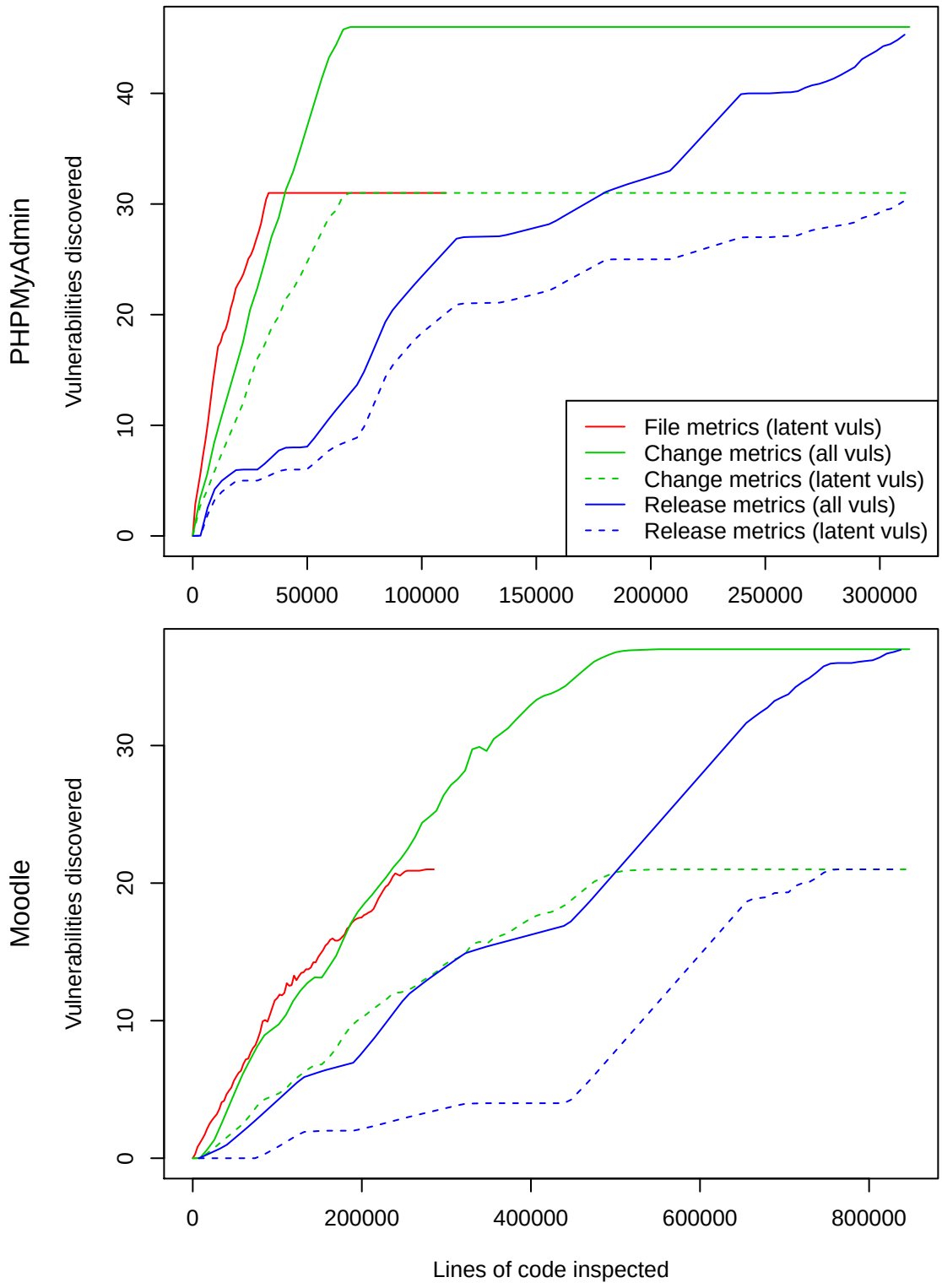


Figure 6.20: Absolute performance comparison of file, change, and release predictors by lines of code inspected when detecting matched sets of vulnerabilities

Table 6.39: Vulnerability discovery techniques for PHPMyAdmin using metric or atom predictors

History	Objects	Predictors	r_{20}	i_{20}	i_{40}	i_1	auec	$aucec_{50}$
At 3.1.0	Files	Atoms	0.910	3550	8848	110477	11315	0.795
At 3.1.0	Files	File	0.788	3433	7374	110477	12108	0.781
To 3.1.0	Change	Change	0.965	10564	24029	312923	30025	0.808
To 3.1.0	Release	Release+cur	0.248	51098	82168	312923	130685	0.358
Whole	Change	Change	0.888	12563	34465	456885	47289	0.793
Whole	Change	Change+rel	0.901	12613	30750	456885	45977	0.799
Whole	Release	Atoms	0.249	74238	116651	456885	159113	0.389
Whole	Release	Release+cur	0.349	66387	102253	456885	165147	0.405

Table 6.40: Vulnerability discovery techniques for Moodle using metric or atom predictors

History	Objects	Predictors	r_{20}	i_{20}	i_{40}	i_1	auec	$aucec_{50}$
At 2.0.0	Files	File	0.307	36395	74964	285198	109250	0.376
To 2.0.0	Change	Change	0.398	72533	164582	847692	214651	0.502
To 2.0.0	Release	Release+cur	0.126	195305	248149	847692	390475	0.261
Whole	Change	Change	0.428	71093	182003	961791	234553	0.519
Whole	Change	Change+rel	0.389	87086	194905	961791	236114	0.512
Whole	Release	Release+cur	0.292	158481	255193	961791	401121	0.316

1. Do file-level or change-level models discover vulnerabilities in some categories more easily than vulnerabilities in other categories?

2. Do file-level and change-level models tend to discover the same vulnerabilities, or different ones?

3. Is the speed at which a vulnerability was discovered naturally (the amount of time it persisted in the codebase before being fixed) correlated with the ease that it could have been discovered by the model?

4. When training the predictor multiple times (varying the random seed each time), does the predictor consistently find the same vulnerabilities each time?

We first examine the first question, the relationship between vulnerability category and discoverability. Recall that building a cost-effectiveness curve involves

Table 6.41: Median number of ROCs that discovered each vulnerability, by category

variable	Moodle		PHPMyAdmin	
	Change	Code	Change	Code
Code Injection			541	1302
CSRF	586		532	
Disclosure	483	548	549	1282
Lack of Input Validation	395	422	623	
Path Disclosure	490		549	1255
Priv. Bypass	494	416	582	1236
Priv. Elevation	544	557		
SQL Injection	466	482		
UNKNOWN	462	248		
XSS	447	207	590	1266
p	0.5787	0.2316	0.0769	0.6815

training models with various values of m, and then repeating the training process 10 times to compensate for the randomness inherent in the algorithms. This process yields a large set of models (sometimes over 1000), and we define the discoverability of a vulnerability to be the number of models in this set that found the vulnerability in the file or change. We then perform a Kruskal-Wallis rank sum test to test if the median discoverability measurements of vulnerabilities of each category are significantly different.

The results of these tests are shown in Table 6.41. At the .05 significance level, none of the vulnerability categories were correlated with discoverability – in other words, no category of vulnerabilities was significantly easier to discover than others.

Putting the concept of vulnerability categories aside, next we examine if different vulnerability discovery techniques tend to find vulnerabilities in the same order. The three vulnerability discovery techniques that are compared in this way are file-level predictors, change-level predictors, and natural discovery (the vendor's real-life announcement that the vulnerability was discovered and patched). For the predictors, order is determined by the discoverability measure defined earlier in this section – more discoverable vulnerabilities are presumed to have been found earlier,

Table 6.42: Correlations of vulnerability discoverability (number of ROCs that discovered each vulnerability) by prediction method

Application	Object 1	Object 2	*p*	*rho*
PHPMyAdmin	Change	Date	0.35814	-0.1197
PHPMyAdmin	Code	Change	0.23814	0.2183
PHPMyAdmin	Code	Date	0.20474	-0.2303
Moodle	Change	Date	0.76174	-0.0516
Moodle	Code	Change	0.00177	0.6519
Moodle	Code	Date	0.31302	0.2313

because the models would have directed code inspection to them first. For natural discoverability, discovery order is defined by the introduction-to-fix time. (Note that the chronological order of when vulnerabilities were discovered in real life is not used here.)

We assess if the order of vulnerability discovery across predictors is consistent by computing the Spearman correlation coefficient between each possible pair of techniques, and then testing if the correlation is significant. The results of these tests are presented in Table 6.42. In general, the order of vulnerability discovery between techniques was not significant at the .05 level. The exception was the correlation between file-level and change-level predictors for Moodle, which was highly significant at $p = .002$; however, the same was not observed for PHPMyAdmin. The natural speed of discovery was not correlated with discoverability by a predictive model in any case.

Finally, we evaluate the *consistency* of vulnerability discovery between iterations of the same predictor; in other words, the tendency of a model training algorithm to yield predictors which find the same vulnerabilities every time it is run. This investigation is motivated by the poor correlations which were observed when addressing the previous questions – after all, if the same vulnerabilities are not even found from run to run of the same experiment, then there is less of a prospect for any kind of correlations across experiments.

In order to measure this, we focus on the consistency of *easily discovered vulnerabilities*. In the previous correlation experiments, the discoverability of a vulnerability was measured by determining how many of the experiment's predictors found it. Note that in a single experiment, multiple predictors are trained in two, nested loops – first, a large set of predictors is trained to locate all the points in the cost-effectiveness curve, and afterwards, the first step is repeated 10 times to compensate for the randomness inherent in the training process.

When constructing any given cost-effectiveness curve (i.e. when performing 1 out of the 10 iterations), consider that, if some vulnerabilities actually are found more easily found than others, then some vulnerabilities will be covered by more of the curve's points than others. For any given curve, consider the top one-third most frequently discovered vulnerabilities. This is the set of *easily discovered vulnerabilities* for that curve. To see if this set remains *consistent* across iterations, we examine the extent that the set's membership changes from iteration to iteration (i.e. the 10 iterations of the experiment).

Let $a_1, a_2, \ldots a_n$ be indicator random variables for each vulnerability, modeling if the vulnerability was (1) or was not (0) in the set of easily discovered vulnerabilities during an iteration. Note that these random variables are not independent (because the size of the set is constant) and their distribution must be considered jointly. However, it still holds that $\mathbf{E}(\Sigma a_i) = \Sigma \mathbf{E}(a_i)$.

We now consider the probability that a given vulnerability is in *both* sets of easily discovered vulnerabilities across *two* iterations. This is the probability that two samples of the random variable are both 1, or $(\mathbf{E}(a_i))^2$. Following the same reasoning as before, the size of the set of vulnerabilities that were easily discovered in both iterations is $\Sigma(\mathbf{E}(a_i))^2$. If the set of easily discovered vulnerabilities is completely consistent, then $\Sigma(\mathbf{E}(a_i))^2 = \Sigma \mathbf{E}(a_i)$ because all a_i are 1 or 0. However, if the set is not completely consistent, then the size measurement will be lower, and

the magnitude of the difference provides an intuitive guide as to how "inconsistent" it is.

Finally, we consider the probability that a vulnerability appears in both sets of easily discovered vulnerabilities when performing two *different* experiments, or when attempting to find the same vulnerability with two different experimental setups. Let b_i be the probability that a vulnerability was easily discovered in the second experiment. The size of the set of vulnerabilities that were easily discovered in both experiments is then $\mathbf{E}(\Sigma a_i \cdot b_i) = \Sigma \mathbf{E}(a_i) \cdot \mathbf{E}(b_i)$, because a_i and b_i are drawn from different joint distributions and are hence independent. This measurement will get smaller as the set of easily discovered vulnerabilities becomes more inconsistent across experiments.

In Table 6.43, we use the above formulas to measure the average sizes of various sets of easily discovered vulnerabilities. The expected values ($\mathbf{E}$) of the random variables for each vulnerability are estimated by counting how often the vulnerability was easily discovered over all 10 iterations. Each row of this table represents some experimental setup which is considered alone, or considered in conjunction with a second repetition of the same setup, or a different setup. *Change* refers to a cost-effectiveness curve built with a set of change-level predictors. *Code* refers to the same built with a set of file-level predictors. Finally, *Random* refers to the size of a set of vulnerabilities chosen at random, or the expected size of the intersection of two such sets. Note that a row of the table that lists just one predictor acts as an upper bound on the size of the set (because a single iteration is completely consistent with itself), while a row listing two random predictors acts as a lower bound (because practically, if the set of easily discovered vulnerabilities is completely inconsistent, then we would expect to see it vary across iterations as if it were completely random). For all the other rows of the table, we can then judge the consistency of the predictor by gauging if the measurement is closer to the upper bound than the lower bound.

Table 6.43: Average number of vulnerabilities that random pairs of predictors have in common, by prediction method (prediction at recall=.33)

Predictor 1	Predictor 2	PHPMyAdmin	Moodle
		Vuls Common	Vuls Common
Change		10.00	7.00
Change	Change	4.32	3.52
Code		10.00	7.00
Code	Change	3.39	2.59
Code	Code	4.62	3.32
Random		10.00	7.00
Random	Random	3.23	2.33

The results indicate that there was little consistency between multiple iterations of the same kind of predictor, and even less consistency between iterations of different kinds of predictors. When looking at how many easily-discovered vulnerabilities two iterations of the same predictor had in common, the total was closer to what would have been expected from random selection (the no-consistency scenario) rather than what a completely consistent process would have yielded. For example, when training two sets of file-level predictors for PHPMyAdmin, comparing the sets with the 10 most easily found vulnerabilities from each predictor, on average, the two sets only have 4.62 elements in common.

There is at least some consistency to which vulnerabilities are easily found, as the average set size did notably exceed the lower bound in all cases. In addition, file-level and change-level prediction methods were more consistent with themselves than they were with each other, indicating that there are differences in which vulnerabilities are found by these two types of predictors. In general, though, the poor consistency may partially explain our failure earlier in this section to find any patterns in vulnerability discovery – it was inevitable, considering that repeating the same process twice yields two very different predictors. (However, although different vulnerabilities tend to be found during each prediction iteration, the overall *performance* of the iterations is more consistent – if this wasn't the case, then few

statistically significant results would have been encountered due to the high performance variance. In other words, consecutive iterations tended to perform similarly – they just found vulnerabilities in a different way.)

Chapter 7

Conclusions, threats to validity, and future work

In this thesis, we have studied, developed, and evaluated a number of tools, metrics, datasets, algorithms, and experimental setups for predicting the presence of security vulnerabilities in source code by using machine learning. Our major contributions and findings are:

A system for defining and constructing families of code and change metrics, incorporating the following:

- A method to construct harmonized sets of static code metrics and change metrics which represent the same quantities
- Representations of many common metrics within this system, allowing for common code metrics and corresponding change metrics to be constructed
- Formulas allowing for additional metrics to be derived by combining other metrics, in cases where a metric cannot directly be represented within this system
- Proofs that any family of metrics constructed in this system will contain a subset of change metrics that satisfy the distance axioms, supporting the theoretical validity of these change metrics.

A formalization of methods to transform software repositories and defect data into datasets for prediction, encompassing the following:

- An in-depth review of how past defect and vulnerability prediction studies constructed machine learning features and set up experiments

- A taxonomy of techniques for feature construction and experimental setup that were identified in this review

- Formal definitions for composable operators which implement selected techniques from the taxonomy, allowing us to set up experiments that use both static code metrics and change metrics for file-level, change-level, or release-level vulnerability prediction

Several machine learning meta-learning algorithms which enable effort-sensitive training and evaluation of vulnerability predictors under diverse experimental setups, incorporating:

- Definitions of performance indicators for evaluating experimental results, allowing for different kinds of vulnerability prediction techniques to be compared with each other

- A meta-learning algorithm which utilizes example weighting and class flipping during the model training process, in order to build effort-sensitive models that attempt to maximize the benefit of using the model while minimizing the cost

- A search algorithm that interpolates a cost-effectiveness curve for any type of machine learning model, spanning the entire range of effort budgets that a user of a model would likely choose, allowing for a fair and unbiased comparison of the effectiveness of two models

Tools and datasets to support predictive experiments, including:

- A dataset detailing the evolution of two PHP web application, including fine grained information on how a set of vulnerabilities was introduced, evolved over time, and was fixed in each application

- The public release of this dataset to encourage additional empirical research of this type

- Algorithms to efficiently compute code and change metrics from an AST-based representation of PHP source files, ensuring that the theoretical properties guaranteed previously in this work (such as the satisfaction of the distance axioms) were satisfied

- A metric development and computation tool which allows for metrics to be developed within a graphical user interface leveraging the functional representation of metrics defined previously in this work

- A visualization based on animated scatterplots that depicts the relationship between metrics in an evolving software system, along with their associations with the presence of vulnerabilities

A series of experiments in vulnerability prediction, covering machine learning and experimental setups for prediction, encompassing the following:

- An evaluation of vulnerability prediction models under three setups: file-level (predicting which files at a given point in time are more likely to be vulnerable), release-level (predicting which releases are more likely to introduce new vulnerabilities into the codebase than others) and change-level (predicting which file changes, or deltas, are more likely to introduce new vulnerabilities at a given release)

- A consistent treatment of each setup, evaluating each with the same metrics and similar feature transformation methods, to ensure that the effects of each factor affecting performance are examined in isolation

- A method to ensure that dissimilar experiments are run with equivalent, matched sets of vulnerabilities, eliminating confounding factors and ensuring

that the results are comparable with each other

- Estimates of the human effort required to find vulnerabilities in various scenarios when assisted by machine learning models, quantifying both the raw amount of effort expended (measured in lines of code) and the proportion of effort expended (compared to the maximum possible effort in that scenario)

- A method to mix multiple predictive techniques, such as file-level and change-level prediction, potentially allowing for the particular benefits of each technique to be realized simultaneously

- Simulations of vulnerability introduction and discovery over time, simulating how each prediction technique would affect the speed at which vulnerabilities are discovered, the number of vulnerabilities discovered within a certain time window, and the number of vulnerabilities that would be caught during quality assurance techniques before they are discovered "in the field"

- Statistical tests for patterns in vulnerability discoverability measurements, examining the factors that may make some vulnerabilities easier to discover (by machine learning models) than others

The following experimental conclusions:

- Effort-sensitive training algorithms improve the effectiveness of machine learning models under effort-sensitive evaluation criteria but worsen their effectiveness under non-effort-sensitive criteria.

- The random forest algorithm performed better than other machine learning algorithms in almost all cases, although in one case, the Naive Bayes algorithm performed substantially better than others.

- Our novel distance-based change metrics usually outperformed code-delta change metrics for prediction tasks, although the performance differences were small.

- It was best to use all available metrics, rather than a subset of the available metrics, when training predictive models. However, there was no clear benefit to augmenting basic metric family members with extra, derived combinations of metrics.

- For release-level prediction, it was best to aggregate all the changes made in a release with a Shannon entropy-based aggregator, rather than with other mathematical operations such as sum or mean.

- Augmenting simple code and change metrics with more complex features (such metadata on releases or recent change history) had no significant, consistent benefit across both studied applications.

- If the primary quality improvement goal is to maximize the total number of vulnerabilities found over time (regardless of how they are found), allocating one's effort toward a single round of file-level prediction is most effective.

- However, if the primary quality improvement goal is to maximize the number of vulnerabilities found by quality assurance teams (rather than by the general public), it is most effective to continuously apply predictive models, allocating one's effort toward change-level predictors or mixtures of file-level and change-level predictors.

- In most cases, our machine learning models more readily identified (discriminated) vulnerable changes than vulnerable, static files. However, because inspecting changes often meant inspecting the same functionality over and over again, due to high code churn, the absolute amount of effort required to utilize file-level predictors was less than for change-level predictors.

- Due to random factors inherent to the model training process, predictors tended to be inconsistent, in the sense that different models tended to find

different vulnerabilities on the same dataset.

7.1 Experimental setup contributions

In the previous section, we summarized the primary contributions of this study. In this section, we discuss in more detail this study's unique contributions to the *experimental setup* aspect of prediction, describing specific ways in which this aspect of our study extended the previous work.

Predicting vulnerability-introducing changes with machine learning models: Our study focuses on the prediction of *vulnerability-introducing changes*, or changes to software which induce a security vulnerability which was not previously present. From the standpoint of a software development practitioner, avoiding vulnerability-introducing changes is paramount. Only by detecting and remediating new vulnerabilities before the release of the software can the "window of opportunity" for attackers to exploit a system be eliminated. Unlike with some other, less critical, classes of defects, avoiding vulnerabilities is essential even if the users of the software are not experiencing any apparent issues, as the damage done by a single security breach may be irreparable.

As demonstrated in the table of related work, most defect prediction studies do not attempt to predict defect-introducing changes (or vulnerability-introducing changes); rather, they only attempt to predict the presence of defects at a single point in time. Out of the subset of experiments related to the characteristics of vulnerable code changes [23, 39, 90, 152], none attempt to predict which code changes are likely to be vulnerable for the purposes of optimizing a code inspection task.

Tracking vulnerabilities as they migrate across files over time: Although some related work focuses on predicting defect-introducing commits or patches [43, 44, 46, 69, 74, 97], none explores this question for security vulnerabilities. However, several factors distinguish vulnerabilities (or security defects) from defects in

general. Vulnerabilities are relatively rare and long-lived. Because vulnerabilities are relatively rare [179], problems related to imbalanced data in machine learning models are enhanced. As we demonstrate in Chapter 5, vulnerabilities tend to be long-lived, possibly because they do not interrupt the normal (non-malicious) usage of a system, and hence there is no pressure to fix them from users who are not specifically seeking out vulnerabilities. This makes the SZZ [144] algorithm for finding defect-introducing changes, used by nearly all defect-introducing change prediction studies [43, 44, 46, 69, 74], unsuitable for our purposes. The only alternative is to perform a full root-cause analysis to find the inception of the defect [97] or vulnerability. As we discuss in Chapter 5, the long life of vulnerabilities ensures that an SZZ-like approach will yield highly inaccurate estimates of when a vulnerability was introduced, and furthermore, the vendor's own data on vulnerability lifetimes tends to dramatically underestimate or overestimate the actual amount of the time that the vulnerability was present.

Even when performing file-level prediction, a proper root-cause analysis of a vulnerability is helpful because knowing when it was introduced and fixed ensures that the vulnerability was actually present in the file at the time that prediction is performed. Most defect prediction studies technically build predictive models using data on *defect-prone* files, rather than *defective* files, building the model such that a file is a "hit" if it had ever been defective during some range of time, even if the defect had been fixed by the time the model was built. Although this is usually inconsequential, because an isolated defect fix may only result in a very small change in the features of the file, this methodology could prevent the recognition of major code overhauls which increase the quality of a file and reduce the probability of defects.

Ensuring that defects are actually present in a file when building a model necessitates that the location of each defect as it migrates from file to file be tracked.

Because we have collected this data for our vulnerability dataset, all experiments in this study are done with this assurance.

Release-level prediction: Existing studies in machine learning for defect-introducing change prediction focus on the prediction of *defect-introducing commits* (or on commit-like modification requests [97]). In this study, we explore the alternative problem of predicting vulnerability-introducing changes on a *release* level, predicting which releases would introduce vulnerabilities (or, on a more granular level, which files in which releases will become vulnerable). Because release-level prediction has not been well-studied, the expected benefit of performing such prediction (measured by a reduction in inspected or tested code) had not yet been quantified before this study. Furthermore, many features which have been used for change-level commit prediction are not suitable for release-level change prediction.

The release level is a natural unit of granularity for prediction because of the quality-related activities that may occur before a release. Pre-release testing or code inspection could be enhanced by using release-level models, and users are not concerned with which commit introduced a vulnerability (as this information is not actionable to them); rather, it is only important when a new installation or upgrade to a new release results in a new vulnerability. In addition, aggregating changes into releases sidesteps methodological issues related to commit-level prediction, because commits in the midst of a release cycle may contain partially implemented functionality. This is a particular problem with the open-source projects that we study, which use simple source control systems that don't aggregate commits into larger changesets.

Prediction with features not defined at the file level: Nearly all defect prediction studies that use metrics act at a *file level* – more specifically, the metrics of a specific file are used to predict the occurrence of defects in the same file. The potential disadvantages of this approach are twofold. First, information on the code

in other parts of the program is not considered when predicting defects in a specific file (except to a limited extent with metrics such as coupling metrics), preventing the context in which a file is used from being considered when predicting defects. Second, specialized metrics, such as attack surface security metrics [83], cannot be incorporated into such a model at all. Even if effective defect prediction can be accomplished without using such specialized metrics, demonstrating utility of these metrics for defect or vulnerability prediction can serve as a powerful tool for validating the metric [89, 146] for use in other quality-related activities.

In Section 3.4, we introduce several feature construction operators that work on release/feature matrices. Any metric that is not defined at the file level can be incorporated into such a matrix and used in prediction experiments with various experimental setups.

Systematic, consistent comparison of defect predictor setups: In this study, we build and evaluate three distinct defect prediction setups – file-level prediction, release-level prediction, and change-level (release+file level) prediction. Because all three of these machine learning predictors are evaluated on the same datasets, we can perform a *systematic* comparison of the relative benefit that would be realized by using each predictor, which was previously unknown. This systematic comparison is enabled through the use of a lines-of-code effort measure to compare the performance of dissimilar experiments and an effort-aware model training process introduced in Chapter 4 which can optimize a variety of classes of machine learning models for effort-aware prediction.

It is also important that the performance comparison of multiple defect predictor setups be *consistent*, in that the same features are used for prediction in each context to ensure that the observed performance differences are due to the different prediction setups, rather than the different features. In our survey of related work earlier in this chapter, we observe that different features have traditionally been

used for each kind of prediction problem, with software metrics being more common for file-level prediction and code churn or commit properties being more common for commit-level prediction. This is due to the fact that traditional software metrics are typically defined on the level of static source code files – rather than changes – and many of these metrics have no immediately obvious analog for code changes.

7.2 Threats to validity and future work

In the following sections, we discuss several threats to validity (unexplored issues that should be explored before utilizing these predictors in real-world applications) and areas for future work (improvements to the techniques described in this paper, or additional areas of inquiry opened by the groundwork performed in this thesis).

7.2.1 Cross-project prediction

Note that in this work, all machine learning work was done with a *cross-validation* experimental design. In other words, each model was trained on some subset of the training data and evaluated on the rest of the data. For example, for change-based models, a subset of the program's vulnerable and non-vulnerable changes was used during training, and the rest of the changes were used during evaluation. This cross-validation methodology is a common and standard practice in the machine learning community, and it minimizes confounding factors while ensuring that the benefit associated with using the entire dataset is realized.

Threat to validity: All experiments were done with a cross-validation setup, which does not precisely replicate how predictive models would be used in a production environment.

In a production environment, one of the following two approaches would have

been used instead:

- **Next-release prediction**: When building a model, the organization uses an application's past development history and vulnerability data to build a model. This model is then used to predict vulnerabilities in the next release of the product. With next-release prediction, the model is highly calibrated to a particular product; however, the application must suffer from many past vulnerabilities before the model will become effective.

- **Cross-project prediction**: With this kind of prediction, a model is trained on data from one application and then used to predict vulnerabilities in another application. This means that the organization using the model need not go through the process of preparing a dataset and training a model, and the organization can begin to realize the benefits of prediction immediately, without suffering from numerous vulnerabilities first.

Cross-project prediction tends to perform poorly when compared to the kind of prediction we performed in this study (within-project prediction) due to differences between the application used to build the model and the target application being predicted. However, this problem has been well-studied, and a number of methods to alleviate it have been proposed. Some methods revolve around project clustering [68, 91], the practice of ensuring that the model is trained on a project that has similar characteristics to the project under prediction. Other methods [118, 159] focus on filtering, or selectively retaining the portions of the training data most similar to the project under prediction. Additional methods include ensemble voting [46] or normalization [169, 175] (ensuring that the attributes in the training project and the project under prediction have similar ranges).

We note that the normalization issue is similar to the effort-sensitivity issue which we explored throughout this work, in the sense that models need to be "tuned"

to be more or less sensitive in some sense. Normalization has had great success in narrowing the gap between within-project and cross-project prediction, in one case [175] closing the gap completely. Similarly, effort-sensitive evaluation made the gap between cross-project and within-project prediction seem much smaller than it appeared with other evaluation measures [126]. Note that the search algorithm we used during model training tunes the weights of the input data until it hits a desired effort budget. This tuning may help compensate for non-normalized data, especially since [126] effort-sensitive evaluation did so much to improve cross-project performance.

Future work: Determine if our new effort-sensitive training methodology improves cross-project prediction performance.

An alternative approach, proposed in our previous work [148], improves cross-project prediction performance with Confirmatory Factor Analysis. When using Confirmatory Factor Analysis as part of the prediction process, before predicting vulnerabilities, a structural model is constructed, which models how software metrics are related to more general, underlying characteristics of the software, such as complexity or cohesion. By fitting the same model on multiple projects, the differing relationships between these metrics and the characteristics that they measure can be accounted for and accommodated. Our preliminary results with this technique show that when performing file-level prediction, the Confirmatory Factor Analysis technique improves cross-project prediction performance in nearly all cases. Full details can be found in the cited work.

Next-release prediction is an additional alternative to cross-project prediction which could be facilitated by our vulnerability dataset. Note that our dataset contains the dates when each vulnerability was announced and patched. These dates could act as a reasonable approximation to the dates when the developers of each application became aware of the vulnerabilities because, unlike with commercial

software development organizations, it is difficult for open-source developers to hide the existence of vulnerabilities, and hence there is an incentive to patch and announce them quickly. If these vulnerability discovery dates were incorporated into the simulations performed as part of this work, then the real-world performance of next-release prediction could easily be estimated.

Future work: Repeat the experiments performed in this work under a next-release prediction setup, as opposed to a cross-validation setup.

7.2.2 Moving beyond PHP

In this work, we studied how the presence of vulnerabilities (or the introduction of vulnerabilities) in PHP web applications could be predicted with machine learning models. Although our evaluation was limited to PHP web applications, defect and vulnerability prediction is not limited to this domain by any means, and past studies have applied these techniques to a wide variety of languages and platforms.

Threat to validity: The conclusions made in this thesis may not generalize to domains beyond PHP web applications.

One domain where vulnerability predictive models may be especially useful is in Linux distribution packages. Most Linux distributions disseminate a wide variety of software (beyond the operating system itself) in the form of standardized packages. Normally, vulnerability patches are delivered in the form of package updates, providing a convenient single source for security updates.

In our previously published work [147], we demonstrated how this source of vulnerability packages could also serve as a vulnerability data source for research purposes. For future work, we propose that this channel of vulnerability data could be used to train *predictive models*, allowing end-users to measure the risk of installing or upgrading a Linux package by predicting if the installation or upgrade action will introduce new, unknown security vulnerabilities into a system. This prediction

would ultimately be done in the same way that we predict PHP web application vulnerabilities in this work, training models with the source code of packages instead of the source code of PHP files.

Future work: Apply the predictive techniques introduced by this thesis to security vulnerabilities in Linux packages.

7.2.3 Enabling reproducible research

As we demonstrated in Chapter 3, previous studies in defect prediction have used a variety of attributes, experimental setups, and feature construction techniques. The diversity of techniques makes it difficult for practitioners to compare the results of various studies and determine what techniques would best be applied in their unique situations. For example, a meta-analysis of past defect prediction studies [136] found that the explanatory factor with the greatest apparent impact on the experimental results was the identity of the research group performing the experiment. Clearly, the success or failure of an experiment should not depend on who is performing it; rather, the identity of the research group was serving as a proxy for other, more complex details of experiments that were difficult to model in a meta-analysis.

The ability to reproduce experimental results in software engineering is important, as reproduction makes it possible to compare several experimental practices or discover hidden confounding variables in another's work. To this end, the PROMISE [94] repository makes a large number of datasets available to the research community, and we have made our dataset available as part of our previously published work [165]. However, as observed by [87], publishing the raw data that one used in an experiment is not enough, due to the importance of also replicating the experimental techniques and models that were used. In response to this issue, we published a set of scripts as a companion to our previous work [165], which repli-

cate the experiments and results described in that study. However, developing and publishing shell scripts for replicating large, complex research results is cumbersome.

In Chapter 5, we described a graphical metric development tool which allows for a variety of metrics to be developed and visualized for an evolving codebase. We have prototyped a set of extensions for this tool enabling the *collaborative* development of metrics and execution of prediction experiments. With these extensions, authorized users can freely design metrics (i.e. machine learning features), select parameters to train machine learning models, perform prediction experiments, plot performance curves, and share the prediction results with other authorized users of the system. We intend to make this tool accessible to the research community, allowing for both the dissemination of experimental results and for those results to be improved upon by other members of the community.

Future work: Port the experiments performed in this thesis to a collaborative metric development and experimental tool, allowing for the larger research community to examine and improve upon our results.

7.2.4 Introducing security-specific features by improving release-level prediction

In this study, we used traditional software metrics to train vulnerability predictive models. Although it was possible to build effective predictive models with these metrics, we propose the investigation of *security-specific features* which are specifically designed to perform the task of vulnerability prediction.

Some defect prediction work in the past utilized sophisticated features that were intended to indicate the likelihood of defects from the very beginning. For example, in one study [78], code style warnings were used as a predictive features for the presence of defects. Because poor style could hypothetically increase the likelihood of coding errors, a predictive model trained on these warnings can statistically

associate poor coding practices to the defects that they cause.

Code style warnings are easy to incorporate into a predictive model, because they are isolated to a single location and can be counted in the same way that many other metrics are. Other security-specific characteristics such as dataflow-based metrics [30] could be utilized in the same way. However, several other characteristics of software quality which could potentially act as vulnerability indicators are somewhat more difficult to validate:

- **Attack surface**: The informal concept of limiting a system's attack surface (or limiting the array of opportunities for an attacker to access the system) has long been linked to security. Recent work in quantifying the attack surface of software systems [63, 83] defined several metrics that can be used to derive a numerical attack surface measurement from the software's architecture. Being able to associate such metrics with the presence of vulnerabilities could serve as additional validation that the attack surface metrics can provide a practical benefit for developers.

- **Secure design patterns**: Other work related to security design patterns [25, 31, 55] proposes that certain kinds of software architectures or architectural transformations could be beneficial to software security. These security design patterns could be used to craft additional metrics which gauge the extent that each secure (or insecure) pattern is exhibited in the codebase.

Future work: Incorporate security-specific concepts, such as attack surface and security design patterns, into vulnerability prediction models, in order to assess the validity of these concepts and improve performance of the models in practice.

As we originally observed in previous work [146], one hindrance to using concepts such as attack surface for vulnerability prediction is that the resulting metrics are not linked with vulnerabilities on the level of individual files; for example, the at-

tack surface in one module may enable the exploitation of vulnerabilities in another. This would prevent a file-level or change-level prediction setup from identifying the statistical associations required to build a model. However, the release-level prediction setup which we introduced in this work could incorporate such metrics.

Release-level prediction has several advantages over the prediction setups (file-level and change-level) that tie vulnerabilities to individual files. Aside from having the ability to utilize metrics not defined on individual files, this prediction setup is more closely aligned to the ways that certain stakeholders may benefit from using predictive models. For example, an organization may want to gauge the risk of upgrading a software package from one version to another. Release-level prediction could help estimate the likelihood that the upgrade will introduce a new security vulnerability into the application. Predictive models could be applied in a similar way to Linux packages as described previously. However, in this work, the cost/benefit tradeoffs of our release-level models was generally poor (although better than with no model at all).

Threat to validity: Performance of release-level models in this work was poor, leaving doubt that release-level prediction would be a useful technique in practice.

One potential problem with release-level prediction is that the granularity of the source code being measured is simply too large. For example, a model may identify that a major release is highly likely to introduce new vulnerabilities, but inspecting the major release is difficult because it contains many code changes. If releases could be divided into more granular units (such as development sprints or feature branches), similar benefits could be realized with better performance. This would also bring an increased ability to distinguish effective metrics from ineffective ones, providing a way to validate which attack surface metrics and design patterns should actually be considered when attempting to reduce the incidence of vulnera-

bilities in a new software design.

Future work: Experiment with decreasing the granularity of release-level prediction setups, in order to improve the performance of this style of prediction.

7.2.5 Discoverability and categories of vulnerabilities

In Chapter 6, we noted that the discoverability of vulnerabilities was generally unstable, in the sense that training successive predictive models resulted in different vulnerabilities being discovered every time. The variation in predictive models is ultimately caused by random decisions made when building these models, including the way that cross-validation folds are formed and the random decision trees constructed when building random forest models.

As we mentioned previously, this random variation is harmless from an experimental standpoint because, although different vulnerabilities were discovered by different models, there was consistency in the *number* of vulnerabilities that were discovered. Nevertheless, this instability merits further study to ensure that it will not adversely affect the utility of these models in practice.

Threat to validity: The decisions made by vulnerability predictors were unstable, and different vulnerabilities were discovered from run to run of the same experiment.

Note that the instability of the discovered vulnerabilities could actually be beneficial if it could be used to build an *ensemble* model. Ensemble models combine the results of multiple, conflicting predictors in order to produce a composite predictor that often functions better than any of its component parts. In our case, due to the observed non-linear relationship between inspection ratio and recall, a composition of several low-inspection-ratio predictors could perform better than a single high-inspection-ratio predictor. However, note that the algorithm used throughout most of this work (random forest) already has some characteristics of an ensemble

predictor if it is permitted to build a sufficient number of trees.

Future work: Determine if the observed instability in the predictors was inherent, or if it was an artifact of the cross-validation setup or model tuning parameters. If it was inherent, leverage this instability with ensembles of predictors to improve prediction performance.

Additional work is also warranted to better determine how vulnerability *categories* affect prediction. In this work, vulnerabilities of all categories were combined when building predictive models and testing them – in other words, no specific effort was made to discover vulnerabilities with models trained exclusively with vulnerabilities from their own category. Because our dataset was sampled without regard to vulnerability category, performing such an experiment is difficult, as the existing data is too dispersed across categories. However, more concentrated data collection focused on selected categories could determine if certain models (or certain metrics) are better at finding vulnerabilities in certain categories.

Future work: Determine if vulnerabilities of a given category are found more easily when using predictive models trained from vulnerabilities in the same category.

7.2.6 Vulnerability dataset size and model performance

In this study, the prediction performance of Moodle was generally worse than that of PHPMyAdmin. This could have occurred for multiple reasons – for example, it could have been because Moodle has many more files than PHPMyAdmin, or the characteristics of the application may make it genuinely more difficult to predict vulnerabilities in its codebase. However, the dataset also contained fewer Moodle vulnerabilities than PHPMyAdmin vulnerabilities, and we cannot exclude the possibility that this effect was simply due to having fewer vulnerabilities in its training set.

Threat to validity: The Moodle dataset in this work had fewer vulnerabilities than the PHPMyAdmin dataset, potentially hindering the performance of the experiments done with Moodle.

This raises the possibility of exploring how many vulnerabilities must be present in a training set before an effective predictive model can be constructed. In machine learning, more training data generally builds better predictors (at least until the point of overfitting). However, there is a real-world cost associated with accumulating lots of training data, as each collected vulnerability represents another security threat which potentially (or actually) made its way into a production system.

Future work: Perform simulations to determine how many examples of known vulnerabilities a predictive model must "see" before being able to effectively predict unknown vulnerabilities in the future.

7.3 Summary

In summary, the major contributions of this work revolved around defining multiple setups in which machine learning could be used for vulnerability prediction – file-level, change-level, and release-level – and defining metrics, features, algorithms, and performance indicators such that predictive models under any of these setups could be built in an equivalent and comparable way. Another major contribution was in the preparation and use of fine-grained vulnerability evolution data, which made several of the above prediction setups possible. Finally, we had some success in predicting vulnerabilities by using information defined at the granularity of entire programs, rather than individual files – through a release-level prediction setup – although the effectiveness of these predictors was worse than with the other two setups.

Bibliography

[1] Martin Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. Control-flow integrity principles, implementations, and applications. *ACM Trans. Inf. Syst. Secur.*, 13:4:1–4:40, November 2009.

[2] F Abukhodair, Bernhard E Riecke, H Erhan, and Chris D Shaw. Does interactivity improve exploratory data analysis of animated trend visualizations. In *Proceedings SPIE*, volume 8654, 2013.

[3] Luca Allodi and Fabio Massacci. Comparing vulnerability severity and exploits using case-control studies. *ACM Transactions on Information and System Security (TISSEC)*, 17(1):1, 2014.

[4] N. Antunes and M. Vieira. Benchmarking vulnerability detection tools for web services. In *Web Services (ICWS), 2010 IEEE International Conference on*, pages 203 –210, july 2010.

[5] Nuno Antunes and Marco Vieira. Comparing the effectiveness of penetration testing and static code analysis on the detection of sql injection vulnerabilities in web services. In *Proceedings of PRDC'09: 15th IEEE Pacific Rim International Symposium on Dependable Computing.*, pages 301–306. IEEE, 2009.

[6] Ivan Arce, Kathleen Clark-Fisher, Neil Daswani, Jim DelGrosso, Danny Dhillon, Christoph Kern, Tadayoshi Kohno, Carl Landwehr, Gary McGraw, Brook Schoenfield, Margo Seltzer, Diomidis Spinellis, Izar Tarandach, and Jacob West. Avoiding the top 10 software security design flaws. *IEEE Center for Secure Design*, 2014.

[7] Erik Arisholm, Lionel C Briand, and Magnus Fuglerud. Data mining techniques for building fault-proneness models in telecom java software. In *Software Reliability, 2007. ISSRE'07. The 18th IEEE International Symposium on*, pages 215–224. IEEE, 2007.

[8] Erik Arisholm, Lionel C Briand, and Eivind B Johannessen. A systematic and comprehensive investigation of methods to build and evaluate fault prediction models. *Journal of Systems and Software*, 83(1):2–17, 2010.

[9] Muhammad Asaduzzaman, Michael C Bullock, Chanchal K Roy, and Kevin A Schneider. Bug introducing changes: A case study with android. In *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, pages 116–119. IEEE Press, 2012.

[10] A. Austin and L. Williams. One technique is not enough: A comparison of vulnerability discovery techniques. In *Proceedings of Empirical Software Engineering and Measurement (ESEM), International Symposium on*, pages 97 –106, Sept 2011.

[11] Dejan Baca, Bengt Carlsson, and Lars Lundberg. Evaluating the cost reduction of static code analysis for software security. In *Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, PLAS '08, pages 79–88, New York, NY, USA, 2008. ACM.

[12] Alberto Bacchelli, Marco DAmbros, and Michele Lanza. Are popular classes more defect prone? In *Fundamental Approaches to Software Engineering*, pages 59–73. Springer, 2010.

[13] V.R. Basili, L.C. Briand, and W.L. Melo. A validation of object-oriented design metrics as quality indicators. *Software Engineering, IEEE Transactions on*, 22(10):751 –761, Oct 1996.

[14] T. Basso, P.C.S. Fernandes, M. Jino, and R. Moraes. Analysis of the effect of java software faults on security vulnerabilities and their detection by commercial web vulnerability scanner tool. In *Dependable Systems and Networks Workshops (DSN-W), 2010 International Conference on*, pages 150 –155, 28 2010-july 1 2010.

[15] Abraham Bernstein, Jayalath Ekanayake, and Martin Pinzger. Improving defect prediction using temporal features and non linear models. In *Ninth international workshop on Principles of software evolution: in conjunction with the 6th ESEC/FSE joint meeting*, pages 11–18. ACM, 2007.

[16] Dirk Beyer and Ahmed E Hassan. Animated visualization of software history using evolution storyboards. In *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, pages 199–210. IEEE, 2006.

[17] P. Biggar. *Design and Implementation of an Ahead-of-Time Compiler for PHP*. PhD thesis, Trinity College Dublin, 2009.

[18] Philip Bille. A survey on tree edit distance and related problems. *Theoretical computer science*, 337(1):217–239, 2005.

[19] Christian Bird, Nachiappan Nagappan, Harald Gall, Brendan Murphy, and Premkumar Devanbu. Putting it all together: Using socio-technical networks to predict failures. In *Software Reliability Engineering, 2009. ISSRE'09. 20th International Symposium on*, pages 109–119. IEEE, 2009.

[20] Christian Bird, Nachiappan Nagappan, Brendan Murphy, Harald Gall, and Premkumar Devanbu. Don't touch my code!: examining the effects of ownership on software quality. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 4–14. ACM, 2011.

[21] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, ICSE '76, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[22] Cathal Boogerd and Leon Moonen. Evaluating the relation between coding standard violations and faultswithin and across software versions. In *Mining Software Repositories, 2009. MSR'09. 6th IEEE International Working Conference on*, pages 41–50. IEEE, 2009.

[23] Amiangshu Bosu, Jeffrey C. Carver, Munawar Hafiz, Patrick Hilley, and Derek Janni. Identifying the characteristics of vulnerable code changes: An empirical study. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2014, pages 257–268, New York, NY, USA, 2014. ACM.

[24] L.C. Briand, S. Morasca, and V.R. Basili. Property-based software engineering measurement. *Software Engineering, IEEE Transactions on*, 22(1):68 –86, jan 1996.

[25] Koen Buyens, Bart De Win, and Wouter Joosen. Resolving least privilege violations in software architectures. In *Software Engineering for Secure Systems, 2009. SESS'09. ICSE Workshop on*, pages 9–16. IEEE, 2009.

[26] Cagatay Catal. Software fault prediction: A literature review and current trends. *Expert systems with applications*, 38(4):4626–4636, 2011.

[27] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[28] Cesar Couto, Christofer Silva, Marco Tulio Valente, Roberto Bigonha, and Nicolas Anquetil. Uncovering causal relationships between software metrics and bugs. In *Software Maintenance and Reengineering (CSMR), 2012 16th European Conference on*, pages 223–232. IEEE, 2012.

[29] Jacek Czerwonka, Rajiv Das, Nachiappan Nagappan, Alex Tarvo, and Alex Teterev. Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 357–366. IEEE, 2011.

[30] Dan DaCosta, Christopher Dahn, Spiros Mancoridis, and Vassilis Prevelakis. Characterizing the 'security vulnerability likelihood' of software functions. In *Proceedings of the IEEE ICSM 2003 International Conference on Software Maintenance*, pages 266–, 2003.

[31] Asish Kumar Dalai and Sanjay Kumar Jena. Evaluation of web application security risks and secure design patterns. In *Proceedings of the 2011 International Conference on Communication, Computing & Security*, ICCCS '11, pages 565–568, New York, NY, USA, 2011. ACM.

[32] Marco D'Ambros, Alberto Bacchelli, and Michele Lanza. On the impact of design flaws on software defects. In *Quality Software (QSIC), 2010 10th International Conference on*, pages 23–31. IEEE, 2010.

[33] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Proceedings of MSR: 7th IEEE Working Conference on Mining Software Repositories*, pages 31–41. IEEE, 2010.

[34] Edsko de Vries and John Gilbert. Design and implementation of a PHP compiler front-end. Technical report, Technical report, Trinity College Dublin, Ireland, 2007.

[35] Sun Ding, Hee Beng Kuan Tan, Lwin Khin Shar, and Bindu Madhavi Padmanabhuni. Towards a hybrid framework for detecting input manipulation vulnerabilities. In *Software Engineering Conference (APSEC, 2013 20th Asia-Pacific*, volume 1, pages 363–370. IEEE, 2013.

[36] Pedro Domingos. Metacost: A general method for making classifiers cost-sensitive. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '99, pages 155–164, New York, NY, USA, 1999. ACM.

[37] Maureen Doyle and James Walden. An empirical study of the evolution of php web application security. In *Security Measurements and Metrics (Metrisec), 2011 Third International Workshop on*, pages 11–20. IEEE, 2011.

[38] Marco DAmbros, Michele Lanza, and Romain Robbes. Evaluating defect prediction approaches: a benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[39] Sebastian G Elbaum and John C Munson. Software evolution and the code fault introduction process. *Empirical Software Engineering*, 4(3):241–262, 1999.

[40] Ural Erdemir, Umut Tekin, and Feza Buzluca. E-quality: A graph based object oriented software quality visualization tool. In *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on*, pages 1–8. IEEE, 2011.

[41] Jon Eyolfson, Lin Tan, and Patrick Lam. Do time of day and developer experience affect commit bugginess? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 153–162. ACM, 2011.

[42] Tom Fawcett. An introduction to roc analysis. *Pattern recognition letters*, 27(8):861–874, 2006.

[43] Javed Ferzund, Syed Nadeem Ahsan, and Franz Wotawa. Empirical evaluation of hunk metrics as bug predictors. In *Software Process and Product Measurement*, pages 242–254. Springer, 2009.

[44] Javed Ferzund, Syed Nadeem Ahsan, and Franz Wotawa. Software change classification using hunk metrics. In *Software Maintenance, 2009. ICSM 2009. IEEE International Conference on*, pages 471–474. IEEE, 2009.

[45] J. Fonseca, M. Vieira, and H. Madeira. Testing and comparing web vulnerability scanning tools for sql injection and xss attacks. In *Dependable Computing, 2007. PRDC 2007. 13th Pacific Rim International Symposium on*, pages 365 –372, dec. 2007.

[46] Takafumi Fukushima, Yasutaka Kamei, Shane McIntosh, Kazuhiro Yamashita, and Naoyasu Ubayashi. An empirical study of just-in-time defect prediction using cross-project models. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 172–181. ACM, 2014.

[47] Michael Gegick, Pete Rotella, and Laurie Williams. Toward non-security failures as a predictor of security faults and failures. In *Proceedings of the 1st International Symposium on Engineering Secure Software and Systems*, ESSoS '09, pages 135–149, Berlin, Heidelberg, 2009. Springer-Verlag.

[48] Michael Gegick, Laurie Williams, Jason Osborne, and Mladen Vouk. Prioritizing software security fortification through code-level metrics. In *Proceedings of QoP '08: 4th ACM workshop on Quality of protection*, pages 31–38, New York, NY, USA, 2008. ACM.

[49] Emanuel Giger, Marco D'Ambros, Martin Pinzger, and Harald C. Gall. Method-level bug prediction. In *Proceedings of the ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '12, pages 171–180, New York, NY, USA, 2012. ACM.

[50] Emanuel Giger, Martin Pinzger, and Harald Gall. Using the gini coefficient for bug prediction in eclipse. In *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th annual ERCIM Workshop on Software Evolution*, pages 51–55. ACM, 2011.

[51] Emanuel Giger, Martin Pinzger, and Harald C Gall. Comparing fine-grained source code changes and code churn for bug prediction. In *Proceedings of the 8th Working Conference on Mining Software Repositories*, pages 83–92. ACM, 2011.

[52] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[53] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on*, 26(7):653–661, 2000.

[54] Tibor Gyimothy, Rudolf Ferenc, and Istvan Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *Software Engineering, IEEE Transactions on*, 31(10):897–910, 2005.

[55] Munawar Hafiz, Paul Adamczyk, and Ralph Johnson. A catalog of security-oriented program transformations. *Submitted to ECOOP 2009*, 2009.

[56] Gregory A Hall and John C Munson. Software evolution: code delta and code churn. *Journal of Systems and Software*, 54(2):111–118, 2000.

[57] Mark Hall, Eibe Frank, Geoffrey Holmes, Bernhard Pfahringer, Peter Reutemann, and Ian Witten. The weka data mining software: An update. *SIGKDD Explorations*, 11(1), 2009.

[58] Tracy Hall, Sarah Beecham, David Bowes, David Gray, and Steve Counsell. A systematic literature review on fault prediction performance in software engineering. *Software Engineering, IEEE Transactions on*, 38(6):1276–1304, 2012.

[59] Ahmed E Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[60] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Reconstructing fine-grained versioning repositories with git for method-level bug prediction. *IWESEP 10*, pages 27–32, 2010.

[61] Hideaki Hata, Osamu Mizuno, and Tohru Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering*, pages 200–210. IEEE Press, 2012.

[62] Sallie Henry, Dennis Kafura, and Kathy Harris. On the relationships among three software metrics. In *ACM SIGMETRICS Performance Evaluation Review*, volume 10, pages 81–88. ACM, 1981.

[63] Thomas Heumann, Sven Türpe, and Jörg Keller. Quantifying the attack surface of a web application. In *ISSE/Sicherheit 2010: Information Security Solutions Europe - Sicherheit, Schutz und Zuverlässigkeit.*, volume P-170 of *Lecture Notes in Informatics (LNI)*, pages 305–316. Gesellschaft für Informatik (GI) e.V., Bonner Köllen Verlag, 2010.

[64] A. Hovsepyan, R. Scandariato, W. Joosen, and J. Walden. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th international workshop on Security measurements and metrics*, 2012.

[65] Yue Jiang, Bojan Cuki, Tim Menzies, and Nick Bartlow. Comparing design and code metrics for software quality prediction. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 11–18. ACM, 2008.

[66] Kanta Jiwnani and Marvin Zelkowitz. Maintaining software with a security perspective. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 194–203. IEEE, 2002.

[67] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: a static analysis tool for detecting web application vulnerabilities. In *Security and Privacy, 2006 IEEE Symposium on*, pages 6 pp. –263, may 2006.

[68] Marian Jureczko and Lech Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 9. ACM, 2010.

[69] Yasutaka Kamei, Emad Shihab, Bram Adams, Ahmed E Hassan, Audris Mockus, Anand Sinha, and Naoyasu Ubayashi. A large-scale empirical study of just-in-time quality assurance. *Software Engineering, IEEE Transactions on*, 39(6):757–773, 2013.

[70] K. Kendall. *A database of computer attacks for the evaluation of intrusion detection systems*. PhD thesis, Massachusetts Institute of Technology, 1999.

[71] Taghi M Khoshgoftaar and Kehan Gao. Count models for software quality estimation. *Reliability, IEEE Transactions on*, 56(2):212–222, 2007.

[72] Taghi M Khoshgoftaar, Kehan Gao, and Naeem Seliya. Attribute selection and imbalanced data: Problems in software defect prediction. In *ICTAI (1)*, pages 137–144, 2010.

[73] Taghi M Khoshgoftaar, Kehan Gao, and Robert M Szabo. An application of zero-inflated poisson regression for software fault prediction. In *Software Reliability Engineering, 2001. ISSRE 2001. Proceedings. 12th International Symposium on*, pages 66–73. IEEE, 2001.

[74] Sunghun Kim, E James Whitehead, and Yi Zhang. Classifying software changes: Clean or buggy? *Software Engineering, IEEE Transactions on*, 34(2):181–196, 2008.

[75] Sunghun Kim, Thomas Zimmermann, E James Whitehead Jr, and Andreas Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[76] B.A. Kitchenham, L.M. Pickard, and S.J. Linkman. An evaluation of some design metrics. *Software Engineering Journal*, 5(1):50 –58, jan 1990.

[77] Kelvin Ku, Thomas E. Hart, Marsha Chechik, and David Lie. A buffer overflow benchmark for software model checkers. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 389–392, New York, NY, USA, 2007. ACM.

[78] Luigi Lavazza, Sandro Morasca, Davide Taibi, and Davide Tosi. Predicting oss trustworthiness on the basis of elementary code assessment. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 36. ACM, 2010.

[79] Luigi Lavazza, Sandro Morasca, Davide Taibi, and Davide Tosi. On the definition of dynamic software measures. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 39–48. ACM, 2012.

[80] Stefan Lessmann, Bart Baesens, Christophe Mues, and Swantje Pietsch. Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *Software Engineering, IEEE Transactions on*, 34(4):485–496, 2008.

[81] Marek Leszak, Dewayne E Perry, and Dieter Stoll. Classification and evaluation of defects in a project retrospective. *Journal of Systems and Software*, 61(3):173–187, 2002.

[82] Virtual Machinery. Jhawk – the java metrics tool.

[83] P.K. Manadhata and J.M. Wing. An attack surface metric. *Software Engineering, IEEE Transactions on*, 37(3):371 –386, May-June 2011.

[84] Mika V Mäntylä, Kai Petersen, and Dietmar Pfahl. How many individuals to use in a qa task with fixed total effort? In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 311–314. ACM, 2012.

[85] Girish Maskeri, Deepthi Karnam, Sree Aurovindh Viswanathan, and Srinivas Padmanabhuni. Bug prediction metrics based decision support for preventive software maintenance. In *Software Engineering Conference (APSEC), 2012 19th Asia-Pacific*, volume 1, pages 260–269. IEEE, 2012.

[86] Thomas J. McCabe. A complexity measure. *Software Engineering, IEEE Transactions on*, (4):308–320, 1976.

[87] Thilo Mende. Replication of defect prediction studies: problems, pitfalls and recommendations. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, page 5. ACM, 2010.

[88] Thilo Mende and Rainer Koschke. Effort-aware defect prediction models. In *Software Maintenance and Reengineering (CSMR), 2010 14th European Conference on*, pages 107–116. IEEE, 2010.

[89] Andrew Meneely, Ben Smith, and Laurie Williams. Validating software metrics: A spectrum of philosophies. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 21(4):24, 2012.

[90] Andrew Meneely, Harshavardhan Srinivasan, Ayemi Musa, Alberto Rodríguez Tejeda, Matthew Mokary, and Brian Spates. 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*, 2013.

[91] Tim Menzies, Andrew Butcher, David Cok, Andrian Marcus, Lucas Layman, Forrest Shull, Burak Turhan, and Thomas Zimmermann. Local versus global lessons for defect prediction and effort estimation. *Software Engineering, IEEE Transactions on*, 39(6):822–834, 2013.

[92] Tim Menzies, Jeremy Greenwald, and Art Frank. Data mining static code attributes to learn defect predictors. *Software Engineering, IEEE Transactions on*, 33(1):2–13, 2007.

[93] Tim Menzies, Zach Milton, Burak Turhan, Bojan Cukic, Yue Jiang, and Ayşe Bener. Defect prediction from static code features: current results, limitations, new approaches. *Automated Software Engineering*, 17(4):375–407, 2010.

[94] Tim Menzies, Carter Pape, Corbin Steele, and Mitch Rees-Jones. The teraPROMISE repository. `http://openscience.us/repo/about/`, 2014.

[95] P. Meunier. Classes of vulnerabilities and attacks. *Wiley Handbook of Science and Technology for Homeland Security*, 2008.

[96] Osamu Mizuno and Hideaki Hata. An integrated approach to detect fault-prone modules using complexity and text feature metrics. In *Advances in Computer Science and Information Technology*, pages 457–468. Springer, 2010.

[97] Audris Mockus and David M Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[98] Sandro Morasca. Refining the axiomatic definition of internal software attributes. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, ESEM '08, pages 188–197, New York, NY, USA, 2008. ACM.

[99] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. Analysis of the reliability of a subset of change metrics for defect prediction. In *Proceedings of the Second ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 309–311. ACM, 2008.

[100] Raimund Moser, Witold Pedrycz, and Giancarlo Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 181–190. IEEE, 2008.

[101] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 284–292. IEEE, 2005.

[102] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, pages 452–461, 2006.

[103] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th international conference on Software engineering*, pages 521–530. ACM, 2008.

[104] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of CCS: the 14th ACM conference on Computer and communications security*, pages 529–540, New York, NY, USA, 2007. ACM.

[105] Viet Hung Nguyen and Fabio Massacci. The (un)reliability of NVD vulnerable versions data: An empirical experiment on Google Chrome vulnerabilities. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ASIA CCS '13, pages 493–498, New York, NY, USA, 2013. ACM.

[106] Viet Hung Nguyen and Le Minh Sang Tran. Predicting vulnerable software components with dependency graphs. In *International Workshop on Security Measurements and Metrics (MetriSec)*, 2010.

[107] Allen P Nikora and John C Munson. Developing fault predictors for evolving software systems. In *Software Metrics Symposium, 2003. Proceedings. Ninth International*, pages 338–350. IEEE, 2003.

[108] Allen P Nikora and John C Munson. An approach to the measurement of software evolution. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(1):65–91, 2005.

[109] Gary Nilson, Kent Wills, Jeffrey Stuckman, and James Purtilo. Bugbox: A vulnerability corpus for PHP web applications. In *Presented as part of the 6th Workshop on Cyber Security Experimentation and Test*. USENIX, 2013.

[110] NIST. National vulnerability database. `http://nvd.nist.gov/`, 2013.

[111] Mel Ó Cinnéide, Laurence Tratt, Mark Harman, Steve Counsell, and Iman Hemati Moghadam. Experimental assessment of software metrics using automated refactoring. In *Proceedings of the ACM-IEEE international symposium on Empirical software engineering and measurement*, pages 49–58. ACM, 2012.

[112] Offensive Security. Exploit db. http://www.exploit-db.com/.

[113] Thomas J Ostrand, Elaine J Weyuker, and Robert M Bell. Where the bugs are. In *ACM SIGSOFT Software Engineering Notes*, volume 29, pages 86–96. ACM, 2004.

[114] Osvdb: The open source vulnerability database. `http://www.osvdb.org/`, 2013.

[115] Andy Ozment and Stuart E. Schechter. Milk or wine: Does software security improve with age? In *Proceedings of the 15th Conference on USENIX Security Symposium - Volume 15*, Berkeley, CA, USA, 2006. USENIX Association.

[116] Kai Pan, Sunghun Kim, and E James Whitehead. Bug classification using program slicing metrics. In *Source Code Analysis and Manipulation, 2006. SCAM'06. Sixth IEEE International Workshop on*, pages 31–42. IEEE, 2006.

[117] Mateusz Pawlik and Nikolaus Augsten. Rted: A robust algorithm for the tree edit distance. *Proc. VLDB Endow.*, 5(4):334–345, dec 2011.

[118] Fayola Peters, Tim Menzies, and Andrian Marcus. Better cross company defect prediction. In *Mining Software Repositories (MSR), 2013 10th IEEE Working Conference on*, pages 409–418. IEEE, 2013.

[119] Maurizio Pighin and Anna Marzona. An empirical analysis of fault persistence through software releases. In *Empirical Software Engineering, 2003. ISESE 2003. Proceedings. 2003 International Symposium on*, pages 206–212. IEEE, 2003.

[120] G. Poels and G. Dedene. Distance-based software measurement: necessary and sufficient properties for software measures. *Information and Software Technology*, 42(1):35 – 46, 2000.

[121] Daryl Posnett, Abram Hindle, and Prem Devanbu. Got issues? do new features and code improvements affect defects? In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 211–215. IEEE, 2011.

[122] Łukasz Puławski. Software defect prediction based on source code metrics time series. In *Transactions on rough sets XIII*, pages 104–120. Springer, 2011.

[123] Danijel Radjenović, Marjan Heričko, Richard Torkar, and Aleš Živkovič. Software fault prediction metrics: A systematic literature review. *Information and Software Technology*, 55(8):1397–1418, 2013.

[124] Foyzur Rahman and Premkumar Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.

[125] Foyzur Rahman, Sameer Khatri, Earl T Barr, and Premkumar Devanbu. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 424–434. ACM, 2014.

[126] Foyzur Rahman, Daryl Posnett, and Premkumar Devanbu. Recalling the imprecision of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 61. ACM, 2012.

[127] Jacek Ratzinger, Martin Pinzger, and Harald Gall. Eq-mine: Predicting short-term defects for software evolution. In *Fundamental Approaches to Software Engineering*, pages 12–26. Springer, 2007.

[128] Daniel Rodriguez, Israel Herraiz, Rachel Harrison, Javier Dolado, and José C Riquelme. Preliminary comparison of techniques for dealing with imbalance in software defect prediction. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering*, page 43. ACM, 2014.

[129] R. Scandariato, J. Walden, A. Hovsepyan, and W. Joosen. Predicting vulnerable software components via text mining. *Software Engineering, IEEE Transactions on*, 40(10):993–1006, Oct 2014.

[130] R. Scandariato, J. Walden, and W. Joosen. Static analysis versus penetration testing: A controlled experiment. In *Proceedings of ISSRE: IEEE 24th International Symposium on Software Reliability Engineering*, pages 451–460, Nov 2013.

[131] Riccardo Scandariato and James Walden. Predicting vulnerable classes in an android application. In *Proceedings of the 4th international workshop on Security measurements and metrics*, 2012.

[132] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[133] Lwin Khin Shar and Hee Beng Kuan Tan. Mining input sanitization patterns for predicting sql injection and cross site scripting vulnerabilities. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 1293–1296. IEEE Press, 2012.

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

[135] Lwin Khin Shar, Hee Beng Kuan Tan, and Lionel C Briand. Mining SQL injection and cross site scripting vulnerabilities using hybrid program analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 642–651. IEEE Press, 2013.

[136] Martin Shepperd, David Bowes, and Tracy Hall. Researcher bias: The use of machine learning in software defect prediction. *Software Engineering, IEEE Transactions on*, 40(6):603–616, 2014.

[137] Emad Shihab, Ahmed E Hassan, Bram Adams, and Zhen Ming Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 62. ACM, 2012.

[138] Emad Shihab, Zhen Ming Jiang, Walid M Ibrahim, Bram Adams, and Ahmed E Hassan. Understanding the impact of code and process metrics on post-release defects: A case study on the eclipse project. In *Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, page 4. ACM, 2010.

[139] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *Software Engineering, IEEE Transactions on*, 37(6):772–787, 2011.

[140] Yonghee Shin and Laurie Williams. Is complexity really the enemy of software security? In *Proceedings of the 4th ACM workshop on Quality of protection*, QoP '08, pages 47–50, New York, NY, USA, 2008. ACM.

[141] Yonghee Shin and Laurie Williams. 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*, pages 1–7. ACM, 2011.

[142] Yonghee Shin and Laurie Williams. Can traditional fault prediction models be used for vulnerability prediction? *Empirical Software Engineering*, 18(1):25–59, 2013.

[143] Joseph F Shobe, Md Yasser Karim, Motahareh Bahrami Zanjani, and Huzefa Kagdi. On mapping releases to commits in open source systems. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 68–71. ACM, 2014.

[144] Jacek Sliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? *ACM SIGSOFT software engineering notes*, 30(4):1–5, 2005.

[145] Ben Smith and Laurie Williams. Using SQL hotspots in a prioritization heuristic for detecting all types of web application vulnerabilities. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 220–229. IEEE, 2011.

[146] J. Stuckman and J. Purtilo. Comparing and applying attack surface metrics. In *Proceedings of MetriSec: International Workshop on Security Measurements and Metrics*, 2012.

[147] Jeffrey Stuckman and James Purtilo. Mining security vulnerabilities from linux distribution metadata. In *Software Reliability Engineering Workshops (ISSREW), 2014 IEEE International Symposium on*, pages 323–328. IEEE, 2014.

[148] Jeffrey Stuckman, James Walden, and Riccardo Scandariato. The effect of dimensionality reduction on software vulnerability prediction models. *IEEE Transactions on Reliability (submitted)*, 2015.

[149] Zhendong Su and Gary Wassermann. The essence of command injection attacks in web applications. In *Conference record of POPL '06: The 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 372–382, New York, NY, USA, 2006. ACM.

[150] Kuo-Chung Tai. The tree-to-tree correction problem. *Journal of the ACM (JACM)*, 26(3):422–433, 1979.

[151] Alexander Tarvo. Using statistical models to predict software regressions. In *Software Reliability Engineering, 2008. ISSRE 2008. 19th International Symposium on*, pages 259–264. IEEE, 2008.

[152] Alexander Tarvo, Nachiappan Nagappan, and Thomas Zimmermann. Predicting risk of pre-release code changes with checkinmentor. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 128–137. IEEE, 2013.

[153] Josée Tassé. Using code change types in an analogy-based classifier for short-term defect prediction. In *Proceedings of the 9th International Conference on Predictive Models in Software Engineering*, page 5. ACM, 2013.

[154] Piotr Tomaszewski, Jim Håkansson, Håkan Grahn, and Lars Lundberg. Statistical models vs. expert estimation for fault prediction in modified code–an industrial case study. *Journal of Systems and Software*, 80(8):1227–1238, 2007.

[155] Piotr Tomaszewski, Lars Lundberg, and Håkan Grahn. The accuracy of early fault prediction in modified code. In *Proceedings of the Fifth Conference on Software Engineering Research and Practice in Sweden (SERPS)*, pages 57–63, 2005.

[156] Piotr Tomaszewski, Lars Lundberg, and Håkan Grahn. Improving fault detection in modified codea study from the telecommunication industry. *Journal of Computer Science and technology*, 22(3):397–409, 2007.

[157] Roberto Tonelli, Giulio Concas, Michele Marchesi, and Alessandro Murgia. An analysis of sna metrics on the java qualitas corpus. In *Proceedings of the 4th India Software Engineering Conference*, pages 205–213. ACM, 2011.

[158] Andrea Torsello and Edwin R Hancock. Matching and embedding through edit-union of trees. In *Computer VisionECCV 2002*, pages 822–836. Springer, 2002.

[159] Burak Turhan, Tim Menzies, Ayşe B Bener, and Justin Di Stefano. On the relative value of cross-company and within-company data for defect prediction. *Empirical Software Engineering*, 14(5):540–578, 2009.

[160] K. Vikram, Abhishek Prateek, and Benjamin Livshits. Ripley: automatically securing web 2.0 applications through replicated execution. In *Proceedings of the 16th ACM conference on Computer and communications security*, CCS '09, pages 173–186, New York, NY, USA, 2009. ACM.

[161] Tatiana von Landesberger, Sebastian Bremm, Peyman Rezaei, and Tobias Schreck. Visual analytics of time dependent 2d point clouds. In *Proceedings of the 2009 Computer Graphics International Conference*, pages 97–101. ACM, 2009.

[162] S. Wagner, J. Jürjens, C. Koller, and P. Trischberger. Comparing bug finding tools with reviews and tests. *Testing of Communicating Systems*, pages 316–316, 2005.

[163] J. Walden and M. Doyle. Savi: Static-analysis vulnerability indicator. *Security Privacy, IEEE*, 10(3):32 –39, may-june 2012.

[164] J. Walden, M. Doyle, R. Lenhof, and J. Murray. Java vs. php: Security implications of language choice for web applications. In *International Symposium on Engineering Secure Software and Systems (ESSoS)(February 2010)*, 2010.

[165] J. Walden, J. Stuckman, and R. Scandariato. Predicting vulnerable components: Software metrics vs text mining. In *Proceedings of ISSRE: IEEE International Symposium on Software Reliability Engineering*, pages 23–33, Nov 2014.

[166] James Walden, Maureen Doyle, Grant A. Welch, and Michael Whelan. Security of open source web applications. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 545–553, Washington, DC, USA, 2009. IEEE Computer Society.

[167] Shuo Wang and Xin Yao. Using class imbalance learning for software defect prediction. *Reliability, IEEE Transactions on*, 62(2):434–443, 2013.

[168] Gary Wassermann and Zhendong Su. Sound and precise analysis of web applications for injection vulnerabilities. In *ACM Sigplan Notices*, volume 42, pages 32–41. ACM, 2007.

[169] Shinya Watanabe, Haruhiko Kaiya, and Kenji Kaijiri. Adapting a fault prediction model to allow inter languagereuse. In *Proceedings of the 4th international workshop on Predictor models in software engineering*, pages 19–24. ACM, 2008.

[170] E.J. Weyuker. Evaluating software complexity measures. *Software Engineering, IEEE Transactions on*, 14(9):1357 –1365, sep 1988.

[171] Elaine J Weyuker, Thomas J Ostrand, and Robert M Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, 2008.

[172] Chris Williams. Anatomy of OpenSSL's Heartbleed: Just four bytes trigger horror bug. `http://www.theregister.co.uk/2014/04/09/heartbleed_explained/`, 2014.

[173] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the 15th conference on USENIX Security Symposium - Volume 15*, USENIX-SS'06, Berkeley, CA, USA, 2006. USENIX Association.

[174] B. Zadrozny, J. Langford, and N. Abe. Cost-sensitive learning by cost-proportionate example weighting. In *Data Mining, 2003. ICDM 2003. Third IEEE International Conference on*, pages 435–442, Nov 2003.

[175] Feng Zhang, Audris Mockus, Iman Keivanloo, and Ying Zou. Towards building a universal defect prediction model. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 182–191. ACM, 2014.

[176] Kaizhong Zhang and Dennis Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.

[177] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for Eclipse. In *Proceedings of PROMISE'07: ICSE Workshop on Predictor Models in Software Engineering*, page 9, May 2007.

[178] Thomas Zimmermann and Nachiappan Nagappan. Predicting defects using network analysis on dependency graphs. In *Proceedings of the 30th international conference on Software engineering*, pages 531–540. ACM, 2008.

[179] Thomas Zimmermann, Nachiappan Nagappan, and Laurie Williams. Searching for a needle in a haystack: Predicting security vulnerabilities for windows vista. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 421–428. IEEE, 2010.