A New Approach for Predicting Security Vulnerability Severity in Attack Prone Software Using Architecture and Repository Mined Change Metrics

By

Daniel D. Hein

B.S., Computer Engineering (Iowa State University, 1999)
M.S., Computer Engineering (University of Kansas, 2004)

Submitted to the Department of Electrical Engineering and Computer Science and
the Graduate Faculty of the University of Kansas
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy

Hossein Saiedian, Ph.D., Professor and Advisor

Arvin Agah, Ph.D., Professor

Perry Alexander, Ph.D., Professor

Committee Members:

Reza Barati, Ph.D., Assistant Professor, External Member, Chemical and Petroleum Engineering

Prasad Kulkarni, Ph.D., Associate Professor

Nancy Mead, Ph.D., External Member,
Carnegie Mellon University

Date defended: April 21, 2017

The Thesis Committee for **Daniel D. Hein** certifies
that this is the approved version of the following thesis:

A New Approach for Predicting Security Vulnerability Severity in Attack Prone Software Using Architecture and Repository Mined Change Metrics

Hossein Saiedian, Ph.D., Professor and Advisor

Date approved: April 21, 2017

Abstract

Billions of dollars are lost every year to successful cyber attacks that are fundamentally enabled by software vulnerabilities. Modern cyber attacks increasingly threaten individuals, organizations, and governments, causing service disruption, inconvenience, and costly incident response. Given that such attacks are primarily enabled by software vulnerabilities, this work examines the efficacy of using change metrics, along with architectural burst and maintainability metrics, to predict modules and files that might be analyzed or tested further to excise vulnerabilities prior to release.

The problem addressed by this research is the residual vulnerability problem, or vulnerabilities that evade detection and persist in released software. Many modern software projects are over a million lines of code, and composed of reused components of varying maturity. The sheer size of modern software, along with the reuse of existing open source modules, complicates the questions of where to look, and in what order to look, for residual vulnerabilities.

Traditional code complexity metrics, along with newer frequency based churn metrics (mined from software repository change history), are selected specifically for their relevance to the residual vulnerability problem. We compare the performance of these complexity and churn metrics

to architectural level change burst metrics, automatically mined from the `git` repositories of the Mozilla Firefox Web Browser, Apache HTTP Web Server, and the MySQL Database Server, for the purpose of predicting attack prone files and modules.

We offer new empirical data quantifying the relationship between our selected metrics and the severity of vulnerable files and modules, assessed using severity data compiled from the NIST National Vulnerability Database, and cross-referenced to our study subjects using unique identifiers defined by the Common Vulnerabilities and Exposures (CVE) vulnerability catalog. Specifically, we evaluate our metrics against the severity scores from CVE entries associated with known-vulnerable files and modules. We use the severity scores according to the Base Score Metric from the Common Vulnerability Scoring System (CVSS), corresponding to applicable CVE entries extracted from the NIST National Vulnerability Database, which we associate with vulnerable files and modules via automated and semi-automated techniques. Our results show that architectural level change burst metrics can perform well in situations where more traditional complexity metrics fail as reliable estimators of vulnerability severity. In particular, results from our experiments on Apache HTTP Web Server indicate that architectural level change burst metrics show high correlation with the severity of known vulnerable modules, and do so with information directly available from the version control repository change-set (i.e., commit) history.

Acknowledgements

You are a direct reflection of the expectations of your peer group.

— Anthony Robbins

I am fortunate to have an uplifting and supportive peer group. There are so many people whose support and help made this work possible. First and foremost of these is my Advisor, Committee Chair, and friend, Dr. Hossein Saiedian. Dr. Saiedian's support, patience, and encouragement helped me push forward through difficulties, always reminding me to keep my focus to complete this final work. Additionally, I credit our one-on-one conversations with driving creative insights and realizations that are sprinkled throughout this dissertation.

I am likewise grateful to my wife Candy who provided both emotional and household support. A large project like this can make a person doubt their abilities and whether they are really up to the task. Candy provided unwavering faith in my abilities, helping me to believe in myself and find the strength to persevere when times got tough. Candy also took on many of our household and family obligations so that I could have the the focused time I needed for this work. I owe her an extreme debt of gratitude for all that she endured and took on while I was otherwise immersed in this work.

There are many others in my network of close friends and extended

family that have offered encouragement, support, and help. Thanks to all of you. In particular, my dear friend Nicholas Kuzmich was *especially* helpful, greatly aiding me with his unparalleled efficiency and proficiency with MS-Excel which was invaluable for quickly visualizing large datasets generated and saving me precious time (thank you Nick). In addition to close friends and family, are several of my managers and co-workers at Garmin who deserve mention.

I would like to acknowledge my managers at Garmin for their encouragement and understanding as I took time off to complete this work. My current manager, Jay Dee Krull has been especially supportive and encouraging, especially considering I have recently transitioned into a cyber-security focused role within the year. In addition, my previous manager, Travis Marlatte, has been a constant advocate, always encouraging me and rooting me on to the finish line.

They probably do not realize it, but many of my co-workers at Garmin also played a part in making this work what it is. From lunch outings, to casual "coffee-pot conversations", I can think of many discussions where I gleaned information and tips that found application in this dissertation. A few notable mentions from my Garmin coworkers include Jason Anderson, Austin Morgan, and Daniel Wilson (a co-worker, who with me, experienced the heightened awareness in automotive cyber-security during my time in the Automotive OEM department). I would like to thank Jason for openly sharing his knowledge of Python data science tools, such as Anaconda Python. Likewise, I am grateful to Austin Morgan for his insights on Mozilla Firefox development history and conventions. Finally,

Daniel Wilson was always ready to offer computation resources, as well as engage in insightful discussions regarding my data.

Contents

List of Figures

List of Tables

Chapter 1

Security Vulnerabilities and Software

The economic externalities stemming from exploited security vulnerabilities in software are a multi-billion dollar problem [79]. Each year, world-wide economies, corporations, and individuals shoulder financial damage stemming from successful attacks on computers and networks–attacks ultimately enabled by software vulnerabilities. Such attacks lead to lost productivity, data disclosure, and identity theft. In response, several organizations have adopted secure software development practices, including practices such as code review, use of static analysis tools, and penetration testing to remove these attack enabling vulnerabilities.

The goal of secure software development practices is to eliminate vulnerabilities before they enter the field. However, exhaustive application of secure development practices for all code is often not feasible or cost effective. While it may seem reasonable for an organization to apply secure development practices to all newly written code, one must simultaneously acknowledge that not all code is new. Modern software systems are composed of multiple components, each possibly of varying maturity. Some of these components may have been developed before adopting secure development practices. Moreover, it is often prohibitively expensive to require

that all pre-existing modules be re-written or that secure development practices be retroactively applied.

The cost associated with re-engineering and re-testing pre-existing modules is often not considered worthwhile when pitted against newer development efforts competing for the same limited development resources. Development teams must therefore decide where to focus energies for extended security reviews, penetration testing, and emphasis to place on fixing defects identified by static analysis tools.

The defect search space in modern software systems is enormous–often several million lines of code. As such, the level of effort to unearth vulnerabilities and remove them under time-to-market pressure is at odds with the level of effort required by attackers to find and exploit vulnerabilities. The effort required by an attacker to violate software security defenses is linear, requiring an attacker to find a single weakness to exploit. As Bellovin [19] states, "whatever the defense, a single well-placed blow can shatter it." On the other hand, the effort required to *assure* that software is secure is exponential, requiring exhaustive and comprehensive knowledge of the software and all its possible interactions with its environment; this simply isn't tractable [19].

Identification of attack-prone components is an important step in preventing vulnerabilities from entering the field. With knowledge about which components are most vulnerable, managers can allocate more resources for extended security reviews, automated static analysis, and penetration testing. The added scrutiny and focused effort can help to identify and eliminate vulnerabilities prior to software release.

1.1 The Residual Vulnerability Problem

The problem addressed by this research is the residual vulnerability problem – the presence of attack enabling security vulnerabilities in released software. Residual vulnerabilities in modern software systems such as mobile phones, personal computers, Web servers, and various embedded devices (e.g. SCADA) allow attackers to compromise both the host systems and the networks to which these systems are connected. Secure software engineering practices seek to identify and remove these vulnerabilities prior to release, but the application and management of these practices may be nontrivial in practice. Project size, maturity, and development team culture all play a part in effectively applying secure engineering practices. As an example, consider the practice of using a static analysis tool to search a product's source for frequently abused function calls, memory (mis)management, and other problematic patterns.

A development team can typically keep pace with warnings generated by a static analysis tool for newly developed code. In contrast, running a static analysis tool on a large, pre-existing code base may produce such a high warning count that the task of addressing the tool warnings becomes its own development activity, with the term *backlog* often used to differentiate tool warnings on pre-existing code from warnings on newly authored code. A dedicated team, or a managed effort spanning several teams, may be required to "triage" (inspect and evaluate) the warnings in order to determine whether further action is needed. Time spent triaging false positives, or warnings that do not actually translate into actual runtime faults, is essentially wasted time. The development team, overwhelmed with the large number of warnings, may be unsure how to proceed with triaging warnings and subsequently prioritizing fixes for legitimate defects. Consequently, the development team may

be at a loss as to how use their time effectively.

This research addresses the residual vulnerability problem by identifying and prioritizing attack-prone code units to better guide review and penetration testing efforts. First, we seek to reduce the vulnerability search space in a given software product by correctly predicting modules and files containing exploitable vulnerabilities. Second, we further consider the practical question of how to rank prediction results. Toward this end, we evaluate the feasibility of using architectural graphs to prioritize further development activity induced by the predictions (e.g. triage, patching, and verification).

Code units vary in granularity, with modules or packages at the coarse end of the spectrum, files and classes in the mid range, down to methods and lines at the fine end. To facilitate brevity and generalize discussion across the code units, this work coins the term "vulnerability suspects" or simply "suspects" to refer to prediction results at these various levels of granularity. This becomes relevant when presenting ideas related to prioritization. Since the prioritization and ranking scheme is derived from architectural structure (i.e. graphs of modules), it follows that the ranking, $R(m)$ for a module m can be extended downward to the classes, files, methods, and lines contained within m.

1.2 Vulnerability Research Questions and Hypotheses

Many existing works in defect and vulnerability prediction attempt to model the influence of size and complexity on residual defects. Often, the size and complexity metrics are extracted directly from the source code. The premise underlying existing prediction works is that increased size and complexity increase the difficulty of comprehension and coverage. Difficulty in comprehension impacts the ability of

developers and reviewers to understand the more detailed workings of the code as well as how unintended side effects could be unknowingly introduced when code is added, deleted, or modified. Difficulty in coverage impacts the ability of testers to adequately test a program. The common theme explored in existing prediction works is that increased size and complexity lead to an increased number of residual defects in released software. By comparison, fewer studies examine higher level factors that could similarly increase defect introduction and reduce defect detection.

Higher level factors likely to influence residual defects include developer characteristics (such as experience and familiarity with changed code), the rate of code changes, and whether or not those changes are made to a system or component amenable to change. For example, unintended side effects are likely to result when modifying code of poor maintainability that uses several shared global variables (i.e. exhibits poor modularity and encapsulation). This work intends to extend existing change-based predictors, augmenting them with architectural metrics indicative of maintainability and modularity. Below, φ is a predicted vulnerability. A predicted vulnerability can come from various prediction models and may include defect warnings from a static analysis tool.

Research Question 1. *Can change metrics reflecting frequent, high volume, or recent changes be used to predict vulnerable code units φ?*

Research Question 2. *Can architectural metrics reflecting poor modularity or maintainability be used to predict vulnerable modules?*

Research Question 3. *Can architectural metrics reflecting modularity or maintainability be used to prioritize change-based predictions, φ, relative to one another?*

Recent work has shown promising performance for change-based fault and vulnerability prediction metrics. This work seeks to expand on existing churn and developer based predictors by evaluating whether or not additional architectural

metrics can be used to prioritize change-based vulnerability predictions. *We hypothesize that changes, already indicative of a vulnerable file, made to a module of poor maintainability or modularity will be more likely than its peers to positively correlate with the severity of associated residual vulnerabilities.*

Using the additional architectural context, one can also ask what components are more likely to process attacker crafted data. That is, one can investigate how data might funnel through a system given its front line functions [25], attack surface [51], and a component's relation to the same. The component relationship may be expressed as a call path distance or coupling. Figure 1.1 shows a labeled graph for a program with $m = 100$ functions, a FLF density of 2%, and a path length of $p = mk = 100(0.02) = 2$, after the reverse breadth first search algorithm detailed in [25]. The `read()` function in this case is the front line function. Components sitting at the outer edges of the system interfacing with network connections, databases, and input fields are more prone to attacker manipulation. *We hypothesize that methods closer to so called front line functions are more attack prone than other methods in the system.*

1.3 Significance

The significance of this research is in optimizing resource utilization. One of the most expensive and limited resources in modern software development is developer time. Additional review and testing of various software artifacts (e.g. architectural designs, drivers, libraries, modules, and application code) is only warranted if there is reason to believe those artifacts may be attack prone. Moreover, the supply of time and expertise from security-competent developers and testers is arguably more restricted due to the specialized knowledge required to unearth security vulnerabil-

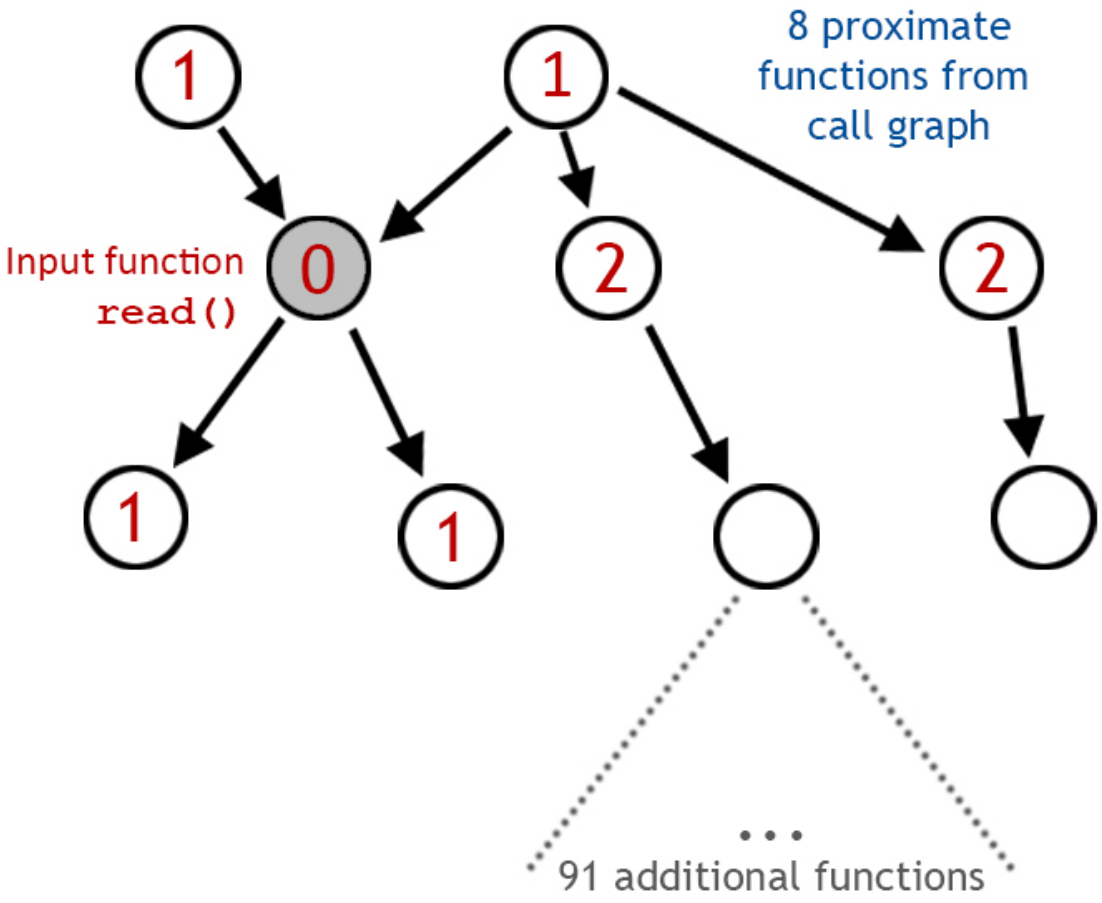

Figure 1.1: Front line function `read()`, with path length $p = 2$

ities. This research aims to intelligently focus these energies on those artifacts most likely to exhibit vulnerabilities. The analogy to resource utilization is much the same as the surgical laser analogy used justifying lightweight formal methods: "A surgical laser has less power and coverage than a traditional light bulb, but makes the most efficient use of the energy it uses, with often more impressive results." [42]

This research complements existing tools and predictive approaches which have already examined the use of code-based metrics as predictors. However, this work more closely examines evolutionary and architectural aspects of the software by:

1. Investigating how change metrics correlate with residual vulnerability severity, and
2. Examining how the software module structure either exposes or masks residual vulnerabilities.

This work approaches attack-prone prediction from a contextual and practical perspective. As such, when compared with other work in vulnerability prediction, the predictors in this work have the potential to reveal relationships between a product's security (measured by residual vulnerabilities) and how the product's code was developed. For example, the following questions characterize the types of questions we might reason about:

- Do rapid and scattered code changes possibly indicate increased development activity under time-to-market or competitive pressure?
- Do atypically frequent and repeated changes to a module or file possibly indicate a poor understanding of requirements, or a challenging technical issue, such that the module or file requires ongoing rework?
- Do the environmental conditions implied by the above factors also correspond to a large number of security failures?

- Do security failures seem either more severe or more prevalent for components of poor modularity?
- Do security failures seem either more severe or more prevalent for components of poor maintainability?

Additionally, we relate prediction models to the attack surface of a system along its boundaries with the surrounding environment after [51]. Attacker-formed data will enter the system at these boundaries. This is the practical aspect missing from the bulk of defect and vulnerability prediction research, save that as more recently acknowledged by Younis, Malaiya, and Ray [92] [94]. Consider a legitimate vulnerability identified in the system. With the exception of the aforementioned researchers, vulnerability prediction literature is largely devoid of any discussion of the *likelihood of its execution*, or the *anticipated severity* of such a vulnerability. This work investigates if architectural information can fill the void with respect to severity. Likelihood is examined, albeit indirectly in that we draw on similar past research and concepts as described in the following paragraphs.

Likelihood of execution is assessed using graph metrics characterizing a component's connection to system entry points [51], such as front line functions [25], exit points, channels, and untrusted data. The intuition is that components exhibiting shorter distance or higher coupling (inter-component cohesion) to elements at the boundary between the system and its environment, after [51], are more likely to process attacker-crafted data.

Severity may be further assessed using relative component entropy, calculated utilizing a data flow connection graph after [12]. The greater the relative entropy is for a component, so also goes the potential for damage when executing a contained vulnerability, and conversely, potential benefit resulting from vulnerability removal.

The addition of architectural considerations, and metrics reflecting the same,

better enable the results from vulnerability prediction models to be used as another touch point within secure software engineering life cycle models, as another static analysis tool for security improvement. The addition of architectural considerations provide practical benefits as they mirror ideas from operational security such as risk analysis, and so called value-of-protection calculations, that require likelihood and severity as input [70].

The majority of developers and testers are trained to verify conformance to functional requirements, statements about specific tasks or functions the software is to provide. Security assurance, on the other hand, probes for and tests the negative – that the software will not have unintended side affects when pushed outside its specifications. To remain secure, the software must not perform unintended actions even when used in unanticipated ways by an intelligent attacker. As a consequence, developers and testers specializing in security are trained to think like attackers and to look for specific classes of defects across a wide range of technologies. These specialized resource constraints are at odds with the amount of code requiring coverage. The goal of this research is to help focus these limited resources on those artifacts most likely to contain security vulnerabilities.

1.4 Contributions

The key contributions of this research are as follows:

- Aid security improvement by reducing the search space for residual vulnerabilities,
- Evaluate the feasibility of using information theoretic architectural metrics to further prune and prioritize predictions (i.e. vulnerability indicators),

- Contribute new knowledge to the field related to the relationship among security, complexity, and architectural quality attributes (modularity and maintainability),
- Provide open source tools for measuring information theoretic architectural metrics describing modularity and maintainability, and
- Add change burst empirics to the growing body of literature on defect and vulnerability prediction in open source projects.

The overall goal of this research is to improve security of consumer software products by enabling organizations to more effectively find and remove vulnerabilities prior to release. The goal of this research is to make vulnerability identification and removal more tractable and cost effective by reducing the number of components suggested for extended security review and penetration testing. This work speculates that organizations developing consumer software will be more motivated to expend additional effort on a more feasible and narrowly focused objective. This is in contrast to taking no action because either (1) the probability of finding real vulnerabilities is near infinitesimal (i.e. wasting time searching for the proverbial needle in a haystack), or (2) because tool-predicted vulnerabilities are so numerous that the development team is simply overwhelmed. On the latter point, the sheer number of defects is an issue because false positive rates of modern tools necessitate that development teams manually inspect, or “triage", each identified issue. On both points, triage efforts typically lose out to other development activities such as fixing observed bugs or developing new features.

The new and novel part of this work is the addition of suspect prioritization enabled by architectural metrics. The re-application of architectural metrics (characterizing modularization and maintainability) provide the foundation needed for prioritizing predicted components. Such relative prioritization is largely absent from existing defect and vulnerability prediction literature. Nevertheless, establishing

priority is important because it recognizes that not all vulnerabilities are created equal. Paraphrasing a common saying, “Nothing is top priority when everything is top priority".

Priority enables an organization to set customized goals, such as “fix all severe, high-impact vulnerabilities”, that better align with near and long term business strategies. In effect, the addition of priority better recognizes the socioeconomic realities and human factors characterizing the environment in which software is developed. For example, consider how a start-up may cease to exist if they cannot release a working product before their seed money expires. In general, competitive time-to-market pressures, along with modern companies’ need to rapidly innovate, imply that companies will not release perfect software. It is common practice to release a software product with known issues, as long as those issues are minor. This research’s addition of priority would enable such priority-based go/no-go decisions.

An additional benefit realized by developing a relative ranking among modules on the basis of their potential vulnerability severity, is that this ranking is complementary for use with other defect identification and removal techniques. As mentioned in Section 1.1, a relative ranking at the module level could be extended downward to contained functions and lines. By extension, the ranking could also be extended to tools that find defects at the line level, such as existing static analysis tools. A relative module ranking might be used to prioritize line level alerts output from a commercial static analysis tool (e.g., identifying the top n “actionable alerts”).

Although techniques have been published for calculating various information theoretic metrics from the structural and development views of a software architecture [77], there is currently no publicly available tool support for doing the same. Tool support is important for practical technique application as well as comparing results via repeatable and verifiable experiments [29]. Availability of a publicly ob-

tainable tool reduces the barrier to entry for researchers interested in evaluating the sensitivity and discriminatory power of architectural metrics. Since this work must have access to such a tool, we intend on building this tool and subsequently making it available to others.

1.5 Research Scope

The scope and context of this work is important in understanding intended contribution. This work assumes a commercial development context characterized by time-to-market pressure, code reuse, and limited development resources (e.g. man-days). The release of many software products occurs inside this type of environment–a competitive landscape where vendors must hit target dates as part of a larger business or marketing strategy. Time pressures and limited mandays for developing new sale-generating features leave little time for performing extended security reviews of reused code, especially when the search space for possible vulnerabilities is impossibly large. The development context outlined above is important because it drives the need for prioritization and effective resource utilization.

As residual vulnerability detection is a pre-requisite for removal, this work focuses on detection. Detection methods can be broadly classified into static and dynamic (e.g., test based) techniques. Prediction can be of various forms as well, but this work focuses on static prediction techniques which do not require system execution. Static approaches are particularly attractive for vulnerability removal since future attacks are not known a priori by those developing the software. Static approaches can also be applied earlier in the SDL, since it is not necessary to have a running system.

Given the activity of the research community on defect and vulnerability pre-

diction derived from change (AKA process) metrics, we focus less on re-applying those techniques to our study subjects. Instead, our focus is on evaluating the utility of architectural modularity and maintainability metrics for vulnerability prediction. Therefore, we utilize the most promising change based metrics (e.g. churn and change bursts) from related research and additionally investigate architectural metrics.

Vulnerability prediction is fundamentally a classification exercise. Software objects are classified as *vulnerable* or *neutral*. Such classification takes place on software objects of varying granularity with results at the module, file, function, or line level. Although some classifiers work at the level of individual lines, the present study considers that existing commercial static analysis tools already work at the line level to generate alerts for known vulnerability patterns. Additionally, existing works [81] have already explored structural dependency graphs from a function level. Therefore, the focus of the present study is on source code at the file level and above. While tools we use, such as architectural recovery tools, may output function level invocation relations, we carry out additional processing to group such relations at the file and module level.

1.6 Organization

The rest of this work is organized as follows:

Chapter 2. A Survey of Secure and Empirical Software Engineering discusses the background in from both Secure Software Engineering and Empirical Software Engineering in which this work is couched. The chapter covers relevant terms and concepts, as well as related work in vulnerability detection.

Chapter 3. Vulnerability Prediction Modeling and Evaluation provides a more detailed view of the prediction model building process. We also provide additional detail on metrics extracted, and formulas used for data evaluation.

Chapter 4. Detailed Repository Mining Approach outlines our research approach and data mining methodology. We describe software studied (i.e., our case-study subjects) and rationale used to select specific version ranges for the same.

Chapter 5. Experimental Results, and Analysis of Vulnerability Predictions describes experimental results and analysis for each software case study.

Chapter 6. Contributions and Future Work concludes the work with parting thoughts and ideas for future work that may build upon this work.

Chapter 2

A Survey of Secure and Empirical Software Engineering

This work bridges the fields of secure software engineering (SSE) and empirical software engineering (ESE). Secure software engineering is primarily concerned with methods and techniques to both prevent the introduction of, as well as detect and remove, vulnerabilities prior to release [38]. Empirical software engineering seeks to understand software quality through experimentation, data collection, and analysis [87], [43].

Within ESE, mining software repositories (MSR) has emerged as its own area of research within the last decade [36]. MSR uses artifacts from a software project for knowledge discovery (e.g. frequent item set and association rule mining) [90] as well as to support or refute investigative questions (e.g. do last minute changes introduce vulnerabilities?). Commonly mined artifacts include the software version control system, bug reports, and mailing lists of software projects [89].

The following sections highlight various concepts and terms from SSE, ESE, and MSR relevant to this research. After a brief summary of the secure software landscape and component based systems, related terminology is discussed. Various terms such as defect and vulnerability are discussed in the context of this work.

Finally, a review of related ESE and MSR based prediction work is presented.

2.1 Secure Software Landscape

Implementation level defects (e.g. defects in code) account for approximately half of all exploited security vulnerabilities [55] and therefore represent an important class of defects to address for security improvement. Moreover, implementation level security defects are amenable to static analysis since many of these defects have surfaced repeatedly over the years, implying the existence of detectable bug patterns. Such implementation level defects include the notorious buffer overflow and it's more general class of input-validation-related defects [84]. In fact, many such defects are presently detectable by static analysis tools.

Common techniques for determining if one software product is "more secure" than another involve comparing the number of security advisories logged against each version in a post-release fashion, with vulnerabilities identified *after* release. For example, although looking at completely different things, both Sachitano et al. [75] and Alhazmi et al. [7] use this post-release approach for source data and comparison. Furthermore, this post-release advisory counting technique only works on products from the same product line or that perform identical functions. In the case of Sachitano et al. [75], noted previously, the security attributes of two mail programs were being compared. In the case of Alhazmi et al. [7], a model for vulnerability discovery rate in subsequent operating system versions was being validated.

2.1.1 Component Based Systems

Large software systems such as Web servers and mobile phones are component based systems. The reuse of existing components is common in industry. Mod-

ern software systems are synthesized from pre-existing components, often being built on an application platform.

By application platforms, we mean a system of software components working together in order to provide (often tailored and domain specific) capabilities that form the foundation of modern systems. As such, application platforms are more than the operating system itself. Linux proper is actually the Linux kernel, but the term "Linux" is often meant as the application platform including libc, execution shells, and command line utilities. Colloquially, various application platforms are informally referred to by their names such as Linux, Windows, Android, and iOS. These platforms make extensive reuse of pre-existing software modules. Android, for example, makes use of several large components, including libraries to render web pages (WebKit), coordinate media playback (Media Framework), render fonts (FreeType), as well as provide the basic services needed to load and execute programs (Android Runtime). Figure 2.1, a high level overview provided by Google, shows the various layers, libraries, and frameworks within the Android application platform.

The use of pre-existing components in modern application platforms is analogous to viewing icebergs–the integrator cannot see 90% of what lurks below the surface. The many possible variations and configurations of such modules make exhaustive dynamic testing nearly unfeasible. Additionally, products aggregating platform components cannot re-inspect the source every time a new version of a new module or library is released; as stated previously, developer time is often focused on implementing new features in new code. Hence, static prediction models seem well suited to the task of narrowing the vulnerability search space of such systems.

Predictors that operate on source code and repository information are relevant to any arrangement where the source code and its repository are openly accessible.

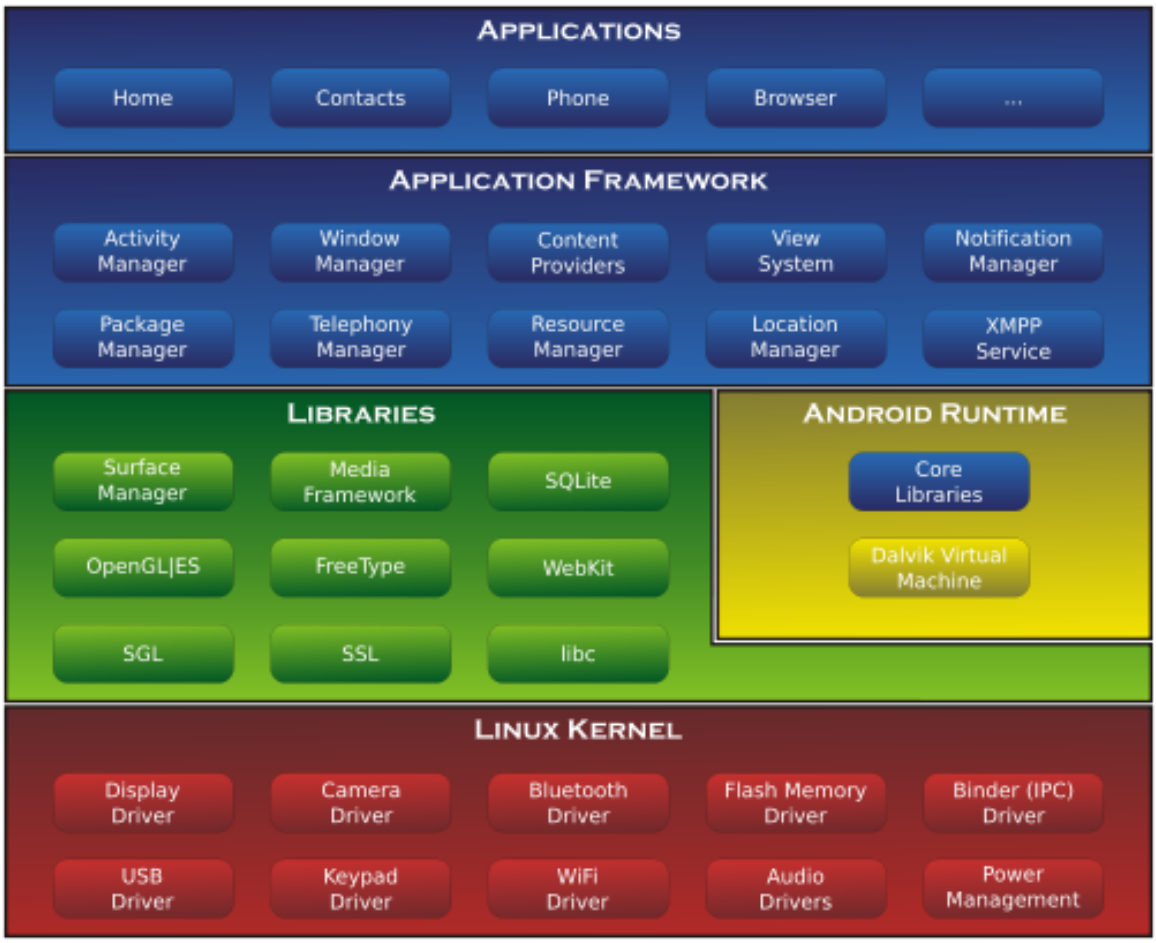

Figure 2.1: Android architecture

This arrangement manifests in vertically integrated technology companies as well as open source projects. Moreover, predictors driven by architectural relations may be used even when source code is not available.

Architectural relationships, such as function call invocations and variable references, can be recovered from linked objects and libraries. Metrics based on link level relationships can therefore be used when integrating closed source commercial off the shelf (COTS) components. Although link level relationships do require the software to build and link, many static techniques also impose a similar build requirement. The build requirement is needed so that static tools can obtain the correct environmental options and compiler configuration switches used for building the target executable [20].

2.1.2 Role of Static Analysis Tools

Automated static analysis tools are capable of scanning millions of lines of code, searching for patterns indicative of programmer mistakes, inefficient code, and error-prone language features (e.g. such as C's `'str'` functions). Such tool capabilities have proven effective at controlling costs while improving software quality in general and security in particular [17].

Reviewing code from a security perspective requires a level of expertise in vulnerability assessment and attacker-minded thinking that few developers possess [85]. Such expert knowledge, at least with respect to recurring vulnerability patterns, can be programmed into static analysis tools to help fill the void in software security expertise. Studies such as [40], [17], and [95] demonstrate the promise of static analysis tools for detecting many common and high frequency security-related implementation level defects. In [88], Jeannette Wing states that "We have the technical

solutions in hand to detect or prevent these attacks; so it is a matter of deploying them in an effective, scalable, and practical way.' Practical deployment of automated tools entails understanding (1) how they complement more traditional defect detection and removal techniques, and (2) how to effectively identify and prioritize actionable alerts.

Regarding the first point (1), static analysis tools complement other more traditional techniques such as structured code reviews and manual code inspections. Wagner et al. [86] showed that manual code reviews catch different types of vulnerabilities than those caught by automated tools. In addition, Gegick's work [28] provides evidence that non-security related defect warnings from static analysis tools can be used to identify attack-prone components.

Static analysis tools further complement manual review efforts by overcoming human time and stamina limitations. A human can only visually inspect a fraction of a project's code in the time it takes an automated tool to check a project's entire code-base. Additionally, since multiple code paths are checked simultaneously, corner cases harboring vulnerabilities are more likely to be uncovered by static analysis tools than conventional (dynamic) testing techniques [27]. A tool forges ahead, automatically executing each of its checks against every file; a human performing similar tedious actions is more likely to tire, increasing the chances for error.

Regarding the second point (2), the question around how to address tool generated warnings is not always so straightforward, especially for component based systems. Adding static analysis to an existing software project can unearth several thousand defects. Administrative questions about code and defect ownership, team loading, and fix assignment instantly surface. A natural extension of these issues are questions over which defects developers should fix first.

Although many tools bin detected defects into broad categories such as high,

medium, and low, these default categories may not be adequate in practice. Heckman [37] notes the inadequacy of default prioritization schemes offered by analysis tools "out of the box". A fundamental limitation of the default defect prioritization schemes offered by shrink-wrapped tools is lack of higher level insight into system structure and context. The default impact classifications may not align with the domain or operating environment of the software.

The author knows of at least one static analysis tool that classifies program hangs and data race conditions as a medium priority defects. The classification as a medium priority defect might be fine for a desktop application that can be reset or restarted with relative ease. However, resetting or restarting an application is not feasible (or is at the very least a last resort) for embedded systems. In the context of medical devices or other critical embedded systems, defects such as program hangs and data race conditions manifest as serious operational field failures. For example, the author knows of an automotive system where the driver must turn off the vehicle and exit the vehicle for two minutes before the system would shut down completely. Such a sequence is commonly referred to as a key cycle. In such a context, a program hang would render the automotive system unusable until the driver performed a key cycle.

2.1.3 Metric Based Prediction Models

Prediction models within ESE typically look at various attributes or properties of a software file (or its history) as opposed to scanning files line by line for a specific problematic pattern; in contrast, commercial static analysis tools scan files line by line for problematic functions, data flow sequences, and control flow sequences. The various attributes examined by prediction models are often colloquially referred to

as *predictors* or *indicators*, but these attributes, or their derivatives, either form the explanatory variables in statistical regression models, or form the inputs for (typically supervised) machine learners. Such attributes are also called features (or feature vectors in multivariate schemes). The said features typically utilize one or more code or change measures directly, or are themselves metrics derived from the same.

In contrast with commercial static analysis tools, which tend to focus on particular coding practices, or use of problematic functions, ESE defect prediction models typically approach model building from the standpoint of evaluating metrics with a bearing on (or that otherwise describe) human factors. For example, as stated in Section 1.2, MCC and KLOC can be viewed as representations of cognitive work factor, quantifying the degree to which code complexity and size exceeds the psychological capacity (e.g., short term memory) of the average developer. In other words, the MCC and KLOC *code metrics* measure attributes of code that have a bearing on human information processing and retention capabilities.

Code, Architecture, and Change Metrics In related ESE literature, *code metrics* typically refer to file or line level complexity metrics such as McCabe's Cyclomatic Complexity (MCC), or thousand lines of code (KLOC). Our work, and a few others, make reference to *architectural metrics* that characterize various properties of software dependency, invocation, and data flow graphs. Such architectural metrics might include, fan-in, fan-out, and various modularity and maintainability metrics noted in Section 2.2.4. In contrast, *change metrics* typically refer to historical attributes of file modifications mined from projects' VCSs. Promising historical metrics in recent ESE literature include relative code churn and change bursts.

Descriptive Capabilities of Change Metrics In addition to also describing complexity in the code change process, *change metrics* may also be representative of underlying technical or environmental factors impacting developers or the code change process [35]. For example, several observations of an abnormally high number of changes to a file or module prior to release might be indicative of one or more of the following:

- The file (or module) was revised several times because the requirements were not complete, were ambiguous, or had technical errors. Nagappan et. al [66] note that requirements may only stabilize after multiple implementation attempts due to the involvement of conflicting organizations.
- The developers working on the file (or module) were not familiar with the file (or module) and therefore changed it frequently to make corrections as they gained additional knowledge or discovered faults through additional testing.
- The file (or module) was not adequately tested–repeated changes were therefore made to the file (or module) as new edge cases were discovered [66].
- The module had so called *hairy bugs*, only partially fixed with a band aid solution, without understanding true root cause. Therefore, failures resulting from these defects reappear and result in repeated changes in attempts to fix the issues [66].

The characteristics and pattern of the changes to the project over defined time periods (e.g., a week), could also be indicative of various project level issues. Hassan [35] proposes capturing such patterns by describing the complexity of the code changes with entropy based measures. He notes that entropy based historical change metrics may be indicative of various development scenarios:

> *…a spike in entropy may be due to an influx of developers working on too many aspects of the system concurrently, or to the complexity of the code, or to a refactoring or redesign of many parts of the system* [35].

We also reason that change patterns may also be able to reflect time-to-market pressure. That is, pressure to deliver certain (possibly a highly anticipated) feature by an agreed delivery date. We characterize time-to-market pressure according to the size or ambitiousness of the feature, compounded with the urgency and importance of the feature (i.e., as a distinguishing product feature) to the organization developing the software. As the ambitiousness or scope of the feature increases, we might expect more widespread changes to result. As the urgency or importance increases, we might expect the changes to be more ad hoc, being made under duress, as opposed to more thoughtfully architected and coordinated changes. Considering such factors, we might expect time to market pressure to be indicated by frequent, possibly last minute changes, scattered across multiple files.

Promising Change Metrics: Churn and Change Bursts Using projects' version control systems (AKA software repositories), recent ESE studies have been able to analyze code churn and change bursts with promising results. Code churn refers to the number of code lines added, deleted, or changed over a specified time interval; a convenient time interval specification is often the time between product releases [66].

Change bursts refer to consecutive changes over a period of time [65]. Change bursts are described by gap and burst size. The gap is the maximum distance (e.g., in days) between successive changes, such that those changes are considered within the same burst. The burst size is the minimum number of successive changes required to be considered a burst. Different change bursts are shown in Figure 2.2, taken from Nagappan et al. [65].

Recent studies [60, 66] consistently find relative code churn outperforming other more traditional metrics such as MCC and KLOC. The study on change bursts in Windows Vista by Nagappan et al. [65] was particularly impressive as they found that

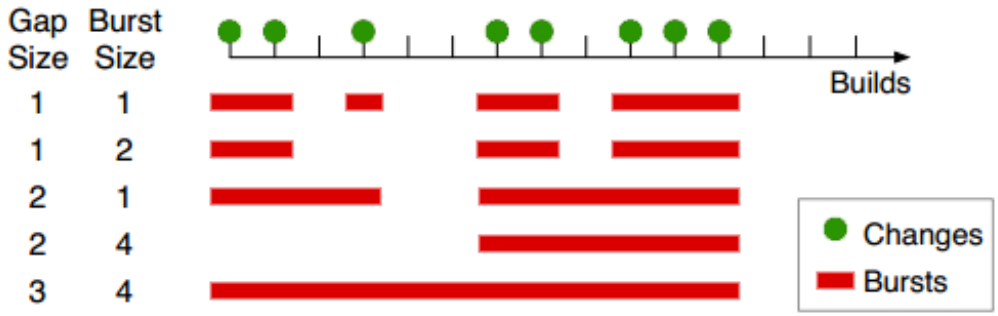

Figure 2.2: Example change bursts with varying gap and burst size parameters [65]

change burst metrics outperformed all previous predictors, such as code complexity, code churn, and organizational structure as predictors for Windows Vista.

In said studies, performance is evaluated in terms of higher precision and recall for statistical logistic regression models and machine learning based classifiers. Performance for statistical linear regression models is often evaluated in terms of statistically significant error reduction. Additional details on regression models and their evaluation can be found in Section 3.2.

2.2 Terminology and Relationships

This section disambiguates and more clearly defines terminology used throughout the remainder of the work. While clearly defining the use of various terms, relationships among terms is also explained.

2.2.1 Terms in Software Measurement

The terms *module* and *component* are often used interchangeably and depending on context, can represent various levels of granularity; although this work uses *component* in a less rigid sense to denote larger subsystems, such as runtime libraries,

rendering engines, etc. In contrast, *module* is formally defined, but the definition is such that it can be adapted to varying levels of granularity. Note that the varying levels of granularity echo the various levels of entity abstraction observed in software designs. For example, we submit that such entities could be any of the following:

- binaries,
- libraries,
- requirements,
- commits (i.e., VCS transactions)
- architectural abstractions such as layers,
- packages,
- modules,
- aspects,
- files,
- classes,
- program slices,
- functions,
- blocks, or
- lines

This work examines various graph and information theoretic measures relating to modularity and maintainability. Architectural dependency graphs can be of various granularity with nodes depicting individual methods or files, such as call graphs, with file A invoking methods in file B, to higher level block diagrams, where nodes represent libraries of functions, services, or even separate binaries. All such examples fit the formal definition of *module* outlined by Briand et. al. [22].

This work follows the basic concepts definitions established by Briand et al. [22] for *systems* and *modules*, *modular systems*, *size*, and *complexity*. In their framework, *cohesion* and *coupling* are measures that only apply to systems that can be decomposed into modules. Modules are defined as subsystems within a system S. The *system* S is composed of elements, E, and relations R, where R is a binary relation on E, ($R \subseteq E \times E$); $S =< E, R >$. A *module*, or subsystem of S is defined as $m =< E_m, R_m >$ if and only if $E_m \subseteq E, R_m \subseteq E_m \times E_m$ and $R_m \subseteq R$ [22]. This generic and intuitive framework affords architects and metric designers freedom in choosing both the particular structure and granularity of partitions defining modules within a system. As an example, Sarkar et al. [77] use the directory structure of open source projects such as Apache, Mozilla, and the Linux kernel as the definition of modularity for their experiments while staying within the framework outlined by Briand et al.

2.2.2 Defects, Faults, and Failures

We make frequent use of the terms defect and vulnerability. Faults and failures are also frequently mentioned as this research compares and contrasts with other work. There are various subtleties across the use of these terms in both academic literature and software development practice. Another fact that has bearing on this work is that not all defects result in runtime errors, with non-conformance to a project's coding standard being a prime example. Below, these terms are outlined starting with *defect*, describing some error in software design or code, extending to the eventual manifestation of a *failure*.

Defect is one of the most overloaded and loosely used terms in related literature. For this work, the use of the term *defect* is synonymous with *error* or *anomaly* as defined by IEEE [41]. The goal of review and testing efforts is to identify and fix

defects prior to release. Depending on the use of term defect, it can describe a coding or documentation anomaly found prior to release, or a known defect discovered after release. This work consistently applies *residual defect* to denote undetected anomalies which persist in the software after release.

A *defect warning* or *alert* is the notification of the probable likelihood of a defect; a pre-release time frame is assumed. The term *alert* is often used when discussing warnings from a static analysis tool. By contrast, a *residual defect* refers to an undiscovered defect that remains in the software after release. A *fault* occurs when a residual defect is executed at runtime. If the fault, or the execution of a residual defect, prevents the software from providing its specified service, then such deviation is termed a *failure* (or bug) [15].

As previously alluded, there is not necessarily a direct correspondence between residual defects and failures (also often called field failures). Developers and researchers are often interested those residual defects, and by extension, defect warnings (or predictions) that eventually manifest as runtime failures (or predictions often indicating runtime failures). In contrast, some tools issue defect warnings that never manifest as run time failures. For example, tool warnings over stylistic issues such as or brace placement within the code rarely cause operational failures. In the context of externally perceived product quality, such stylistic defect warnings are considered benign. The semantics of program execution are unchanged; the compiled code is blind to the style of the source code. No runtime failure occurs.

Although such stylistic concerns may help identify non-compliance with a coding standard, they are unlikely to cause an observable field failure [74]. In terms of quality metrics, the residual defect density of a software product is an item of interest since it is these defects that may be executed during system operation and result in unexpected behavior [47]. The execution of a residual defect commonly

referred to as a fault. Whether or not such a fault prevents the software from delivering specified service determines whether or not the fault can be called a failure [15]. Generally, the likelihood of encountering a failure during system operation increases as residual defect density increases.

2.2.3 Vulnerabilities

This research uses the term *vulnerability* to indicate an instance of a security related defect present in the software's code or architecture. In this work, a security related defect is contrasted with a more general class of defects impacting product functionality and conformance to requirements. We briefly elaborate on this context and the use of terms in the larger body of current literature.

Krsul [49] concisely and comprehensively defines vulnerability as "an instance of a [fault] in the specification, development, or configuration of software such that its execution can violate the [implicit or explicit] security policy." The use of *fault* follows directly from it's standard use in dependability literature such as [15]. The term *defect*, identical to fault, is used more extensively in risk analysis literature such as [7] and [13]. Popular SSE literature (e.g [53, 54, 56]) also favors the use of *defect*. This work attempts to consistently use the word vulnerability to label a security related defect. In this work, defect can be either a functional fault or a security related fault.

Security assurance is ultimately a subset of software quality assurance [26]. Software quality assurance is concerned with assuring that the software conforms to it's requirements. Fault-induced failures, ultimately caused by the execution of defects, prevent the software from meeting specified requirements. In the more general field of standard-issue software engineering, approaches to improving software quality

focus on preventing and/or removing defects throughout SLC stages. SSE is concerned with eliminating (or at least reducing) software defects/faults threatening security; such defects are commonly called vulnerabilities. In this regard, SSE seeks to improve the quality of security attributes by preventing and/or removing vulnerabilities throughout the SLC [38].

2.2.4 Entropy and Uncertainty

Entropy characterizes the average degree of uncertainty in a discrete random variable, X. In information theory, entropy, or the Shannon Uncertainty as it is also called, is often used to determine the average number of bits required to represent, or encode the range of expected values of X. Shannon Uncertainty is also frequently referred to in the literature as Shannon Entropy.

Equation 2.1 shows the general Shannon Entropy formula, while Equation 2.2 shows the formula applied to a uniform random distribution. In the case of uniform random distributions, $P(x_i) = \frac{1}{n}$, because each event is equiprobable, and the $\frac{1}{n}$ term can be factored out of the sum. Uniform random distributions model scenarios where code modification is the same across all entities:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \lg P(x_i) \tag{2.1}$$

$$= -\frac{1}{n} \sum_{i=1}^{n} \lg \frac{1}{n} \tag{2.2}$$

Entropy characterizes patterns and redundancy, and as such, is frequently used to evaluate data compression techniques. The more pattern and structure, the lower the entropy. As the distribution of the expected values of X approach equiprobabil-

ity, its entropy likewise increases, reaching *maximum entropy* for a uniform random distribution.

Entropy is applicable to various aspects of software and software development. Entropy calculations performed on various software artifacts can reflect organization or discord with respect to software development and maintenance activities, can characterize architectural structure of the software, or can characterize the potential for data flow at runtime. Moreover, these relationships are relevant to the context of vulnerability prediction. Section 2.3.2 gives an expanded discussion of entropy as a software metric and describes its application to this research.

Rao et al. [73] extend the notion of Shannon Entropy to continuous distributions defining a new measure termed cumulative residual entropy (CRE). The CRE measure has some attractive properties that we note for reference purposes and that may find application in this work. In particular, they note that Shannon Entropy is not sensitive to extreme differences in value and that this fact can be detrimental in situations where the actual values of a discrete random variable has bearing on realizing a payoff, such as in a game of chance (or in our case, perhaps line by line inspection effort). They give the example of two finite sample sets of two discrete random variables, X and Y, where $X = \{1, 2, 3, 4, 5, 6\}$ and $Y = \{1, 2, 3, 4, 5, 10^6\}$, noting that Y contains a sample value (10^6) much greater than any item found in X: $10^6 \gg 6$. The Shannon Entropy (here using $\log_{10}$ for the log function) is oblivious to this extreme difference, but CRE is not. CRE has the additional attractive property that it can be computed from sample data and the computations asymptotically converge to the true values of the underlying distribution. We may consider utilizing CRE in metrics involving lines of code, where we might add a cost component for an expected inspection effort.

2.3 Related Work

Our study is informed by similar empirical vulnerability and fault prediction studies by Shin [81], Ayanam [16], Gimothy et. al. [30], and Bozorgi et. al. [21]. Our work is most closely related to that of Shin and Ayanam, as both researchers investigated coupling metrics as vulnerability predictors. Shin's and Ayanam's respective works build on a long tradition of complexity metrics used to predict faults. Vulnerability ranking approaches are informed by the work of Bozorgi et. al.

2.3.1 Metrics Based Fault Prediction

Code Metrics. Several studies [23, 64, 66, 83] have examined the relationship between residual defects and static metrics extracted from source code. For example, sheer size of a file in lines of code (LOC), or more commonly in thousands of lines of code (KLOC), has been studied as having a bearing on residual defects based on the premise that larger more complex code is more difficult to understand and comprehend. By extension, reviews of difficult to understand and comprehend code are less likely to unearth defects. Another popular metric is McCabe's Cyclomatic complexity (MCC) which measures the number of paths through a program. The premise behind McCabe's cyclomatic complexity relates to the difficulty in achieving adequate branch and path coverage during testing.

In the area of fault prediction, Gimothy et al. [30] set several precedents for fault prediction studies: use of large, real-world open source software, applying linear and logistic analysis, independent evaluation of univariate predictors, as well as applying machine learning techniques, and 10-fold cross validation for training and testing. These staple analysis patterns appear regularly across fault prediction studies, and by extension, recent vulnerability investigation works as well. Gimothy et al. applied

the CK [23] metrics suite for objected oriented (OO) software to seven versions of Mozilla Firefox, covering 3,192 extracted classes. They found high correlation between the CK CBO metric, coupling between objects, and fault proneness of modules, with precision and recall values over 0.69 of on predictive models based on CBO.

A related work by Janzen and Saiedian [44–46] considered a large number of software architecture metrics to examine the impact of test-driven development (TDD) on software architecture. Their objective was to provide a comprehensive and empirically sound evidence and evaluation of the TDD impact on software architecture and internal design quality. They conducted formal controlled experiments in undergraduate and graduate academic courses, in a professional training course, and with in-house professional development projects in a Fortune 500 company. The experiments involved over 230 student and professional programmers working on almost five hundred software projects ranging in size from one hundred to over 30,000 lines of code. The research also included a case study of fifteen software projects developed over five years in a Fortune 500 corporation. Their research result demonstrated that software developers applying a TDD approach are likely to improve some software quality aspects at minimal cost over a comparable test-last approach. In particular, their research shows statistically significant differences in the areas of code complexity, size, and testing. These differences can substantially improve external software quality (defects), software maintainability, software understandability, and software reusability.

Even though the objective of Janzen and Saiedian's research was different from ours, we can benefit from their comprehensive treatment and examination of software architecture metrics (e.g., for complexity, coupling, cohesion, size, productivity, test coverage, class-level interface-level, etc.). For example, for the class-level alone, they considered some 40 different metrics (e.g., depth of inheritance tree, number

of instance variables, number of static variables, number of instance methods, number of static methods, number of primitive methods, etc.). Many of the metrics studied by Janzen and Saiedian were also studied by Shin [81], noted earlier. The sheer volume of metrics along with a comprehensive statistical analysis and empirical methods will be beneficial to our work and will suggest not only various metrics to consider but also how to combine various statistical indicators to arrive at useful conclusions.

Change Metrics: Churn and Change Bursts. In the last decade, a large number of studies have gone past direct code-based attributes to examine how these attributes change over time as the code is developed. The difference in these more recent studies is that they are one level removed, extracting metrics from the version control system rather than from individual files. We refer to such metrics as *change metrics* or *historical metrics* to distinguish them from more traditional static *code metrics*, that do not require a VCS for calculation, being directly obtainable from a version archive of the source code. Two such change metrics are churn, introduced by Munson and Elbaum [63], and change bursts. Note that churn and change burst metrics each can be decomposed into various metric suites; that is the terms churn and change burst do not themselves define concrete measures, but apply as a moniker to the historical traits common to such measures. Below, we summarize related work in fault prediction and defect estimation inspires our efforts in the context of vulnerability prediction.

Khoshgoftaar et al. [48] use churn relative to bug changes, as the number of lines added or changed to fix the bug. Khoshgoftaar used the amount of code changed along with 16 other static code metrics to build a module fault-proness predictor. Their case study on two successive releases of a telecommunications system con-

taining 171 modules with over 38,000 functions yielded precision and recall values over 78%.

Nagappan and Ball [66] demonstrated how to use relative code churn as an estimator for system defect density. Their case study on Windows Server 2003 showed that various churn related metrics were able to discriminate between fault-prone and not fault-prone binaries with an accuracy of 89%.

Moser et al. [60] compared 18 change metrics (called process metrics in their work) to 31 traditional code complexity metrics in Java source for the Eclipse project. They used a cost-sensitive prediction model to allow for different costs in prediction errors. They found that change metrics outperformed the traditional complexity metrics by a margin of about 10 percentage points for both true positive rate and recall when using their cost based model. In addition, the models based on change metrics had nearly half the rate of false positives when compared to the models based on static code metrics.

Bell, Ostrand, and Weyuker [69] studied change metrics related to individual programmers. They analyzed change reports filed by 107 programmers for 16 releases of a system with 1,400,000 LOC and 3100 files. A “bug ratio" was defined for programmers, measuring the proportion of faulty files in release R out of all files modified by the programmer in release R-1. The study compares the bug ratios of individual programmers to the average bug ratio, and assessed the consistency of the bug ratio across releases for individual programmers. Their results that counts of the cumulative number of different developers changing a file over its lifetime can help to improve fault predictions, while other developer counts were not helpful. They concluded that information related to particular developers were not good predictors.

In another study, Bell, Ostrand, and Weyuker [18] examine several churn metrics,

such as lines added, deleted, and modified, where churn = added + deleted + changed. They evaluate the independent predictive capability of several different types of both relative and absolute churn, using 18 successive releases of a large software system. They also study the extent to which faults an be predicted by the degree of churn alone, and in combination with other code characteristics. Their findings indicate that various churn measures have roughly the same predictive capability. Bell, Ostrand, and Weyuker conclude that including *some* change measure from a prior release, R_{i-1}, is a critical factor in fault prediction models for release R_i.

Nagappan et al. more recently studied change bursts as predictors of residual defects in Windows Vista. For their study on Windows Vista, they found that change burst metrics outperformed all previous predictors, such as code complexity, code churn, and organizational structure, yielding precision and recall values over 90% [65].

2.3.2 Entropy Based Software Metrics

As described in Section 2.2.4, entropy characterizes the average degree of uncertainty in a discrete random variable, X, and this also translates in different ways to a software project. In this section, we review entropy metrics from other works that we intend to utilize as explanatory variables in our prediction models.

Hassan [35] presents several complexity metrics based on historical changes, calculating entropy for the file modifications within a change period. A change period (or interval) is a period of time over which files change as a result of development progress. A fixed calendar time period (e.g., a week) offers the most straightforward method to establish the change period; however, Hassan presents several different methods for defining the period. The different period derivation methods result in

the concept of differing period types. That is, the period types differ with respect to the method or technique used to define the period. Hassan presents the following methods for change period derivation:

1. **Fixed time:** establishing the change period based on a fixed calendar time (e.g., a week),
2. **Number of modifications:** establishing the change period over a fixed number of file modifications, across all modified files, or
3. **Burst pattern:** establishing the change period based on the continuity of successive changes or lack thereof

Hassan's entropy based, historically derived measures were shown to out perform both prior faults and prior modifications as a predictor of future faults for the open source systems he studied. As mentioned, in addition to his results on entropy based historical change metrics, Hassan also directly compared the prediction capability of prior faults and prior modifications.

In addition to his entropy based historical metrics, Hassan found differences in the prediction capability of prior faults versus the prediction capability of prior modifications; again, the goal being prediction of future faults. In particular, for his studied software subjects (NetBSD, FreeBSD, OpenBSD, Postgres, KDE, and KOffice), Hassan found that prior faults (as opposed to prior modifications) were a better predictor of future faults.

These findings are notable since Chowdhury and Zulkernine [24] specifically found that prior vulnerabilities *did not* perform well in estimating residual vulnerabilities; that is, although Hassan found that prior faults *were* a good predictor of *future faults*, Chowdhury and Zulkernine found that prior vulnerabilities *were not* a good predictor of *future vulnerabilities*. Said another way, accurate and precise fault

prediction models do not always translate to accurate and precise vulnerability prediction models. However, to our knowledge, Hassan's historical complexity metrics have not specifically been evaluated for vulnerability prediction.

We will evaluate Hassan's entropy based historical complexity metrics, HCM, for vulnerability prediction. We are inspired by existing efforts to re-evaluate fault prediction metrics for vulnerability prediction. In particular, Shin [82], as well as Chowdhury and Zulkernine [24], adapt various fault prediction metrics for the purpose of vulnerability prediction. We feel that entropy based metrics may be especially well suited for vulnerability prediction since:

- Entropy based historical change metrics outperformed prior faults as a predictor of future faults–in experiments to date, the notion of prior faults predicting future faults hasn't been empirically supported for vulnerabilities (i.e., past vulnerabilities have not been shown to be predictors of future vulnerabilities),
- The level of entropy will increase as changes become more scattered across files and modules,
- The change period can be determined automatically using change bursts, and
- the presence of said change bursts may themselves be indicative of a large development push or refactoring effort where vulnerabilities may likely be introduced.

Sarkar et al. [77] describe a number of information theoretic metrics that represent module interactions in a system, or modularity. We submit that the modularity principles outlined by Sarkar et al. such as similarity of purpose, acyclic dependencies, and encapsulation also characterize the classic security design principles of Saltzer and Schroeder [76]. The associations between security design principles and detailed modularity principles enumerated by Sarkar et al. [77] are shown in Table 2.1. For example, Saltzer and Schroeder's design principle of complete mediation, where every object access must be checked for proper authority, is enabled by a

Security Design Principle	Modularity Principle
Economy of Mechanism	Size
Economy of Mechanism	Acyclic Dependencies
Economy of Mechanism	Unidirectionality in Layered Architecture
Complete Mediation	API-based Inter-Module Call Traffic
Complete Mediation	Purpose Dispersion
Complete Mediation	Similarity of Purpose

Table 2.1: Related security and modularity principles

design that routes all inter-module call traffic through a well defined API. Sarkar et al.'s Module Interaction Index, (*MII*), is a modularity metric characterizing the modularity principle of *maximization of API-based inter-module call traffic*–an underlying principle of encapsulation. *MII* is the ratio of external calls made to a module's API functions relative to the total number of external calls made to the module. Low *MII* could indicate direct usage of shared memory or direct global memory references. We might expect *MII* to inversely correlate with security vulnerabilities manifesting from unmediated changes to global variables, ultimately characteristic of poor encapsulation.

Anan et al. [12] discuss how entropy calculations on software data flow relationships can be used to derive a maintainability profile for a given software architecture. In their work, the maintainability profile quantifies the effort needed to modify a module given a particular architecture. Code modification is modeled uniformly and randomly across modules, with the probability of modification for any given module as $\frac{1}{n}$, where n is the number of modules in the system; however, unintended side effects of that modification are estimated using the probability of information flow by using the number of incident edges linking the given module to other modules in the system. This research investigates if the maintainability of a module can be used to rank predictions powered by change based metrics. The intuition is that

change based metrics might generate a number of predictions, based on frequently changed files. It seems reasonable that changes to files inside of a module, where said module already high maintainability score (relative to its peers), would be more likely to cause unintended side affects, simply because there are larger data flows directed from it to other modules.

Entropy surfaces again in Abdelmoez et al. [6] during analytical derivation measures for quantifying error propagation probability in a given architecture. Assuming an error injected into a given module, their measure characterizes the likelihood of it reaching other modules. This is similar in spirit to the maintainability metric, but seems to hold promise for mirroring propagation of attacker crafted data.

2.3.3 Metrics Based Vulnerability Prediction

Ayanam studied coupling metrics derived descended from the lineage of metrics inspired by Chidamber and Kemerer [23]. The notion of coupling is embodied in the some of the metrics from Sarkar et. al. [77] and Anan et. al. [12] that we also investigate. As mentioned in Section 2.2.4, we seek to investigate architectural modularity metrics that characterize economy of mechanism and Complete Mediation. We are also interested in information flow metrics, based on the idea that attacks are often executed by manipulating input data.

As mentioned previously, a closely related study for vulnerability prediction was provided by Yonghee Shin. The results from Shin's study indicate that certain change and developer oriented metrics are able to discriminate between vulnerabilities and the larger class of standard issue defects. Shin [81, 82] examined churn in addition to several other change metrics mined from software projects' version control systems. Shin's study was likewise focused on security and vulnerability prediction.

Shin sought to answer whether or not these metrics could also be used to identify *vulnerable* files. Shin also examined developer oriented graph metrics. Shin's results showed developer oriented metrics and change metrics yielding the best performance on the projects she studied.

2.3.4 Vulnerability Scoring and Ranking

Scoring and ranking vulnerabilities requires expert knowledge about both the severity and likelihood of exploitation. Various vulnerability rating systems exist to distill expert knowledge concerning accessibility, ease of exploitation, and severity into a numeric score or a qualitative classification such as high, medium, or low. Security advisory organizations such as US-CERT and Secunia provide mailing lists to update systems administrators on newly discovered vulnerabilities. Systems administrators use these scoring systems and advisory services to prioritize patches to operational systems. Although various criticisms exist with respect to using these rating systems for prioritizing the application of operational patches to running systems, we consider that the Common Vulnerability Scoring System (CVSS) [57], despite said criticisms, offers a metric usable for the purposes of evaluating our ranking approaches.

The past ten years have seen widespread adoption of the common vulnerability scoring system (CVSS) as a common and standardized assessment system. CVSS scores are often included on security advisories from organizations such as US-CERT, Microsoft, Cisco, and Secunia. These scores are also listed in the on-line National Vulnerability Database (NVD) [68]. Such scores encapsulate expert vulnerability knowledge and provide a basis for ranking vulnerabilities. In particular, the base metric from CVSS represents intrinsic characteristics of a vulnerability as six components: access vector, access complexity, authentication, and impact to con-

fidentiality, integrity, and availability. We expand on these notions briefly, more completely explained in [57] below:

Access vector. Defined by three binary values, $< L, A, N >$, the access vector reflects how a vulnerability is exploited:

- **L** = local access; attacker must have physical access to the vulnerable system or a local account
- **A** = adjacent network access; attacker must be on the same broadcast or collision domain; examples include local IP subnet, Bluetooth, IEEE 802.11, and local Ethernet segment
- **N** = network access; often termed “remotely exploitable", the attacker does not have to have local or adjacent access

Access complexity. Defined as a qualitative measure of difficulty, the access complexity reflects how difficult it is for the attacker to trigger the vulnerability. Access complexity can take one of three values: high, medium, or low:

- **High (H)** = most difficult to trigger; attacker must have elevated privilege, is easily detectable, or is rarely an issue given the configuration of production systems
- **Medium (M)** = somewhat difficult to trigger; scope of attack is limited or confined to untrusted users, requires additional data gathering and surveillance to launch, or is only present for a limited number of systems given typical configurations of the same
- **Low (L)** = easiest to trigger; applicable to a wide range of systems and users, and is accessible in common or default configurations

Authentication. A qualitative measure indicating the number of times an attacker must authenticate during and attack. The possible values are:

- **Multiple (M)** = requires authentication two or more times
- **Single (S)** = requires attacker to authenticate once
- **None (N)** = no authentication required

CIA impact metrics. Three additional metrics quantifying impact on security properties confidentiality, integrity, and availability (i.e., CIA) are specified with one of three values. The possible value for impact on each of these properties is specified as N, P, or C, for none, partial, or complete impact respectively:

- **None (N)** = there is no impact to the property
- **Partial (P)** = there is some impact to the property, but it is contained or isolated
- **Complete (C)** = there property is completely compromised

The above components are rolled up into a single CVSS score. We intend to evaluate our vulnerability ranking techniques based on architectural metrics against the order imposed by CVSS base scores. We are aware of the criticisms of CVSS base scores by Bozorgi et. al. [21] as a standard against which to evaluate ranking, but we submit that our usage is different in the context of ranking the predictions of residual vulnerabilities. The following paragraphs recapitulate Bozorgi et al.'s critique and then compares the differences in the context of our application and intent.

Bozorgi et. al.'s work also provides a critique of CVSS scores, noting that CVSS is subject to "categorical magic numbers" and that the score aliases too many details of the security advisory (from the perspective of prioritizing patch application based on exploitation likelihood). Further, Bozorgi e.t al. note that derivation of factors in the CVSS base score is not clear and that there are no empirical investigations of whether or not CVSS base scores are truly representative of exploitation likelihood.

The following are important differences between our work and that of Bozorgi et. al. [21], considering that their work is concerned with prioritizing patch selection and application to operational systems:

Prediction error: Our prediction models will have some prediction error, such as a given false positive rate. This factor doesn't impact Bozorgi et. al. since their false positive rate with respect to this dimension is 0; that is, they already know the vulnerability exists, as well as the fix.

Time independence: Bozorgi et. al. notes a significant difference on exploitation likelihood based on time.

Bozorgi et. al. [21] focus on the classification of a vulnerability as exploited or not exploited in order to train a support vector machine (SVM) learner to predict likelihood of exploitation. Their work is different than ours as their focus is not on the software itself, but the vulnerability reports, such as listed in the NVD [68]. Their work recognizes that although a vulnerability may be found, there may be little interest from attackers in developing its exploit. The interest of their work is in prioritizing the application of patches by software vendors. Our interest, in contrast, is in prioritizing the creation of those patches, from the perspective of a software vendor. Rather than advisory reports, our inputs consist of architectural features.

A key difference between our context and that of Bozorgi et al. discussed above is that of time. In their work, the age of a vulnerability was a significant factor in determining exploitation likelihood–attackers may be less likely to exploit a vulnerability the older it gets, since, it is reasonable that system administrators may have already patched older vulnerabilities. In contrast, our context is one where any vulnerability could potentially be a zero day exploit. A residual vulnerability is by definition a vulnerability that evades detection and persists in released software.

Despite the criticisms of CVSS, we submit that our context is different and use CVSS as an evaluation of rankings created from modularity and maintainability metrics.

In addition to Bozorgi, a number of works [8–11, 61, 62, 91–94] either further dissect the suitability and adequacy of CVSS scores, or discuss the relationship between CVSS scores and the concept of severity related to exploited vulnerabilities. We elaborate on these more recent and notable works since many of the ideas share common themes with our research (i.e., discuss CVSS scores) and draw upon similar background concepts such as likelihood of exploitation that naturally follow from examination of DaCosta et al.'s [25] Front Line Functions and Manadhata's Attack Surface Metric [51].

Allodi, Luca, Massacci, and Fabio [9, 10] study the accuracy of CVSS scores in capturing risk/severity when compared against additional exploit databases constructed from compiling exploits in the wild into (1) their EKITS database, as well as (2) those available from Symantec's Threat Database and the Exploit-db. They conclude that CVSS is not an appropriate representation of risks stemming from exploits in the wild because it unfairly favors exploits and proof-of-concept code found in exploit kits.

Allodi, Shim, and Massacci [8] additionally explore black market trading activity as it relates to vulnerability exploits, as an indicator of risk (aka severity) to Internet users. That is, the existence of an exploit on the black market represents security risk to users of the targeted software. In [11] they find that largest risk reductions result from fixing security vulnerabilities based on their presence and availability on the black market.

Let it be noted that we decided to utilize CVSS v2 [57] at the earliest stages of our research despite known criticisms as discussed in previous paragraphs. We further note that we moved forward with CVSS v2 because it is a tangible embodiment

of expert security knowledge lending itself directly to automation, which was a key practical consideration for our work. In fact, despite their criticism of CVSS, Younis et al. acknowledge that "CVSS metrics are the de facto standard for measuring the severity of vulnerabilities" [93]. The characteristic of CVSS scores being directly amenable to automation will further be shown in later chapters where we detail our approach to training set construction which leverages CVSS scores to automatically ascribe vulnerability severity. Moreover, we note development of improved standards such as CVSS v3 that compensate for shortcomings [34] in CVSS v2. Therefore, despite criticisms leveled against CVSS v2, we have confidence that better scoring schemes (such as CVSS v3) could be directly substituted for CVSS v2 scores used in our approach. Finally, it may be possible to better combine our traditional complexity metrics, VCS-centered change metrics, and architectural metrics in our approach with techniques and methods elaborated in the aforementioned recent works that share similar background ideas, albeit different research goals.

A common theme within the aforementioned works is the relationship with available (or potentially available) malicious exploits targeting vulnerabilities and the notion of whether or not patching is required. As noted previously, exploit consideration and patch timing were considerations for the research conducted by Bozorgi. We feel it is important to differentiate our work by pointing out that (1) we do not consider existence of exploits in our quantification of severity, and (2) our research relies on call graph metrics (and Front Line Functions) only to the extent required to compute architectural modularity metrics; that is, we do not directly attempt to further quantify reachability. Again, we reiterate that our research questions are more closely aligned with earlier design and implementation phases of the SDLC than with the later operational and maintenance phases. That is, our research questions seek to better understand human factors possibly responsible for vulnerability

introduction prior to release as opposed to assessing likelihood of attacker exploitation after release. The metrics we've selected embody some elements of reachability which is a theme most prominently appearing in the work of Younis, Malaiya, and Ray [93, 94]. Any reachability thus captured is implied only, resulting as a byproduct of architectural metric calculation (i.e., since such calculations depend on call graph information). However, outside our selected architectural metrics, we do not attempt to directly refine or quantify reachability.

We conclude this section by reiterating that CVSS scores are already widely used and are available directly on, or directly cross-referenced from, various vulnerability advisory systems. These advisory systems, along with vulnerability databases (i.e., NVD) constitute the source of our training and evaluation data. Therefore, a natural extension of classification (e.g., file (module) is vulnerable) for vulnerability prediction is to utilize these available scores, encapsulating expert knowledge, for the purpose of ranking vulnerability predictions generated by our prediction models.

2.4 Background Summary

This chapter has presented an overview of related terms and concepts relevant to the intersection of empirical software engineering and secure software engineering. We've also noted work by others in the area of empirical software engineering research and have discussed the difference between code metrics and change metrics. As noted in Section 1.3, to our knowledge, entropy based architectural maintainability and modularity metrics have not yet been studied from the context of vulnerability prediction. These metrics seem promising not only for their potential to help us in vulnerability prediction, but in terms of understanding other factors in the software development environment, process, or working conditions that such

metrics might indicate.

Chapter 3

Vulnerability Prediction Modeling and Evaluation

This chapter describes the detailed steps involved in building and evaluating prediction models. We discuss particular approaches to ranking and classification, as well as evaluation of the same. The following sections provide detail on regression models, metric correlation Analysis, and evaluation. Evaluation is relative to expected reduction in inspection effort, as well as rank correlation with advisory CVSS [57].

3.1 Building ESE Prediction Models

Shröter, Zimmerman, and Zeller [80], along with Nagappan, Ball, and Zeller [67], as well as Chowdhury and Zulkernine [24], clearly describe the process of building predictive classification and ranking models based on post-release defects. They differentiate between classification, a primarily binary relation, and ranking which provides a partial or total order among predictions based on a scoring function.

Shröter, Zimmerman, and Zeller largely adhere to ESE model building process outlined by Nagappan, Ball, and Zeller [67]. Nagappan, Ball, and Zeller outline prescriptive steps for model building and specifically explain how to carry out the process using post-release defects, which is relevant to our approach. Their model build-

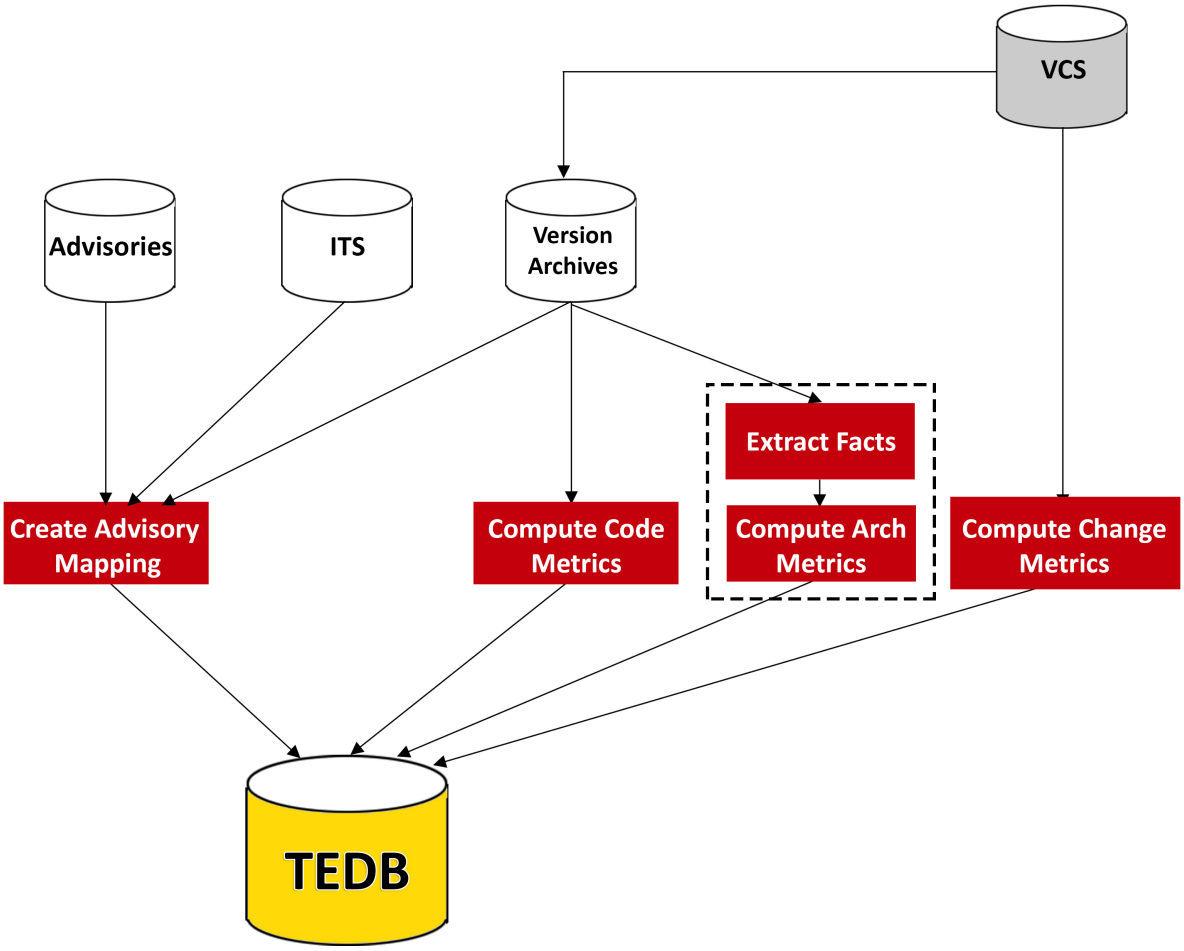

Figure 3.1: Prediction system data and processing

ing process is relevant because we are using security advisory reports to identify residual vulnerabilities. Residual vulnerabilities are a subset of the more general post-release defects (i.e., used in the predictive model building process). We list the following prescriptive steps,from Naggappan, Ball, and Zeller, along with our own embellishments (also depicted in Figure 3.1), to describe the process of building prediction models for a project:

1. **Collect input data** from bug databases (i.e., ITS), VCS, and project source code. These data sources and processing steps are graphically depicted in Figure 3.1. Note that the version archives can be obtained by obtaining source snapshots

from the project VCS, typically by checkout operation specifying a "tag" corresponding to each release.

2. **Map post-release advisories to vulnerabilities in entities**, where entities can represent any number of software artifacts or components (e.g., requirements, modules, files, lines, etc.). Our entities consist of files, modules, and layers. Note that in order to carry out the mapping, we will have to provide additional input describing higher level architectural groupings. The most straightforward example of such additional input would be a list of module and layer associations: the additional input encodes a list of layers, $\mathcal{L} = \{L_1, \ldots L_n\}$, and module assignment to each layer; $M_L(i) = \{m | m \in L_i\}$.

3. **Compute metrics and extract facts** using the source code, VCS, and fact extraction tool (e.g., UnderstandC++). Fact extraction occurs on the file level using UnderstandC++. Module and layer relations are used to sum extracted facts at the module level. Fact extraction results are fed into the architectural modularity and maintainability calculations detailed in Section 3.8. The end results of said calculations are stored in the TEDB.

4. **Determine relevant attributes (metrics)** using techniques such as *analysis of variance* (ANOVA) [59], *information gain* [31], or *principal component analysis* (PCA) [32]. PCA is used to remove attributes that are inter-correlated (exhibit multicollinearity). Multicollinear attributes must be removed when applying linear regression, otherwise large standard errors and incorrectly apportioned coefficients may result. Section 3.2 discusses linear regression in more detail.

5. **Generate predictions** using relevant attributes. At this final step, we utilize the TEDB to train and evaluate various machine learners. In this step, we also examine performance of the prediction model, using various performance evaluation metrics discussed in Section 3.4.

The above steps, 1–4, are repeated for several versions (releases) of a product. The end result, over several versions of a product, is a completed training and evaluation database (TEDB). Each record in the TEDB contains several metrics computed per release. This database is then used to build prediction models by using statistical techniques (e.g. least squares regression) and machine learning classifiers on a portion of the collected data (e.g, the training data). After the prediction models are built, a different portion of the collected data (e.g. the evaluation or test data) is used to test the predictions generated by the models. Since the test data is already labeled

as “vulnerable” ($Vuln^{+}$) or “not vulnerable” ($Vuln^{-}$), a confusion matrix relating the accuracy of the predictions to the actual values can be generated.

		Actual		
		Positive, +	Negative, –	Total
Predicted	Positive, +	TP	FP	$TP + FP$
	Negative, –	FN	TN	$FN + TN$
	Total	$TP + FN$	$FP + TN$	N

Table 3.1: Detailed confusion matrix

In Table 3.1, note that N represents the total number of samples (e.g., files or modules). The following show various counts and ratios used in the evaluation of the prediction (or classification as vulnerable):

$$N = Count_{samples} = Count^{+}_{actual} + Count^{-}_{actual} \tag{3.1}$$

$$Count^{+}_{actual} = TP + FN \tag{3.2}$$

$$Count^{-}_{actual} = FP + TN \tag{3.3}$$

Equation 3.1 shows the total number of samples, N, expressed from the perspective of the labeled oracle.

$$N = Count_{samples} = Count^{+}_{predicted} + Count^{-}_{predicted} \tag{3.4}$$

$$Count^{+}_{predicted} = TP + FP \tag{3.5}$$

$$Count^{-}_{predicted} = FN + TN \tag{3.6}$$

Equation 3.4 shows the total number of samples, N, expressed from the perspective of the generated predictions.

3.2 Linear and Logistic Regression

This section provides a brief review of linear and logistic regression as these statistical regression techniques are used extensively in both evaluating individual predictors and serving as a prediction model. In our context, linear regression can be used to estimate the *number* of residual vulnerabilities in a file or a module from either a single explanatory variable (known as simple linear regression) or multiple explanatory variables (known as multiple linear regression). Logistic regression, on the other hand, serves as a binary classifier, mapping the the response of the dependent variable into one of two classes:

$Vuln^+$ – a file or module contains one or more vulnerabilities

$Vuln^-$ – a file or module is assumed to be neutral with respect to vulnerabilities

Simple Linear Regression Simple linear regression (SLR), also known as least squares regression, shown in Equation 3.7, is used extensively in fault prediction literature to perform univariate evaluation of individual metrics, calculating β_1 so as to minimize the sum of squared residuals ($\sum_{i=1}^{n}(y_i - x_i)^2$), where residuals are the difference between sampled observations of the dependent variable y and the explanatory variable x.

$$y = \beta_0 + \beta_1 x \tag{3.7}$$

Multiple Linear Regression Multiple linear regression is an extension of SLR for more than one variable. Rather than attempting to fit a single line to minimize

error, multiple coefficients inside β are used to fit X to Y.

$$\mathbf{Y} = \beta_0 + \beta\mathbf{X} \tag{3.8}$$

Shin [81], as well as Chowdhury and Zulkernine [24] note the danger of imbalances in the training data and resulting impact on regression based models. Essentially, since there are far fewer instances of training entries labeled as vulnerable, $Vuln^+$, than those labeled as not vulnerable $Vuln^-$, there is a danger of overfitting the model to the non-vulnerable classification. That is, the regression line comes to favor the training class of higher cardinality.

Shin refers to this phenomenon as the “needle effect”, referencing the analogy that finding a vulnerability is like finding a needle in a haystack, and thus conveys the notion that far fewer“needles”, or training samples labeled as $Vuln^+$ are available for training. Shin presents a graph depicting the distribution of vulnerabilities, faults, and neutral files in Firefox. Shin’s data show that, at least for Firefox, files labeled as vulnerable account for less than 6% of the total dataset–363 files out of 11,051 [81].

The danger of failing to account for the “needle effect" in vulnerability prediction is that the learned models will generally suffer more false negatives and lower precision if the training data is not somehow balanced. In both cases above, the researchers provide balance by randomly sampling a subset of training data labeled not vulnerable, $Vuln^-$. The number of $Vuln^-$ items randomly sampled is set equal to the number of available vulnerable, $Vuln^+$, items.

3.3 Correlation Analysis

With our data sets sanitized and labeled, we will compute the Pearson correlation coefficient [59], r, for each of our architectural modularity and maintainability metrics, for each version, with respect to the number of advisories logged against affected files and modules. The equation for the Pearson sample correlation is shown in Equation 3.9.

$$r = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2}\sqrt{\sum_{i=1}^{n}(Y_i - \bar{Y})^2}} \tag{3.9}$$

The Pearson correlation coefficient enables us to test for any relationships between the independent variable, our architectural metrics, and the dependent variable, number of vulnerabilities per module. We will be looking for values above 0.5.

We also examine the Spearman rank order correlation [59], ρ (Equation 3.10), between a module's various architectural metrics (x_i) and the sum of CVSS scores (y_i) for any advisories logged against the module for each release.

$$\rho = \frac{\sum_i (x_i - \bar{x})(y_i - \bar{y})}{\sqrt{\sum_i (x_i - \bar{x})^2 \sum_i (y_i - \bar{y})^2}} \tag{3.10}$$

Notice we use the form of Spearman correlation, ρ that accounts for tied ranks. Given the limited range of CVSS values, we reason that we could have several results "bin" to identical ranks.

3.4 Evaluating Classification Performance: P, R, and FP_{rate}

Precision, recall, false positive rate, and F-measure are often used in the evaluation ESE prediction models (as well as machine learning and information retrieval) [52]. This section provides the equations for these evaluation metrics and reasons about their application.

These metrics characterize the quality of the prediction models relative to the confusion matrix (Refer to Table 3.1). As such, they yield results on the quality of the prediction, characterizing how often a prediction model was correct, with how often it was wrong.

Precision, P, shown in Equation 3.11, is a measure of the correctly predicted results relative to the total predictions.

$$P = \frac{TP}{Count^{+}_{predicted}} = \frac{TP}{TP + FP} \tag{3.11}$$

Precision on its own can be misleading, because it ignores false negatives, or samples actually containing a vulnerability that go undetected by the prediction model. The model fails to classify the sample as $Vuln^{+}$. This phenomenon is also frequently called a type II error. As an extreme case, consider a model that predicts only one real vulnerability and misses all the others out of 100 vulnerable samples. In this case, there is no misclassification; $FP = 0$. Therefore, the precision in this extreme example is 1, implying that the model correctly classified 100% of samples it detected as being vulnerable. This example, although extreme, provides motivation for defining Recall.

Recall, R:, or true positive rate is shown in Equation 3.12

$$R = \frac{TP}{TP + FN} \tag{3.12}$$

Recall, R, (also called sensitivity) provides intuition as to how likely it is that a prediction model will detect a Vulnerability prone file, assuming the real data actually does contain a a vulnerability. Recall is used in conjunction with precision, P, and a false positive rate, FP_{rate}, to characterize the performance of a classifier.

False positive rate, FP_{rate}:

$$FP_{rate} = \frac{FP}{N} = \frac{FP}{FP + TN} \tag{3.13}$$

The false positive rate indicates how likely it is that a prediction model will flag a file or other entity as vulnerable, when it really is not. Note that cost and penalty functions can be derived from the false positive rate, since we would not know a priori which modules our prediction model would have us inspect further.

F-measure, is show in in Equation 3.14.

$$F = \frac{(\beta^2 + 1)PR}{\beta^2 P + R} \tag{3.14}$$

$$\beta^2 = \frac{1 - a}{a} \tag{3.15}$$

The F-measure, yields a value between 0 and 1, indicating the balance of precision to recall for the prediction model. The beta parameter, when set to 1, equally favors precision and recall; so common is this configuration that the F-measure is often just denoted as F_1. Increasing beta above 1 favors recall. Conversely, decreasing beta below 1 favors precision [52].

3.5 Evaluating Ranking Performance: ρ and ROC

Ranking prediction results for inspection is a further optimization in inspection effort reduction. Firstly, we require that the prediction model has identified a set of entities (e.g., files in our case) as vulnerability prone; that is, it is believed that the entity contains at least one residual vulnerability. However, rather than simply provide a large list of files to the development team to triage and inspect, we can attempt to rank the predictions. However, in order to rank items relative to one another, we need a measure of severity which is to our context and application, what relevance is to Information Retrieval.

In Information Retrieval (IR), there is some distance measure that approximates the relevance of documents resulting from a user query. Precision, P and recall R in an IR system are binary evaluations of relevance, respectfully indicating the fraction of retrieved documents that are relevant, and the fraction of relevant documents that are retrieved (i.e. retrieved from the entire corpora). However, given the enormity of the document space, it is common for millions of documents to be returned. The user, issuing a query, typically does not desire to inspect each document retrieved. Therefore, IR systems have adopted various measures of relevance in order to put those items of greatest relevance at the top of the list returned to the user. We submit that the commercial development context could make use of an analogous relevance measure for vulnerabilities.

To our knowledge, no other work has proposed a ranked inspection for vulnerability predictions. Our conjecture as to why ranked inspection has not been discussed in the context of security is that the forces of time and delivery schedule in a commercial development environment have not been completely considered. That is, the few researchers who have explored vulnerability prediction emphasize the

importance of recall to ensure that type II errors are minimized. While we appreciate the importance of correctly detecting all vulnerability prone modules, it must also be recognized that the vulnerabilities themselves may be of varying severity and that management may not desire to expend development resources on such tasks considering that prediction models have some false positive rate. In a commercial context, developers are likely to be tasked with more deterministic tasks that more directly impact the bottom line. That is, chasing possible security vulnerabilities will likely lose out to more well defined tasks developing differentiating features.

In a commercial context, project managers often have to make so called go/no-go decisions to ship software or hold it back because of known defects. However, because residual vulnerabilities are by definition undetected, it is not likely, even with prediction models such as ours, that a product would be held back because the predictive model indicated vulnerability-prone files. Reasoning based on our experience working professionally in a commercial context for over fifteen years, a manager would not push back a ship date simply because there was a possibility that a vulnerability exists. A parallel for this line of thinking and justification can also be drawn from practices surrounding static analysis tools, as described in Section 2.1.2, which also often highlight security defects. Because of the aforementioned indeterminate nature of static methods, and the potential to waste valuable development resources on false positives, the field of static analysis tools has come up with alert prioritization schemes around a backlog of defects; See sec:role-static-analysis-tools.

In summary, we assert that industrial application of vulnerability prediction models would likely benefit from a quality prioritization scheme. What exactly fits our IR relevance analogy in the context of vulnerability prediction is likely a rich topic for future research. For now, we satisfy ourselves with proposing a module based ranking approach, evaluated based on strength of correlation (ρ) with CVSS base

metrics.

As an aside, we also considered calculating a more complex rank-order metric such as Mean Average Precision, or MAP [52] from IR. However, Given that MAP is an IR rank order metric, some re-interpretation of its parameters such as information need q provides an interesting thought exercise. Essentially, in our context, we have one query, “find vulnerable files”. The effect of a single information need effectively reduces MAP to a $Precision@K$ metric [52] because $|Q| = 1$ for a single query. However, this exercise sparks new ideas for how one might go about searching a code base for specific types of security vulnerabilities or security violations; in other words, we could leverage more of IR to satisfy queries like “show me potential security violations related to cross site scripting (XSS)", or “show me potential security violations with respect to use of cryptography".

Figure 3.2 shows an ROC curve. ROC stands for relative operator characteristic, and is a way to visualize Recall (sensitivity) as FP_{rate} is allowed to vary. Good prediction models, returning ranked results will be indicated by ROC curves that rise sharply on the left, keeping type II errors low as well as the false positive rate. Calculating the area under the ROC curve provides a way to compare different ranking approaches and quantifies recall over a spectrum of parameter settings for ranked results. For example, one could derive an ROC plot by evaluating different recall and FP_{rate} values as the K, in $Precision@K$ is varied.

3.6 Metrics Extracted and Calculated

This section details the various metrics we intend to collect, discussing the implications of using the various metrics for residual vulnerability prediction. Although we are primarily interested in evaluating *architectural* and *change* based metrics, we

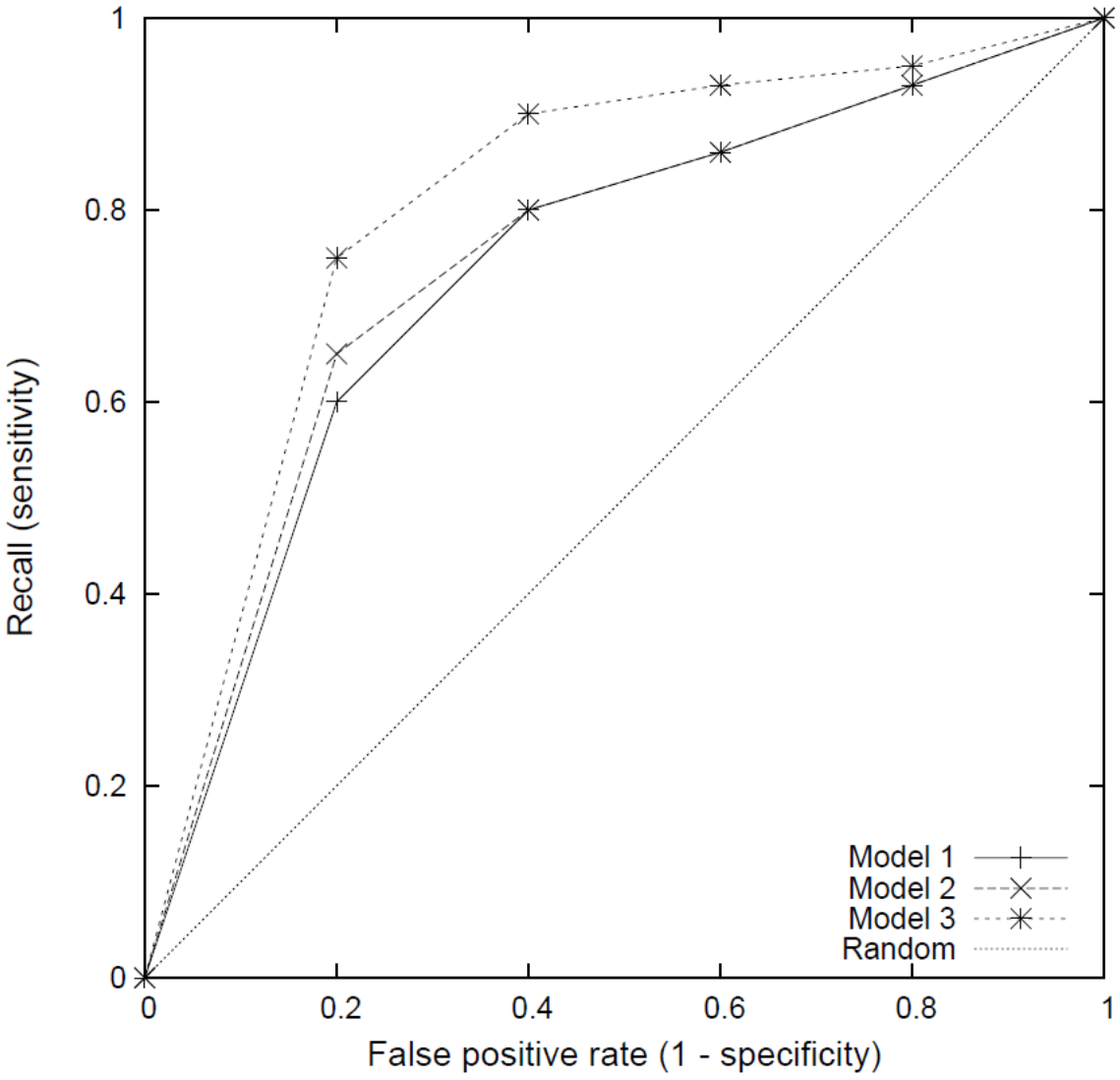

Figure 3.2: Example ROC curve

also include some traditional *code metrics* such as McCabe's and KLOC for baseline evaluation and comparison to past studies.

3.7 Notation

The following sections elaborate on the various metric categories, providing metric calculation formulas. The formulas are dependent on a notation for a software system, S. Note that we combine various notation schemes from Sarkar et. al. [77] and Hassan [35]:

- S consists of a set of modules, $\mathcal{M} = \{m_1, m_2, \ldots m_M\}$, where $|\mathcal{M}| = M$, the number of modules.
- Functions in S are $\mathcal{F} = \{f_1, f_2, \ldots f_F\}$, where $|\mathcal{F}| = F$, and it is assumed that the functions are distributed over the set of $\mathcal{M}$ modules.
- f^a denotes a function that belongs to a module's API.
- $\mathcal{L}$ denotes a set of layers $\{L_1, \ldots L_p\}$.
- $K(f)$ denotes calls made to function f. $K_{ext}(f)$ denotes the number of calls to a file from other modules. $K_{int}(f)$ denotes calls made from within the same module, and where $K(f) = K_{ext}(f) + K_{int}(f)$.
- $K_{ext}(m)$ is the number of external function calls made to module m. A module with $f_1 \ldots f_n$ functions will have $K_{ext}(m) = \sum_{f \in \{f_1 \ldots f_n\}} K_{ext}(f)$
- $\hat{K}(f)$ is the total number of calls made by function f.

3.8 Architectural Modularity Metrics

The architectural modularity metrics presented in this section quantify modularity principles enumerated by Sarkar et al [77]. The modularity metrics quantify modularity principles that may likewise impact security properties. The module interaction index, MII, for example, quantifies the extent to which external calls to a

module, honor the API provided by the module, as opposed to calling directly into private functions and methods. The *MII* measures the portion of all calls made to a module that are also routed through that module's API. We submit that such a property is also useful for security. The following paragraphs frame this concept more concretely using a hypothetical authorization module as an example.

Consider a module that is responsible for handling authorization. It is possible for the *MII* values to range between 0 and 1: $0 \leq MII(authorization) \leq 1$. An authorization module with *MII* of 0 implies that the module is either not being used, or that any users of the module are bypassing the authorization module's API. In the ideal case, $MII = 1$, indicating that all calls to the authorization module in fact utilize that module's API.

In general, we theorize that *MII* values closer to 0, and perhaps below some project specific threshold, are inversely correlated with security advisories and patches involving a module or its called functions, even if the module is not directly related to a security feature or function. Unintended side effects resulting from system maintenance or modification would be more likely since encapsulation is violated. Such side effects have the potential to violate security.

An example would be a module maintaining a processing state, perhaps representing progress through an online game, where the progress state is updated by non-API (private) module functions; in other words, the module utilizes internal function calls to update the internal state. It is possible that although the module is not specifically labeled as security related, an attacker could carry out attacks on the methods or functions directly calling the private state update functions to gain unfair advantage in the game. In online games, such cheating may impact other gamers financially, since they are often purchasing (with real money) such advantage via virtual goods like special weapons or shielding; in other words, cheating

would devalue in-game currency. In this case, although the module is not specifically related to a security feature or function of the software, it nevertheless has the potential to impact the integrity of the game currency; by extension, any exchange rate between real world currency and game currency is similarly impacted.

The following equations utilize the notation from Section 3.7 to describe relevant metrics:

Module interaction index, (*MII*):

$$MII(m) = \frac{\sum_{f^a \in \{f_1^a \ldots f_n^a\}} K_{ext}(f^a)}{K_{ext}(m)} \tag{3.16}$$

Rationale: The *MII* was discussed extensively in at the introduction to Section 3.8.

API function usage index, (*APIU*):

$$APIU(m) = \frac{\sum_{j=1}^{k} n_j}{n * k}; 0 if n = 0 \tag{3.17}$$

Rationale: *APIU* gives an indication regarding the maturity and degree to which the module has been vetted. Modules providing a large collections of unused cryptography routines would likely have low *APIU*. We expect this metric to inversely correlate with vulnerabilities, especially those related to use of cryptography. A change in a project that introduces a call into such a module to use a previously unused cryptography routine would be treading new territory; the newly called routine is assumed to not have the operational time represented by other public functions in the module.

3.9 Change and Churn Metrics

Unless otherwise stated, when referring generically to churn, we mean the sum total of additions, deletions, and modifications. That is:

$Churn(E) = NumItem_{additions} + NumItem_{deletions} + NumItem_{modifications}$.

As an example:

$Churn(FILE) = NumLines_{Added} + NumLines_{deleted} + NumLines_{Modified}$.

NumberOfChanges – Number of releases, or commits in which the entity E, (e.g., the file or module), has changed.

Rationale: There is support from fault prediction studies showing that the more a component changes, the more likely it is to have defects; we evaluate this notion in the context of vulnerability prediction. Access to a VCS provides a fine granularity since each check-in (i.e., “commit") can be counted. When only version archives are available, the count is limited to detected changes across snapshots.

ChurnTotal – Total churn over the lifetime of an entity E, i.e., churn(E). Rationale: We assume that the more has changed, the higher the likelihood defects will be introduced.

3.10 Change Burst Metrics

Change Bursts represent a family of metrics, characterized by change bursts, $CB(\mathcal{G}, \mathcal{B})$, with gap, $\mathcal{G}$, and burst $\mathcal{B}$ parameters (See Section 2.1.3). We may refer to the change burst as simply CB, without the parameters ($\mathcal{G}$, $\mathcal{B}$, in cases where we are discussing concepts and the particular parameter settings are unimportant. The notation $bursts(E)$ corresponds to the bursts for element E. That is, element E has a change history

$E = \langle e_1, e_2, \ldots \rangle$ and its bursts are $bursts(E) = \langle B_1, B_2, \ldots \rangle$. The following are burst metrics presented by Nagappan et al. [65], adapted slightly for our context:

NumberOfConsecutiveChanges – Number of consecutive builds, versions, or releases for a given gap size, $\mathcal{G}$. This is $|bursts(E)|$, with $\mathcal{B} = 0$. Rationale: Accounts for all consecutive changes for a given gap size.

NumberOfChangeBursts – Number of change bursts corresponding to a particular gap, $\mathcal{G}$, and burst size, $\mathcal{B}$. The cardinality of CB, or $|bursts(E)|$. Rationale: Burst patterns are indicative of risky behavior.

TotalBurstSize – Number of changed builds/versions/releases in all change bursts, i.e. $\sum_{B \in bursts(E)} |B|$. Rationale: Assuming that change bursts indicate risky activities, a high number of changes during these bursts could be particularly risky.

Churn-Burst Metrics In the following definitions, let $churn(e_i)$ be the number of lines that were added, deleted, or modified during the changes to the entity e_i. By extension, let us also apply churn to sets, as in $churn(E) = \sum e_i \in E churn(e_i)$ [65].

TotalChurnInBurst – Total churn in all change bursts, i.e., churn(bursts(C)). Rationale: The amount of change involved may be particularly predictive.

MaxChurnInBurst – Across all bursts, this is the maximum churn, $max\{|churn(B)||B \in bursts(E)\}$. Rationale: Looking for extremes across change bursts.

3.11 Entropy Based, Historical complexity metrics

A Family of metrics, presented by Hassan [35], denoted as HCM, that utilize the entropy of files changed over a change period. Note that the change period can be

established as a burst with a gap and burst size. Higher values in these metrics reflect more scattered and widespread changes. Lower values of *HCM* correspond to smaller, more isolated changes to a few files. We expect that more widespread and scattered changes, characterized by larger *HCM* values, will be more likely to introduce vulnerabilities.

3.12 Code Metrics

The code metrics included here are less extensive than other studies. We include some of the better performing coupling and complexity metrics for comparison with other studies. Metrics are also selected based on our conjectures regarding how these metrics might be compared with our architectural modularity and maintainability metrics. For example, we include the Henry Kafura (*HK*) metric because it mirrors the information flow concept also embodied in Anan et al.'s [12] module maintainability index, *MMI*.

In general, we are interested in metrics that have the potential to characterize information flow through entities such as functions, files, and modules, as well as complexity metrics that might provide barriers to human comprehension. We reason that security vulnerabilities may manifest as the combination of information flow and difficulties in comprehension, such as defects introduced unknowingly by developers because parameter passing or variable accesses from one function (or module) to another is dubious or suspect. An example of a suspect variable access would be indirect access to global variables. Below are a list of code metrics we are interested in:

SLOC: Lines of source code is a size based complexity metric related to human

comprehension–the larger the code, the more difficult it is to modify without introducing an error; included because of all various metrics it is one of the oldest and most studied complexity metrics. Moreover, SLOC is one of the easiest metrics to compute.

McCabe: McCabe's complexity metric for comparison with similar studies.

FanIn: An information flow metric indicative of the number of inputs into a function. Any global variables read by the function are included in the count. This metric is of interest since many attacks are carried out by specially crafted attacker input. The more paths existing for such input therefore increases likelihood of exploitation.

FanOut: An information flow metric indicative of the output from a function. Any global variables written by the function are included in the count. This metric is of interest since it hints at the potential for errors in one function to propagate outward to other functions, causing unintended side effects, especially with respect to global variables.

Nesting: A complexity metric indicating the depth of control statements handling conditional evaluation and looping `if, else, do, while, switch`. Deeply nested control structures may lead to developer error. We posit that such errors may likely include omissions of authorization checks, or cases that inadvertently "fall through" to logic that may ultimately enable security compromise.

HK: Henry Kafura metric, HK, is an structural complexity metric indicative of the information flow through functions or modules. $HK = SLOC \times (FanIn \times FanOut)^2$ [24]. The HK metric is interesting because it characterizes information flow through a function. As mentioned previously, many attacks attempt to manipulate input.

3.13 Model and Metrics Summary

In this chapter, we have provided an overview of model building and have described the metrics collected by our study. In so doing, we have have outlined the theoretical basis and rationale for our study. Along the way, we have attempted to provide the rationale for studying the metrics selected, and have provided the measures we utilize for evaluation.

Chapter 4

Detailed Repository Mining Approach

In this chapter, we examine the problem more deeply, providing details relevant to our approach. After a brief overview is provided, we detail the steps used to carry out our investigation and specify the output or goal of each step. As the steps are enumerated, key constructs, equations, tools, and resources are identified.

4.1 Investigation Overview

This work is an exercises in basic research that builds upon empirical benchmarking and established empirical software engineering (ESE) precedents from mining software repositories (MSR) and sensitivity analysis. A rich history of fault and vulnerability prediction literature utilize a post-release analysis approach on open source software (OSS) projects. Researchers mine project repositories such as issue tracking systems (ITS) and software version control systems (VCS) using information about known and fixed vulnerabilities as an oracle.

Using data from the project ITS (e.g., Bugzilla) and VCS (e.g., git), researchers trace bug reports back to particular file revisions and software releases. The bug

reports, along with affected files, enable researchers to build a training and evaluation database (TEDB) where each record consists of a file (or other entity of interest), various metrics, and a classification label of “defective” (or“vulnerable"). Generally speaking, a classification of defective ($Defective^+$) is applied to a file (or other entity) if its code underwent modification when resolving a field failure logged in the ITS. A classification of vulnerable($Vuln^+$) is applied to a file if its code underwent modification as the result of patching a vulnerability noted in a published advisory.

The training and evaluation database (TEDB) is, in data mining terms, a data mart [33], although it is not typically referred to as such in current ESE literature. However, we note that the TEDB fits the definition of a data mart as it is used to aggregate data from various external data sources to support overall knowledge discovery in the field of ESE and MSR.

The TEDB is used to produce a confusion matrix for evaluation of generated prediction models. The confusion matrix is used to evaluate the precision, recall (sensitivity), and false positive rate of the predictive model against the file corpora relative to a given classification, e.g. “file f is vulnerable”. A confusion matrix is shown in Table 4.1. The confusion matrix summarizes the number of correct and incorrect classifications generated by the prediction model relative to the real classifications of the actual samples. These numbers (e.g. true positives and false negatives) are used to derive various performance measures (e.g., precision and recall) that together describe the performance of the prediction model. Disciplines outside of computer science refer to this type of statistical evaluation as sensitivity analysis.

A k-fold cross validation technique is often used for model evaluation as researchers often use the same (limited) data set for both training and predictive evaluation. Note that since there are often significantly fewer vulnerabilities than standard-issue defects, the k-fold cross validation technique is more often used for

		Actual	
		Positive, +	Negative, –
Predicted	Positive, +	TP	FP
	Negative, –	FN	TN

Table 4.1: Example confusion matrix

evaluation in studies focused on vulnerability prediction. In general, largely owing to the diversity among software domains and projects, such learned models are not transferable, but are instead relevant only to the project from which they were trained.

At this time, several well established repositories [78] supporting empirical evaluation exist, as well as tools facilitating analysis and mining activities. These established tools and repositories support existing, published studies against which this research may also be compared. This research evaluates security in a post-release fashion, evaluating specific software repositories against vulnerabilities reported in corresponding issue tracking systems.

4.2 Research Steps

The following sections document the detailed steps performed in our investigation. In order to extract items of interest, we perform several preprocessing steps to construct the TEDB, shown in Figure 4.1. After constructing the TEDB, we analyze and evaluate the gathered data using a variety of statistical and machine learning techniques. Where possible, we leverage existing software to carry out preprocessing and analysis calculations.

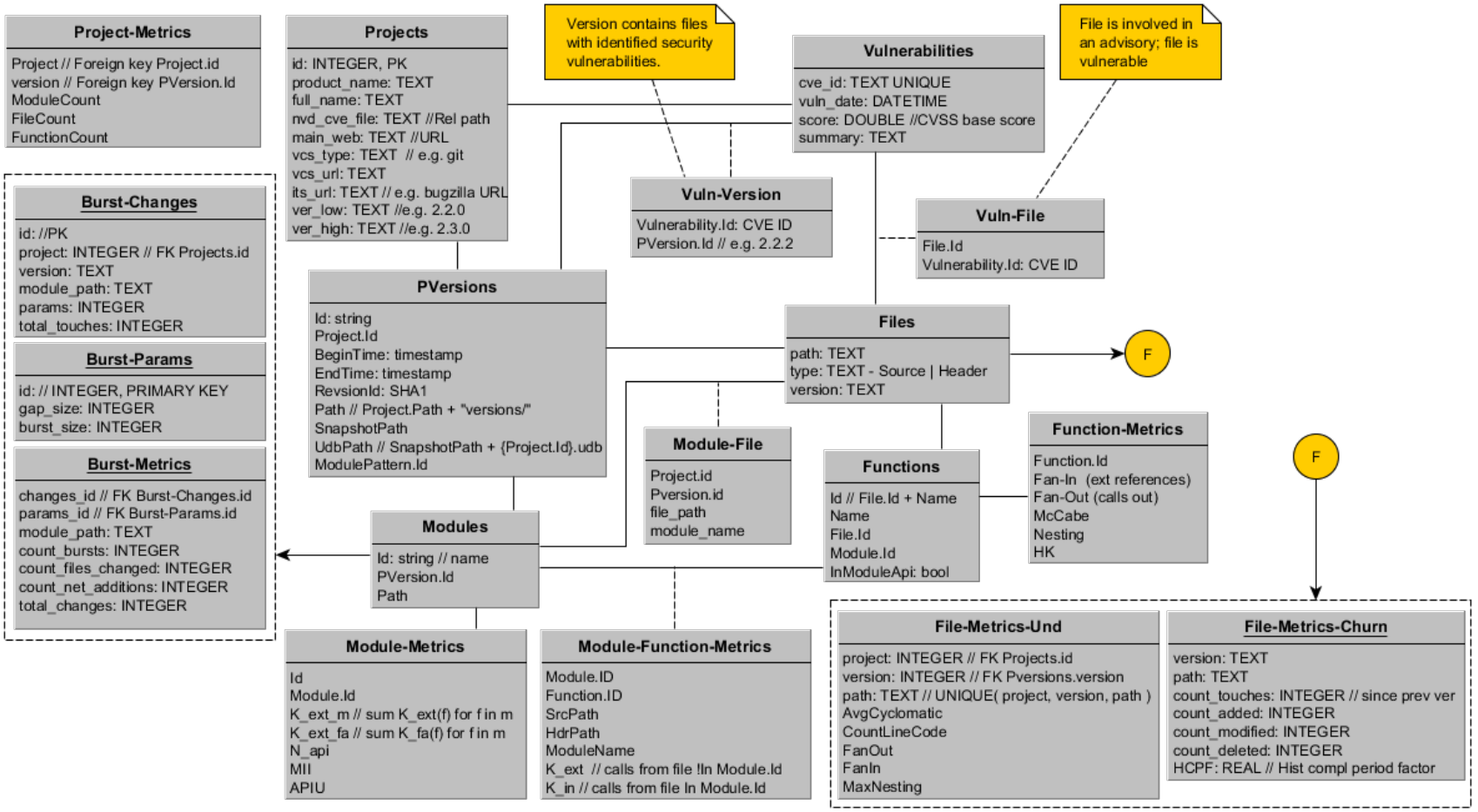

Figure 4.1: Training and evaluation database design

4.2.1 Acquire Study Subjects

Our study subjects are selected due to their significance and relevance, source code accessibility, and popularity among ESE researchers. Our study subjects are large, widely used, real world software projects. Moreover, many of these projects are widely deployed across the Internet and are used by non profit, government, and commercial organizations. Finally, several other researchers in ESE have also used the following open source projects:

Mozilla Firefox: a free and open source Web browser with over a hundred million users world wide [2].

Apache HTTP Server: a free and open source Web server powering over 50% of the active Web sites on the Internet, with a developer base over 103 million [1]. Specifically, we study **httpd2** at version 2.2.x.

MySQL: a popular open source relational database management system (RDBMS).

Building on these attributes, we gain confidence that the results of our empirical investigation will be generalizable to other software of similar size and function. The fact that many of these projects are widely used across the Internet add to the significance and noteworthiness of our results. We can also compare our study to that of similar empirical vulnerability and fault prediction studies by Shin [81], Ayanam [16], and Gimothy et. al. [30].

4.2.2 Acquire Extraction and Modeling Tools

Several extraction tools exist for examining software repositories as well as analyzing the code to automatically compute traditional metrics. Architectural recovery of dependency graphs and calculation of various metrics is known as fact extraction.

Related studies, as well as our own research have guided our selection of SciTools Understand and Anaconda Python.

SciTools Understand is a commercial tool that provides architectural recovery of call graph information and several complexity metrics [3]. In addition to being used widely in the ESE literature, Understand provides Python APIs to access the fact database it generates for a project [4, 5]. Listing 4.1 shows Python code to read information about entities within the Understand database.

```
1
2 import understand
3
4 db = understand.open("test.udb")
5
6 #Generate an image fore each function showing its callers
7 #Each image named as 'callby_<func>.png'
8 for func in db.ents("function,method,procedure"):
9     file = "callby_" + func.name() + ".png" print
    (func.longname(),"->",file)
10     func.draw("Called By",file)
11
12     #Print print all files in the project for file in
    db.ents("file"):
13     print(file.longname())
```

Listing 4.1: Example Python interface to Understand

Anaconda Python is a collection of Python packages used by data scientists for so called "data munging" or "data wrangling", handling statistical computations needed for sensitivity analysis, as well as machine learning algorithms. Using the interactive

IPython shell [72] from within JuPyter Notebook, we leverage several of Anaconda's bundled packages as follows:

Pandas is used to calculate traditional statistics such as mean, various percentiles, and standard deviation.

SciPy is used for more advanced statistics calculation such determining coefficients of correlation.

SciKitLearn is used for normalization and standardization preprocessing, as well as to experiment with machine learning models for prediction.

Matplotlib is used for data visualization.

4.2.3 Develop Custom Tools and Mine Repositories

We leverage existing tools where possible to focus our analysis efforts on the research itself, evaluating the discriminatory power of architectural metrics for vulnerability prediction, but as in many data mining activities, there is still much work needed to extract, normalize, and pre-process data from various sources.

The following items represent large portions of Python software developed to conduct this research:

- **Repository Diff Extractor:** Software to abstract the interface to version control systems for the purpose of extracting patches, diffs, or commits between project version ranges. Although the interface may be extended to support additional version control systems, we currently implement only Git. We use Git because the source code of all our case study subjects make their development history available using Git.

  We utilize Git distributed version control system (DVCS) differencing tools (e.g. `git diff`) in order to calculate coarse change metrics such as files changed

between versions, number of versions, etc. Additional custom software (diffparser) provides more precise change metrics, but at increased computation time. By more precise, we mean relative to the information provided by `git diff`. For each file changed in particular changeset, our diffparser provides lines added, modified, and deleted. Git, on the other hand, provides only total lines added and deleted and therefore does not provide information on when an addition and subsequent deletion is actually a modification.

- **Custom Fact Extractor (CFE):** Scripts to calculate Sarkar et al's modularity metrics [77] and Hassan's *HCM* metrics [35]. The scripts in this tool category are of central importance and required to calculate many of the metrics under study, since there is no other publicly available tool support. Scripts within the CFE interface with the fact database created by architectural recovery tools. Utilities and scripts in this tool category are also be used to sum function and file level relations at the module level. Note that we use the directory structure of the project as a guide to modularity.

We more completely describe our custom tools and data mining activities in Section 4.3.

4.2.4 Perform Modeling and Analysis of Subject Programs

For each subject program in Section 4.2.1, carry out prediction modeling and evaluation according to the details discussed in Chapter 3. Details for each case study are enumerated in Chapter 5, where we also consolidate our empirical findings.

4.3 Data Mining Activities

This section provides a more detailed examination of our data mining activities. Aside from representing an important (and non-trivial) part of this work, we provide this discussion for two reasons. First, we wish to inform other ESE and MSR researchers seeking additional information and insight into our methods. Secondly, these activities are important because our analysis and results depend on the data available for each software project studied.

Below, we refer to each software as a *project* to convey the notion that mining activities transcend the resulting software *product*. That is, the project encompasses multiple data sources used by the development team (and other actors) in the development and use of the resulting software product. For example, the development team tracks issues on the software product in an ITS (e.g., Bugzilla). Security actors (both offensive and defensive) utilize information from public vulnerability databases (e.g., NVD). The following steps highlight the mining activities carried out for this research:

1. Mine NIST NVD [68] for security advisories applicable to the project
2. Analyze mined advisory data to determine versions of study (i.e., the version range)
3. Mine the project VCS to extract facts related to each version
4. Determine affected files (i.e., our vulnerable files)

The following paragraphs provide additional detail as needed to understand the case studies presented in Chapter 5. Specifically, we describe the criteria we use to arrive at the studied version range as well as the different techniques employed

to determine the set of files that were modified in order to fix a publicly disclosed vulnerability.

Advisory data in this work consists primarily of CVE [58] entries extracted from the NVD for each project. CVSS [57] scores related to specific advisories are extracted from NVD records. We note here that other researchers engaged in similar empirical studies [24] built advisory extraction tools specific to each subject studied. For example, for Mozilla Firefox, both Shin [81], as well as Chowdhury and Zulkernine [24] utilized parsing scripts specific to Mozilla Foundation Security Advisories (MFSA). We employed similar parsing/scraping techniques for determining the set of files patched to fix a given vulnerability. We provide additional detail on our approach below.

For each project, we limit our study to a viable version range. We define viability with respect to the following factors:

Frequency: Versions occurring more frequently in CVE summaries are are more likely to offer data relevant to a vulnerability study.

Popularity: Popular versions of the project are likely to garner more widespread interest from industry and the research community.

Recency: Recent versions often, but not always, coincide with popularity. Recency is used as a tie-breaker in cases where frequency and popularity result in multiple or overlapping version ranges.

As mentioned, we use different methods to determine the set of files modified to fix a vulnerability. This set of files modified is used for marking our training data with the vulnerable classification ($Vuln^{+}$). For convenience, we refer to this set as *the vulnerable file set*. A common pre-requisite for each method is Web scraping

(and/or crawling) of the project's release notes and security advisories. We developed spiders using the ScraPy Web crawling package. The method used to arrive at the vulnerable file set depends on the information available in the release notes (or security advisory Web-page) of each project and information available in the project's ITS and VCS. For example, both Firefox security advisories and MySQL release notes include specific bug identifiers (IDs), but Apache HTTP Server does not. The inclusion of specific bug IDs in Firefox and MySQL facilitate automated patch retrieval (i.e., direct lookup by bug ID), while the absence of specific bug IDs in Apache HTTP Server require additional manual effort to be used with semi-automated search techniques. Additionally, Firefox's ITS entries, directly referenced by the aforementioned bug ID, contain patches associated with the fix. MySQL, in contrast, does not provide patches associated with a fix for its specific bug ID. This leads to two primary methods for determining the vulnerable file set: download from ITS and extract from VCS.

Download from ITS: This method is used when a bug ID is provided in the release notes and patches are available from the project's ITS. The ITS is indexed with the bug id (and further scraped) to determine the vulnerable file set.

Extract from VCS: This method is used when patches are not accessible from the project's ITS and/or when a specific bug id is not provided in the release notes.

For projects where the vulnerable file set can be downloaded from the ITS, our spiders obtain resolution and status terms, as well as perform automated text classification to determine which of the attached patches contain the vulnerable file set. For example, a particular bug entry may contain several attached patches, some of which may be a proof of concept or test code for the actual vulnerability fix. An example snippet from one such Bugzilla entry is shown in Listing 4.2. The spider

```
<tr id="a2" class="bz_contenttype_text_plain bz_patch">
<td class="bz_attach_desc" valign="top">
<a href="attachment.cgi?id=762008" title="View the content of
    the attachment">
  <b>Patch</b></a>
  <span class="bz_attach_extra_info">
      (8.59 KB,patch)
    <a href="#attach_762008"
       title="Go to the comment associated with the
    attachment">
         2013-06-13 06:39 PDT
    </a>
</span>
</td>
<td class="bz_attach_flags" valign="top">
    <span title="Reviewer Fullname
    (:reviewer_user)">reviewer_user</span>:
         review+
    <br>
    <span title="Approver Fullname
    [:approver_user]">approver_user</span>:
         approval+
    <br>
</td>
```

Listing 4.2: Example HTML from Bugzilla page parsed by Firefox spider

looks for keywords in the status fields associated with each attachment to determine the patch with the vulnerability fix. Listing 4.3 shows Python source from the spider that is used to search specific page element contents for the keywords **review** and **approval**. Additional details are highlighted in the comments within the listings.

Additionally, although not filtered by the spider directly, resolution terms such as FIXED, WONTFIX, and DUPLICATE are used when further filtering the scraped data. Following Chowdhury [24], we are interested only in FIXED bugs that have a life cycle status of VERIFIED or CLOSED.

For projects where the vulnerable file set cannot be determined from the ITS, we use one of the following techniques to determine the vulnerable file set by searching the VCS:

```
# Locate the table
  table = response.xpath('//table[@id="attachment_table"]')

# Prerequisites for parsing attachments for affected files
# Ensure the bz_attach_flags column contains both review and
    approval
# Ensure bz_attach_extra_info' is a patch (e.g., contains
    "patch")

attach_xpath = './tr/td[@class="bz_attach_flags" and\
    contains(.,"review") and contains(.,"approval")]\
    //preceding-sibling::td[@class="bz_attach_desc"]\
    /span[@class="bz_attach_extra_info" and
    contains(.,"patch")]\

    //preceding-sibling::a[contains(@href,"attachment.cgi")]/@href'

links = table.xpath(attach_xpath).extract()
```

Listing 4.3: Patch determination

- When a bug ID can be obtained, automatically search the VCS change logs (commit history) for references to the bug (i.e., in the commit message), or
- When a bug id cannot be obtained, query associated CVE summary for keyword stems that are subsequently used to search the VCS change logs in a semi-automated fashion.

The semi-automated keyword approach was used primarily for the Apache HTTP Server. Most of the CVE summaries for Apache HTTP highlight filenames or contain keywords that can be used to search the VCS repository. For example, to search the VCS for a change between two dates for path `mod_status` where the CVE summary mentions XSS (cross-site scripting), one could use the following command:

```
$ git log --after="2006-12-31" --before="2008-01-01" \
          --grep 'XSS' -- *mod_status*
```

Apache HTTP Server was the unique case study subject in this regard since we used the above semi-automated keyword approach to manually determine the vul-

nerable file set from information in the CVE alone. This was only possible due to the detail included in the CVE entries for Apache HTTP Server. In the case of both Firefox and MySQL, we were able to use the specific ITS entry identifiers (i.e., bug identifiers) to use the aforementioned automated methods: Download from ITS, and Extract from VCS.

4.4 Discussion Regarding Training Set Construction

Note that the construction of a suitable training set from the mined information sources is non-trivial. Real world data sources contain information of varying maturity, completeness, and accuracy. The variance and inconsistency in vulnerability data presents multiple challenges to building a viable data set for training supervised machine learning algorithms. Figure 4.2 shows a Venn diagram to visualize the required vulnerability information. For each software application, we require information about individual, *publicly disclosed*, security vulnerabilities. Such information includes the vulnerable file set, affected versions, and a severity score (from CVE entry).

The mapping of the required vulnerability information is needed to build out the TEDB tables and associations showing in Figure 4.3. The primary source for rows in the Vulnerabilities table are CVE entries from the NIST NVD. Figure 4.4 shows an example CVE entry. The Files table is populated for each Version of a Project based on information in the VCS. The association between exactly what Version in a Project contains a VulnerableFile is determined based on information from both CVE and ITS entries, and is described at length in subsequent sections.

The previous sections discussed various challenges and approaches for obtaining the vulnerable file set (VulnerableFile in Figure 4.3). However, such challenges first

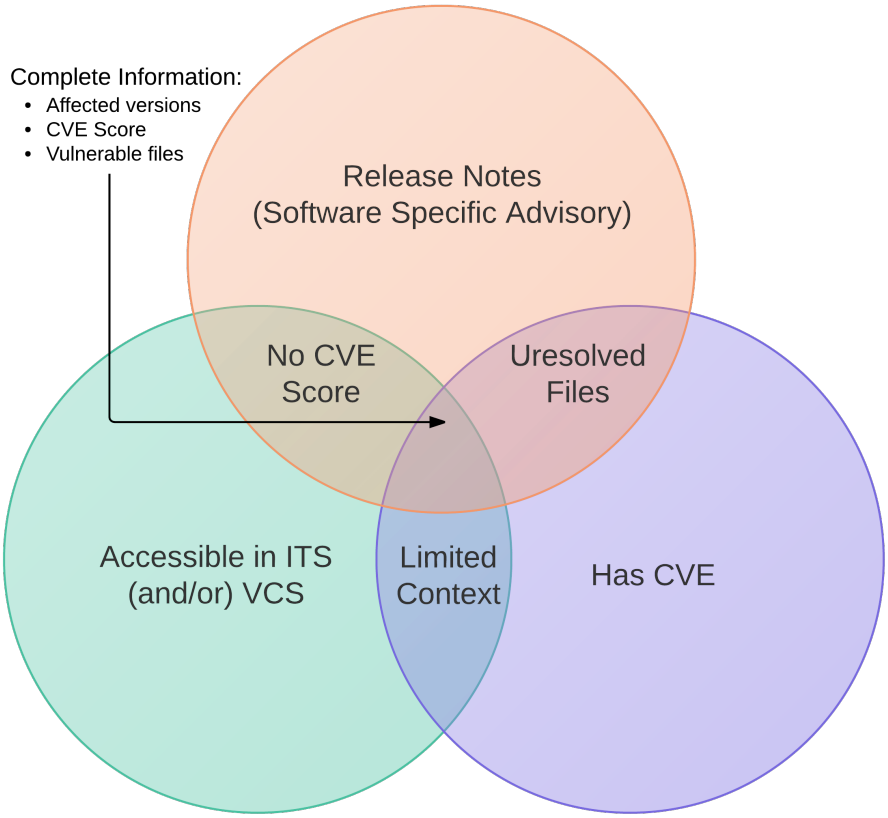

Figure 4.2: Required vulnerability information

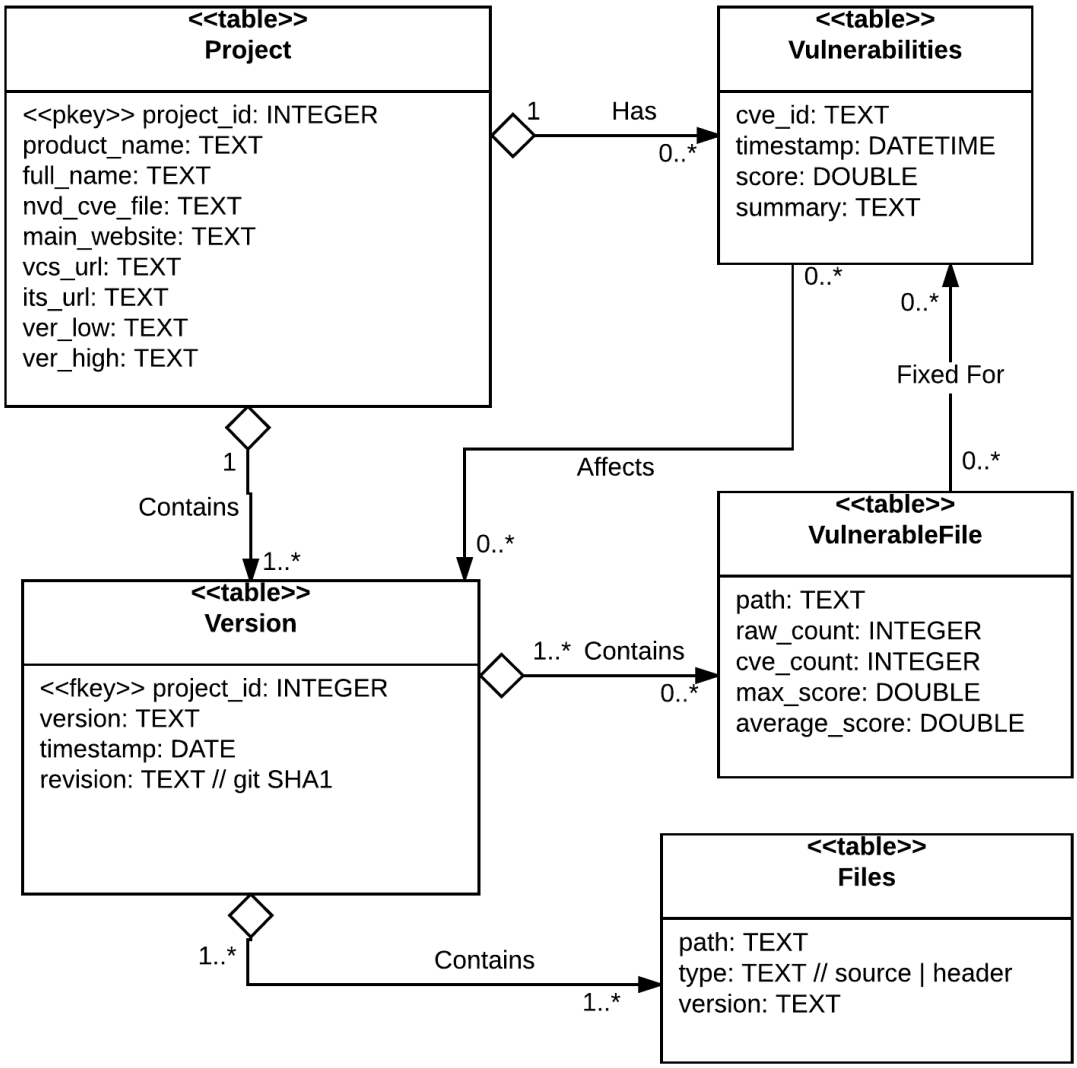

Figure 4.3: Training and evaluation database tables

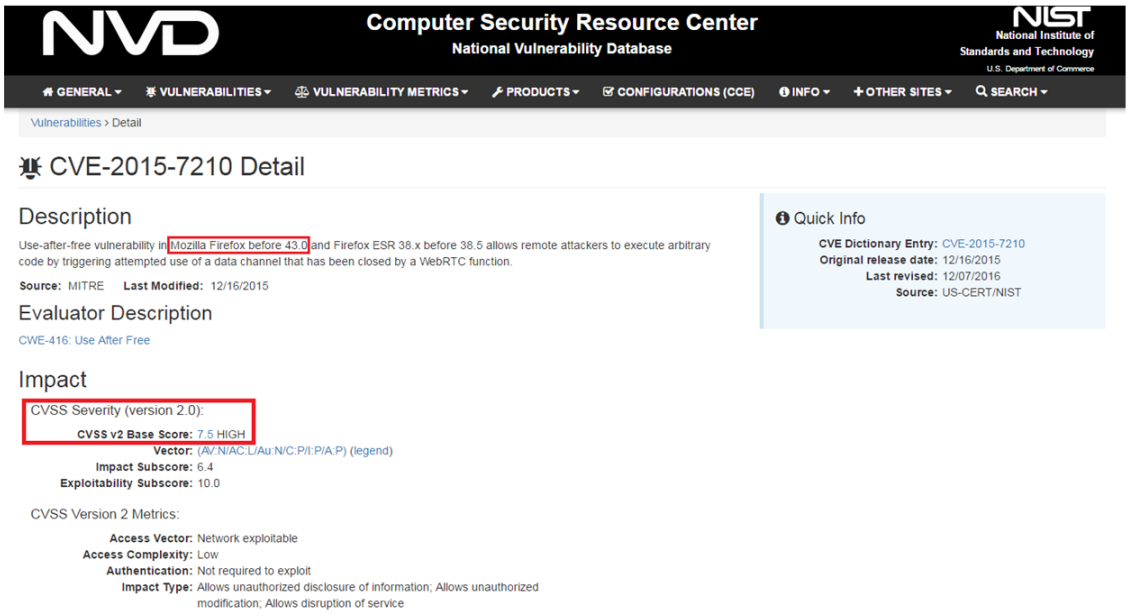

Figure 4.4: Example CVE entry with fix version and CVSS score highlighted

assume that vulnerability information is indeed *publicly disclosed*. Because our study examines security vulnerabilities that are of a potentially sensitive nature, information on the most recent and severe vulnerabilities are often not publicly accessible. We encountered this situation for approximately 5%, or 138 ITS entries out of a total of 2,747 ITS entries, that we scraped from the security advisory pages for Mozilla Firefox (i.e., MFSA pages). Additional challenges include accurate association of a particular vulnerability with a particular software revision, as well as consistently determining a severity score.

Because we train our models to differentiate vulnerable from non-vulnerable files and modules within each version of a software, it is necessary to accurately associate a publicly disclosed vulnerability with the vulnerable file set in each version. As shown by Figure 4.4, the summary text in the CVE entry typically contains mention of versions prior to a given fix version that are affected by a vulnerability. CVE en-

tries from NVD contain a list versions of the software that are affected by the specific vulnerability disclosed in each CVE. However, in the case of the Apache HTTP Server, we found that in more than one instance, the release notes (aka security report) for version 2.2 [14] described a set of affected versions that were not otherwise found in the corresponding CVE entry. For example, CVE-2010-1623 is specifically mentioned by the security advisory pages for Apache HTTP Server version 2.2, but CVE-2010-1623 itself does not list any affected version after 1.3.9. In the case of Apache HTTP Server, we chose to use the affected version information from the Apache-specific security advisory pages for version 2.2. In the cases of Mozilla Firefox and MySQL, the security advisories and release notes only listed the fix version, as opposed to a detailed list enumerating each individual version. Given the absence of a detailed list of affected versions from the security advisories and release notes for Firefox and MySQL, we turned to the CVE data for additional insight. We performed a study of the affected versions listed in CVE entries corresponding to Firefox and MySQL and found that in our version ranges of interest (i.e., the inspection interval $\mathcal{I}$) for each application, all versions prior to the listed fix version were affected. When building our training data for MySQL and Firefox, we therefore assumed that a vulnerable file was also vulnerable in previous versions, of course requiring that the file actually existed in the earlier version. We will refer to this key assumption as the *Previously Vulnerable Assumption*. Below, we describe additional detail around our investigation of the affected versions listed in CVE data to validate this assumption.

In the course of undertaking the aforementioned investigation related to affected versions listed in the CVE data from the NIST NVD, we also noticed a change in the completeness of the affected version list in said CVE entries. For example, Firefox CVE entries corresponding to version 36 and earlier (from years 2011 to 2015) often enumerate all affected versions, listing each one with its own `<vuln:product>` tag.

However, CVE entries after and including CVE-2015-2706 (for years 2015 and 2016) only list one affected version, (not counting so called ESR, or extended service release versions). An example of the affected version list is shown for CVE-2016-1935 in its raw form in Listing 4.4. Since we only look at the CPE entries denoted as an application (i.e., `cpe:/a:`) and exclude ESR entries, the extraction processing for this entry results in only Firefox version 43.0.4. Although the `<vuln:vulnerable-software-list>` (i.e., affected software list) contains only 43.0.4, the summary text for CVE-2016-1935 indicates that prior versions are also affected. We found this was also the case for all other entries (over our inspection interval $\mathcal{I}$).

As mentioned previously, our additional validation effort led us to the *Previously Vulnerable Assumption*, and a new view of training set construction which is discussed at length in later sections.

Previously Vulnerable Assumption: *In the absence of trustworthy data related to specific versions of a software that are impacted by a vulnerability fixed in version N, we assume that all prior versions, over the inspection interval $\mathcal{I}$, up to and including version $N-1$ are also impacted.*

Impacted versions are thus expressed as V_i, where:

- V_i denotes a vulnerable version number $i = \{i_1, i_2, \ldots, N-1\}$
- $\mathcal{I}$ denotes the inspection interval, $\mathcal{I} = \{R_{Min}, \ldots, R_{Max}\}$,
- R_{Min} and R_{Max} correspond to the minimum and maximum release versions studied respectively.
- Note that $i \in \mathcal{I}$, therefore $\forall i, i \geq R_{Min}$ and $i < N \leq R_{Max}$

Note that the *Previously Vulnerable Assumption* is contingent on validation over interval $\mathcal{I}$. Using Firefox as an example, interval $\mathcal{I}$ corresponds to the inclusive range 6..49.

```
<entry id="CVE-2016-1935">
<!- ** NOTE: other blocks omitted for brevity ** ->
<vuln:vulnerable-software-list>
    <vuln:product>cpe:/a:mozilla:firefox_esr:38.1.0</vuln:product>
    <vuln:product>cpe:/a:mozilla:firefox_esr:38.2.0</vuln:product>
  <vuln:product>cpe:/o:novell:leap:42.1</vuln:product>
  <vuln:product>cpe:/a:mozilla:firefox:43.0.4</vuln:product>
  <vuln:product>cpe:/o:oracle:linux:5.0</vuln:product>
  <vuln:product>cpe:/o:novell:opensuse:13.1</vuln:product>
  <vuln:product>cpe:/o:novell:opensuse:13.2</vuln:product>
    <vuln:product>cpe:/a:mozilla:firefox_esr:38.5.0</vuln:product>
  <vuln:product>cpe:/o:oracle:linux:6.0</vuln:product>
  <vuln:product>cpe:/o:oracle:linux:7.0</vuln:product>
  <vuln:product>cpe:/a:mozilla:firefox_esr:38.0</vuln:product>
    <vuln:product>cpe:/a:mozilla:firefox_esr:38.3.0</vuln:product>
    <vuln:product>cpe:/a:mozilla:firefox_esr:38.4.0</vuln:product>
</vuln:vulnerable-software-list>
<vuln:summary>Buffer overflow in the BufferSubData function in
Mozilla Firefox before 44.0 and Firefox ESR 38.x before 38.6
allows remote attackers to execute arbitrary code via crafted
    WebGL
content.
</vuln:summary>
<vuln:cve-id>CVE-2016-1935</vuln:cve-id>
</entry>
```

Listing 4.4: CVE version snippet for CVE-2016-1935

Our additional goal of relating the severity of vulnerabilities to various features of our training data requires the use of a standardized measure of severity. At the outset of this work, we decided to use the CVSS score for each vulnerability (i.e., a CVE entry in the NIST NVD) as an indicator of the severity. Two different cross-reference consistency complications related to the use of the CVSS score from the CVE presented itself in practice: unknown CVE identifiers and missing CVE identifiers. We describe each in additional detail below.

Unknown CVE Identifiers result from inconsistent mapping among security advisories, release notes, and ITS entries. This mapping inconsistency presents an issue when attempting to cross-reference a CVE identifier from a mined ITS entry. For example, the security advisory pages and ITS entry for one vulnerability in Firefox mapped to CVE-2011-1187, which is actually cataloged in the NIST NVD under Chrome. Because this entry shows up under a different browser software, this particular CVE was not found in our local database built by filtering the Firefox CVE data from the NIST NVD.

Missing CVE Identifiers occur when a security vulnerability is noted on a security advisory page (or in release notes), but does not list an associated CVE. That is, the advisory page may list only the ITS entry, but fail to list any associated CVE identifier. While one might try to determine a comparable score from the ITS entry, we found that some entries were inaccessible (read sensitive). In addition, our experience with scraping the data informed us that, in several instances, a single CVE may be associated with multiple ITS entries, and in turn, various combinations of files. These relationships among the data sources mined for information are depicted by the diagram in Figure 4.5. Rather than fabricate an overall score, we decide to omit

such entries from correlation analysis.

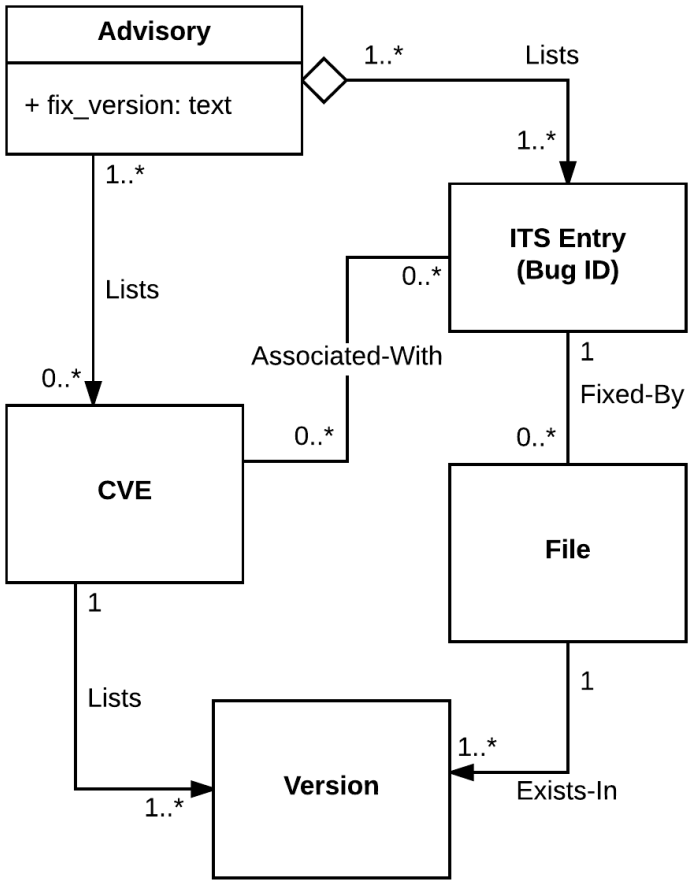

Figure 4.5: Relationship between vulnerability entities

The above complications limit our ability to reliably use CVSS scores in all cases. Because we are able to resolve the ITS entry to the individual affected files, we can classify them as vulnerable, but we are unable to perform correlation analysis related to severity.

Time	Version → 1	2	3	4	5
1	`foo.c` not in Vuln		FixedIn,0	0	0
2			0	0	0
3	1	1	0	0	0
4	1	1	0	0	0
5	2	2	1	1	FixedIn,0
6	2	2	1	1	0
7	2	2	1	1	0

Table 4.2: Marking technique for vulnerable files

4.5 Vulnerable File Marking Approach

The combination of the Previously Vulnerable Assumption and the notion of CVE identifiers attributing a scored severity, leads to a vulnerable entity marking technique. We present a view of this marking pattern in Table 4.2.

Table 4.2 depicts a marking for a single vulnerable file in the TEDB. The marking shown above is for an associated CVE. File-to-CVE relationships are stored in the TEDB outside this table. The table is interpreted as follows, for some single file `foo.c`, in iterating through mined vulnerability fix data for over the inspection interval *1*.

1. At version 1, `foo.c` is not in the vulnerability table; no record or marking; same at version 2

2. At version 3, the table building logic observes mined data indicating a security fix for version 3

    - The table building logic adds entries for `foo.c` in versions 1 and 2
    - The table building logic increments CveCount to 1 in records corresponding to versions 1 and 2, and adds any corresponding CVE relationship the

File-to-CVE table

3. At version 4, no mined security fix information is available for `foo.c`

4. At version 5, the steps at version 3 are repeated for the new fix information by applying the Previous Vulnerability Assumption

    - Table building logic back-propagates the relationship for any newly associated CVE fixed in 5
    - The CveCount for prior entries of `foo.c` are incremented, resulting in the pattern depicted in Table 4.2

Note that the *FixVersion* is *not* included in the marking depicted by Table 4.2. Inclusion of the fix version artificially inflates a rank order assessment by causing metrics assessed with Spearman correlation to artificially "lock on" to these "security fix oracles". Our approach seeks to better understand factors occurring in development of a software product that leads to severe vulnerabilities, as opposed to measuring the development effort applied to the development of the security fixes themselves. Direct measurement of the development activity related to the security issue fixes is less relevant for better understanding the origins surrounding the original introduction of the vulnerabilities (e.g., lack of developer focus, poor understanding of requirements, time to market pressure leading to shortcuts).

4.5.1 Vulnerability Distributions and Related Signals

One must ensure that the vulnerability table reflects an accurate *vulnerability* distribution at any point in time and *not* the *fix distribution* (e.g., over time when characterized as successive version slices). This is a phenomenon we experienced in our

model building process where a sanity check of correlation values revealed an echo of the *fix distribution*, rather than accurately characterizing desired relationships with the *vulnerability distribution*, i.e., within a vulnerability slice. As we discovered, this issue is especially acute for repository mined change metrics counting number of files changed or LOC, as they will directly measure the development activity coinciding with the creation of security fixes for known vulnerabilities, thereby artificially inflating correlation results when evaluating change based metrics for vulnerability prediction. Thinking of change based metrics across version slices leads to interpretation of the overall patterns as vulnerability and fix signals respectively. We can think of this phenomenon as failure to remove the "security fix signal", which forms a vulnerability oracle and has the effect akin to including a label as a feature during training. In other words, including labeled observations (or oracle entries) as features when feeding data to a learning model. A model trained on oracle data for its test set would show 100% precision, but would be completely useless in accurately predicting vulnerabilities in practical real world scenarios.

Assuming a typical scenario where a security fix for version $N + 1$ is developed in the prior version N, correlation evaluation of size based repository mined change metrics will show inflated results. Because change metrics measure the interval $N-1$ to N, an observer at version N, making predictions for version $N + 1$ from mined data, will have a vulnerability oracle for the fix included with the associated change based metric (measured over the interval $N - 1$ to N). A review of the literature (see Section 2.3.1) will find hints of analogous phenomenon during discussion of partial correlation correction for LOC compensation, although we have not seen the rationale for needed compensation so clearly explained as removal of the security fix signal. This also means that data from research on change based metrics are likely inflated if the researcher failed to specifically mention accounting for the impact of

this effect by removing the counts of known security fix data from the repository mined change metric.

Aside from exercising caution to avoid the above pitfalls when building our training tables, we force additional delay into change based metrics by phase shifting the change based signal to prior versions $N-2$ or $N-3$. The rationale for this particular phase shift compensation technique for change based metrics is to remove the effect of the fix oracle from the current observation interval $N-1$ to N. In effect, relegating the impact of the oracle to previous version slices in this manner "melts" the oracle effect into the known vulnerability distribution, which is more consistent with our approach to feature evaluation for vulnerability prediction, and helps to remove a bias that otherwise blinds us from picking up on more useful "signal".

4.6 Module and API Identification

Throughout this work, we make extensive use of the term module. In earlier sections, we noted that we had decided upon directory structure as a way of practically carrying out our mining activities. We provide additional detail here, in order to inform discussion in Section 4.7, covering our change burst detection methods, as well Chapters 5 and 6, where we discuss experimental findings and conclusions.

Our method for module identification is based on identifying patterns in the directory path, relative to the project's root repository. Path names that are less than two parts deep use the corresponding directory structure path as the module identification string. Path names longer than two parts deep use a heuristic to provide the most relevant module identifier by using the directory structure. We note that "most relevant" is subjective, based on our experience as software professionals, and on our observations of common approaches to code organization across soft-

ware projects. The module identification logic first looks for the keyword 'lib' in the path components, and then applies further heuristics, such as searching for common names (i.e., 'src', 'test', 'include') to consolidate the subtrees under the lowest applicable level. As a clarifying example, we simply show a few the test cases in Listing 4.5 that exercises our technique for module identification.

```
1  test_data = [
2              {'INPUT':
     "security/nss/external_tests/google_test/gtest/src",
3               'EXPECTED': "security/nss"
4              },
5              {'INPUT': "storage/test",
6               'EXPECTED': "storage"
7              },
8              {'INPUT': "security/nss/lib/pki",
9               'EXPECTED': "security/nss/lib/pki"
10             },
11             {'INPUT': "nsprpub/lib/libc/src",
12              'EXPECTED': "nsprpub/lib/libc"
13             },
14             {'INPUT': "toolkit/components/build",
15              'EXPECTED': "toolkit/components"
16             }
```

Listing 4.5: Python test snippet for module identification

Recall the module metrics MII and APIU are relative to a module and characterize interaction with other modules. In both metrics, there is a concept of identifying a public API for the module. We note that for a large projects, containing millions of lines of code, it is necessary to devise an approximation for what constitutes the public API, since it is unfeasible to search through the entire codebase. Our method relies in part on information from the Understand tool, as well as heuristics of our own design.

As we process functions (or methods) in the codebase using the understand tool, we identify front line functions [25] from the standard C library, according to the list given by Manadhata [51]. We consider such front line functions part of a `STDLIB`

meta-module. All front line functions thus identified are considered part of the `STDLIB` public API. Automated identification of the functions in the public API of other modules (that is, modules that are not the `STDLIB` meta-module) are based on the filename, its extension, or information reported by the Understand tool. We briefly expand on these steps in the following paragraphs.

After determining a function's possible inclusion in the `STDLIB` meta-module, we subsequently examine the filename reported to contain the function, and take different branches of additional processing, depending on whether the file is a header file (i.e., `.h`), or a source file (i.e., `.c`).

Functions (methods) in header files are classified as API functions if the header declaring them meets any of the following:

- ends with `.IDL`, as such define interfaces (i.e., API) calls
- contains the string 'PUBLIC' anywhere in the name, 'PUB/' in a path stem, or 'PUB.' preceding the file extension
- contains the string 'API/' in a path stem, or 'API.' preceding the file extension

Functions (methods) in source files: are classified as API functions if the name of the function starts with the name of its file stem. An example is would be `sqlite3_exec()`, found in `sqlite3.c`. The function name `sqlite3_exec()` starts with stem `sqlite3`

Finally, we look at the information reported by the Understand tool, related to the function declaration. If the declaration is either 'VIRTUAL FUNCTION', or 'STATIC MEMBER FUNCTION', we classify the function as an API function because, in our experience, the majority of such functions define a class interface. Virtual functions often form the base of inheritance, in order to provide a uniform interface that can be subclassed. Likewise, static member functions are often used as builder or factory objects, as interfaces to allocate resources or return handles to dynamically created objects.

Similar techniques are used prior to public determination, based on the above heuristics, based on whether the string 'PRIVATE' and its variants appear in the filename, as well as similar techniques given to the 'STATIC' and 'PRIVATE' keywords, depending on a variety of nuances best examined directly from the source code for our mining software.

As noted above, calculation of the module metrics require that functions be classified as API functions.We elaborate the preceding detail regarding our techniques, since as stated, they directly impact calculation of module the metrics. Because our mining and training software performs no additional compilation (and we perform no additional preprocessor definition for the Understand tool, inbound calls are overestimated for conditionally compiled code. In such cases, Understand cannot tell what code would actually be compiled into the resulting build. This was a compromise made in order to study several sequential versions of large open source projects such as Firefox; again, it simply is not feasible to define all these by hand for each version of our large study subjects. However, because inbound calls are overestimated, and API calls are likely underestimated, the overall approach is conservative in terms of keeping with our theme of severity. As we will see, this is reflected by our empirical data that show good discrimination performance based on the count of public API functions (n_api). That is, N_API is a simple characterization of the interconnected of a module.

4.7 Change Burst Detection

Change bursts and their related metrics are discussed in detail in Sections 2.1.3 and 3.10. This section provides additional detail around the mining and extraction of change burst metrics. We provide additional insight on our change burst detection

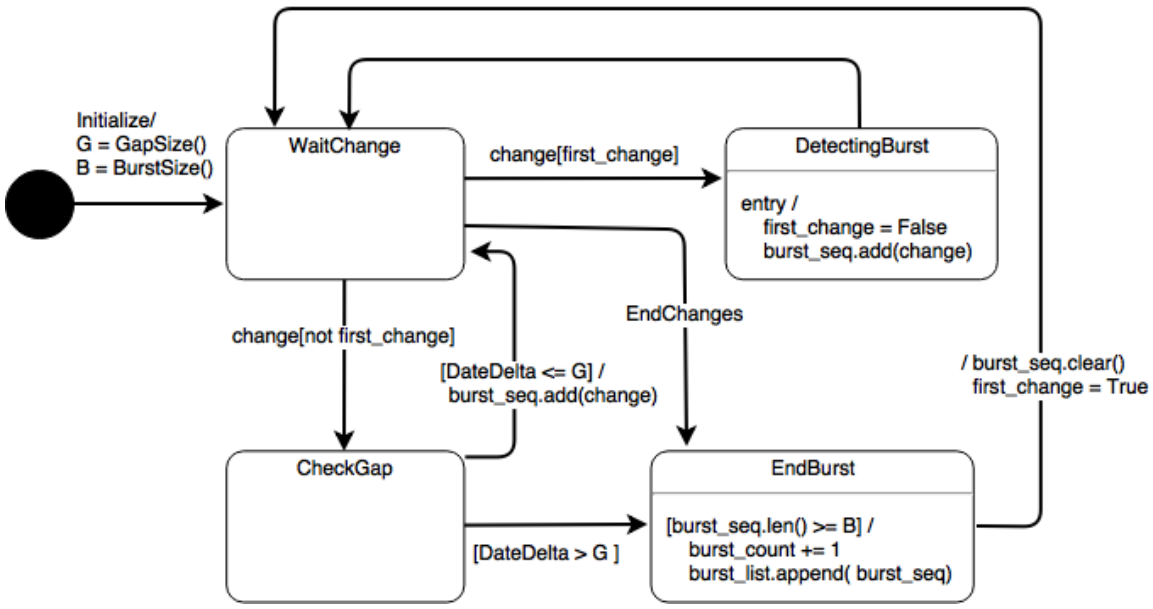

Figure 4.6: Burst detection state machine

approach and detail caveats and assumptions made when calculating change burst metrics.

Recall that a change burst, $CB(\mathcal{G}, \mathcal{B})$ is parameterized with with gap size $\mathcal{G}$ and burst size $\mathcal{B}$. The gap size, $\mathcal{G}$, is the maximum distance between successive changes, such that those changes are considered within the same burst. The burst size is the minimum number of successive changes required to be considered a burst. Figure 4.6 shows the state machine logic used to detect these burst sequences with parameters $\mathcal{G}$ and $\mathcal{B}$.

Adjusting the the gap size and burst size enables additional filtering on the related metrics. Increases in burst size decreases across the maximum absolute values for related metrics as the shorter burst sequences are eliminated from consideration. Increases in gap size result in longer burst sequences that in turn yield increasingly larger maximum absolute values. These relationships are shown in Figure 4.7 relative to the number of files changed within a burst.

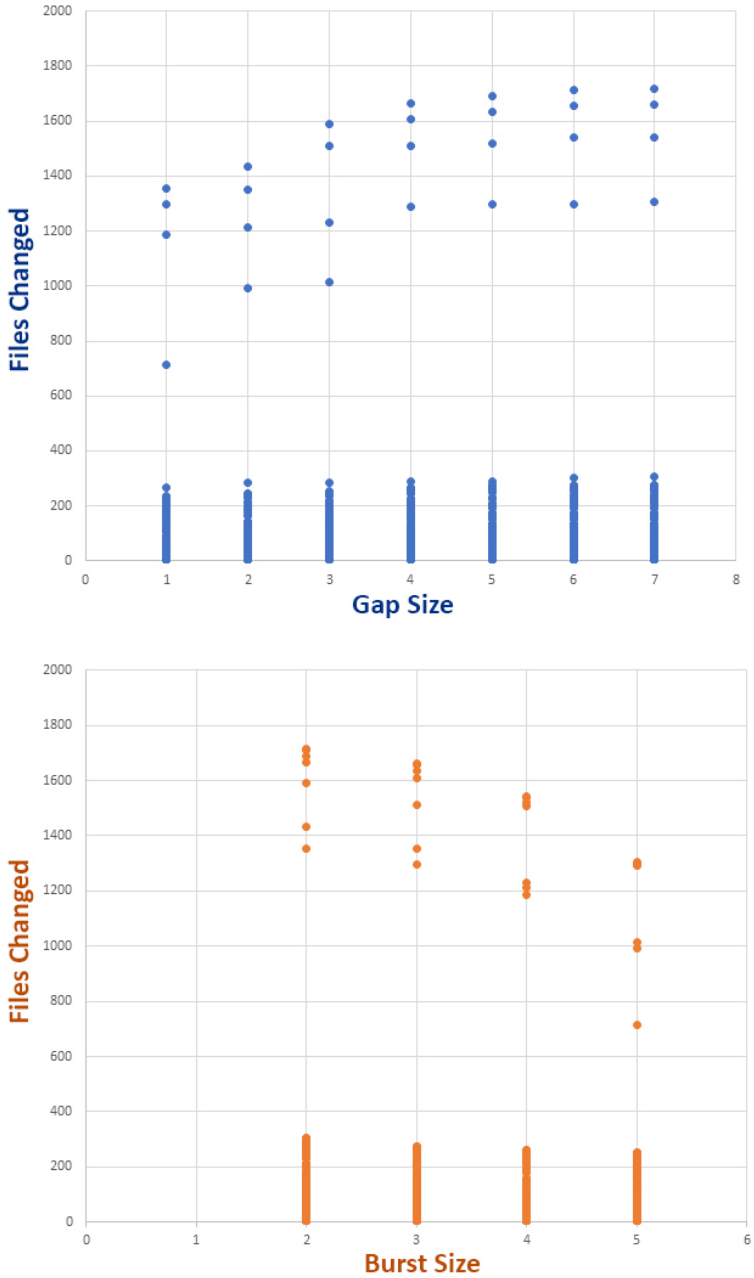

Figure 4.7: Change burst parameter effect on files changed metric

Nagappan et. al. [65] note that a controlled change process is required in order for bursts to provide a meaningful signal against the noise of regular development activity. This implies that changes go through some type of review before committed to the central repository. A common practice in industry is to first review the change with an online review tool before finally “submitting”, or uploading it into the central VCS.

Our mining software uses the git “commit date" associated with each changeset in order to perform comparisons against a gap size, G, which is specified in days. A phenomenon complicating the construction of the training database is the fact that dates associated with changesets in a VCS do not necessarily follow the sequence in which those changesets are merged into the VCS. This often occurs when as different branches are merged together, or may also result when a changeset is under review for a long period. Additionally, at least under git, it is also possible to manipulate the commit date. We encountered this phenomenon, which we refer to as “a reversal”, for both Firefox and MySQL repositories. A reversal occurs when the date associated with a subsequent change $Change_{i+1}$, precedes the current change $Change_i$. That is, we expect the mapping between change sequence and its corresponding date sequence to maintain relative ordering, $Date(Change_i) \preceq Date(Change_{i+1}), i \in \mathcal{I}$. However, a reversal is the violation of this expected ordering.

In the case of Firefox, we attempted to use supplementary information available to correct the date. Specifically, our mining software searches through the comments in the associated ITS entry, in order to determine when staged changes were pushed to the Mozilla Central development trunk. Briefly, our method is outlined as follows:

1. Parse the ITS entry (bug_id) from the git commit message

2. Fetch the ITS entry by bug_id

3. Search comments for the Mozilla Central URL

```
  RE_HG_MOZ_URL = re.compile( re.escape(\
"https://hg.mozilla.org/mozilla-central/rev/") + "(\w+)" )
```

4. Search for the push date needed for the correction

```
 RE_PUSH_DATEMATCH = re.compile(\
".*\s+at\s+(\d{4}\-\d{2}\-\d{2}\s+\
\d{2}\:\d{2}\s+[\-|\+]\d{4})")
```

Because it is not possible to determine a correction in all cases, we also log this phenomenon as two new burst measurements we call *CountReversals* and *MaxSeqByDate*. In cases where the mining software encounters no date reversals in the VCS between versions $N-1$ and N, then *MaxSeqByDate* is equivalent to the number of commits (i.e., *CountTouches*) over the interval defined by $N-1$ to N. In cases where a reversal occurs, *MaxSeqByDate* represents the largest contiguous sequence before encountering a date reversal.

4.8 Approach Summary

This chapter provided a review of key aspects of our approach. We reviewed the overall mining and model building approach and then highlighted specific details relevant to both the experimental results and other researchers. In particular, we reviewed our rational for deciding on specific version ranges of our study subjects, reviewed challenges faced, and discussed our workarounds for the same. Overall, the key points to take away from this section are experiences regarding burst collection, and our definition of the *Previous Vulnerability Assumption*.

With our approach to change burst collection, we look at each individual commit or VCS change-set for the purposes of gathering the sequential changes on a module, which we define as a directory. Because of challenges associated with a VCS, and the manner in which code is committed to a central repository, we have additionally introduced the concept of a reversal. A reversal occurs when the date associated with a subsequent change precedes the current change. We add additional measurements *CountReversals* and *MaxSeqByDate* to quantify the reversal phenomenon.

The *Previous Vulnerability Assumption* aids us in setting up our training data for evaluating correlation between individual metrics and the severity of security vulnerabilities. In the face of uncertainty as to which versions of a file a CVE might apply, we assume all previous versions are vulnerable (assuming of course that associated files exist in said previous version). Since the assumption applies retroactively (i.e., looks backward in time, applying to previous versions), we believe it is a *better* ***approximation of*** *the* ***true population*** *of vulnerable files at that historical point in time.*

Although this historical viewpoint may be of limited utility for carrying out predictions at present, we posit that it offers a superior test and evaluation environment for building and evaluating the performance of vulnerability prediction models; if for no other reason than there is more information available on which to train and evaluate prediction performance. Moreover, we don't make these statements without supporting empirical evidence. In Chapter 5, we provide data that empirically supports this claim.

Chapter 5

Experimental Results, and Analysis of Metrics and Vulnerability Predictions

In this chapter, we enumerate the experimental results for our software case studies. Results within each section are prefaced by an overview of the the project and important characteristics, such as versions studied, project size, and the approximation of the project's true vulnerability density given by our historical training tables (i.e., that define the vulnerable training set). A detailed discussion of observations that affect prediction model building are discussed according to the top performing metrics, evaluated by Welch's two-tailed test and Spearman rank correlation.

As we lead into the subsequent sections, we note the impact on the view of the vulnerability tables built for a project, by version "slices", moving forward in time across said slices. As the data presented in this section will show, we find new insights with this view of training as we observe higher average precision in early version slices and subsequently observe it "roll off" as we advance our example learning framework through those slices.

Note that both Welch's two-tailed test and the Spearman correlation do not make any assumptions about the distribution of the true populations from which our experimental samples are evaluated. That is, both Welch's test and the Spearman cor-

relation are non-parametric. Non-parametric evaluation techniques are especially relevant for vulnerability model building because:

1. The true (i.e., real) distribution of residual vulnerabilities is unknown

2. Vulnerable entities are the minority class

Regarding the second item, vulnerabilities are far fewer in number, if not orders of magnitude so, when compared to standard issue defects. We also note that the t-test is used to evaluate class discrimination, while Spearman is used to evaluate relative ranks of values. As also noted by Shin [81], we note here that *because Spearman is a rank order correlation, it does not make any quantification related to linear relationships or magnitudes of the same.* We feel this last point is important to stress in order to distinguish from the more commonly used Pearson correlation, which does assess linear relationship and magnitude.

5.1 Test Harness and Evaluation Approach

For each case study, we use the same experimental framework for evaluation. Our framework for evaluation consists of a training and test harness built with Scikit-learn [71] that provides a ten-fold stratified random split of the data, withholding 33% of the data for test evaluation. The framework is applied on select metrics, iterating the ten-fold stratified split for 10 iterations (e.g. 10 x 10-fold cross-validation), as is a widely accepted practice in vulnerability model evaluation.

We note here salient elements of our overall evaluation, adopted as a convention in our reporting. Because we are searching for residual vulnerabilities, their true population distribution is unknown by definition. Therefore we are always assuming that any statement about the vulnerability population is an approximation.

Moreover, from the samples we have (i.e., the files we've already labeled as vulnerable and neutral) it is evident that the distributions are heavily skewed, with neutral files outnumbering vulnerable files by a factor of roughly ten to one. Hence, we use the random stratified sampling technique to preserve the skewed distribution in our datasets, while training across the ten folds prevents our model from over-fitting on neutral files (i.e., the majority class).

5.2 Training Labels and Metrics Definitions

This section provides a roadmap for the case studies presented in this chapter. We describe the training scores and labels associated with vulnerable files and review the definitions for the metrics analyzed.

Vulnerable and neutral entities (i.e., files and modules) are distinguished based on a binomial column label Vuln. We set Vuln $\bar{1}$ for vulnerable entities that we mined from security advisories and release notes as described in Section 4.3. Neutral entities, or entities where no vulnerability has yet been found are labeled with Vuln = 0. We remind the reader with the caveat that although neutral entities are not yet known to be vulnerable, a label of Vuln = 0 is not an unequivocal assertion that the entity is not yet vulnerable in reality. Rather, Vuln = 0 means we don't know the entity is vulnerable.

5.2.1 Counting Vulnerabilities and Quantifying Severity

In addition to the binomial Vuln label, we can also count how many security related issues that the file was associated with from the ITS, VCS, or determined vulnerable using some other reference source. Other reference sources in our context are primarily release notes or CVE entries. The number of security related issues thus

Metric	Definition
Vuln	The binomial label indicating whether or not the entity is vulnerable or neutral. For modules, this is set to 1 if the module contains at least one vulnerable file.
RawCount	The ordinal count of security issues (most commonly counted as ITS entries) associated with a file in version $N + 1$. For modules, this is the sum of the RawCount across all files within the module.
CveCount	The ordinal count of CVE entries associated with a file in version N. For modules, this is the sum of the CveCount for all files in the module.
AvgScore	TotalScore ÷ CveCount. Simply, the average score; refer to TotalScore and CveCount defined in other rows. AvgScore is equal to the CVSS Base score metric in cases where CveCount is 1, and is 0 when CveCount is 0.
MaxScore	The maximum CVSS score, from among associated CVEs (floating point). For modules, this is the maximum across all files contained in the module.
TotalScore	The sum of CVSS scores from associated CVEs (floating point). For modules, this is the sum of the TotalScore across all files contained in the module.

Table 5.1: Definitions for classification labels and expected scores

determined is stored as RawCount. The RawCount for an entity in version N, is the number of security related issues fixed in the *next* version, $N + 1$, and represents the likelihood that the entity is vulnerability prone.

The CveCount is an ordinal value indicating the number of unique CVE entries with which an entity is associated, irrespective of version. For example, if a file (`foo.c`) is associated with three different CVE entries in version N, then CveCount = 3.

Note that RawCount indicates the number of security issues with which the entity was associated, without requiring the entity to be resolved (cross-referenced) to a CVE. By resolved, we typically mean that the ITS entry (e.g., Bugzilla id `1226423`) referenced a corresponding CVE identifier (e.g., CVE-2015-7223), *and* that said CVE identifier was also in the official list we mined from NIST NVD as described in Section 4.3. For those entities which we were able to cross-reference to a CVE entry, we define additional CVE-based counts and scores. These additional CVE-based scores are defined in Table 5.1, along with Vuln and RawCount.

We undertake to stress to the reader that "vulnerability", and even "vulnerability count" are nebulous terms. In this work, assume that "vulnerability count", or VulnCount as we call it in many of our figures, is the same as the CveCount. We elaborate on these nuances and relationships here to clarify that it is important to be specific about one's definition of "vulnerability" and "count of vulnerabilities". The use of a vulnerability count measure without specific definition is inherently ambiguous and has the potential to lead to inconsistent results and confusion.

RawCount and CveCount may differ for a vulnerable entity, especially when that entity is involved in more than one vulnerability. We will use a file to explain the point, but a module would also be similarly affected. Using a file as our example, we *assume that a vulnerable file must be associated with at least one ITS entry* (i.e., RawCount = 1), to know that it is vulnerable. Following from the relationships dis-

cussed in Section 4.3, and illustrated in Figure 4.5, the following relationships are possible for our example file:

RawCount == CveCount when a single unique ITS entry maps to a single unique CVE identifier.

RawCount < CveCount when a single unique ITS entry maps to multiple unique CVE identifiers.

RawCount > CveCount when multiple unique ITS entries map to either

- a single unique CVE identifier, or
- no CVE identifiers

As stated above, the preceding relationships are presented, for example purposes, assuming the entity (e.g., a file) is associated with an ITS entry (as was the case for Firefox). However, as discussed in Section 4.3, vulnerable entities are not always classified as vulnerable through association with an ITS entry. Recall that a vulnerable classification may also be derived from the VCS, by mention of a CVE identifier (as was the method used for MySQL Database), or named directly in a CVE entry (as was the method used for Apache HTTP Web server). For our purposes, RawCount helps to identify vulnerable entities we learned about through our mining activities, but were not resolved to a particular CVE entry. Entities with a RawCount > CveCount, and where CveCount = 0, represent entities that we have learned are vulnerable, but for which severity cannot also be ascribed.

Throughout this work, we use CveCount interchangeably with "vulnerability count". By standardizing on CveCount, we build on the already well-formed definition for CVE entries (and their use in industry) as synonymous with identified security vulnerabilities in software. Finally, our decision to standardize on CveCount additionally

facilitates our use of CVSS scores that are exclusively associated with CVE entries as *severity quantification metrics*. Note that when CveCount = 1, the average, max, and total are all equal. Table 5.1 summarizes the quantification metrics.

As stated, the CVSS-derived scores enable us to quantify *severity*. We note here that although RawCount and CveCount can be used as estimators for vulnerability proneness, they may fail to accurately characterize (i.e., estimate) *severity*. Although it is true that RawCount can tell us whether or not a file is a likely "repeat offender" (that is, a party to more than one reported security fix), there is no additional quantification of severity. For purposes of discussion, we will refer to such current approaches as "RawCount estimators".

The failure of RawCount based estimators to better characterize severity is a shortcoming of contemporary approaches because they do not correspond well with the real problem. Contemporary model building approaches built only on RawCount estimators fail to account for the properties of one-off severe bugs in the cyber security domain. That is, a software entity may only be involved in one security fix during its existence (i.e. RawCount = 1), but it is entirely plausible that the vulnerability corresponding to said fix, when exploited by a bad actor, results in full information security compromise. That is, the affected victim suffers a complete loss of confidentiality, integrity, and/or availability to their information and computation resources. Because current approaches emphasize estimation of the fix probability alone, they may miss the more important identification of vulnerable files with severe impact. Moreover, this phenomenon is in fact exacerbated since, as our data show, and as past case studies in the related literature repeatedly show, *repeat offenders are in the minority of vulnerable entities*.

Because empirical evidence suggests that repeat offenders are in the minority of vulnerable entities, coupled with the fact that current approaches (to building

vulnerability prediction models) emphasize estimators against RawCount, it follows that current approaches also likely fail to accurately reflect the true vulnerability distribution of a software project. Counting security fixes, and not otherwise emphasizing severity density and severity, strongly suggest that current approaches to model building fail to capture the reality that a single vulnerability, in some central module or file, can have a severe security impact. Hence, we not only examine the discriminative power of our selected metrics (for predicting neutral vs vulnerable class membership), but we also evaluate their rank order correlation against our severity quantification scores: AvgScore, MaxScore, and TotalScore. Our CVSS-derived scores are shown in Table 5.1. For all the reasons outlined in preceding paragraphs, we believe metrics specifically selected for their relevance to the residual vulnerability problem, and that also estimate our CVSS-derived scores, provide better correspondence with the true vulnerability distribution, and imbue models built from them with an additional severity estimation component.

5.2.2 Complexity Metrics

All file level complexity metrics are extracted relative to the `git` commit revision corresponding to our collected version N. As such, all metrics thus mined form a set of observations at N for predicting $N + 1$.

We used SciTools Understand tool to extract file level complexity metrics. The understand tool exports several metrics, detailed descriptions of which can be found at [3]. We list the top few metrics that were repeatedly selected by the ANOVA [59] filter in our classification experiments (i.e., primarily using the logit classifier), in addition to a few metrics that are widely recognized across defect and vulnerability literature. Refer to Figure 5.1 for a visualization of the the relative frequency among

the top six metrics automatically selected based on the `classif` ANOVA prefilter we used in Scikitlearn based learner pipeline when set to select K best features based on F-score (or F statistic). Note that we also experimented with K at additional values $K < 5$.

The visual in Figure 5.1 shows consistent selection of the same metrics, despite using stratified random selection and iterating across all versions of Firefox over our inspection interval. An observant reader may question why we show six bars, when the feature selection limit, K is set to 5. Although the the top five features were selected (i.e., $K = 5$), the chart displays six bars. This is not an error; whatever random combination of vulnerable and neutral files selected in each fold determine the corresponding variance and the subsequent selection corresponding metrics as a coefficient in determining fit. Our test harness simply increments a frequency count corresponding to the feature selected at each individual iteration. The interesting result is not that there are six features, but the remarkable consistency required to select the same subset of these few features over so many randomly perturbed iterations.

5.2.3 File Level Churn Metrics

A review of file level churn metric definitions is provided in Table 5.3 to facilitate quick reference in the following sections. Note that all churn metrics begin collection by walking the reverse topographically sorted revision history provided by the `git log` command from the revision corresponding to the interval $N - 1$ to N. The observation point for prediction is still at version N. As such, all metrics thus mined form a set of observations at N for predicting $N + 1$.

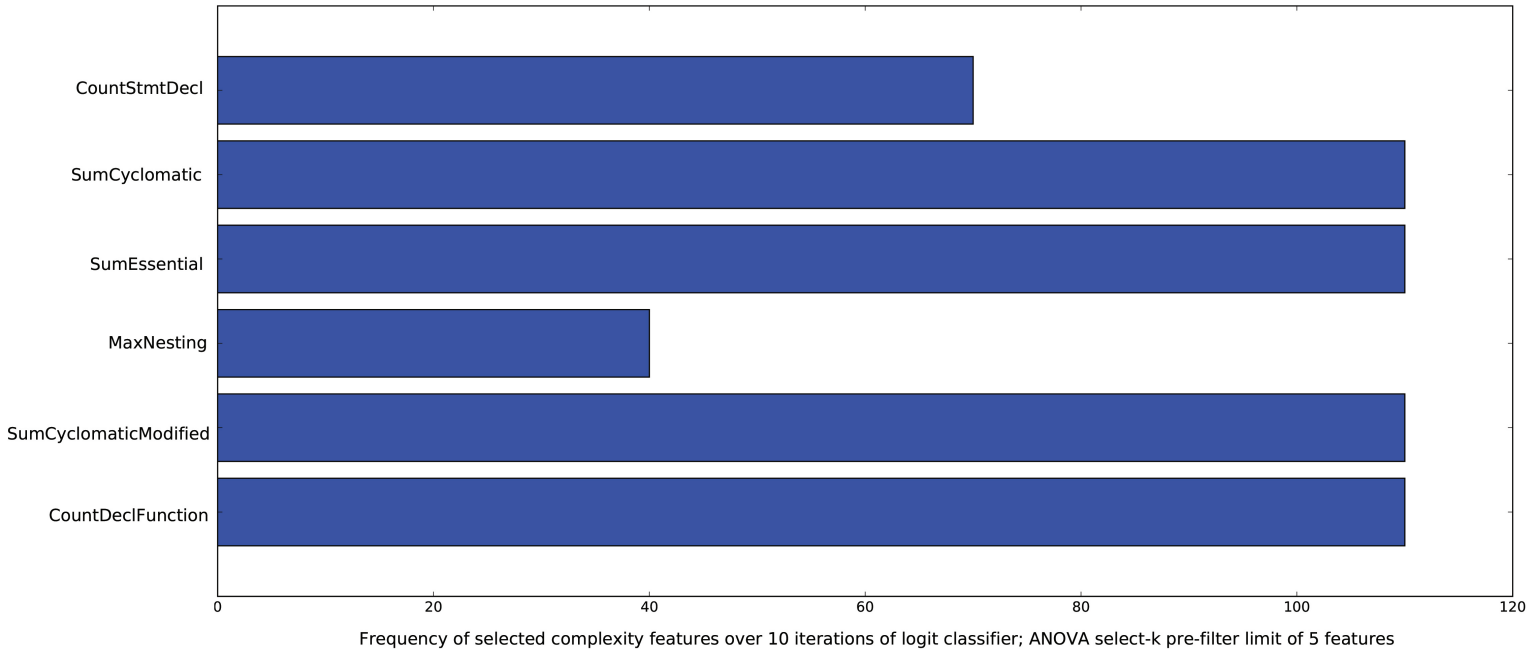

Figure 5.1: Most frequently selected complexity metrics; 10 iterations over 43 versions of Firefox.

Metric	Definition
CountDeclFunction	Number of functions.
CountLineCode	Number of lines containing source code. [LOC]
CountLineCodeDecl	Number of lines containing declarative source code.
CountLinePreprocessor	Number of preprocessor lines.
CountStmt	Number of statements.
CountStmtDecl	Number of declarative statements.
CountStmtExe	Number of executable statements.
MaxCyclomaticModified	Maximum modified cyclomatic complexity of nested functions or methods.
MaxCyclomaticStrict	Maximum strict cyclomatic complexity of nested functions or methods.
MaxNesting	Maximum nesting level of control constructs.
RatioCommentToCode	Ratio of comment lines to code lines.
SumCyclomatic	Sum of cyclomatic complexity of all nested functions or methods. [aka WMC]
SumEssential	Sum of essential complexity of all nested functions or methods.

Table 5.2: Definitions for file complexity metrics at version N

Metric	Definition
CountTouches	The number of changsets (e.g., VCS commits) touching the file since $N-1$ [aka. NumChanges]
CountTouchesP1	CountTouches for this file from $N-2$
CountTouchesP2	CountTouches for this file from $N-3$
HCPF	The historical complexity period factor, an entropy based metric indicating the file's degree of contribution to entropy over the interval $N-1$ to N
LinesAdded	The number of lines added since $N-1$
LinesModified	The number of lines modified since $N-1$
LinesDeleted	The number of lines deleted since $N-1$
TotalTouches	The cumulative sum of CountTouches at N, from all prior intervals

Table 5.3: Definitions for file churn metrics at version N

Metric	Definition
CountTouches	The number of changsets (e.g., VCS commits) touching the module path since $N-1$ [aka. NumberOfConsecutiveChanges] 3.10].
CountReversals	The number of commit date reversals encountered when scanning the (e.g., VCS commits) changes within the module since $N-1$.
MaxSeqByDate	Largest sequence of consecutive changes by date when CountReversals > 0, otherwise this metric is equivalent to CountTouches.
CountBursts	The total count of bursts in the module since $N-1$ [aka. NumberOfChangeBursts].
CountFilesChanged	The total count of files changed in the module since $N-1$.
NetAddedInBurst	A churn metric characterizing the net positive number of lines added since $N-1$. This is LinesAdded - LinesDeleted, when LinesDeleted $\leq$ LinesAdded, otherwise 0. This quantity is LinesAdded when LinesDeleted = 0.
TotalChurnInBurst	The number of lines modified, as the sum of lines added and lines deleted since $N-1$ (as reported by git diff).

Table 5.4: Definitions for module change burst metrics observed at version N

5.2.4 Module Level Burst and Architectural Metrics

Module level burst metrics are defined in Table 5.4 to facilitate quick reference in the following sections. Note that all churn metrics begin collection by walking the reverse topographically sorted revision history provided by the `git log` command from the revision corresponding to the interval $N-1$ to N. The observation point for prediction is still at version N. As such, all metrics thus mined form a set of observations at N for predicting $N+1$.

In the course of our study we added additional metrics not found in the original literature 3.10. In particular, we have added CountReversals and CountFilesChanged as discussed in Section 4.7.

Metric	Definition
APIU	The API function usage index defined in Section 3.8; appears in graphs in lowercase as apiu.
k_ext_fa	Number of calls from external modules to the public API exposed by a given module; Refer to Section 3.8
k_ext_m	Number of external calls from external modules; Refer to Section 3.8.
MII	Module Interaction Index defined in Section 3.8; appears in generated graphs in lowercase mii.
N_API	The number of public API functions or methods exported by a module, discussed in Section 4.6

Table 5.5: Definitions for module metrics observed at version *N*

Version	LOC	CountFunctions	CountMethods	CountFiles
6	2,584,376	145,384	82,529	11,329
28	4,473,921	263,480	105,902	20,050
49	5,818,230	355,352	116,753	25,592

Table 5.6: Firefox project statistics

Table 5.5 provides an overview of our module metric definitions.

5.3 Case Study 1: Mozilla Firefox Web Browser

We studied Mozilla Firefox versions 6 to 49, representing a time span of approximately five years from August, 2011 to August, 2016. Table 5.6 provides an overview of size measurements across the first, middle, and last versions studied. The measurements shown here were collected from the SciTools Understand tool [3]. The files processed include only source and header files (e.g, .c, .cpp, .h, etc.), and additionally exclude documentation and test folders in the project's working tree.

As depicted in Table 5.6, there are significant differences in size across the ver-

sions studied. Version 28, representing the middle version, is nearly double (1.7 times) the size of version 6 when measured in lines of code (LOC). The last version, 49, has more than double (2.25 times) the LOC than in version 6, and is 1.3 times larger than version 28. Likewise, the number of files in the last version, 49, are more than double (2.25 times) the number in version 6. Given such large differences, we would expect that Firefox version 49 is a much different piece of software than it was at version 6.

A concern with Firefox version 49 being much different from version 6 relate to the validity of the Previously Vulnerable Assumption. Recall that the *Previously Vulnerable Assumption* presented in Section 4.4 states that if a vulnerability is fixed in version N, then we assume all previous versions are also vulnerable. In this case, if our data indicates that some file (e.g., `foo.c`) was fixed in version 49, then, contingent on the existence of `foo.c` in each version, we would assume that `foo.c`was vulnerable in all previous versions 6 through 48. However, software which undergoes such significant change can see the vulnerability either masked or exposed more prominently as the topology of the call structure changes [92], as functionality migrates in and out of other files, or the project is otherwise refactored. Conversely, the *Previously Vulnerable Assumption* may find limitation when a file is renamed throughout its lifetime. In that case, using our example of `foo.c` fixed in version 49, our labeling technique would fail to identify `foo.c` in version 48 if the file was renamed between versions 48 and 49. Although we've noted this concern, we remind the reader that our decision to apply it for metric evaluation in Firefox is derived from our careful study of the CVE data associated with Firefox, as discussed in Section 4.4.

Figure 5.2 is a visualization the number of vulnerabilities, binned to the nearest whole-numbered (i.e., major) version of Firefox, over our inspection interval $\mathcal{I} = \{6, \ldots, 49\}$. In this figure, VulnCount represents the number of unique CVE entries

plotted against the versions to which they apply. As shown, the figure represents view of the historical vulnerability density in Firefox over *1*. The figure visually depicts the pattern we noticed in CVE data for Firefox that supports the *Previously Vulnerable Assumption* and forms the basis for feature evaluation when building prediction models for Firefox.

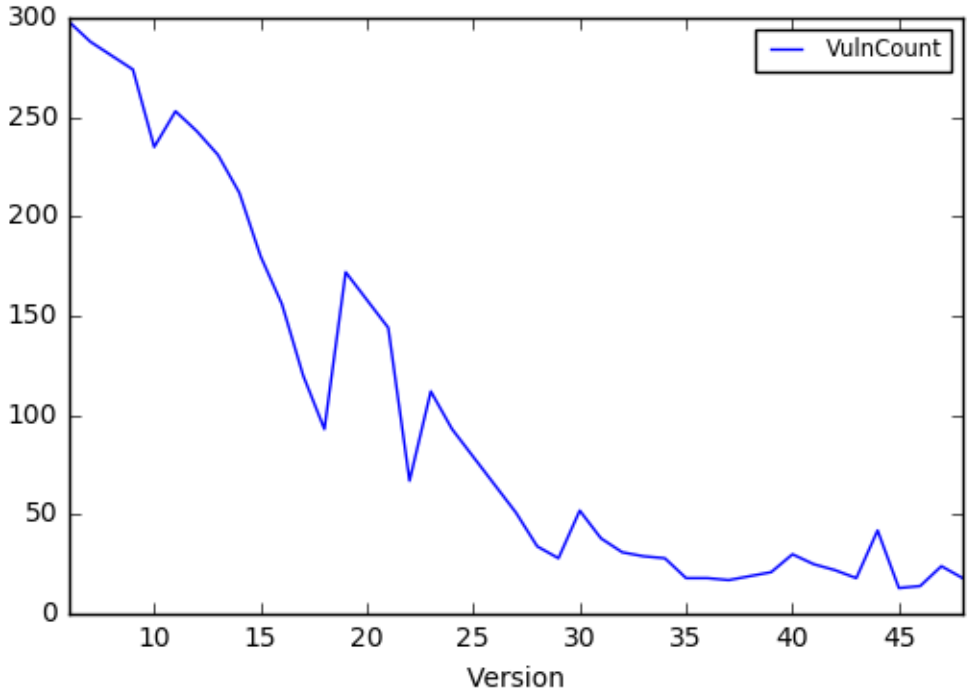

Figure 5.2: Historical vulnerability density for Firefox versions 6..49

The list of applicable affected versions is derived directly from the CVE entries themselves. Firefox related CVE entries in the NIST NVD, for versions 6 to 36, specifically enumerate all prior versions as previously discussed in Section 4.3. A short tabular summary of the top ten versions, sorted by VulnCount per version, is shown in Table 5.7.

The pattern, as depicted by Figure 5.2 and Table 5.7, shows the effect of a retroactive, linearly increasing offset that "swells" the VulnCount of earlier versions; col-

VulnCount	Version
298	6
288	7
281	8
274	9
253	11
243	12
235	10
231	13
212	14
180	15

Table 5.7: Vulnerabilities per Firefox version, 6..49

loquially speaking, we feel the catch phrase a "rising tide lifts all boats" fits well here. Early visualization of this data for Firefox informed our vulnerability table construction technique discussed in Section 4.5 and the also resulted in the Previously Vulnerable Assumption presented in Section 4.3.

Because the vulnerability table used for training is built from files known to be vulnerable, it consists of samples from the residual vulnerability population and thereby forms an approximation of the true vulnerability distribution for Firefox. We refer the reader to the concept of the *vulnerability signal* and *fix signal* discussed in Section 4.5.1. The green NumFixedVulns in Figure 5.3 represents the number of Bugzilla security issues fixed, plotted against each Firefox version – the fix signal. Note that the count of files fixed, CountFixedFiles, closely tracks NumFixedVulns (or the fix signal) in Figure 5.3.

Our overall model building approach and corresponding view of fix and vulnerability signals is reinforced when we plot the number of *fixed* issues from the ITS for each RLS version (see Section 5.3.5) against counts of Firefox CVE entries per RLS version. This plot is shown in Figure 5.4, and shows the impact of known security fixes (the blue line) against the the count of unique CVE entries (the green line) affecting

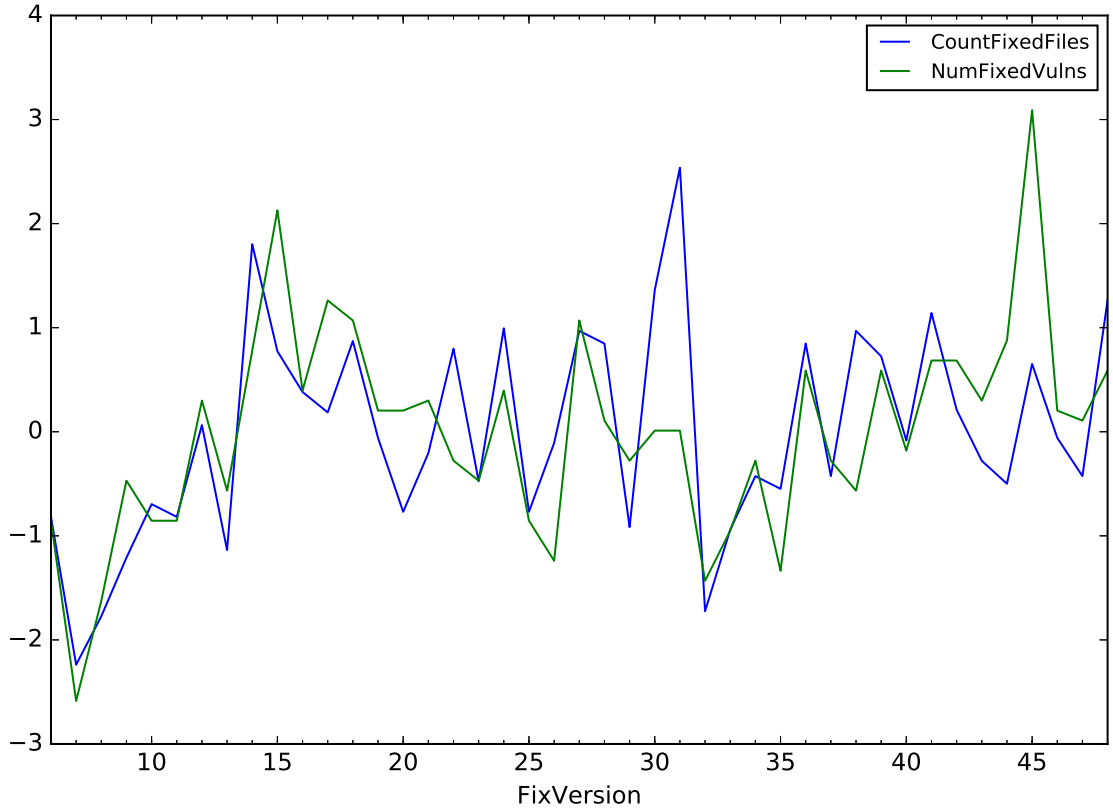

Figure 5.3: Fixed files and fixed security issues; Firefox 6..49

each version of Firefox. As shown, features A and B from fixes in the earlier version slices more severely lag in their appearance in the CVE data due to the effects of the Previously Vulnerable Assumption.

Note that Figure 5.4 simply plots two measured values present in our mined data. We can see that the tip of feature A, between versions 16 to 18, leads its appearance in the CVE data around slice 18. The image also shows what appears to be an increasing phase shift where the fix signal is echoed more rapidly in the CVE vulnerability counts of later slices. Because the overall density curve is standardized by the total sum of all CVE counts for all slices, the density signal flattens out and shows the effects of the spike near version 45 from the fix signal much less prominently. This view

supports the Previously Vulnerable Assumption because the high counts from the spike at 45 feed backward into prior slices, which works to reduce its impact on the density curve at version 45.

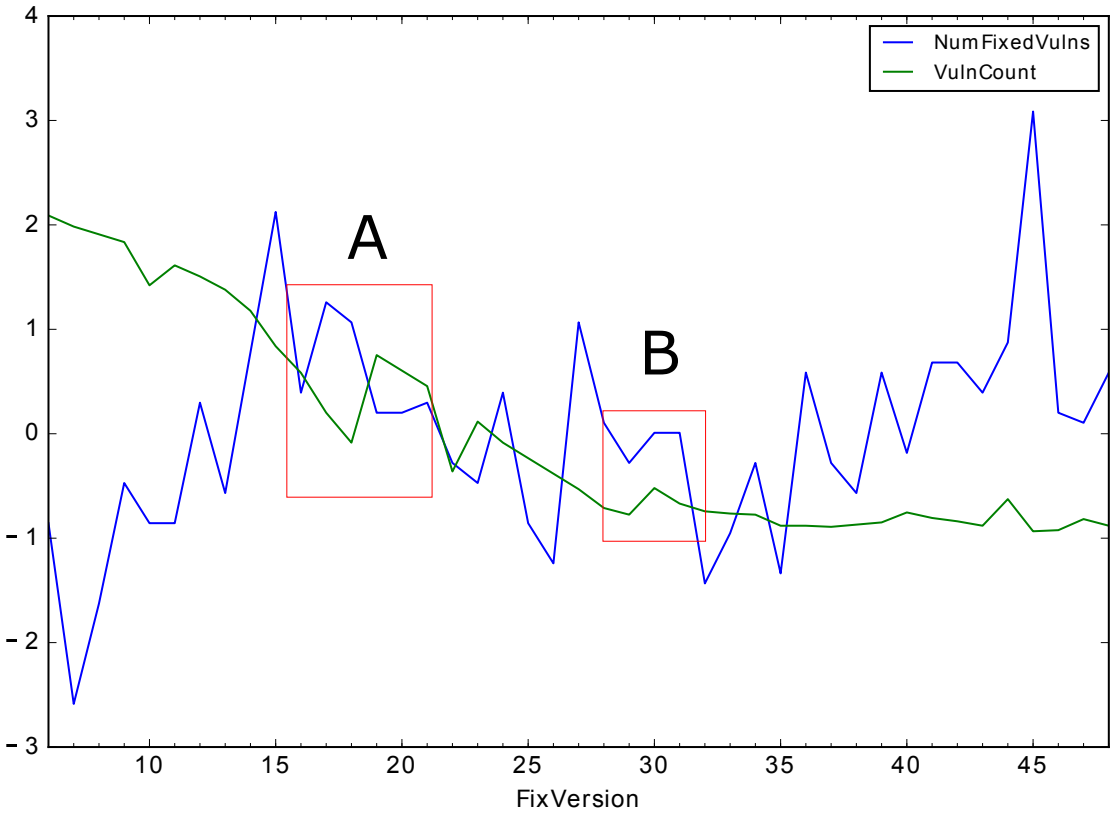

Figure 5.4: Similar trend observations A and B

We remind the reader that our statements and assumptions are based on our automatically mined data for Firefox (and our other subjects) and although we observed similar patterns across all our study subjects, each project's signature is relevant for model evaluation against it alone. We also note that the results of metric evaluation (such as when using Spearman correlation) are contingent on the accuracy of the mined data; the accuracy and shape of the vulnerability density curve is subject to the accuracy of the mined data.The shape of the vulnerability density curve used for

	CveCount	AvgScore	MaxScore	TotalScore
AltAvgLineCode	0.127	0.034	0.018	0.064
AltCountLineCode	0.287	0.059	0.063	0.154
AvgCyclomatic	0.111	0.007	0.041	0.073
AvgCyclomaticModified	0.116	0.009	0.042	0.075
AvgCyclomaticStrict	0.114	0.012	0.037	0.072
AvgEssential	0.125	0.007	0.049	0.083
CountDeclFunction	0.260	0.076	0.041	0.118
CountLineCode	0.278	0.069	0.051	0.141
CountLineCodeDecl	0.286	0.049	0.076	0.162
CountLineCodeExe	0.254	0.064	0.048	0.127
CountLineComment	0.293	0.075	0.048	0.148
CountLinePreprocessor	0.260	0.111	0.004	0.093
CountStmt	0.278	0.064	0.054	0.144
CountStmtDecl	0.279	0.050	0.071	0.155
CountStmtExe	0.254	0.071	0.040	0.121
MaxCyclomatic	0.191	0.053	0.029	0.091
MaxCyclomaticModified	0.197	0.057	0.029	0.093
MaxCyclomaticStrict	0.193	0.058	0.027	0.090
MaxEssential	0.195	0.058	0.031	0.091
MaxNesting	0.190	0.054	0.029	0.089
RatioCommentToCode	0.001	0.017	0.020	0.004
SumCyclomatic	0.256	0.063	0.051	0.128
SumCyclomaticModified	0.260	0.067	0.049	0.128
SumCyclomaticStrict	0.255	0.065	0.049	0.126
SumEssential	0.268	0.066	0.055	0.134

Table 5.8: Average Spearman ρ for file complexity metrics; Firefox 7..26

evaluation of metrics results from the table marking approach discussed in Section 4.5.

5.3.1 File Level Complexity Metrics

Table 5.8 shows the average Spearman correlations of file complexity metrics with severity quantification metrics for Firefox versions 7 to 26. All correlations have $p < 0.05$.

Version	t_stat	M_vuln	M_neut	Mean Ratio
CountDeclFunction	14.244	46.488	9.948	4.673
CountLineCodeDecl	13.451	275.398	73.338	3.755
MaxNesting	19.974	3.203	1.449	2.211
SumCyclomatic	13.572	204.188	38.911	5.248
SumEssential	13.550	122.037	21.888	5.576

Table 5.9: Welch t_stat measures for top performing complexity metrics; Firefox 7..26

Version	CveCount	AvgScore	MaxScore	TotalScore
CountTouches	0.317	0.006	0.139	0.219
CountTouchesP1	0.301	0.010	0.136	0.210
CountTouchesP2	0.284	0.016	0.134	0.202
HCPF	0.302	0.011	0.117	0.197
LinesAdded	0.305	0.007	0.118	0.202
LinesModified	0.299	0.022	0.106	0.186
LinesDeleted	0.307	0.001	0.125	0.208
TotalTouches	0.273	0.055	0.070	0.138

Table 5.10: Average Spearman ρ for file churn metrics; Firefox 7..26

Table 5.9 shows the average t_stat from Welch's test, along with the average mean for the vulnerable and neutral set. Also shown is the ratio of the vulnerable to neutral mean for the top performing complexity metrics (per test harness described in 5.1).

5.3.2 File Level Churn Metrics

Table 5.10 shows the average Spearman correlations of file churn metrics with severity quantification metrics for Firefox versions 7 to 26.

Table 5.11 shows the average t_stat from Welch's test, along with the average mean for the vulnerable and neutral set. Also shown is the ratio of the vulnerable to neutral mean for file churn metrics.

Table 5.12 shows an X where the $p < 0.05$ value from Welch's t test for churn metrics for the versions 7..26 of Firefox. We note that the p fails to indicate statistical

Version	t_stat	M_vuln	M_neut	Mean Ratio
CountTouches	13.555	4.876	0.517	9.430
CountTouchesP1	13.156	10.023	1.223	8.199
CountTouchesP2	12.811	15.026	1.936	7.760
HCPF	6.988	0.006	0.000	18.811
LinesAdded	6.656	55.947	3.047	18.359
LinesDeleted	5.723	49.298	2.433	20.262
LinesModified	7.262	28.755	1.338	21.487
TotalTouches	12.354	124.238	13.870	8.957

Table 5.11: Welch t_stat measures for file churn metrics; Firefox 7..26

significance among the metrics, in some version slices and therefore may not be suitable for use as a class discriminator.

5.3.3 Module Level Burst and Architectural Metrics

Table 5.13 shows the average Spearman correlations of change burst churn metrics CB(2,2) with severity quantification metrics for Firefox versions 7 to 26. Table 5.14 shows the average Spearman correlations of module metrics with severity quantification metrics for Firefox versions 7 to 26. Table 5.15 shows the average t_stat from Welch's test, along with the average mean for the vulnerable and neutral set. Also shown is the ratio of the vulnerable to neutral mean for module metrics. Table 5.16 shows the results of p value testing from Welch's t test for module metrics. The results indicate applicability for discriminative power only over certain version slices.

5.3.4 Firefox Prediction Experiments

In the limited number of experiments we were able to perform with churn and traditional complexity metrics, we noted a marked difference in prediction results, de-

Version	CountTouches	CountTouchesP1	CountTouchesP2	HCPF	LinesAdded	LinesDeleted	LinesModified	Total Touches
7	X	X	X	0	0	0	0	X
8	X	X	X	0	0	0	X	X
9	X	X	X	0	0	0	0	X
10	X	X	X	0	0	0	0	X
11	X	X	X	X	X	0	X	X
12	0	X	X	0	0	X	0	X
13	X	0	X	X	X	0	X	X
14	0	X	0	0	0	0	0	X
15	X	0	X	X	X	0	0	X
16	X	X	0	X	0	X	0	X
17	X	X	X	0	0	X	0	X
18	X	X	X	X	0	0	0	X
19	0	X	X	0	0	0	X	X
20	0	0	X	0	0	0	0	X
21	X	0	0	0	0	0	0	X
22	0	X	0	0	0	0	0	X
23	X	0	X	0	0	0	0	X
24	X	X	0	0	0	0	0	X
25	0	X	X	0	X	X	0	X
26	0	0	X	0	0	0	0	X
27	0	0	0	0	0	0	0	X

Table 5.12: Results of file churn metric pval tests, 7..26; X: $p < 0.05$

	CveCount	AvgScore	MaxScore	TotalScore
CountTouches	0.387	0.071	0.197	0.368
CountReversals	0.361	0.059	0.192	0.344
MaxSeqByDate	0.335	0.082	0.175	0.324
CountBursts	0.373	0.07	0.194	0.356
CountFilesChanged	0.403	0.11	0.231	0.395
NetAddedInBurst	0.396	0.1	0.236	0.39
TotalChurnInBurst	0.397	0.099	0.233	0.389

Table 5.13: Average Spearman ρ for change burst metrics CB(2,2); Firefox 7..26

	CveCount	AvgScore	MaxScore	TotalScore
APIU	0.272	0.298	0.036	0.16
k_ext_fa	.096	0.32	0.172	0.173
k_ext_m	0.167	0.206	0.163	0.22
MII	0.105	0.18	0.025	0.061
N_API	0.38	0.256	0.331	0.427

Table 5.14: Average Spearman ρ for module metrics; Firefox 7..26

Version	t_stat	M_vuln	M_neut	Mean Ratio
APIU	-0.467	0.036	0.047	0.762
k_ext_fa	1.222	53.989	13.352	4.044
k_ext_m	0.600	456.671	306.405	1.490
MII	2.352	0.233	0.085	2.749
N_API	2.148	507.838	112.633	4.509

Table 5.15: Welch t_stat measures for module metrics; Firefox 7..26

Version	APIU	k_ext_fa	k_ext_m	MII	N_API
1	0	0	0	0	0
7	0	0	0	0	0
8	0	0	0	0	0
9	0	0	0	0	0
10	0	0	0	0	0
11	0	0	0	0	0
12	0	0	0	0	0
13	0	0	0	0	0
14	0	0	0	0	0
15	0	0	0	0	0
16	0	0	0	0	0
17	0	X	0	X	0
18	0	X	0	X	0
19	0	X	0	X	0
20	0	X	0	X	0
21	0	X	0	X	0
22	0	X	0	X	0
23	0	X	0	X	0
24	0	X	0	X	0
25	0	X	0	X	0
26	0	X	0	X	0
27	0	X	0	X	0

Table 5.16: Results of module metric pval tests, Firefox 7..26; X: $p < 0.05$

spite the churn metrics p value from the Welch test exceeding 0.05.

5.3.4.1 Firefox experiment comparing complexity and churn, $K = 3$

Table 5.17 shows prediction performance of our prediction test harness selecting the best three complexity features and performing vulnerability predictions for Firefox versions 7 to 26. The three best selected features were, in order: SumCyclomaticModified, SumEssential, and CountDeclFunction.

Table 5.18 shows prediction performance of our prediction test harness selecting the best three file churn features and performing vulnerability predictions for Firefox versions 7 to 26. The features selected were, in order: CountTouches, CountTouchesP1, and TotalTouches.

Comparing predictions from Table 5.17 and Table 5.18, we note generally higher precision values and tighter variance in prediction accuracy within 2σ, or the 95% confidence interval.

5.3.4.2 Firefox experiment comparing complexity and churn, $K = 5$

We increased K from three to five and re-ran the experiment described in Section 5.3.4.1. The model using only complexity metrics selected, in order: SumCyclomaticModified, CountDeclFunction, SumCyclomatic, SumEssential, and CountStmtDecl. Table 5.19 shows the mean precision and the accuracy of mean precision inside the 95% confidence interval.

The model using churn metrics selected, in order: TotalTouches, CountTouches, CountTouchesP1, CountTouchesP2, and HCPF. Table 5.20 shows the mean precision and the accuracy of the mean precision inside the 95% confidence interval. The comparison of the two results shows a marked improvement using the churn based

Version	M_{prec}	**+/− 95%**
7	0.60	0.17
8	0.48	0.51
9	0.47	0.41
10	0.57	0.41
11	0.62	0.10
12	0.61	0.15
13	0.62	0.11
14	0.68	0.11
15	0.58	0.25
16	0.62	0.16
17	0.51	0.24
18	0.55	0.28
19	0.54	0.13
20	0.61	0.16
21	0.59	0.12
22	0.58	0.23
23	0.44	0.39
24	0.53	0.21
25	0.50	0.25
26	0.48	0.19

Table 5.17: Mean precision (M_{prec}) performance using complexity metrics; $K = 3$; Firefox versions 7..26

Version	M_{prec}	**+/− 95%**
7	0.69	0.13
8	0.69	0.10
9	0.67	0.11
10	0.71	0.12
11	0.71	0.08
12	0.68	0.09
13	0.73	0.10
14	0.73	0.11
15	0.69	0.09
16	0.70	0.12
17	0.70	0.11
18	0.70	0.09
19	0.66	0.10
20	0.67	0.08
21	0.67	0.12
22	0.65	0.08
23	0.66	0.10
24	0.63	0.12
25	0.59	0.09
26	0.65	0.11

Table 5.18: Mean precision (M_{prec}) performance using file churn metrics; $K = 3$; Firefox versions 7..26

Version	M_{prec}	+/– 95%
7	0.60	0.25
8	0.64	0.11
9	0.64	0.17
10	0.66	0.14
11	0.63	0.12
12	0.63	0.14
13	0.68	0.13
14	0.71	0.12
15	0.66	0.11
16	0.64	0.10
17	0.55	0.33
18	0.56	0.19
19	0.56	0.15
20	0.60	0.18
21	0.59	0.10
22	0.60	0.15
23	0.43	0.48
24	0.42	0.29
25	0.36	0.42
26	0.47	0.26

Table 5.19: Mean precision (M_{prec}) performance using file complexity metrics; $K = 5$; Firefox versions 7..26

learner. These results are particularly impressive, but require additional validation to ensure that the learning model isn't falling prey to the pitfalls related to measuring vulnerability oracles as discussed in Section 4.5.1. In Section 5.3.5, we review Firefox release numbering as it relates to our techniques to build our training data to ensure we have not unwittingly introduced a vulnerability oracle.

5.3.5 Firefox Release Correspondence to Collected Data

Although the nightly labels are each about a month apart, the calendar duration between the nightly label (NB) and the corresponding release label (RB) may vary anywhere from three to six months. Note that the span between NB and RLS versions

Version	M_{prec}	+/– 95%
7	0.69	0.13
8	0.69	0.10
9	0.67	0.10
10	0.69	0.11
11	0.69	0.08
12	0.69	0.08
13	0.75	0.09
14	0.73	0.10
15	0.70	0.10
16	0.69	0.12
17	0.69	0.10
18	0.70	0.10
19	0.67	0.10
20	0.67	0.07
21	0.67	0.12
22	0.68	0.07
23	0.65	0.08
24	0.62	0.12
25	0.59	0.08
26	0.64	0.13

Table 5.20: Mean precision (M_{prec}) performance using file churn metrics metrics; $K = 5$; Firefox versions 7..26

Date	Version Label	Label Type
04/12/11	6.0a1	NB
05/24/11	7.0a1	NB
06/21/11	5.0	RLS
07/05/11	8.0a1	RLS
08/16/11	6.0	RLS
08/16/11	9.0a1	NB
09/27/11	7.0	RLS
09/27/11	10.0a1	NB
11/08/11	8.0	RLS
11/08/11	11.0a1	NB
12/20/11	12.0a1	NB
01/31/12	10.0	RLS

Table 5.21: Firefox release (RLS) versions vs. first tagged nightly build (NB) versions, ordered by date

is greater than the typical month and a half, or six week release interval following Mozilla's Rapid Release Calendar. Table 5.21 shows an example four month span between April 12, 2001, when the 6.0a1 label was assigned, and August 16, 2011 when version 6.0 was released. We also show a simplified depiction of the relative version sequencing in Figure 5.5; simplified, because we've omitted branch detail for visual clarity.

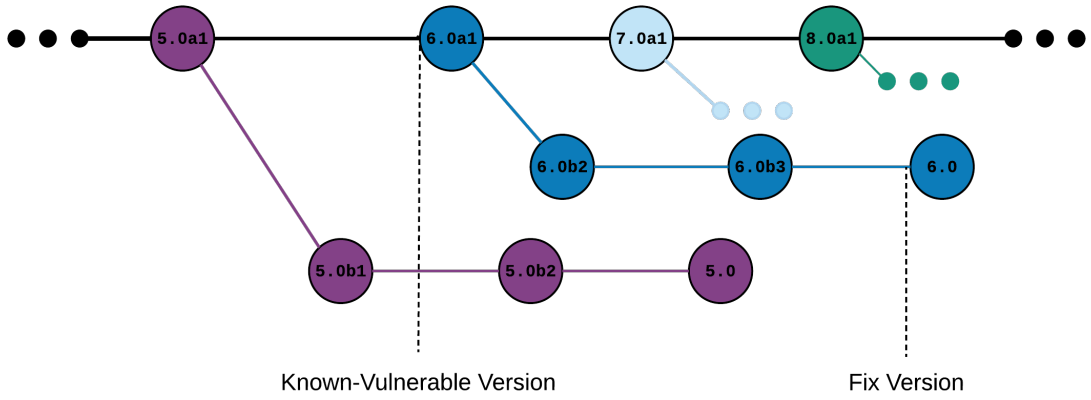

Figure 5.5: Simplified visualization showing nightly version 6.0a1 to release version 6.0

5.3.6 Discussion

Our automated mining techniques result in prediction between the nightly build version and the final release version as discussed in Section 5.3.5. In the construction of our training data, we were careful compare the set of files existing in the first prior nightly build (NB), or the X.0a1 version from the Mozilla Central development trunk, to those files that were *fixed* for a vulnerability in a particular release X. That is, we measure on NB X.0a1, and then compare predictions to know fixes on RB X.

Table 5.21 shows the sequencing of nightly build (NB) versions, (labeled as X.0a1),

relative to the release (RLS) versions (labeled as X.0). For example, NB version 6.0a1 corresponds to RLS version 6.0. Note that the FixVersion, or the version mined from associated ITS reports, corresponds to the RLS version and not the NB version. As such, we believe the development occurring on the main line of development (i.e. "master", or "trunk") provides the needed separation to avoid measuring security fixes as vulnerability oracles as discussed in Section 4.5.

As stated, we mined our VCS based metrics from the `git` repository commit revision corresponding to the NB, and not the RLS label. We also extracted the NB versions as individual snapshots for the collection of conventional complexity metrics, such as shown in Table . In similar form, the changes that define our period based frequency metrics (e.g. file level churn and module level change bursts) were collected from the respective nightly label.

Given the differences in our collection and comparison approach, our numbers may not be directly comparable to past research studying Firefox. First, the versions in our study are years more recent. Second, with the exception of CveCount, we don't evaluate correlation relative to frequency based counts. Third, we have introduced new measurements according to an entirely new characterization of severity. With all the preceding caveats stated, we note relatively higher correlation values for our file complexity results when compared to that reported by Shin [81]. Top Spearman correlation values in her study of Firefox are set next to the correlation values reported for CveCount earlier in this section.

5.4 Case Study 2: Apache Web Server

We studied Apache HTTP Server versions 2.2.0 to 2.2.29, representing a time span of over eight and a half years from December, 2005 to August, 2014. We studied the

Metric	CveCount Corr	Shin Corr
CountDeclFunction	0.260	0.159
LOC	0.278	0.170
CountLinePreprocessor	0.260	0.183
MaxCyclomaticStrict	0.193	0.132
MaxNesting	0.190	0.106
SumCyclomaticStrict	0.255	0.151
SumEssential	0.268	0.154

Table 5.22: Firefox Spearman correlation comparison

Number of Websites	Apache Version
22,970,250	2.2
8,098,197	2.4
328,257	2.0
307,879	1.3

Table 5.23: Apache adoption by version [50]

micro (patch) versions of Apache HTTP Server 2.2.x. Table 5.23 shows the adoption of the 2.2 version as of this writing, thus reinforcing our decision to study 2.2.x.

Table 5.24 provides an overview of size measurements across the first, middle, and last versions studied. The measurements shown here were collected from the SciTools Understand tool and exclude documentation and test folders in the project's source code tree. Note that we've excluded the count of methods from this table because Apache HTTP is implemented exclusively in the C programming language (i.e., contains functions only).

As shown by Table 5.24, the various sizes are relatively stable across the versions studied. There were 18 files added between version 2.0.0 and 2.2.29, or a 6% increase overall. Likewise, there is an approximate 7% increase in the number of lines between versions 2.2.0 and 2.2.9.

Version	LOC	CountFunctions	CountFiles
2.2.0	113,138	2,799	305
2.2.15	118,183	2,898	309
2.2.29	120,950	2,977	323

Table 5.24: Apache HTTP Server project statistics

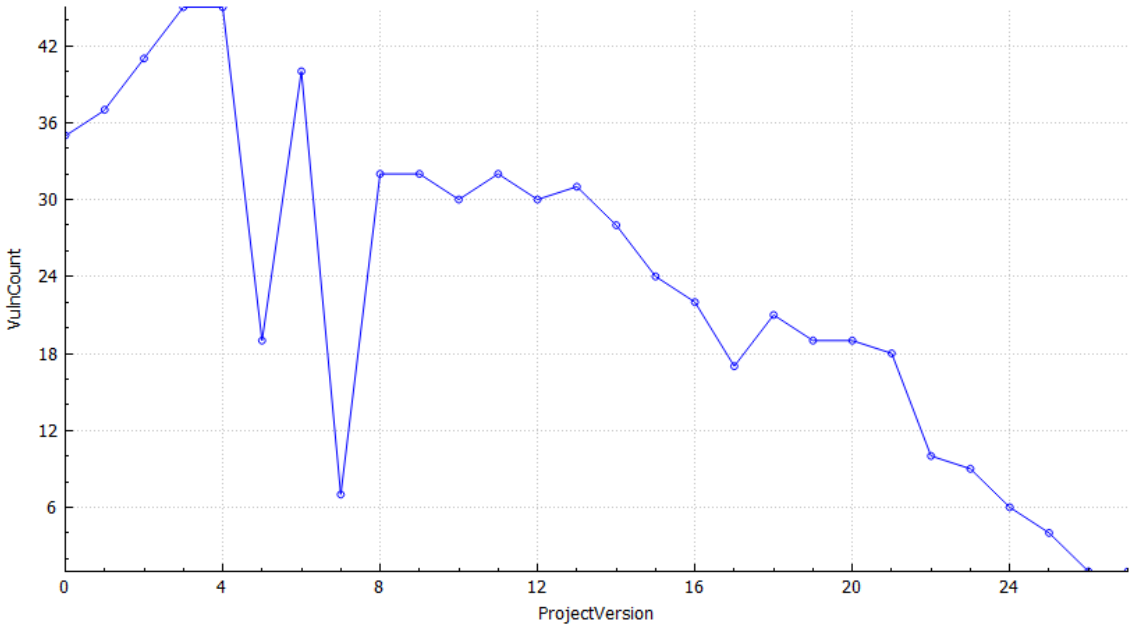

Figure 5.6: Historical vulnerability density in Apache HTTP versions 2.2.x, 0..29

	CveCount	AvgScore	MaxScore	TotalScore
AltAvgLineCode	0.036	0.041	0.135	0.079
AltCountLineCode	0.037	0.116	0.144	0.086
AvgCyclomatic	0.011	0.025	0.06	0.008
AvgCyclomaticModified	0.028	0.034	0.06	0.015
AvgCyclomaticStrict	0.01	0.005	0.075	0.003
AvgEssential	0.075	0.092	0.018	0.097
CountDeclFunction	0.043	0.13	0.173	0.097
CountLineCode	0.04	0.152	0.167	0.108
CountLineCodeDecl	0.007	0.05	0.099	0.026
CountLineCodeExe	0.034	0.157	0.158	0.107
CountLineComment	0.133	0.053	0.052	0.082
CountLinePreprocessor	0.017	0.143	0.072	0.082
CountStmt	0.016	0.14	0.156	0.082
CountStmtDecl	0.007	0.036	0.087	0
CountStmtExe	0.041	0.151	0.153	0.105
MaxCyclomatic	0.144	0.129	0.065	0.051
MaxCyclomaticModified	0.118	0.115	0.059	0.042
MaxCyclomaticStrict	0.085	0.141	0.113	0.012
MaxEssential	0.032	0.182	0.11	0.106
MaxNesting	0.028	0.197	0.173	0.147
RatioCommentToCode	0.113	0.067	0.084	0.119
SumCyclomatic	0.056	0.172	0.188	0.133
SumCyclomaticModified	0.05	0.162	0.18	0.127
SumCyclomaticStrict	0.05	0.158	0.173	0.118
SumEssential	0.039	0.171	0.159	0.11

Table 5.25: Average Spearman ρ for file complexity metrics; Apache HTTP 2.2.0..29

5.4.1 File Level Complexity Metrics

Table 5.25 shows the average Spearman correlations of file complexity metrics with severity quantification metrics for Apache HTTP versions 2.2.0..29. All correlations have $p < 0.05$.

Table 5.26 shows the average t_stat from Welch's test, along with the average mean for the vulnerable and neutral set. Also shown is the ratio of the vulnerable to neutral mean for the top performing complexity metrics (per test harness described in 5.1).

Version	t_stat	M_vuln	M_neut	Mean Ratio
CountDeclFunction	3.616	28.027	7.679	3.650
CountLineCode	3.652	979.564	326.536	3.000
MaxNesting	4.119	4.082	2.197	1.858
SumCyclomatic	3.786	180.744	48.771	3.706
SumEssential	3.783	101.580	27.256	3.727

Table 5.26: Welch t_stat measures for top performing complexity metrics; Apache versions 2.2.x..29

	CveCount	AvgScore	MaxScore	TotalScore
CountTouches	0.185	0.134	0.065	0.044
CountTouchesP1	0.168	0.109	0.061	0.050
CountTouchesP2	0.168	0.108	0.064	0.052
HCPF	0.175	0.083	0.007	0.069
LinesAdded	0.180	0.091	0.020	0.073
LinesModified	0.166	0.061	0.010	0.076
LinesDeleted	0.138	0.087	0.014	0.061
TotalTouches	0.040	0.156	0.134	0.083

Table 5.27: Average Spearman ρ for file churn metrics; Apache versions 2.2.x..29

5.4.2 File Level Churn Metrics

Table 5.27 shows the average Spearman correlations of file churn metrics with severity quantification metrics for Apache HTTP versions 2.2.x..29

Table 5.28 shows the average t_stat from Welch's test, along with the average mean for the vulnerable and neutral set. Also shown is the ratio of the vulnerable to neutral mean for file churn metrics.

5.4.3 Module Level Burst and Architectural Metrics

Table 5.29 shows the average Spearman correlations of change burst churn metrics CB(2,2) with severity quantification metrics for Apache HTTP versions 2.2.x..29. We note that CountBursts shows relatively high correlation with AvgScore, which is one of the more conservative estimates of overall severity, since it is averaged over the

Version	t_stat	M_vuln	M_neut
CountTouches	2.328	0.761	0.204
CountTouchesP1	2.109	4.713	1.510
CountTouchesP2	2.156	8.737	2.815
HCPF	0.869	0.033	0.005
LinesAdded	0.858	3.577	0.351
LinesDeleted	0.173	0.833	0.229
LinesModified	0.737	1.268	0.306
TotalTouches	5.097	136.313	43.138

Table 5.28: Welch t_stat measures for churn metrics; Apache versions 2.2.x..29.

	CveCount	AvgScore	MaxScore	TotalScore
CountTouches	0.378	0.352	0.189	0.315
MaxSeqByDate	0.364	0.332	0.189	0.315
CountBursts	0.277	0.252	0.143	0.249
CountFilesChanged	0.275	0.251	0.158	0.247
NetAddedInBurst	0.252	0.218	0.149	0.244
TotalChurnInBurst	0.259	0.224	0.152	0.25

Table 5.29: Average Spearman ρ correlations for burst metrics; Apache HTTP versions 2.2.x..29.

individuals within the module.

Module metrics for Apache are withheld. Apache HTTP uses many C preprocessor macros and additional add on packages in order to build the software. Being focused on automated techniques, we did not manually provide any of the additional information to help the Understand tool better resolve symbols needed to compile Apache HTTP. Since we did not manually resolve or provide additional resolution for the Understand tool, only k_ext_m, total calls from other modules, appeared in the data, and did so with a suspect correlation of 0.5. We posit that conditionally compiled code regions were double-counted. We describe additional detail related to the pitfalls when calculating module metrics that result from the lack of precision in our automated methods in Section 4.6.

5.4.4 Discussion

An outstanding result of our experiments with Apache HTTP is the high correlation between the burst metrics and our severity quantification metrics (i.e., AvgScore, MaxScore, and TotalScore). This result is especially interesting because the correlation is assessed on a subset of modules already known to be vulnerable (i.e., Vuln = 1). In this context, we *are not* comparing the metrics' discriminative power, but are assessing their relationship to the *severity* of those known-vulnerable modules (i.e., labeled Vuln = 1). Said another way, the correlation values we are presenting in this particular discussion *do not* reflect the respective metrics' *classification* ability, but are presented as a *relative quantification of severity*.

In particular, as shown in Figures 5.7 and 5.8, CountFilesChanged and TotalChurnInBurst show rank order correlation values as high as 0.90 with our aforementioned severity quantification metrics. Figure 5.7 and Figure 5.8 depict a strong correlation in one or more module metrics we can use to enhance our learner.

Initial prediction experiments with using complexity features for vulnerability prediction in Apache HTTP showed terrible performance. Mean precision was consistently below 0.40, with large accuracy values (> 0.30) required to achieve the 95% confidence interval, suggesting a prediction model built with complexity features alone would perform poorly for predicting vulnerabilities in Apache HTTP.

Noting the high correlation between the number of files changed within a burst, and a known vulnerable module, we added an additional score feature used to train our prediction model. When training the model, we cross-reference the file to the module of which it is a member. We then query the TEDB for the burst features associated with that module, extract CountFilesChanged, extending our feature set. Over 100 iterations of a ten-by-ten cross validation, our data show that this change

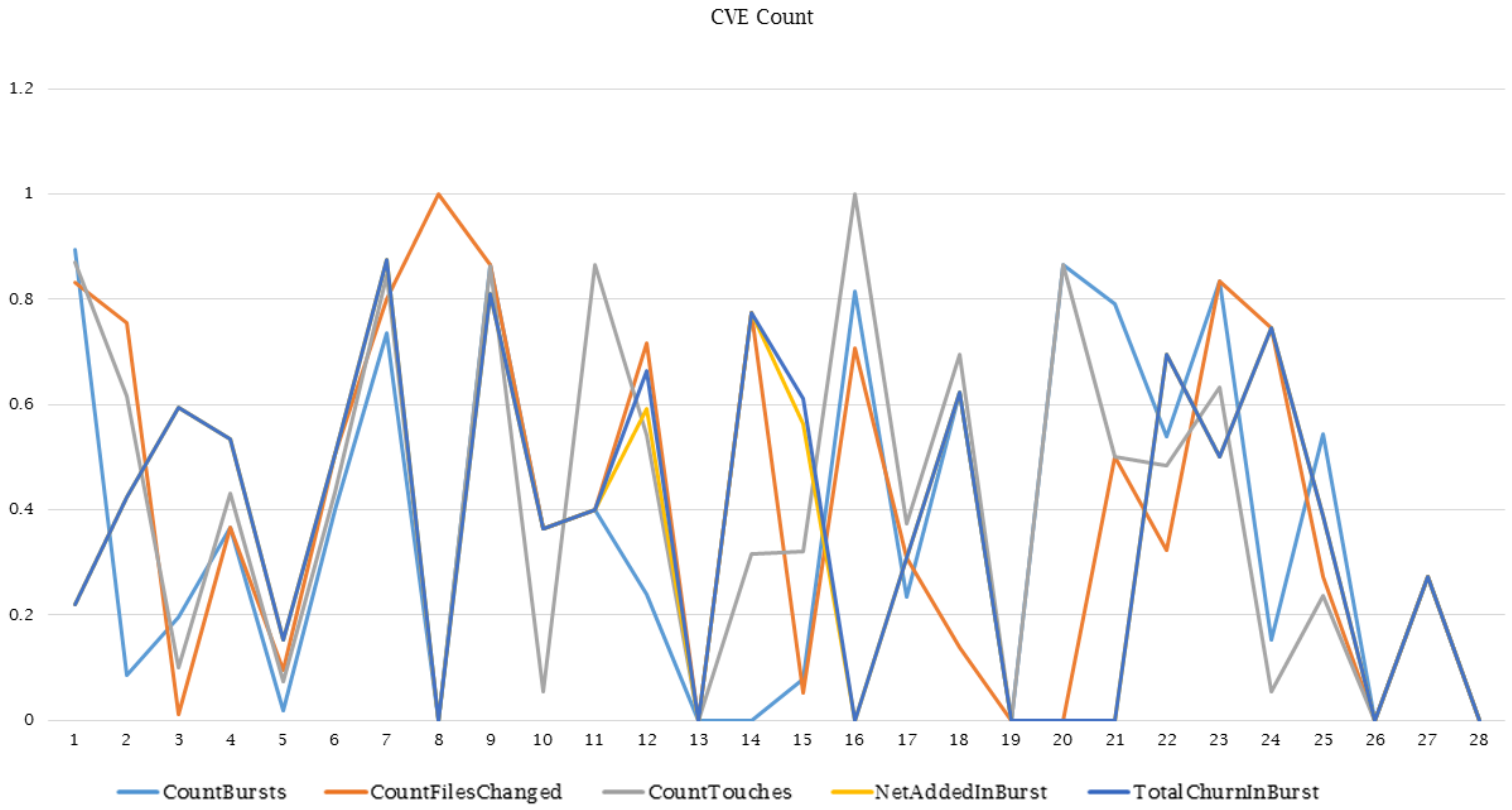

Figure 5.7: Spearman correlation for module level change burst metrics with CVE count; Apache HTTP 2.2.x versions

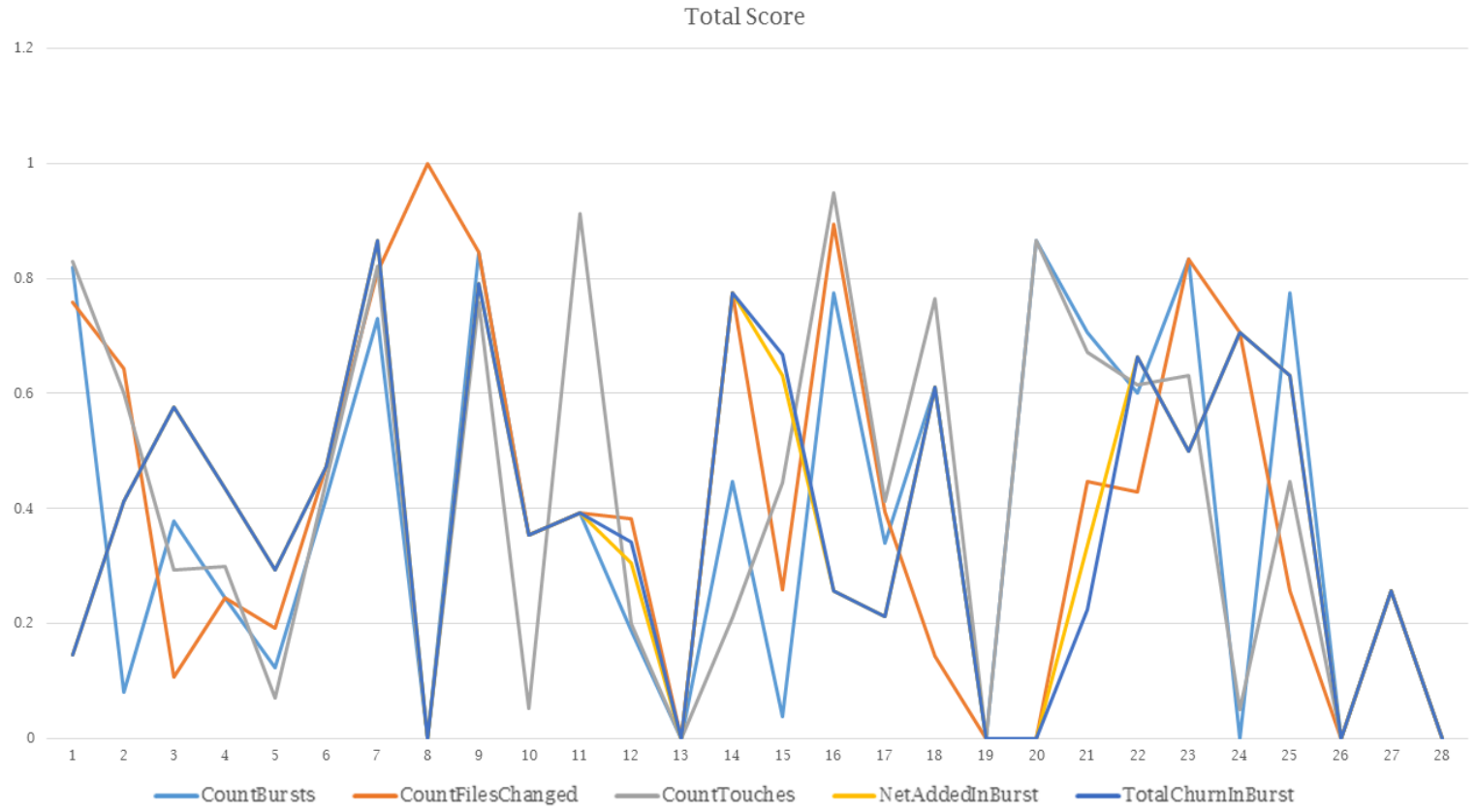

Figure 5.8: Spearman correlation for module level change burst metrics with Total, from CVE entries associated with vulnerable modules in Apache HTTP 2.2.x versions

Version	MP_A	Accuracy A	MP_B	Accuracy B
3	0.57	0.35	0.59	0.25
4	0.55	0.46	0.60	0.37
5	0.58	0.26	0.61	0.21
6	0.54	0.46	0.59	0.37
7	0.46	0.31	0.53	0.26
8	0.55	0.25	0.49	0.28
9	0.55	0.23	0.49	0.35
10	0.38	0.18	0.58	0.25

Table 5.30: Comparison of mean precision and its accuracy over early Apache versions; 2.2.3..2.2.10.

results in improved mean precision, while also narrowing the standard deviation across samples of predicted results in versions 3 through 7. Narrowing the standard deviation improves our 95% confidence interval around the mean precision. Table 5.30 shows the comparison between mean precision evaluated with 10x10 cross fold evaluation with an ANOVA selection filter set for the top five features. Group B (MP_B and Accuracy B) reflects the module metric enhancement.

The result is compelling because the single addition of CountFilesChanged is a simple change, yet results in noticeable prediction performance improvement. One can envision much more elaborate and well designed scoring functions that make better use of our severity quantification measures during training. Moreover, due to the high correlation between CountFilesChanged and severity, as quantified by the sum of CVSS scores from the respective module, it logically follows that correctly predicted results will be of greater significance, as argued throughout this work.

Yet another strong trend exhibited for Apache was that of the churn metrics CountTouchesP1 and CountTouchesP2. Figure 5.9 shows what appear to be strong correlation values in the more vulnerable versions of Apache HTTP server. Spikes in vulnerability fix activity at 12, 22, and 28 are preceded by observable spikes in correlation between churn metrics and Average Score. This correlation with our

quantification of severity is interesting because average score is the most conservative of our CVSS based quantification metrics.

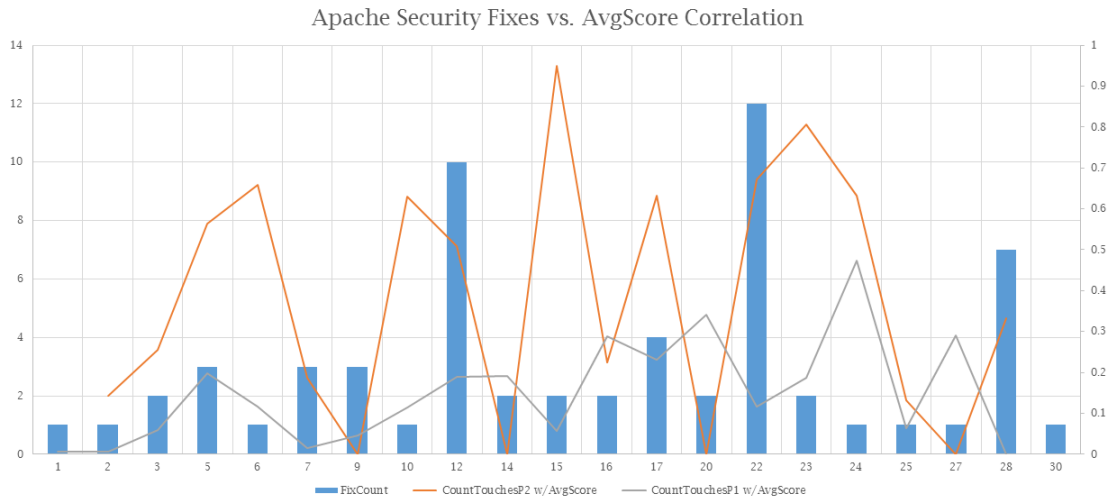

Figure 5.9: Churn metric correlation in vulnerable versions of Apache 2.2.x

5.5 Case Study 3: MySQL Database Server

We studied Oracle MySQL Database Server versions 5.5.0 to 5.5.54, representing a time span of approximately seven years from December, 2009 to November, 2016. Table 5.31 provides an overview of size measurements across the first, middle, and last versions studied. The measurements shown here were collected from the SciTools Understand tool and exclude documentation and test folders in the project's source code tree.

An interesting trend in MySQL that differs from our other case studies is the decreasing size in LOC. There is approximately a 9% reduction in LOC between versions 5.5.0 and 5.5.54. This reduction stands in contrast to our other study subjects

Version	LOC	CountFunctions	CountMethods	CountFiles
5.5.0	741,042	27,292	17,510	2,015
5.5.28	670,181	28,196	17,480	2,058
5.5.54	673,662	28,293	17,509	2,061

Table 5.31: MySQL Database project statistics

that both show increases in LOC between their first and last versions. We do note, however, that despite decreases in LOC, MySQL shows marginal (approximately 4%) increase in the number of its functions and files (only a 2% increase). Assuming the software at version 5.5.54 represents identical or increased functionality, the reduction in LOC and increase in files could represent refactoring activity to remove duplicate code and/or increase the use of templatized or auto-generated code.

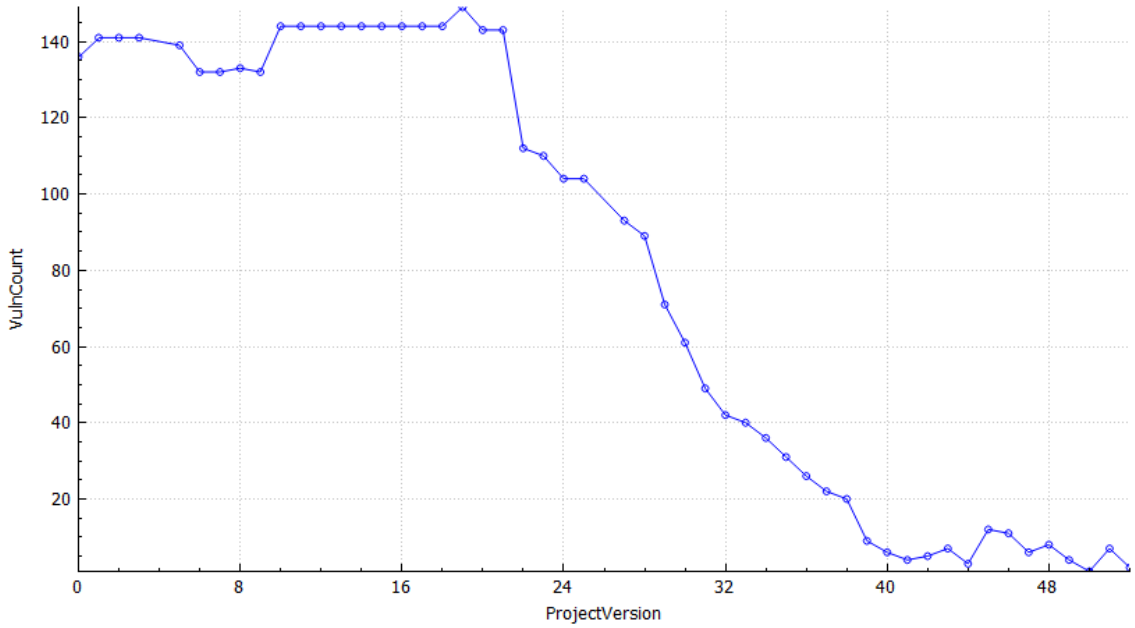

Figure 5.10: Historical vulnerability density in MySQL DB versions 5.5.x, 0..50

As shown from Figure 5.10, the vulnerability distribution is heavily oriented at version 5.5.0. This caused us to restart our automated data mining approach for MySQL, starting at 5.4.9.

Due to differences in the way in which the vulnerability table was built from the VCS mined entries for MySQL, we discovered the results produced to be inaccurate upon analysis. Final results are withheld for a future addendum.

5.6 Discussion

A review of the literature, along with empirical data collected by our study, indicate that the majority of files in a project are only involved in one vulnerability in their lifetime. Note that this doesn't contradict our Previously Vulnerable Assumption. Although one can impart a greater penalty factor to files already known vulnerable, such as when performing a scored ranking of predicted results, the Previously Vulnerable Assumption doesn't make any assertion about future events. However, what we can assert, and what our empirical data support, is that a software project's true vulnerability density (which can never really be known), is better approximated by applying the Previously Vulnerable Assumption to mark files as vulnerable in earlier versions.

We reason therefore, and provide evidence here, that metrics better reflecting a software project's inherent architectural properties, and that also correspond with earlier lifecycle development activity, are better at approximating that project's *Vulnerability Signal*. We define the Vulnerability Signal as a current pre-image of future security fixes likely to be needed for a project, following from that project's true vulnerability density. Every time a security fix is made to a software project, we learn new information that leads us closer to approximating the true vulnerability density. We reason that metrics "locking on" to a projects vulnerability signal at earlier stages will aid earlier detection. Because there is already substantial empirical evidence suggesting that standard issue (i.e., not vulnerability specific) defect fixes

are tightly correlated with development activity, it would stand to reason that some subset of future fix activity will be needed for the correction of residual vulnerabilities.

Metrics better corresponding to a time-based view of a projects' development history are therefore likely to better approximate this its true vulnerability signal. Our data and visualizations support the notion that applying the Previously Vulnerable Assumption to retroactively mark the training set has the advantage of re-evaluating interconnected components in situ, that is, in the context of their architecture and interconnections with other components in earlier stages. In addition, the manner in which a known-vulnerable entity is integrated with it's surrounding architecture, make the negative ramifications of that single vulnerability more significant. Therefore, we view measurement and evaluation of vulnerability prediction metrics better estimated with their relationship to overall severity, and have provided an example of one approach using CVE scoring data to better quantify severity.

Conventional supervised machine learning model building approaches focus on metric correlation with vulnerability frequency (i.e. RawCount). In these models, metrics for files in version N are evaluated for their correlation against RawCount in version $N + 1$. That is, the metrics used as features in such models are selected because of their correlation against the probability of a future security fix. For some version N, the probability of a future security fix in version $N + 1$ is estimated as relative frequency with which files in N (for some known set, labeled with known values from $N + 1$ for training) will undergo a vulnerability fix in version $N + 1$. In these conventional models, repeat offenders, or files that are known to have been a part of more than one security issue fix, will show greater probability because their RawCount is higher relative to the other vulnerable files known in version N. This is most similar to evaluation of RawCount in our labeled data. As shown in Figure

5.11, between versions 6 and 49, there are well over a hundred files that are only ever involved in a single security issue fix, while ten files are involved in two issues, and three files involved in three security fixes.

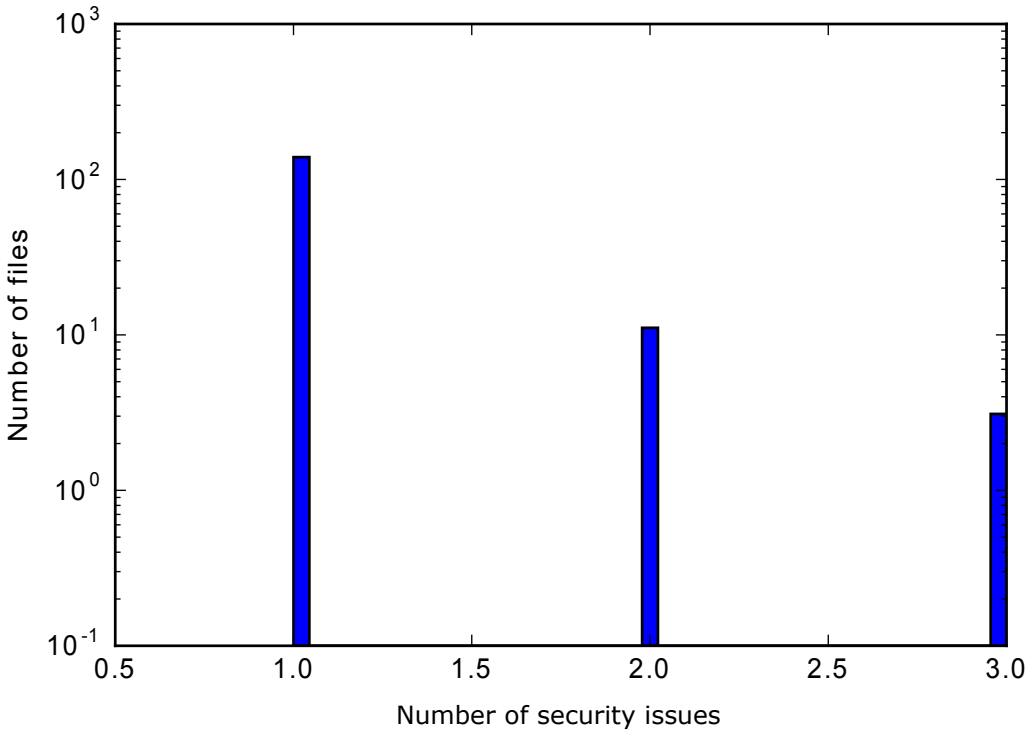

Figure 5.11: Repeat offenders; Firefox versions 6..49

Metrics selected based on linear correlation (i.e., Pearson) or rank order correlation (i.e. Spearman) RawCount, suffer to capture the true residual vulnerability density and corresponding severity of vulnerabilities latent in the released software product. This shortcoming in conventional approaches may be heightened when using automated techniques for model building as we have described in this work. Recall that the vulnerability signal, as we define it, is a property of the overall *true* vulnerability density of a software. While we can never really know the true vulnerability density, assuming we have the appropriate repositories available, we can

better approximate a software's true vulnerability density as security fixes are made over time. That is, every time a released software product is fixed for a security vulnerability, we learn new information about the source code entities involved.

Our experience shows this phenomenon exacerbated when used with automated mining approaches that fail to appropriately pre-filter the source code included in the labeled Vuln = 1 set. As we observed in our experience with Firefox, test files (e.g., updated for regression tests) are often updated in tandem with the set of files needed to fix the vulnerability in the released product. Test files, while part of the development repository, are not included in the final binary released to a software product's end users. Such test files therefore have no bearing on residual vulnerabilities (and their associated severity). Mining and supervised learning approaches that fail to pre-filter such test files will therefore tend to include them in their predicted results.

Using test files as an example, model building techniques that fail to filter test files from their training sets would consequently include such files in their predicted results. The precision and recall rates reported for such learners might appear impressive, but *would fail to accurately reflect the true distribution of residual vulnerabilities in the released software product.* As discussed in the preceding paragraphs, test files are not representative of the product's true vulnerability density once it reaches its user base. Worse yet, because empirical data show that the majority of files that are fixed to correct a vulnerability, are most often done so one time, the inclusion of test files may artificially inflate recall rates and reduce overall precision when said test files are also repeat offenders; in other words, test files would "dilute" the practical application of the predicted results.

The inclusion of test files in predicted results is not only inaccurate with respect each individual vulnerability prediction, but it also prevents the learner from locking

on to the true vulnerability signal, which as we reasoned earlier, is *not* accurately characterized by repeat offenders alone (i.e., RawCount based). Empirical data repeatedly show an order of magnitude more files individually associated with single security fixes (i.e., 1:1), than are associated with multiple security fixes (i.e., 1:N). For a given software product on which we wish to perform vulnerability prediction, if we assume such empirical observations are some approximation of the real population of residual vulnerabilities, then the resulting models (built with RawCount estimators) would suffer to solve the real problem of predicting a wider range of entities responsible for residual vulnerabilities because they favor a minority (repeat offenders) within the vulnerable file population. Our experience with automatically mined test files being included in the vulnerable file set (i.e., labeled Vuln = 1), informs our claims and also serve as an intuitive example where non-essential, less relevant items, dilute model performance.

Chapter 6

Contributions and Future Work

This chapter summarizes our research problem: *the residual vulnerability problem*, or the latent persistence of security related defects, termed *vulnerabilities*, remaining in software after its release. We review the problem's significance as well as the significance of our work in this context. We also highlight the creative, original, and novel aspects of our research contribution and findings. We conclude this chapter with caveats and suggestions for future research.

6.1 Contributions to Residual Vulnerability Prediction in Software

Our emphasis on more accurately characterizing the residual vulnerability problem and focus on practical automated methods for better quantifying the severity of vulnerability impact, leads directly to several of our significant contributions. A brief description of each follows, with references as appropriate for additional detail.

A new approach for metric selection based on correlation with quantified measures of severity, for use in software vulnerability prediction models implemented with supervised machine learners. Through careful explanation and reasoning about

the residual vulnerability problem, we provide a completely new ontology that better *characterizes the relative value of predictions by evaluating their correlation with quantified measures of severity* based on CVE data.

A practical and automated approach to training set construction is presented according to the new approach for metric selection, demonstrating the feasibility of of automating the construction of a training set that is more representative of the characteristic relationships between measurable features and our ontology for learner evaluation. Specific techniques are discussed in Sections 4.4, 4.5, and 4.6.

New insights revealed by our mined data lead to new ways of thinking about vulnerability density over time in a software product. Hence, entirely new concepts and additional explanations for previously observed phenomena emerge on review of mined data (i.e., mined according to our approach for automated training set construction). Specifically, this work discusses the following new concepts:

- **Previous Vulnerability Assumption** introduced as a new concept to aid training set construction, through careful study of CVE vulnerability data available from the NIST National Vulnerability Database for the software projects studied in this work.
- **Vulnerability Signal** introduced as a new concept in 4.5, and empirically demonstrated in Section 5, as the top edge defining a software project's historical vulnerability distribution that appears when visualizing the count of unique CVE entries against affected versions of a software product over time.
- **Fix Signal and Vulnerability Oracle** introduced as a new observed phenomena in sections 4.5.1 and 5 that unify observations from past vulnerability predic-

tion research; specifically the phenomena observed in our study better explain the need for LOC correction and cross-correlation compensation when performing prediction studies.

Empirical data supporting the value of historical evaluation is provided by way of explanation and experiments in Chapter 5. Primarily, there are more samples on which to train a machine learner when the Previously Vulnerable Assumption holds for a software project. We argue that under certain conditions, this approach better approximates the true population of vulnerabilities in the software, and by extension, provides a better test environment in which to select features in order to build and evaluate vulnerability prediction models.

6.2 A Timely Contribution

This work is especially timely, as one only needs to look at the non-stop stream of news headlines publicizing the latest exploits and cyber attacks, testifying to the fact that the state of computer and network security is not improving. In fact, one of the core issues in computer security, one of the central issues enabling modern attacks, is the insecurity of modern software [39]. Crypto-analytic attacks, hardware-based attacks (such as side channel attacks) withstanding, the literature shows, and industry widely recognizes, software borne vulnerabilities as a prime culprit responsible for the majority of modern cyber security attacks against computers and information networks. This state of affairs is at odds with the continual push within the software industry to deliver compelling value added features. The current state of affairs within the software development domain grows ever more acute as software continually grows ever larger and more complex.

The pace of development, increasing complexity, and ever growing distributed nature of software products is at odds with classic security design principles such as economy of mechanism and central mediation, making software more difficult to comprehend when making changes, and more difficult to exhaustively test. Compounding these factors are the forces vying for limited developer time. The commercial sector, finds a feature-centered development culture where security concerns often take a back seat to differentiating features; the return on investment and impact to the bottom line resulting from the differentiating features is more deterministic [53]. It's not that the commercial sector doesn't care about security, but more about spending resources efficiently. Within the software development industry, developer time is a constrained, highly contended, and highly demanded resource. Management therefore wants to make efficient and effective use of developer time. This is the larger context and commercial motivations that just so happen to be aligned against security improvement rather than with it. That is, given that residual vulnerabilities are already known to be difficult to detect, what can be done to motivate commercial development organizations to expend effort, appropriating development resources to search for what is commonly known as "a needle in a haystack"? The answer, from the perspective of this work, is to provide a superior approach to building vulnerability prediction models by quantifying severity and seeking out metrics that show correlation with the same.

We are confident that models built on such relationships will further enable what we refer to as suspect prioritization, such that the "top 10", or top suggested vulnerability predictions are likely to represent vulnerabilities that are exploitable and have the potential to cause severe security violations, compromising confidentiality, integrity, or availability. Ideally, the prioritized "top 10" should be free of false positives, thereby maximizing the efficacy of any resulting triage and inspection efforts

when applied. Although the empirical results from our case studies did not show stable mean precision values free of such false positives, we believe the framework presented for metric evaluation provides a better approximation of, and is more conservative with respect to the true vulnerability density, and vulnerability distribution across files and modules in a software project. Using the framework we presented for metric evaluation, we were able to show that automatically mined, architecture and change-based burst metrics selected for our study outperformed conventional metrics in earlier characterization of severity. We therefore believe architectural level and frequency based change metrics provide additional value as features to consider when building supervised machine learning models for vulnerability prediction.

Our work represents an important advancement in the area of vulnerability prediction and we are already at work on future improvements. We remind the reader of some key points from our work, because looking at the experimental numbers alone fail to properly characterize the significance of our results. First, we largely used *an automated approach to data collection and classification*. Second, aside from the commercial SciTools Understand tool we used to collect conventional metrics (and compute call based metrics), the tools we used for data extraction (i.e., `git`) are widely and freely available. Lastly, our empirical data, and insights from it, suggest future improvements from a new model building approach based on the additional consideration of architectural information and application of what we have presented as the Previously Vulnerable Assumption.

We stress the importance of the automated mining approach used. As working professionals in the software industry, we highlight the importance of automated approaches as key aspects of both the practicality and scalability required to further motivate wider industry adoption (and corresponding overall socioeconomic bene-

fits). We have stressed practicality for industry adoption several times throughout this work. Developer time (not to mention time from developers with security expertise), is an extremely precious and limited resource. Developers busy at work completing product-differentiating features have little time left to hunt through millions of lines of code to identify vulnerabilities. Moreover, when vulnerabilities in a component are known, the usual practice is to patch or move to the subsequent version. However, moving forward is not always possible, especially for already-fielded embedded products lacking connectivity.

The tools we used for data extraction are significant in their simplicity. Einstein is often quoted as saying "Everything should be made as simple as possible, but not simpler". There are few primitives as widely universal as directory structures and VCS control tools such as `git`. We leave the reader with the jovial remark, "how much simpler can you git?" as a quick shorthand to remember our re-interpretation of Naggappan et. al. [65] change-based burst metrics for git repositories, and module identification according to directory structure of the same. The results, especially from our Firefox case study, indicate that the frequency and size characteristics of VCS-derived metrics (i.e., architectural change-bursts) offer earlier detection of what might be considered a projects' "vulnerability signal", of which we provided empirical evidence, and presented by logical reasoning via the concept we presented as the Previously Vulnerable Assumption.

If we believe the Previously Vulnerable Assumption, that is, that a file or module found vulnerable in the future was already vulnerable at present (but simply undiscovered), then the implication is that earlier detection of whatever fundamental properties coincide with their vulnerability potential and severity provides value in solving the residual vulnerability problem. In this ontological framework, we seek closer correspondence between the innate features of such entities and any de-

tectable features that relate not only to their probable occurrence as a vulnerability, but that when exploited by bad actors, cause greater resulting impact. Furthermore, this concept facilitates a mental construct for better characterization of a software project as a whole, given its particular vulnerability density; that is, due to the manner in which we ascribe severity, we would expect the impact from vulnerabilities that are coincident with a project's vulnerability signal, to be more damaging when exploited in the future. For all these reasons, we therefore believe a historical model built by applying these concepts, provides value by better characterizing severity, and by increasing the number of samples on which to train supervised vulnerability prediction models.

6.3 Future Work

There are several future directions for this work, however the most fundamental deal with evaluation of ranked results, further validation of the approach, and more clearly establishing the contexts and caveats characterizing it's suitability an applicability.

As presented, our approach is entirely new. The more modern and novel repository-mined change-based frequency metrics, along with architectural module metrics, showed mixed results for class discrimination largely dependent upon the particular version slice of the respective software project in which they were evaluated. Indeed, although we experimentally "abused" the intent of our metrics in a classification context, this work did not explore evaluation of actual ranked results as was our original intent. Our original premise for the application of repository mined change and architectural metrics was for use in a ranking stage within model pipelines where established classification metrics would discriminate predictions to be ranked. This

speaks to the need for additional validation beyond our initial evaluation of Spearman correlation measurements against our own CVE based quantification of severity.

We are excited to further explore the intended application for scoring. To this end, we also note that CVE-based CVSS scores offered a convenient method for quantifying severity, but the ideas presented are just as applicable to better severity quantification techniques. Additionally, further validation is needed to understand strengths and limitations of this approach. A key example for immediate investigation is research into the extent and applicability of the Previous Vulnerability Assumption.

A caution related to the Previous Vulnerability Assumption is the extent to which we apply it to a given software project. That is, the historical extent to which it is still accurate to label vulnerable samples (e.g., files) for training vulnerability prediction learners. We conjecture that the assumption holds as long as the corresponding software project's physical architecture (i.e., repository organization and inter-module call structure) remains consistent over time (i.e., over successive versions). We note here that future additional areas of research might investigate additional tuning parameters that relate to penalty windows or tail limits that stop application of the Previous Vulnerability Assumption at some prior version when building and labeling the vulnerable training set.

Bibliography

[1] January 2013 web server survey. Netcraft. http://news.netcraft.com/archives/2013/01/07/january-2013-web-server-survey-2.html, accessed May 18, 2013.

[2] Mozilla at a glance. Mozilla Foundation. https://blog.mozilla.org/press/ataglance/, accessed May 18, 2013.

[3] UnderstandC++. Scientific Toolworks, Inc. https://scitools.com/features/, accessed Nov 21, 2016.

[4] UnderstandC++ getting started with the API: Part 2. Scientific Toolworks, Inc. http://scitools.com/blog/api/api-2-entities-references-and-filters, accessed June 1, 2013.

[5] UnderstandC++ Python API manual. Scientific Toolworks, Inc. https://scitools.com/documents/manuals/python/understand.html, accessed June 1, 2016.

[6] W. Abdelmoez, D. M. Nassar, M. Shereshevsky, N. Gradetsky, R. Gunnalan, H. H. Ammar, B. Yu, and A. Mili. Error propagation in software architectures. In *Software Metrics, 2004. Proceedings. 10th International Symposium on*, pages 384–393. IEEE, 2004.

[7] Omar Alhazmi, Yashwant Malaiya, and Indrajit Ray. Security vulnerabilities in software systems: A quantitative perspective. *Data and Applications Security XIX*, 3654(3654):281–294, August 2005.

[8] L. Allodi, W. Shim, and F. Massacci. Quantitative assessment of risk reduction with cybercrime black market monitoring. In *2013 IEEE Security and Privacy Workshops*, pages 165–172, May 2013.

[9] Luca Allodi and Fabio Massacci. A preliminary analysis of vulnerability scores for attacks in wild: The ekits and sym datasets. In *Proceedings of the 2012 ACM Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, BADGERS '12, pages 17–24, New York, NY, USA, 2012. ACM.

[10] Luca Allodi and Fabio Massacci. My software has a vulnerability, should i worry? *arXiv preprint arXiv:1301.1275*, 2013.

[11] Luca Allodi and Fabio Massacci. Comparing vulnerability severity and exploits using case-control studies. *ACM Trans. Inf. Syst. Secur.*, 17(1):1:1–1:20, August 2014.

[12] Muhammad Anan, Hossein Saiedian, and Jungwoo Ryoo. An architecture-centric software maintainability assessment using information theory. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(1):1–18, January 2009.

[13] R. J. Anderson. Security in open versus closed systems — the dance of boltzmann, coase and moore. Technical report, Cambridge University, England, 2002.

[14] Apache. Apache httpd security report for 2.2. Apache Foundation, online. http://httpd.apache.org/security/vulnerabilities_22.html, accessed November 22, 2016.

[15] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions Dependable and Secure Computing*, 1(1):11–33, 2004.

[16] Varadachari S. Ayanam. Software security vulnerability VS software coupling: A study with empirical evidence. Master's thesis, Southern Polytechnic State University, December 2009.

[17] Dejan Baca, Bengt Carlsson, and Lars Lundberg. Evaluating the cost reduction of static code analysis for software security. In *PLAS '08: Proceedings of the third ACM SIGPLAN workshop on Programming languages and analysis for security*, pages 79–88, New York, NY, USA, 2008. ACM.

[18] Robert M. Bell, Thomas J. Ostrand, and Elaine J. Weyuker. Does measuring code change improve fault prediction? In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, Promise '11, New York, NY, USA, 2011. ACM.

[19] Steven M. Bellovin. On the brittleness of software and the infeasibility of security metrics. *IEEE Security and Privacy*, 04(4):96, 2006.

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

[21] Mehran Bozorgi, Lawrence Saul, Stefan Savage, and Geoffrey M. Voelker. Beyond heuristics: Learning to classify vulnerabilities and predict exploits. In *Proceedings of the Sixteenth ACM Conference on Knowledge Discovery and Data Mining (KDD-2010)*, pages 105–113, 2010.

[22] L. C. Briand, Sandro Morasca, and V. R. Basili. Property-based software engineering measurement. *IEEE Transactions on Software Engineering*, 22(1):68–86, January 1996.

[23] S. R. Chidamber and C. F. Kemerer. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, June 1994.

[24] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, March 2011.

[25] Dan DaCosta, Christopher Dahn, Spiros Mancoridis, and Vassilis Prevelakis. Characterizing the 'security vulnerability likelihood' of software functions. In *Proceedings of the International Conference on Software Maintenance*, ICSM '03, Washington, DC, USA, 2003. IEEE Computer Society.

[26] Noopur Davis, Watts Humphrey, Samuel T. Redwine Jr., Gerlinde Zibulski, and Gary McGraw. Processes for producing secure software: Summary of us national cybersecurity summit subgroup report. *IEEE Security and Privacy*, 02(3):18–25, 2004.

[27] Pär Emanuelsson and Ulf Nilsson. A comparative study of industrial static analysis tools. *Electron. Notes Theor. Comput. Sci.*, 217:5–21, 2008.

[28] Michael C. Gegick. *Predicting attack-prone components with source code static analyzers*. PhD thesis, North Carolina State University, 2009.

[29] Georgios Gousios. On the importance of tools in software engineering research. Blog, June 2012. Accessed: 02/20/2013.

[30] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, October 2005.

[31] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*, chapter 5, pages 182–201. Morgan Kaufmann Publishers, San Francisco, 2001.

[32] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*, chapter 3, pages 123–124. Morgan Kaufmann Publishers, San Francisco, 2001.

[33] J. Han and M. Kamber. *Data Mining: Concepts and Techniques*, chapter 1, page 15. Morgan Kaufmann Publishers, San Francisco, 2001.

[34] Seth Hanford. Common vulnerability scoring system, v3 development update. In *Technical report, Forum of Incident Response and Security Teams (FIRST)*, 2013.

[35] Ahmed E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 78–88, Washington, DC, USA, May 2009. IEEE Computer Society.

[36] Ahmed E. Hassan and Tao Xie. Software intelligence: the future of mining software engineering data. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, FoSER '10, pages 161–166, New York, NY, USA, 2010. ACM.

[37] Sarah S. Heckman. *A Systematic Model Building Process for Predicting Actionable Static Analysis Alerts*. PhD thesis, North Carolina State University, May 2009.

[38] Daniel Hein and Hossein Saiedian. Secure software engineering: Learning from the past to address future challenges. *Information Security Journal: A Global Perspective*, 18(1):8–25, 2009.

[39] Greg Hoglund and Gary Mcgraw. Point/Counterpoint. *IEEE Software*, 19(6):56–59, 2002.

[40] David Hovemeyer and William Pugh. Finding bugs is easy. *SIGPLAN Not.*, 39(12):92–106, December 2004.

[41] IEEE. IEEE standard glossary of software engineering terminology. Technical report, IEEE, 1990.

[42] Daniel Jackson and Jeannette Wing. Lightweight formal methods. *IEEE Computer*, 29(4):16–30, 1996.

[43] WA Jansen. NIST IR 7564: Directions in security metrics research, National Institute of Standards and Technology, US Dept. of Commerce, Gaithersburg (2009).

[44] D. Janzen and H. Saiedian. Test-driven development: Concepts, taxonomy, and future direction. *IEEE Computer*, 38(9):43–50, August 2005.

[45] D. Janzen and H. Saiedian. A leveled examination of test-driven development acceptance. In *Proceedings of the 29th ACM International Conference on Software Engineering*, pages 719–722. ACM, May 2007.

[46] D. Janzen and H. Saiedian. Does test-driven development really improve software design quality? *IEEE Software*, 25(2):77–84, March/April 2008.

[47] Stephen H. Kan. *Metrics and Models in Software Quality Engineering*, chapter 4, pages 83+. Addison-Wesley, 1995.

[48] T. M. Khoshgoftaar, E. B. Allen, N. Goel, A. Nandi, and J. McMullan. Detection of software modules with high debug code churn in a very large legacy system. In *Proceedings of the The Seventh International Symposium on Software Reliability Engineering*, ISSRE '96, Washington, DC, USA, 1996. IEEE Computer Society.

[49] Ivan Victor Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, West Lafayette, IN, May 1998.

[50] BuiltWith Pty Ltd. Apache usage statistics. BuiltWith Pty Ltd., online. `https://trends.builtwith.com/Web-Server/Apache`, accessed January 9, 2017.

[51] P. K. Manadhata and J. M. Wing. An attack surface metric. *IEEE Transactions on Software Engineering*, 37(3):371–386, May 2011.

[52] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[53] Gary McGraw. Software assurance for security. *Computer*, 32(4):103–105, 1999.

[54] Gary McGraw. From the ground up: The DIMACS software security workshop. *IEEE Security and Privacy*, 1(2):59–66, 2003.

[55] Gary McGraw. *Software Security: Building Security In*. Addison-Wesley Software Security Series. Addison-Wesley, Upper Saddle River, NJ, 2006. pp. 13-37, 277-298.

[56] Gary McGraw and Bruce Potter. Software security testing. *IEEE Security and Privacy*, 2(5):81–85, 2004.

[57] Peter Mell, Karen Scarfone, and Sasha Romanosky. *CVSS: A Complete Guide to the Common Vulnerability Scoring System Version 2.0*. FIRST: Forum of Incident Response and Security Teams, June 2007.

[58] MITRE. Common vulnerabilities and exposures: About CVE. MITRE Corporation, online. `https://cve.mitre.org/about/`, accessed November 22, 2016.

[59] David Moore. *Introduction to the Practice of Statistics*. Freeman, New York, 2 edition, 1993.

[60] 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*, ICSE '08, pages 181–190, New York, NY, USA, May 2008. IEEE.

[61] Nuthan Munaiah and Andrew Meneely. Beyond the attack surface: Assessing security risk with random walks on call graphs. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, SPRO '16, pages 3–14, New York, NY, USA, 2016. ACM.

[62] Nuthan Munaiah and Andrew Meneely. Vulnerability severity scoring and bounties: Why the disconnect? In *Proceedings of the 2Nd International Workshop on Software Analytics*, SWAN 2016, pages 8–14, New York, NY, USA, 2016. ACM.

[63] J. C. Munson and S. G. Elbaum. Code churn: a measure for estimating the impact of code change. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 24–31. IEEE Comput. Soc, 1998.

[64] J. C. Munson and T. M. Khoshgoftaar. The detection of fault-prone programs. *IEEE Transactions on Software Engineering*, 18(5):423–433, May 1992.

[65] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. Change bursts as defect predictors. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 309–318. IEEE, November 2010.

[66] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th international conference on Software engineering*, ICSE '05, pages 284–292, New York, NY, USA, 2005. ACM.

[67] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 452–461, New York, NY, USA, 2006. ACM.

[68] NIST. NVD: National Vulnerability Database. National Institute of Science and Technology, online. `http://nvd.nist.gov/`, accessed November 22, 2016.

[69] Thomas J. Ostrand, Elaine J. Weyuker, and Robert M. Bell. Programmer-based fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, New York, NY, USA, 2010. ACM.

[70] Raymond R. Panko. *Corporate Computer and Network Security*. Prentice Hall, New Jersey, 2003. pp. 324-330.

[71] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[72] Fernando Pérez and Brian E. Granger. IPython: a system for interactive scientific computing. *Computing in Science and Engineering*, 9(3):21–29, May 2007.

[73] Murali Rao, Y. Chen, B. C. Vemuri, and Fei Wang. Cumulative residual entropy: a new measure of information. *IEEE Transactions on Information Theory*, 50(6):1220–1228, June 2004.

[74] Joseph R. Ruthruff, John Penix, J. David Morgenthaler, Sebastian Elbaum, and Gregg Rothermel. Predicting accurate and actionable static analysis warnings: an experimental approach. In *ICSE '08: Proceedings of the 30th international conference on Software engineering*, pages 341–350, New York, NY, USA, 2008. ACM.

[75] A. Sachitano, R.O. Chapman, and J.A. Hamilton. Security in software architecture: a case study. In *Information Assurance Workshop*, pages 370–376. Proceedings from the Fifth Annual IEEE SMC, IEEE, June 2004.

[76] Jerome H. Saltzer and Michael D. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 9(63):1278–1308, 1975. `http://web.mit.edu/Saltzer/www/publications/protection/`.

[77] S. Sarkar, G. M. Rama, and A. C. Kak. API-based and information-theoretic metrics for measuring the quality of software modularization. *IEEE Transactions on Software Engineering*, 33(1):14–32, January 2007.

[78] J. Sayyad Shirabad and T.J. Menzies. The PROMISE Repository of Software Engineering Databases. School of Information Technology and Engineering, University of Ottawa, Canada, 2005.

[79] Bruce Schneier. Information security and externalities. In *Social and Economic Factors Shaping the Future of the Internet*, pages 19–20. NSF/OECD, January 2007. http://www.oecd.org/sti/ieconomy/37985707.pdf.

[80] Adrian Schröter, Thomas Zimmermann, and Andreas Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering*, ISESE '06, pages 18–27, New York, NY, USA, 2006. ACM.

[81] Yonghee Shin. *Investigating Complexity Metrics as Indicators of Software Vulnerability*. PhD thesis, North Carolina State University, Raleigh, North Carolina, 2011.

[82] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A. Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, November 2011.

[83] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz. What we have learned about fighting defects. In *Software Metrics, 2002. Proceedings. Eighth IEEE Symposium on*, pages 249–258. IEEE, 2002.

[84] Katrina Tsipenyuk, Brian Chess, and Gary McGraw. Seven pernicious kingdoms: A taxonomy of software security errors. *IEEE Security and Privacy*, 03(6):81–84, 2005.

[85] Kenneth R. van Wyk and John Steven. Essential factors for successful software security awareness training. *IEEE Security and Privacy*, 4(5):80–83, 2006.

[86] Stefan Wagner, Jan Jürjens, Claudia Koller, and Peter Trischberger. Comparing bug finding tools with reviews and tests. In *Testing of Communicating Systems*, pages 40–55, 2005.

[87] Greg Wilson and Jorge Aranda. Empirical software engineering. *American Scientist*, 99(6):466+, November 2011.

[88] Jeannette M. Wing. A call to action: Look beyond the horizon. *IEEE Security and Privacy*, 1(6):62–67, 2003.

[89] Tao Xie, Jian Pei, and Ahmed E. Hassan. Mining software engineering data. In *Software Engineering - Companion, 2007. ICSE 2007 Companion. 29th International Conference on*, pages 172–173, Washington, DC, USA, May 2007. IEEE.

[90] Tao Xie, Suresh Thummalapenta, David Lo, and Chao L. Liu. Data mining for software engineering. *Computer*, 42(8):55–62, August 2009.

[91] A. A. Younis and Y. K. Malaiya. Comparing and evaluating cvss base metrics and microsoft rating system. In *2015 IEEE International Conference on Software Quality, Reliability and Security*, pages 252–261, Aug 2015.

[92] A.A. Younis, Y.K. Malaiya, and I. Ray. Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability. In *High-Assurance Systems Engineering (HASE), 2014 IEEE 15th International Symposium on*, pages 1–8, Jan 2014.

[93] Awad Younis, Yashwant Malaiya, Charles Anderson, and Indrajit Ray. To fear or not to fear that is the question: Code characteristics of a vulnerable function with an existing exploit. In *Proceedings of the Sixth ACM Conference on Data*

and Application Security and Privacy, CODASPY '16, pages 97–104, New York, NY, USA, 2016. ACM.

[94] Awad Younis, Yashwant K. Malaiya, and Indrajit Ray. Assessing vulnerability exploitability risk using software properties. *Software Quality Journal*, 24(1):159–202, March 2016.

[95] J. Zheng, L. Williams, N. Nagappan, W. Snipes, J. P. Hudepohl, and M. A. Vouk. On the value of static analysis for fault detection in software. *IEEE Transactions on Software Engineering*, 32(4):240–253, April 2006.

Chapter 7

Appendix 1: Mozilla Foundation Security Advisory Data

This appendix is included for the benefit of others who wish to better understand the companion software for this work, or who are interested in undertaking a similar study utilizing Mozilla Foundation Security Advisories (MFSAs). In particular, depending on investigation time frame, different parsing approaches may be required for MFSA pages. A particular example of a change in the MFSA format over time is the location of the Bugzilla URLs, relative to the CVE (if listed).

MFSA pages published after September, 2016, contain a dedicated section tag with class attribute "cve". Listing 7.1 shows a code snippet from the spider used to parse pages with the dedicated cve section. The cve section in the more recent MFSA pages yields more deterministic parsing results and is especially useful for vulnerability oriented research since the CVE can be consistently determined.

MFSA pages published prior to 2006 often omit CVE, listing only the Bugzilla URL. CVE is not listed consistently, if at all. A majority of the MFSA pages (from 2005 to late 2016) list fix information following a references heading. Most often, the fix information includes one or more Bugzilla entries, *optionally* followed by the associated CVE id. However, some MFSA pages do not follow the aforementioned

structure, despite having been published after 2006.

The snippet in Listing 7.2 shows the formatting and parsing code for a MFSA “summary" block. Note that the summary block is common to most MFSA pages.

```
#
# MFSA 2016-85 introduced the use of cve section blocks
#
# <section class="cve">
#     <h4 id="CVE-2016-5270" class="level-heading">
#       <a href="#CVE-2016-5270">
#       <span class="anchor">#</span>
#       CVE-2016-5270: Heap-buffer-overflow in
#       nsCaseTransformTextRunFactory::TransformString
#       </a>
#     </h4>
#     <dl class="summary">
#       <dt>Reporter</dt>
#       <dd>Atte Kettunen</dd>
#       <dt>Impact</dt>
#       <dd><span class="level high">high</span></dd>
#     </dl>
#     <h5>Description</h5>
#     <p>
#        An out-of-bounds write of a boolean value during text
#        conversion with some unicode characters
#     </p>
#       <h5>References</h5>
#       <ul>
#         <li>
#         <a
#    href="https://bugzilla.mozilla.org/show_bug.cgi?id=1291016">
#                 Bug 1291016
#         </a>
#         </li>
#       </ul>
# </section>
#

xpath_cve_section_block = '//div[@itemprop="articleBody"]\
    /section[@class="cve"]'
sel_cve_id = './*[contains(@id,"CVE")]/@id'
sel_cve_summary_text = './*[contains(@id,"CVE")]\
    /a[contains(@href,"#CVE")]/text()'
sel_cve_summary_block = './dl[@class="summary"]'
sel_cve_summary_impact_item = './dt[contains(.,"Impact")]/\

    ./following-sibling::dd/span/text()'
sel_cve_description_item =
    './*[contains(./text(),"Description")]/\
./following-sibling::p/text()'
sel_cve_bugzilla_url =
    './ul/li/a[contains(@href,"bugzilla")]/@href'
```

Listing 7.1: Example MFSA with section “cve" and xpath parsing code

```
#
# Typical format of a the common summary block.
#
# <dl class="summary">
# <dt>Announced</dt><dd>January 21, 2005</dd>
# <dt>Reporter</dt><dd> Firstname Lastname </dd>
# <dt>Impact</dt><dd><span class="level low">Low</span></dd>
# <dt>Products</dt><dd>Firefox, Mozilla Suite</dd>
# <dt>Fixed in</dt>
#     <dd>
#         <ul>
#             <li>Firefox 1</li>
#             <li>Mozilla Suite 1.7.5</li>
#         </ul>
#     </dd>
# </dl>
#
xpath_summary_block =
    '//div[@itemprop="articleBody"]/dl[@class="summary"]'
sel_date_item_text =
    './dt[contains(.,"Announced")]/following-sibling::dd/text()'
sel_impact_item_text =
    './dt[contains(.,"Impact")]/following-sibling::dd/text()'
sel_fixedin_item_text = './dt[contains(.,"Fixed in")]\

   /following-sibling::dd/ul/li[contains(., "Firefox")]\
                          /text()'
```

Listing 7.2: Example common MFSA “summary" block

Chapter 8

Appendix 2: Supplemental Analysis

This section contains additional graphs and data showing intermediate results gathered during our study.

The scatter matrix in Figure 8.5 reflects intuitive relationships between traditional complexity metrics across all versions sampled. As would be expected, the count based metrics show strong linear relationships with each other.

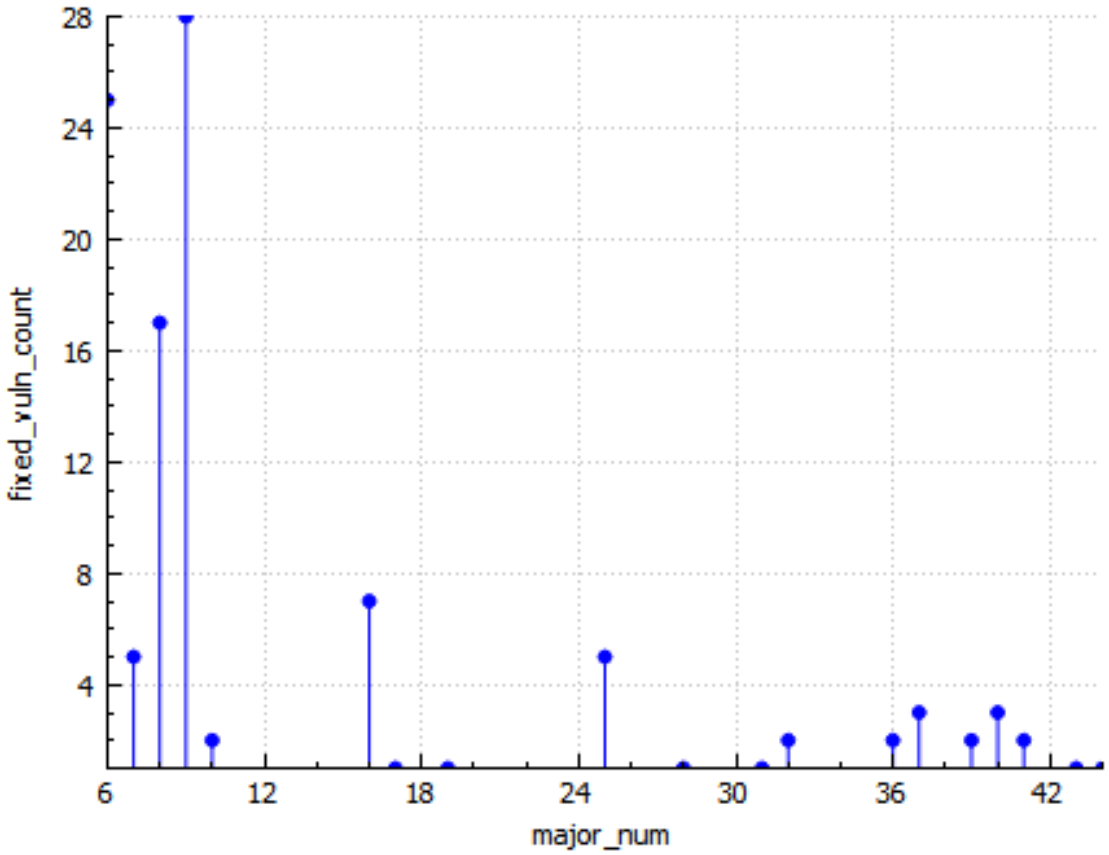

Figure 8.1: VulnCount as number of ITS entries ($RawCount$) fixed at specific versions

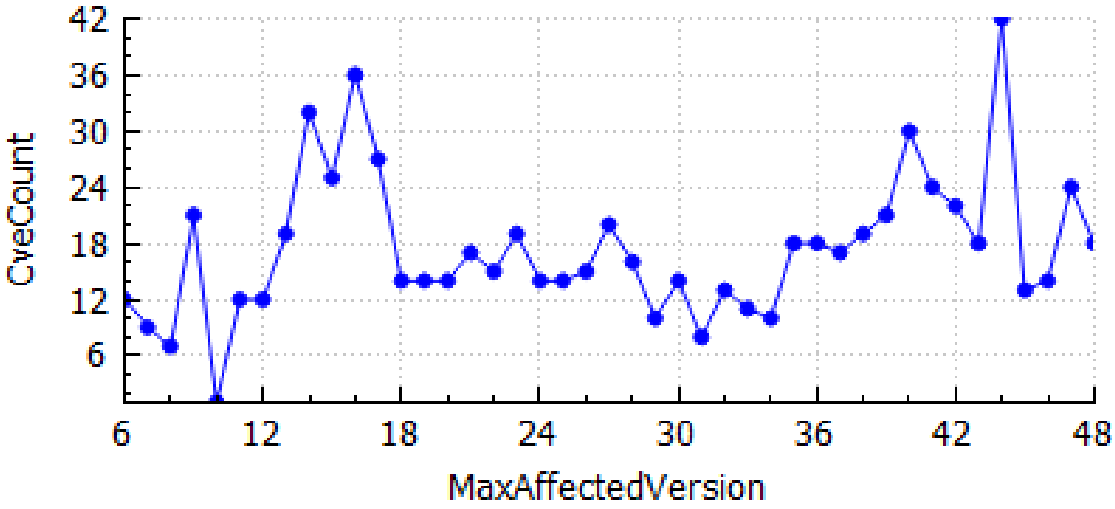

Figure 8.2: Count of CVEs by maximum affected version

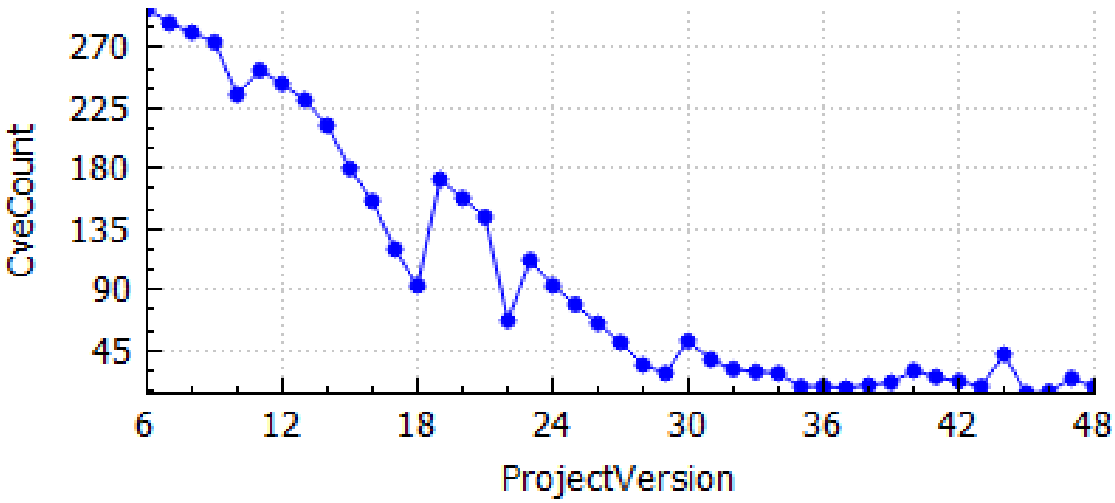

Figure 8.3: CVE density by version

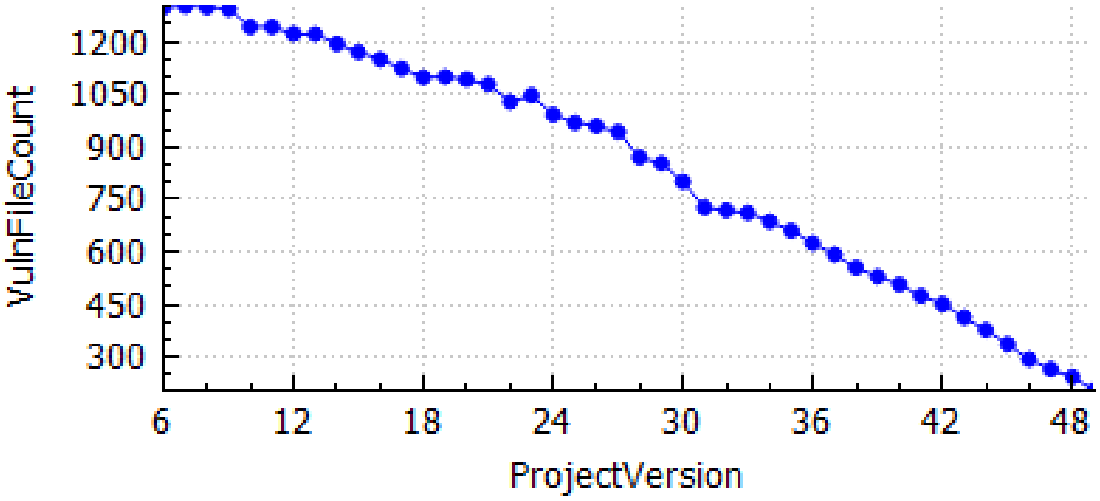

Figure 8.4: Number of vulnerable files by version

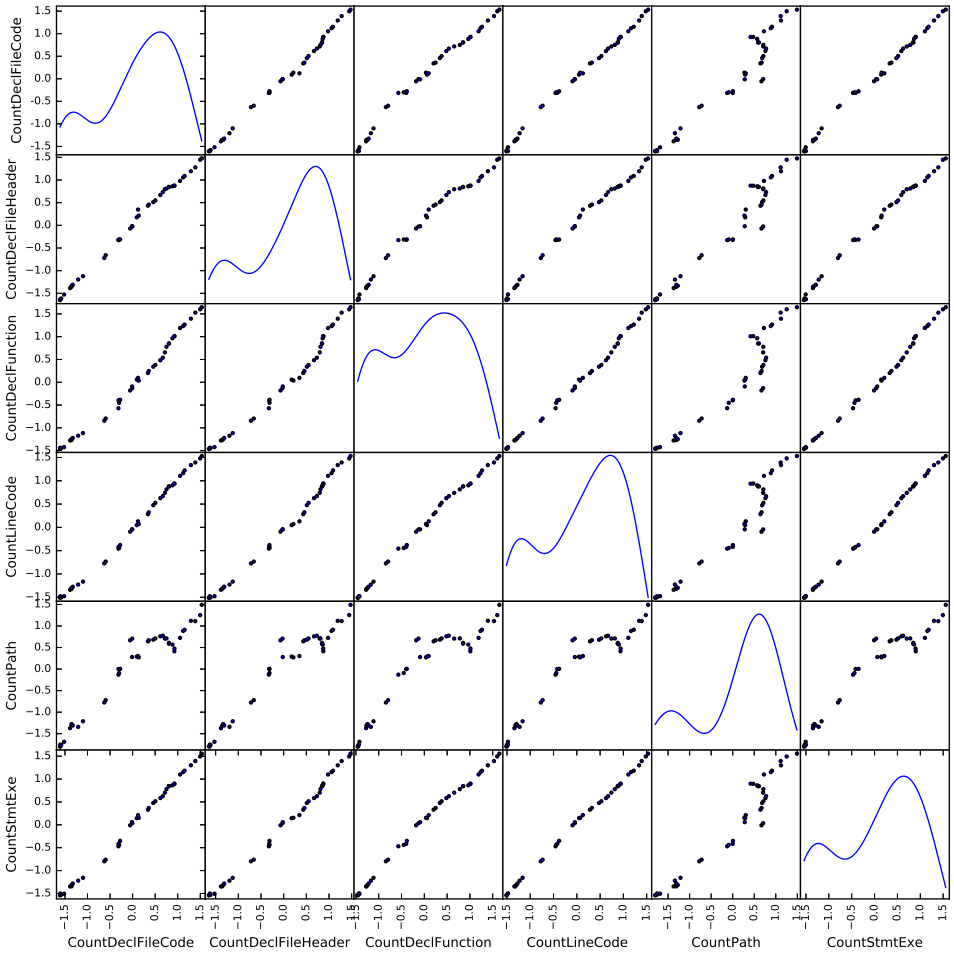

Figure 8.5: Firefox scatter matrix comparing count based metrics

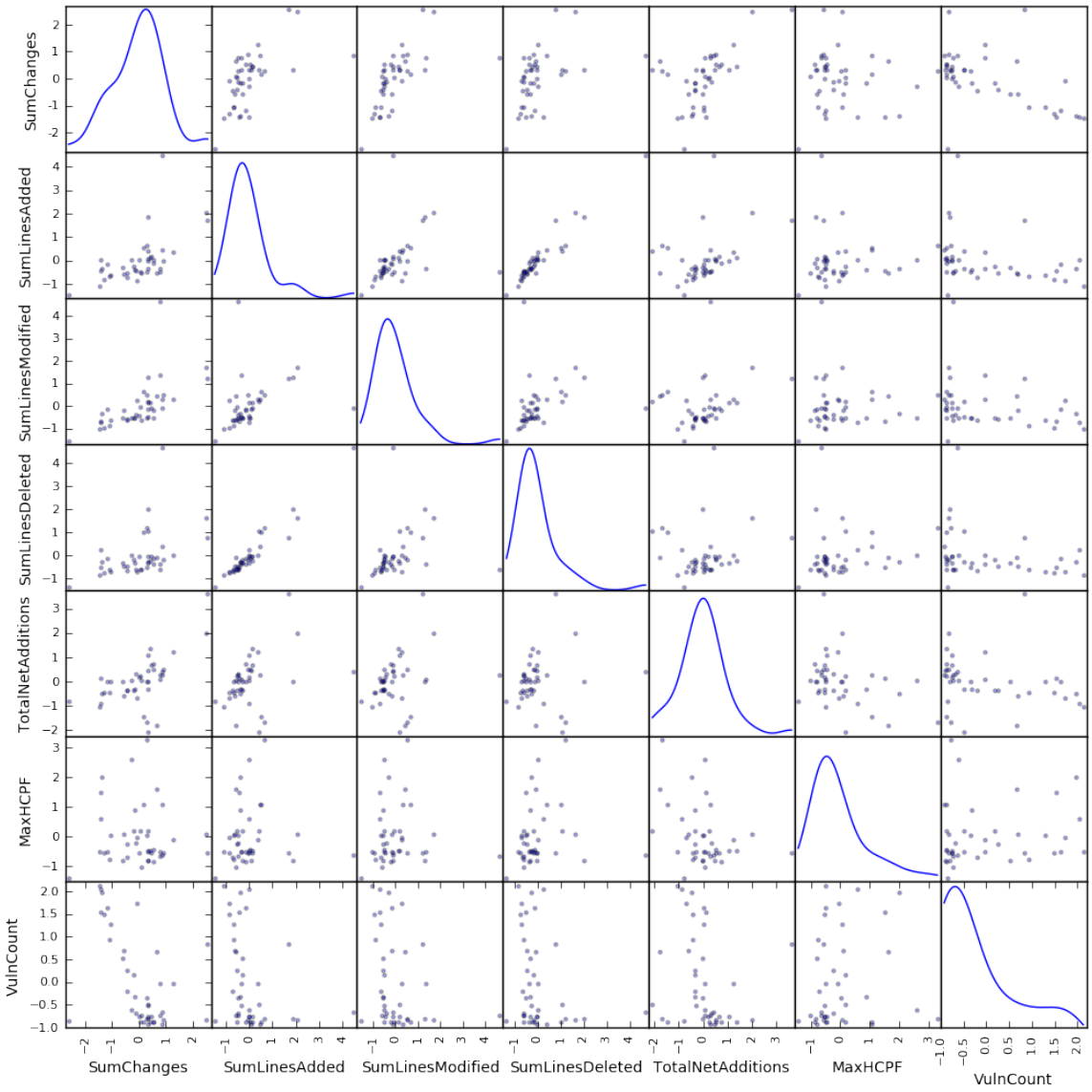

Figure 8.6: Firefox aggregate churn scatter matrix

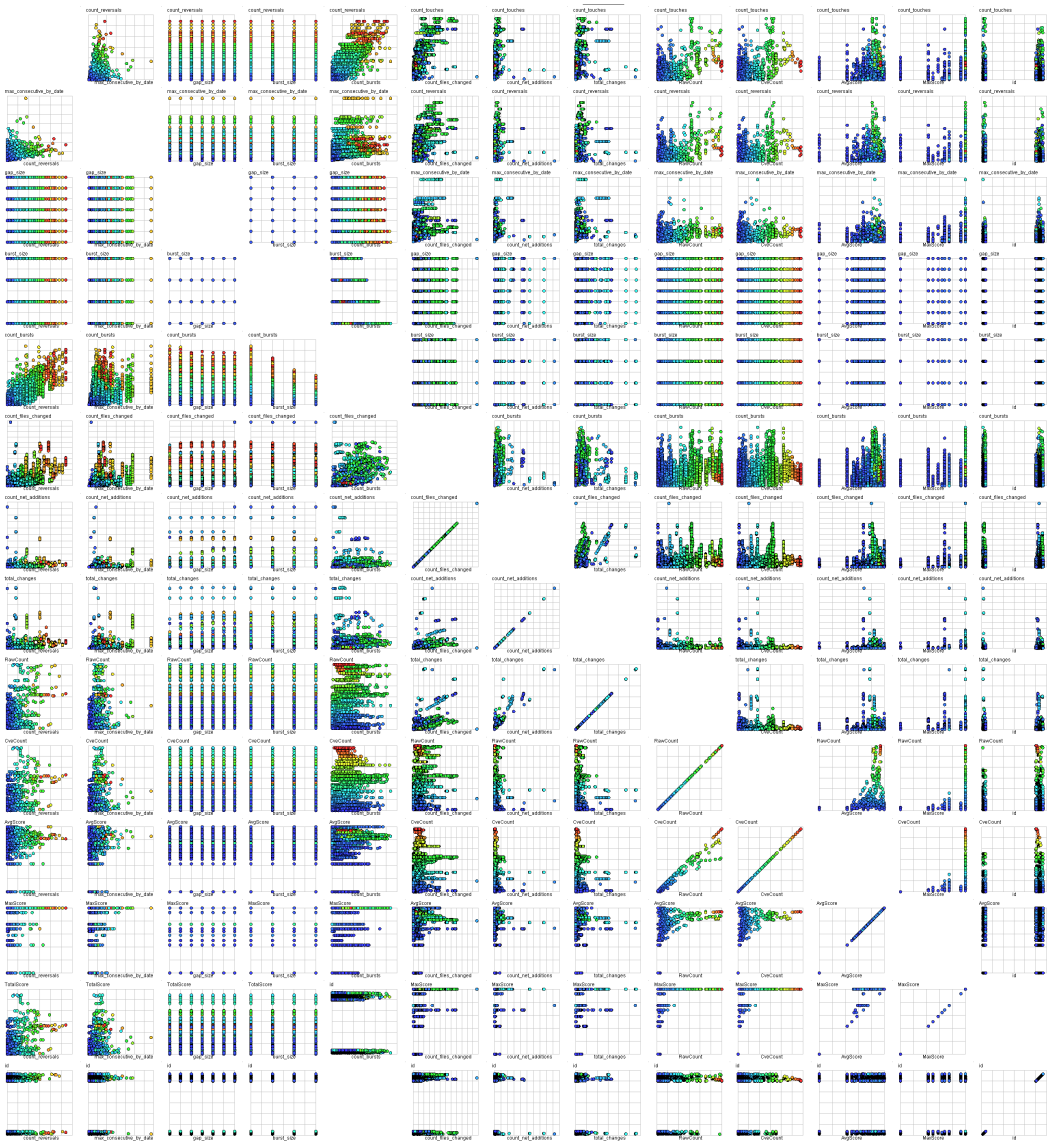

Figure 8.7: Preliminary visualization of aggregate burst metrics for Firefox, across all all combinations of gap and burst size.

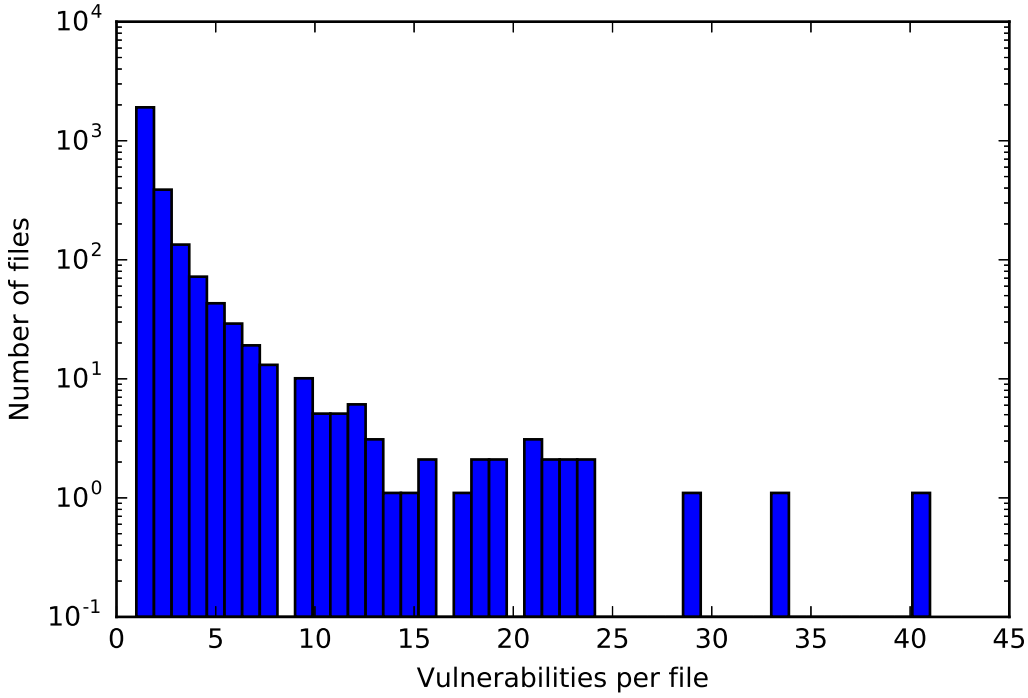

Figure 8.8: Firefox vulnerability distribution; all versions

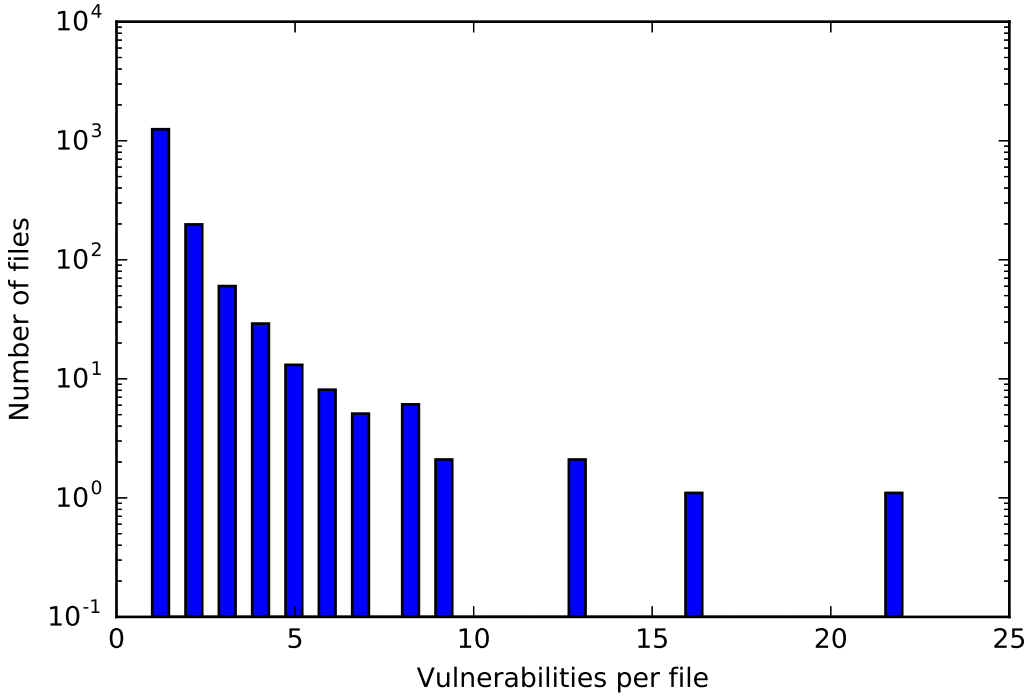

Figure 8.9: Firefox vulnerability distribution; versions 3..49

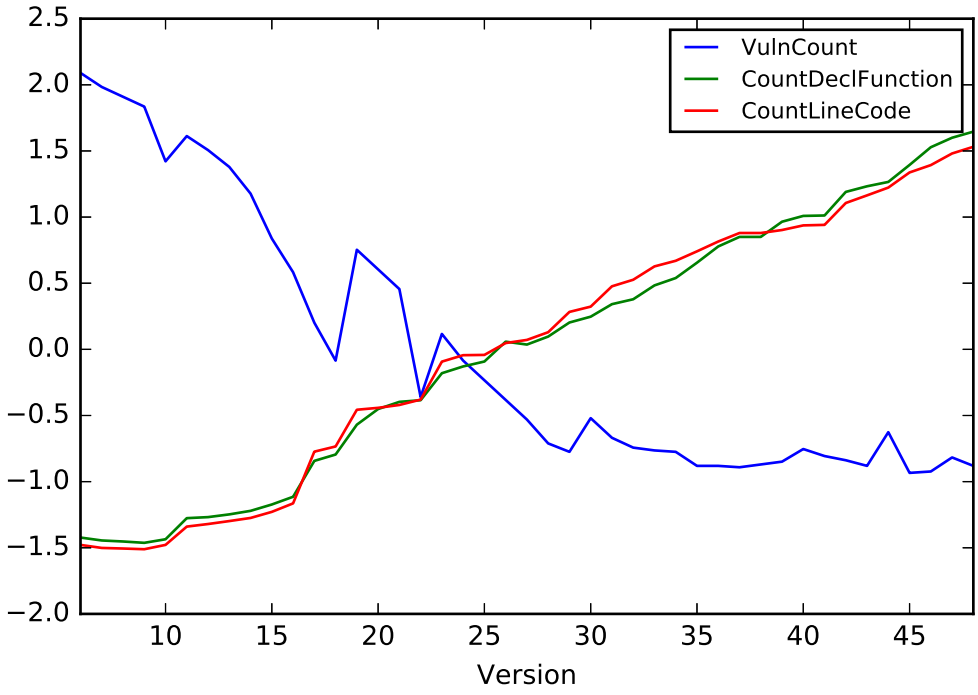

Figure 8.10: Aggregate complexity counts and vulnerabilities by Firefox version

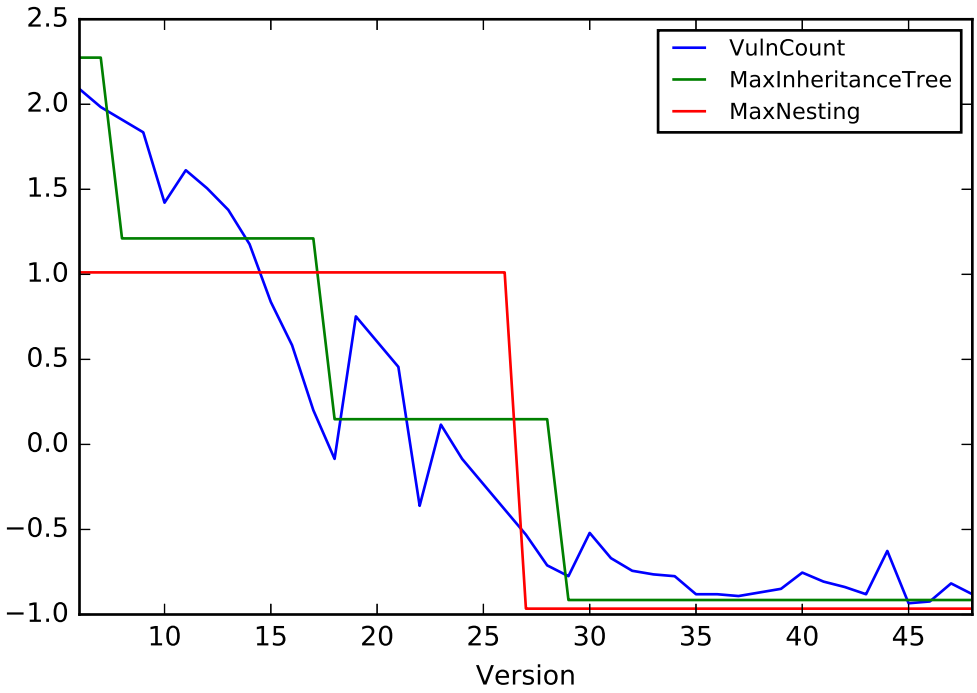

Figure 8.11: Maximum inheritance depth, nesting level, and vulnerabilities by Firefox version

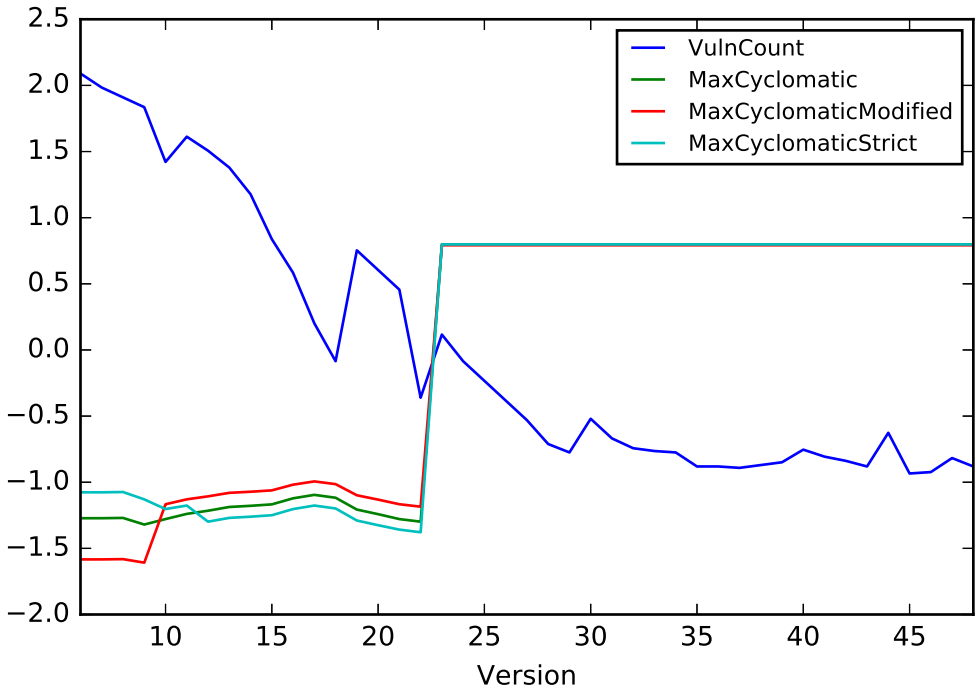

Figure 8.12: Firefox maximum cyclomatic complexity metrics and vulnerabilities by version

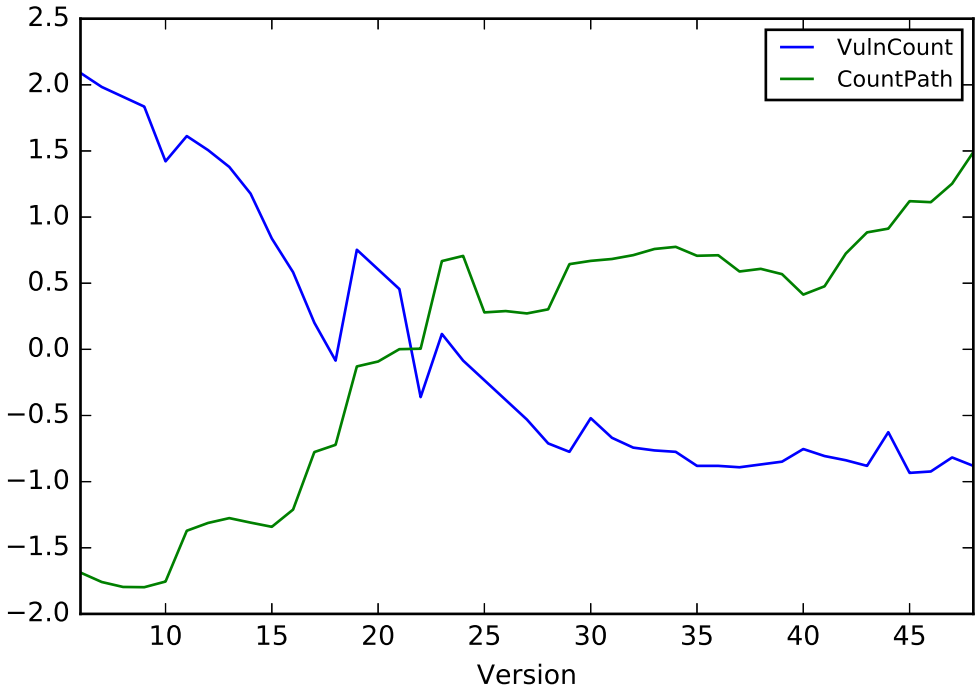

Figure 8.13: Firefox path count and vulnerabilities by version

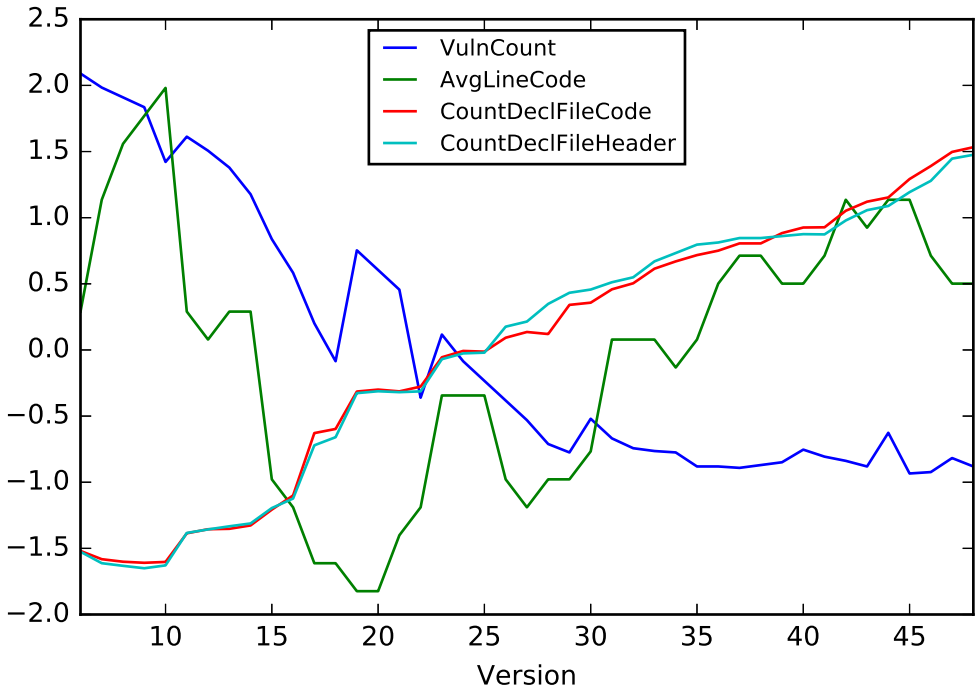

Figure 8.14: Average LOC, source files, header files, and vulnerabilities by Firefox version