Predicting Attack Prone Software Components using Repository Mined Change Metrics

Daniel Hein and Hossein Saiedian
Department of Electrical Engineering and Computer Science, University of Kansas, Lawrence, U.S.A.

Keywords: Secure Software Engineering, Software Vulnerability, Information Security, Vulnerability Prediction Model.

Abstract: Identification of attack-prone entities is a crucial step toward improving the state of information security in modern software based systems. Recent work in the fields of empirical software engineering and defect prediction show promise toward identifying and prioritizing attack prone entities using information extracted from software version control repositories. Equipped with knowledge of the most vulnerable entities, organizations can efficiently allocate resources to more effectively leverage secure software development practices, isolating and expunging vulnerabilities before they are released in production products. Such practices include security reviews, automated static analysis, and penetration testing, among others. Efficiently focusing secure development practices on entities of greatest need can help identify and eliminate vulnerabilities in a more cost effective manner when compared to wholesale application for large products.

1 INTRODUCTION

Current trends toward an increasingly connected world highlight the need for improved security and privacy. In the commercial sector, the cost of improving security competes directly with the cost of feature development. Therefore, cost effective approaches for vulnerability identification, prioritization, and removal are needed in order to improve software security.

Our research looks to prediction models to help identify attack-prone software components and selectively prioritize the application of secure development practices for vulnerability removal. Our goal is to better facilitate the adoption and application of secure development practices(e.g., review, static analysis, and penetration testing) in industry by improving the efficiency and effectiveness of their application. 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. Inspired by existing research from defect and vulnerability prediction, we are researching the feasibility of using historical and architectural metrics from maintainability literature for both identification and prioritization.

Key intuitions guiding our research are based on the reality that modern software based products evolve over time and are composed of different sub-components of varying maturity. In addition, each of those sub-components may be modified by multiple developers with varying knowledge of the system as a complete product. As a consequence, we might expect that unintended side effects are more likely to result when modifying modules already exhibiting poor maintainability. For example, we might expect a higher probability of introducing unintended side effects when several scattered changes are rapidly made to tightly coupled modules with poor encapsulation.

2 BACKGROUND

The majority of attacks on modern connected systems can be traced in one way or another to software defects, and in particular, residual vulnerabilities. Residual vulnerabilities are by definition defects that escape detection and persist in released software. Such vulnerabilities range, in both complexity and severity, from a simple coding error of varying impact, all the way to some fundamental design flaw with far reaching implications (e.g., complete lack of authentication). Because residual vulnerabilities escape detection during development, several experts have suggested integrating additional secure software development practices into the software life cycle to excise these vulnerabilities prior to production re-

Hein, D. and Saiedian, H.
Predicting Attack Prone Software Components using Repository Mined Change Metrics.
DOI: 10.5220/0005812905540563
In *Proceedings of the 2nd International Conference on Information Systems Security and Privacy (ICISSP 2016)*, pages 554-563
ISBN: 978-989-758-167-0

lease. Popular secure development practices include, but are not limited to, scanning code with automated static analysis tools, performing security reviews (for designs and code), adding additional security-focused tests, as well as carrying out penetration tests.

The sheer size of many modern systems means that finding a single defect among thousands, or even millions, of lines of code is like trying to find the proverbial "needle in a haystack". For example, static analysis tools have shown promise toward identifying commonly repeated coding errors, but the sheer size of many systems result in thousands of tool warnings. The ensuing task of verifying (or "triaging") these tool warnings, separating actual defects from false positives, can be overwhelming in itself–especially if there is no strategy defined for eliminating the most critical issues first.

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 vulnerabilities.

For all of these reasons, and especially because developer time is limited, it is essential to prioritize the application of secure software development practices, focusing efforts on the most attack prone areas of code that are likely to result in the greatest impact when exploited. 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." (Jackson and Wing, 1996)

A perquisite is that such attack-prone components can be identified. Therefore, identification of attack-prone components is a crucial step toward preventing vulnerabilities from entering the field. Superior prediction models may significantly help augment and complement existing practices for these purposes.

3 MOTIVATION

Change-based fault and vulnerability prediction metrics have recently shown promising results for identifying vulnerable components. Our work builds on this background, further 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 residual vulnerabilities.*

Our focus on prediction complements existing tools and predictive approaches utilizing code-based metrics as predictors. However, our work more closely examines evolutionary and architectural aspects of the software by:

1. investigating how change metrics correlate with residual vulnerabilities, 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 proposed by 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 maintainability?

Additionally, we aspire to relate prediction models to the attack surface of a system along its boundaries with the surrounding environment after (Manadhata and Wing, 2011) and (Younis et al., 2014). That is, realizing that attacker-crafted data often enters a system along it's attack surface, we reason that modules exhibiting poor maintainability and that are also reachable from said entry-points would be more likely the targets of a 0-day exploit. Because the modules are reachable, it is possible for them to process attacker-crafted data. Because the same modules exhibit poor maintainability, we reason that they are more likely to contain exploitable vulnerabilities.

4 RELATED WORK

Our study is informed by similar empirical vulnerability and fault prediction studies by Shin(Shin et al., 2011), Gimothy et. al. (Gyimothy et al., 2005), and Bozorgi et. al. (Bozorgi et al., 2010). Our work is most closely related to that of Shin, as she investigated coupling metrics as vulnerability predictors. Shin's work builds on a long tradition of complexity metrics used to predict faults. Vulnerability ranking approaches are informed by the work of Bozorgi et. al. The following sections more completely describe related work in the area of metrics based fault prediction, and in particular, predictors based on code metrics, as well as predictors based on change metrics.

4.1 Code Metrics

Several studies (Munson and Khoshgoftaar, 1992), (Nagappan and Ball, 2005), (Chidamber and Kemerer, 1994) 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.(Gyimothy et al., 2005) 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 (Chidamber and Kemerer, 1994) 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 coupling between objects (CBO) metric and fault proneness of modules. Their predictive models based on CBO demonstrated precision and recall values over 69%.

A related work by Janzen and Saiedian (Janzen and Saiedian, 2007) 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. 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. By extension, we reason that security may also be impacted by software architecture and seek to examine relationships between architectural metrics and residual vulnerabilities.

4.2 Change Metrics

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 (Munson and Elbaum, 1998), 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. (Khoshgoftaar et al., 1996) used 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 containing 171 modules with over 38,000 functions yielded precision and recall values over 78%.

Nagappan and Ball (Nagappan and Ball, 2005) demonstrated how to use relative code churn as an es-

timator 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. (Moser et al., 2008) 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 (Ostrand et al., 2010) 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 compared 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 found 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 (Bell et al., 2011) examined several churn metrics, such as lines added, deleted, and modified, where $churn = added + deleted + changed$. They evaluated 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 studied the extent to which faults can be predicted by the degree of churn alone, as well as 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.

Change bursts refer to consecutive changes over a period of time (Nagappan et al., 2010). Change bursts are described by gap and burst size. The gap is the minimum 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 1, taken from Nagappan et al.(Nagappan et al., 2010).

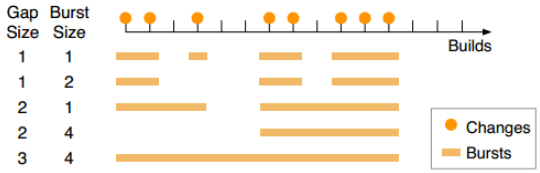

Figure 1: Example change bursts with varying gap and burst size parameters (Nagappan et al., 2010).

Recent studies (Nagappan and Ball, 2005), (Moser et al., 2008) 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. (Nagappan et al., 2010) was particularly impressive as they found that change burst metrics outperformed all previous predictors, such as code complexity, code churn, and organizational structure as predictors for Windows Vista. For their study on Windows Vista, they found that change burst metrics yielded precision and recall values over 90% (Nagappan et al., 2010).

4.3 Entropy based Software Metrics

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 utilize as explanatory variables in our prediction models.

Hassan (Hassan, 2009) 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

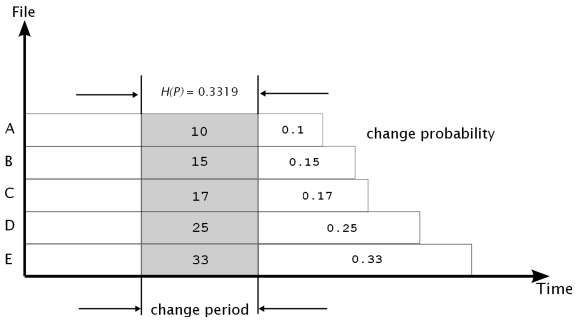

Figure 2: Entropy, $H(P)$, over a change period.

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(NetBSD, FreeBSD, OpenBSD, Postgres, KDE, and KOffice).

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, 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 (Chowdhury and Zulkernine, 2011) 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, Hassan's historical complexity metrics have not specifically been evaluated for 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. (Sarkar et al., 2007) 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 (Saltzer and Schroeder, 1975). The associations between security design principles and detailed modularity principles enumerated by Sarkar et al. (Sarkar et al., 2007) are shown in Table 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 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.

Table 1: Related security and modularity principles.

Security Design Principle	Modularity Principle
Economy of Mechanism	Size
Economy of Mechanism	Acyclic Dependencies
Economy of Mechanism	Unidirectionality in Layers
Complete Mediation	API-based Inter-Module Calls
Complete Mediation	Purpose Dispersion
Complete Mediation	Similarity of Purpose

Anan et al. (Anan et al., 2009) 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. (Abdelmoez et al., 2004) 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.

5 VULNERABILITY PREDICTION

We seek to investigate architectural modularity metrics that characterize economy of mechanism and Complete Mediation. We are also interested in information flow and entropy 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 (Shin et al., 2011), (Shin, 2011) 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.

5.1 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. 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) (Mell et al., 2007), despite said criticisms, offers a metric usable for the purposes of evaluating our ranking approaches.

CVSS scores are often included on security advisories from organizations such as US-CERT, Microsoft, Cisco, and Secunia. These scores are also listed in on-line vulnerability databases such as the National Vulnerability Database (NVD (NIST,)) and the Open Source Vulnerability Database (OSVDB (osv,)). 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 confidentiality, integrity, and availability.

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. (Bozorgi et al., 2010) 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. (Bozorgi et al., 2010), 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. (Bozorgi et al., 2010) 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 CVE-numbered advisories listed in OSVDB (osv,) and NVD (NIST,). 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 propose to use CVSS as an evaluation of ranking we create utilizing modularity and maintainability metrics.

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 such as OSVDB and NVD constitute the source of our training and evaluation data. Therefore, a natural extension of classification (e.g., file or 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.

6 EXPECTED CONTRIBUTIONS

The key contributions anticipated by 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. This research expects 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 say-

ing, “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 released by developing a relative ranking among modules, on the basis of their attack-proneness, is that this ranking is complementary for use with other techniques. Arelative 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 *n* top “actionable alerts”).

Although techniques have been published for calculating various information theoretic metrics from the structural and development views of a software architecture (Sarkar et al., 2007), 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 (Gousios, 2012). Availability of a publicly obtainable 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.

7 CONCLUSIONS

The current state of affairs is rather tenuous as software continually grows larger and more complex, at odds with classic security principles such as economy of mechanism, making software more difficult to comprehend when making changes, and more difficult to test completely. Compounding these factors are the forces vying for limited developer time. Especially in the commercial sector, security concerns often take a back seat to the development of market-differentiating features–the return on investment and impact to the bottom line is more deterministic for feature development (McGraw, 1999). It’s not that the commercial sector doesn’t care about security, but more about spending resources efficiently. Developer time is a constrained, highly contended, and highly demanded resource in software development organizations. 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 our research, is to provide superior prediction models and establish 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. We acknowledge that the notion of ranking vulnerability predictions may be criticized, since a single overlooked security vulnerability has the potential for wide reaching impact.

We believe that advances in vulnerability prediction are possible given the carefully selected metrics, that mirror security considerations and are closely related to our underlying notions about software evolution and development team dynamics. Moreover, drawing on the existing body of work on defect prediction, we believe the more specialized study of vulnerability prediction research offers the potential for new discoveries.

With knowledge of the most vulnerable components, organizations can more effectively prioritize the application of secure development practices (e.g., security review, static analysis, and penetration testing) where they will have the greatest benefit. The added scrutiny from a well-focused effort can help to identify and eliminate vulnerabilities prior to production release in a tractable, efficient, and cost effective manner. The alternative approach is wholesale adoption of secure development practices without a prioritized perspective. Not only is such a wholesale alter-

native likely to be less effective, but it is also likely to be met with political opposition within commercial organizations. A wholesale approach lacks cost-benefit justification and, due to the aforementioned needle in a haystack phenomenon, becomes progressively more futile and psychologically daunting as a product grows in size. We believe prediction models based on change and entropy-based architectural metrics may offer a practical means for identifying and prioritizing attack-prone components.

REFERENCES

OSVDB: Open sourced vulnerability database. osvdb.net (online). http:// www.osvdb.net/, accessed May 30, 2013.

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

Anan, M., Saiedian, H., and Ryoo, J. (2009). An architecture-centric software maintainability assessment using information theory. *J. Softw. Maint. Evol.: Res. Pract.*, 21(1):1–18.

Bell, R. M., Ostrand, T. J., and Weyuker, E. J. (2011). 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. ACM.

Bozorgi, M., Saul, L., Savage, S., and Voelker, G. M. (2010). 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.

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

Chowdhury, I. and Zulkernine, M. (2011). Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313.

Gousios, G. (2012). On the importance of tools in software engineering research. Blog. Accessed: 02/20/2013.

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

Hassan, A. E. (2009). 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. IEEE Computer Society.

Jackson, D. and Wing, J. (1996). Lightweight formal methods. *IEEE Computer*, 29(4):16–30.

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

Khoshgoftaar, T. M., Allen, E. B., Goel, N., Nandi, A., and McMullan, J. (1996). 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. IEEE Computer Society.

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

McGraw, G. (1999). Software assurance for security. *Computer*, 32(4):103–105.

Mell, P., Scarfone, K., and Romanosky, S. (2007). *CVSS: A Complete Guide to the Common Vulnerability Scoring System Version 2.0*. FIRST: Forum of Incident Response and Security Teams.

Moser, R., Pedrycz, W., and Succi, G. (2008). 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. IEEE.

Munson, J. C. and Elbaum, S. G. (1998). 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 Computer Society.

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

Nagappan, N. and Ball, T. (2005). 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. ACM.

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

NIST. NVD:national vulnerability database. National Institute of Science and Technology, online. http:// nvd.nist.gov/, accessed May 30, 2013.

Ostrand, T. J., Weyuker, E. J., and Bell, R. M. (2010). Programmer-based fault prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, PROMISE '10, New York, NY, USA. ACM.

Saltzer, J. H. and Schroeder, M. D. (1975). The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 9(63):1278–1308.

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

Shin, Y. (2011). *Investigating Complexity Metrics as Indicators of Software Vulnerability*. PhD thesis, North Carolina State University, Raleigh, North Carolina.

Shin, Y., Meneely, A., Williams, L., and Osborne, J. A. (2011). Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *Software Engineering, IEEE Transactions on*, 37(6):772–787.

Younis, A., Malaiya, Y., and Ray, I. (2014). 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.