Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista

Thomas Zimmermann [1]
tzimmer@microsoft.com

Nachiappan Nagappan [1]
nachin@microsoft.com

Laurie Williams [2]
williams@csc.ncsu.edu

[1] *Microsoft Research, Redmond, WA, USA*
[2] *Department of Computer Science, North Carolina State University, Raleigh, NC, USA*

***Abstract*—Many factors are believed to increase the vulnerability of software system; for example, the more widely deployed or popular is a software system the more likely it is to be attacked. Early identification of defects has been a widely investigated topic in software engineering research. Early identification of software vulnerabilities can help mitigate these attacks to a large degree by focusing better security verification efforts in these components. Predicting vulnerabilities is complicated by the fact that vulnerabilities are, most often, few in number and introduce significant bias by creating a sparse dataset in the population. As a result, vulnerability prediction can be thought of us preverbally "searching for a needle in a haystack." In this paper, we present a large-scale empirical study on Windows Vista, where we empirically evaluate the efficacy of classical metrics like complexity, churn, coverage, dependency measures, and organizational structure of the company to predict vulnerabilities and assess how well these software measures correlate with vulnerabilities. We observed in our experiments that classical software measures predict vulnerabilities with a high precision but low recall values. The actual dependencies, however, predict vulnerabilities with a lower precision but substantially higher recall.**

Keywords—Vulnerabilities, Prediction, Metrics, Complexity, Churn, Coverage, Dependencies, Organizational Structure

I. INTRODUCTION

Software security is a critical part of the software development process. While there is a significant body of work on predicting defects, unfortunately little is known about the field of vulnerability prediction. Some recent work focused on this topic in the open source domain [9][15][22]. In this paper, we focus on vulnerability prediction for a proprietary commercial product (Windows Vista). We define a component to be vulnerable if it has been changed as part of a security update after it was released publically.

Software security and reliability are two crucial aspects of software engineering research. Software security research spans several domains ranging from better programming language design suited for security to the use of processes like penetration testing, design and use of robust access control policies [10]. For example, a software systems can be reliable (i.e., works as expected) but not secure or a software system can be secure (e.g., adopting threat modeling effectively, eliminating buffer overflows programmatically, etc.) but not reliable (does not work as expected). To better address both security and reliability, it is essential to understand differences and similarities between these two fields.

Towards that end, we leverage existing metrics that have been used in prior research for defect prediction [17][18][19][20] to understand and investigate the efficacy of these metrics for vulnerability prediction. More formally, our research hypothesis is **to investigate and report on the ability of classical defect prediction metrics to be used as predictors for vulnerability prediction.** For this purpose, we study Windows Vista, which is a large and widely-used commercial operating system from Microsoft Corporation. A statistical challenge in our study is motivated by the fact that vulnerabilities are few and widely distributed in the dataset akin to searching for a needle in a haystack. In our study for example, only 66 advisories have been recorded for Vista (40 Million plus lines of code) in the National Vulnerability Database (NVD) [21], and only few of the Windows binaries are affected by security updates.

This statistical challenge involves identifying which of the classical metrics related to code quality can predict vulnerabilities. We extract complexity, churn, coverage, dependency metrics for Vista and used them to predict the vulnerabilities that are found and fixed in Vista as dependent variable. Our results are as follows:

- Metrics correlate with vulnerabilities; however the effect is only small (Section IV).
- Most metrics can predict vulnerabilities with an average to good precision; however the recall is very low (Section V.B).
- Alternative techniques such as using the actual dependencies of a binary to predict vulnerabilities have better recall values (Section V.C).

The paper is organized as follows. Section II describes the metrics that we collected and used in our experiment. Section III characterizes our vulnerabilities based on public data available in the NVD database. Section IV discusses the correlation results between the collected metrics and

vulnerabilities in Windows Vista. Section V presents the results of our experiment on predicting vulnerabilities in Windows Vista. Section VI presents threats to validity of our study. Section VII discusses related work in the context of our experiment and Section VIII concludes with future work.

II. DATA COLLECTION

In this section, we discuss the various metrics that we used in our experiment to predict vulnerabilities. The discussed measures have been used for predicting defects in prior research both within and outside of Microsoft. The measures can be broadly classified into five categories. Throughout the paper, we will refer to the metrics in subsections (i)-(v) as *"classical metrics"*.

(i) Code Churn Measures [17]:

a. *Total Churn*: The total added, modified, and deleted lines of code of a binary during the development of Vista. For our experiments on Windows Vista the churn is measured relative to Windows Server 2003, the release before Vista.
b. *Frequency*: The number of times that a binary was edited during its development cycle. The implication is the greater the number of edits, the greater the risk of vulnerabilities.
c. *Repeat Frequency*: The number of consecutive edits that are performed on a binary. A consecutive edit is when a binary is edited between builds N and N+1 and then again between builds N+1 and N+2. This is a measure of the instability of the binary during its development. The greater the repeat frequency, the greater the instability of the binary during its development.

(ii) Code Complexity Measures [19]:

a. *(Max)(Total) Cyclomatic complexity* [13] measures the number of linearly-independent paths through a program module.
b. *(Max)(Total) Fan-In:* number of functions calling a function.
c. *(Max)(Total) Fan-Out:* number of functions called by a function.
d. (Max)(Total) Lines of Code (LOC).
e. (Max)(Total) Weighted methods per class (if any).
f. (Max)(Total) Depth of Inheritance (if any).
g. (Max)(Total) Coupling between objects (if any).
h. (Max)(Total) Number of sub classes (if any).
i. Total Global variables.

For each of the code complexity metrics, we collect two measures (Max) and (Total) across the entire system (metrics are computed on binary level). *Max* is the maximum value of the metric across all components (or files) in the system, and *Total* is the total value of the metric across the entire system.

(iii) Dependency Measures [18]:

For dependencies we compute both data dependencies and call dependencies at the function level, including caller-callee dependencies, imports, exports, RPC, COM, Registry access. The dependencies are rolled up to the binary level. For each binary, we compute the following dependency metrics.

a. *Incoming direct:* The number of incoming direct dependencies to a binary.
b. *Incoming closure:* The number of incoming indirect dependencies to a binary.
c. *Outgoing direct:* The number of outgoing direct dependencies from a binary.
d. *Outgoing closure:* The number of outgoing indirect dependencies from a binary.
e. *Layer information:* The distance of a binary from the system hardware (CPU), i.e., the Kernel, in the architectural layering of Windows.

(iv) Code coverage Measure:

For each binary within Windows Vista, we compute the total block and arc coverage measures.

a. Block coverage: A (basic) block is a set of contiguous instructions (code) in the physical layout of a binary that has exactly one entry point and one exit point. Calls, jumps, and branches mark the end of a block. A block typically consists of multiple machine-code instructions. The number of blocks covered during testing constitutes the block coverage measure.
b. Arc coverage: Arcs between blocks represent the transfer of control between basic blocks due to conditional and unconditional jumps, as well as due to control falling through from one block to another. Similar to block coverage the proportion of arcs covered in a binary constitute the arc coverage. Arc coverage can also be called branch coverage.

(v) Organizational Measures [20]:

a. *Number of Engineers (NOE):* This is the absolute number of unique engineers who have touched a binary and are still employed by the company.
b. *Number of Ex-Engineers (NOEE):* This is the total number of unique engineers who have touched a binary and have left the company as of the release date of the software system.
c. *Edit Frequency (EF):* This is the total number of times the source code that makes up the binary was edited. An edit is when an engineer checks out code from the version control system, alters it, and checks it in again. This is independent of the number of lines of code altered during the edit.
d. *Depth of Master Ownership (DMO):* This metric determines the level of ownership of the binary depending on the number of edits done. The organization level of the person whose reporting engineers perform more than 75% of the rolled up edits is considered as the DMO. This metric determines the binary owner based on activity on that binary. Our choice of 75% is based on prior historical information on Windows to quantify ownership.

e. *Percentage of Org contributing to development (PO):* The ratio of the number of people directly reporting at the DMO level relative to the total org size at the DMO level.
f. *Level of Organizational Code Ownership (OCO):* The percent of edits from the organization that contains the binary owner (or if there is no owner the percent of edits from the organization that made the majority of the edits to that binary).
g. *Overall Organization Ownership (OOW):* This is the ratio of the people at the DMO level making edits to a binary relative to total engineers editing the binary. A high value is good.
h. *Organization Intersection Factor (OIF):* The number of different organizations that contribute more than 10% of edits, as measured at the level of the overall org owners.

(vi) Actual Dependencies [26][28]:
We use dependency relationships among the binaries of Vista to predict vulnerabilities. The dependencies of a binary are an implicit description of its problem domain. For example, applications that access the Internet will share similar dependencies and also have a similar vulnerability profile.

The use of dependencies is motivated by an earlier study by Schröter et al. [26] who showed that import dependencies can predict defects for Eclipse. We replicated the study for arbitrary dependencies on Windows Vista and found similar result for defects [28]. Neuhaus et al. showed that for Firefox dependencies can also predict vulnerabilities [22].

III. Characterizing Vulnerabilities in Vista

To characterize the vulnerabilities in Windows Vista we used data from the National Vulnerability Database (NVD) [21]. The NVD database contains over 35,000 publicly known security vulnerabilities. For Vista 66 vulnerabilities were reported as of April 2009.

Each entry in the NVD database comes with values for Common Vulnerability Scoring System (CVSS) metrics [14] that capture the characteristics of the vulnerability in terms of access and impact. In this Section, we summarize the CVSS metrics for the 66 Vista vulnerabilities.

The CVSS metrics Access Vector, Access Complexity, and Authentication describe how the vulnerability can be accessed and what conditions are required to exploit it.

- **Access Vector.** This metric indicates from where an attacker can exploit the vulnerability. Of the Vista vulnerabilities, 19 can be exploited only with *physical access* to the machine, one can be exploited through an *adjacent network* (e.g., IP subnet or Bluetooth), and 46 can be exploited *remotely*.
- **Access Complexity.** This metric measures the complexity of attacks exploiting the vulnerability. A vulnerability with low complexity can be for example a buffer overflow in a web server, the vulnerability can be exploited at will. In contrast a vulnerability in an email client can be of high complexity, if the user has to perform several suspicious steps before the vulnerability is accessed. For Vista, the access complexity is *low* for 32 vulnerabilities, *medium* for 31 vulnerabilities, and *high* for three vulnerabilities.
- **Authentication.** This metric counts how often an attacker must authenticate before the vulnerability can be exploited. For 62 vulnerabilities in Vista, *authentication was not required* to exploit the vulnerability; only four vulnerabilities required the attacker to be *logged onto the system*.

The CVSS impact metrics measure how much the vulnerability will affect a user, once it is exploited, with respect to confidentiality, integrity, and availability.

- **Confidentiality Impact.** Of the vulnerabilities in Vista, 17 had *no impact* to the confidentiality of the system, 7 had *partial* information disclosure, and 42 had *total* information disclosure, which means that an attacker is able to read all of the system's files.
- **Integrity Impact.** Of the vulnerabilities in Vista, 15 had *no impact* on the integrity of the system, for 10 vulnerabilities the attacker is able to modify *some files*, and for 42 the attacker is able to modify *all files*.
- **Availability Impact.** For 10 vulnerabilities, there was no impact on the availability of the system, for 8 there was reduced performance and for 48 the attacker is able to shut down the system completely.

The values for the above metrics can be combined into a single **CVSS base score** which takes values from 0 (low severity) to 10 (highest severity). For the 66 Vista vulnerabilities the CVSS base scores range from 1.9 to 10, with an average of 7.5 and median of 7.2.

IV. Correlation Analysis

In a first analysis we computed the correlations between the metrics described in Section II.(i)-(v) and the number of vulnerabilities per binary. We used the Spearman rank correlation, which is a robust technique that can be applied even when the association between values is non-linear [6]. The closer the value of a correlation is to –1 or +1, the higher two measures are correlated—positively for +1 and negatively for –1. A value of 0 indicates that two measures are independent. We also computed the statistical significance of each correlation to ensure that our results are not random. All correlations were significant at $p<0.0001$, except for *Layer information* and *Outgoing closure* from the dependency measures and *Percentage of Org, Overall Organization Ownership,* and *Organization Intersection Factor* from the organizational measures.

Table I shows the metrics for which we found significant correlations. Values greater than 0.10 can be considered a small effect size; values greater than 0.30 can be considered a medium effect size [4]. All our correlations are positive, which means that for an increase in the metric, the number of vulnerabilities increases as well. However, we note that all effects are small.

We can observe the highest correlation values for metrics related to edits *(Edit Frequency, Frequency, Repeat*

TABLE I. SPEARMAN CORRELATION VALUES WITH NUMBER OF VULNERABILITIES.

Metric	rho
Edit Frequency (EF)	0.292
Total Lines of Code	0.281
Frequency	0.279
Total Complexity	0.276
Repeat Frequency	0.273
Number of Ex-Engineers (NOEE)	0.270
TotalFanIn	0.263
TotalFanOut	0.262
Number of Engineers (NOE)	0.261
Total Global Variables	0.255
Total Churn	0.254
Max FanIn	0.224
Max Complexity	0.207
Max FanOut	0.196
Max Lines of Code	0.194
Outgoing direct	0.168
Total ClassMethods	0.167
Max ClassMethods	0.164
Total InheritanceDepth	0.161
Total BlockCoverage	0.157
Incoming direct	0.156
Tota ClassCoupling	0.154
Total ArcCoverage	0.152
Incoming closure	0.148
Total SubClasses	0.141
Max InheritanceDepth	0.137
Max ClassCoupling	0.137
Max SubClasses	0.124
Level of Org. Code Ownership (OCO)	0.123
Depth of Master Ownership (DMO):	0.101

All correlations values are significant at p<0.0001.

Frequency, Editing Ex-Engineers) and size and complexity of *binaries (Total Lines of Code, Total Complexity)*. This observation suggests that binaries with frequent changes by a large number of engineers are more prone to vulnerabilities. Meneely and Williams found a similar phenomenon in a study of Red Hat Linux [15]. Our study indicates an accentuated effect of vulnerabilities in binaries with frequent changes by engineers who have left the company. A possible explanation for this is that when engineers leave, knowledge about the structure and dependencies of the component is lost. Similarly, binaries with many lines of code and high complexity are more prone to vulnerabilities.

Classical metrics correlate with the number of vulnerabilities; however the effect is only small.

V. PREDICTING SECURITY VULNERABILITIES

In this section, we describe an analysis to provide a predicted classification of which binaries in Windows Vista will have vulnerabilities. Every binary is either predicted to have *no vulnerabilities* or to have *one or more vulnerabilities.* We first describe our general experimental setup which consists of 100 random splits (Section 5.1). Next, we discuss the results for logistic regression and metrics (Section 5.2) and for support vector machines on dependency relations (Section 5.3).

A. Experimental Setup.

To evaluate the predictive power of our models, we use a standard evaluation technique: data splitting [16]. That is, we randomly pick two-thirds of all binaries *(training set)* to build a prediction model and use the remaining one-third (*testing set*) to measure the efficacy of the built model. For every experiment, we performed 100 random splits to ensure the stability and repeatability of our results. Whenever possible, we reused the random splits to facilitate comparison of results.

Because the percentage of vulnerable binaries was very low (needle in the haystack), we use *stratified sampling* for choosing the training and testing sets. This ensures that there are always a sufficient number of vulnerable binaries in the training set to learn from (in contrast, when choosing binaries entirely randomly with naïve sampling, the training set might have zero vulnerable binaries, leading to a trivial model that classifies everything the same). In addition, stratified sampling ensures that the ratio of vulnerable binaries in the training and testing sets remains constant across the random splits.

To assess the quality of the prediction models, we computed precision and recall. To explain these two measures, we use the following contingency table.

		Observed	
		Vulnerable	Non-Vulnerable
Predicted	Vulnerable	A	B
	Non-Vulnerable	C	D

The *recall A/(A+C)* measures the percentage of binaries observed as vulnerable that were classified correctly. The fewer false negatives (missed binaries), the closer the recall is to 1.

The *precision A/(A+B)* measures the percentage of binaries percentage of binaries predicted as vulnerable that were classified correctly. The fewer false positives (incorrectly predicted as vulnerable), the closer the precision is to 1.

Both precision and recall should be as close to the value 1 as possible (=no false negatives and no false positives). However, such values are difficult to realize since precision and recall counteract each other.

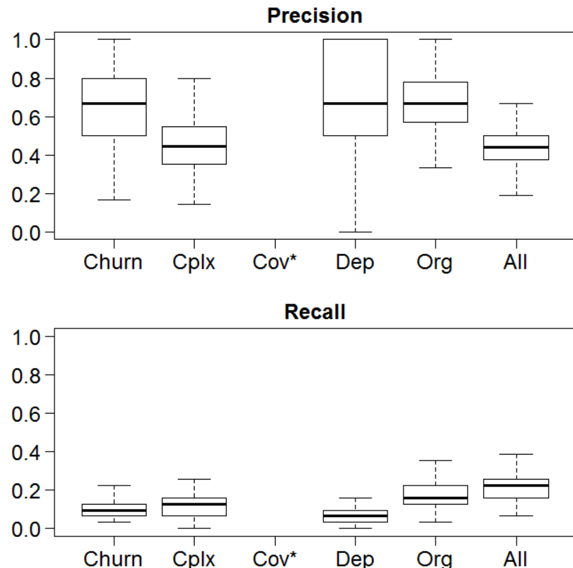

Figure 1. Precision and recall for predicting binaries as vulnerable. [* The coverage models classified all binaries the same, which results either in precision or recall of 1]

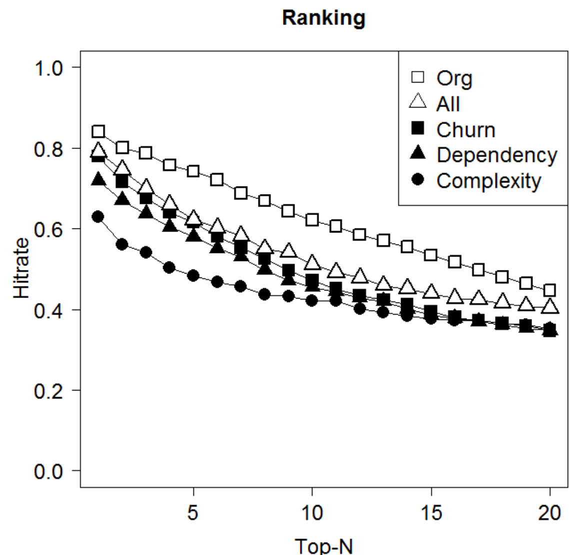

Figure 2. Hitrate for the Top-20 binaries.

B. Predicting with Classical Metrics

To predict vulnerabilities with the classical metrics from Section II.(i)-(v), we used binary logistic regression. Logistic regression predicts likelihoods between 0 and 1. In our case, the likelihoods can be interpreted as the "*vulnerableness*", i.e., how likely a binary contains at least one vulnerability.

For classification, we used a threshold of 0.50, i.e., all binaries with a vulnerableness of less than 0.50 were predicted as free of vulnerabilities, while binaries with a vulnerableness of at least 0.50 were predicted as vulnerable.

We ran six different experiments (recall that a single experiment consists of 100 random splits). We did one experiment for each of the five groups of metrics in Section II: *Churn*, Complexity *(Cplx)*, Coverage *(Cov)*, Dependency Measures *(Dep)*, and Organizational Structure *(Org)*. For the sixth experiment, we used the metrics of all groups combined *(All)*.

The results of the experiments are summarized in Figure 1 as box plots. Each experiment is represented by two box plots, one for precision and one for recall. A box plot shows the minimum value (lowest horizontal line), the maximum value (highest horizontal line), the lower quartiles (lower vertical line of the box), the upper quartile (upper vertical line of the box), and the median (thick vertical line dividing the box). For example, in Figure 1 the box plot for *Precision* and *Churn* shows that in the 100 random splits, the minimum precision was 0.167, and reached up to 1.000. The median precision was 0.667 (thick line).

Sometimes prediction models classify everything the same, which would result in either a precision of 1 (and a recall of 0) or a recall of 1 (and a very low precision). However, in practice such trivial models are useless because they cannot support any decision making. From a modeling perspective, such cases indicate that there is not enough information available to make predictions. For the box plots in Figure 1, we ignored such cases, which except for the Coverage experiment occurred very rarely (in seven out of 500 splits). For Coverage, all 100 random splits yielded trivial models, which is why we cannot show any meaningful box plots in Figure 1.

The highest median precision in our experiments was 0.667 for Churn, Dependency Measures (Dep), and Organizational Structure (Org). In other words, two out of three binaries predicted as vulnerable, are actually vulnerable. However, it has to be noted that in our experiments the precision widely varied across splits, especially for Churn and Depends Measures.

The recall values are disappointing. The highest median recall was roughly 0.2 for the combined model (All)*,* which means that on average only one out of five vulnerabilities can be identified. In some splits, the recall improved to 0.4 (two out of five), which is still not a very high value. Organization Structure had slightly lower, but comparable recall values to the combined model.

We also looked at top binaries predicted as most vulnerable by each model. For this, we predicted for each binary in the testing set the vulnerableness, and then ranked by the vulnerableness (high-to-low). The results of this experiment are in Figure 2, for Top-1 to Top-20. For this experiment, we ignore the models built from Coverage metrics (because as mentioned above they turned out to be trivial models).

When ranking binaries, the top-most binary is vulnerable between 63% (Complexity) and 84% (Org), for the Top-10 binaries the hit rate is still between 42% and 62%. This observation shows that metrics can effectively rank vulnerable binaries to the top. Rather than inspecting binaries

without any order, the binaries should be inspected in order of decreasing predicted vulnerableness.

Most metrics predict vulnerabilities with an average to good precision; however the recall is very low.

C. Predicting with Dependencies

We also built prediction models that use the targets of the dependencies of a binary as input. For example, for a binary that depends on Foo.exe, Bar.dll, and Qux.dll, we used Foo.exe, Bar.dll and Qux.dll to make a prediction. More formally, the input for our model is a high-dimensional bit vector, with one bit for every possible dependency target. In the example above, we would set the bits for Foo.exe, Bar.dll, and Qux.dll. The output is again a classification of whether the binary would contain vulnerability or not. Using just the dependency relationships is inspired by a study of Schröter et al. for software defects [26], which was replicated by Neuhaus et al. for vulnerabilities [22].

Because of the high dimensionality of the input data (every possible dependency target is considered as one dimension), classical regression models would be doomed to overfit the data. Instead we rely on Support Vector Machines (SVMs) [2][25]. They have been used on similar datasets [26], and achieved better results than linear regression, regression trees, and ridge regression, possibly because SVMs are less prone to overfitting.

To make a prediction for a binary from the test set, we compute its dependencies and represent them as a bit vector. The bit vector then serves as the input to the SVM built from the training data. The SVM then classifies the test binary as vulnerable or not vulnerable.

The results of SVMs for predicting vulnerable binaries are shown in Figure 3 in a precision recall diagram. For each random split we create one point with the precision value on the x axis and the recall value on the y axis. The median precision of all experiments is 0.6 which is comparable to the results for the metrics in the previous section. However, for dependency relations the recall increases substantially. In the experiments recall values ranged between 0.2 and 0.6. The median is now 0.40 (compared to 0.2 for metrics).

The dependencies of a binary predict vulnerabilities with better recall values than classical metrics.

D. Discussion

A possible explanation for the increase in recall is that the dependencies of a binary describe its problem domain. Some domains are simply more likely to face vulnerabilities. For example binaries that connect to the Internet will share certain dependencies and a more likely to have vulnerabilities. For an effective prediction of vulnerabilities the domain and functionality of a binary has to be taken into account, which is impossible by just using software metrics.

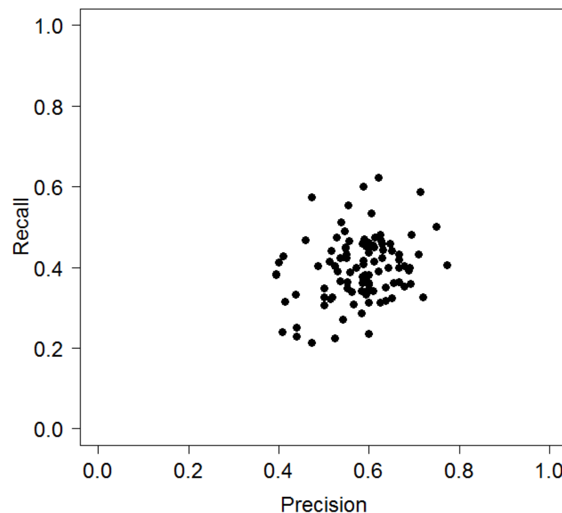

Figure 3. Precision and recall for actual dependencies.

Other relevant factors that should be included for prediction are the complexity of an attack based on a vulnerability. For example, for some binaries it will be very hard for attackers to exploit a vulnerability, maybe because the attacker needs local access to the machine, several passwords, and the user needs to perform suspicious actions. These binaries are less likely to have critical vulnerabilities that matter. The importance of vulnerabilities is hard to capture with existing software metrics. However, finding latent vulnerabilities is still useful because they might be exploited at a later point in time.

In short our results show that predicting security vulnerabilities is possible. However the results can still be improved. We believe that the key for doing this is by (1) developing new prediction techniques that deal with the "needle in the haystack" problem; and (2) finding new metrics that deal with the unique characteristics of vulnerabilities and attacks.

VI. Threats to Validity

As stated by Basili et al. [1], drawing general conclusions from empirical studies in software engineering is difficult because any process depends on a potentially large number of relevant context variables. For this reason, we cannot assume a priori that the results of our study generalize beyond the specific environment in which it was conducted.

Since this study was performed on the Windows Vista operating system and the size of the code base and development organization is at a much larger scale than many commercial products, it is possible that the specific models built for Windows would not apply to other products, or newer versions of Windows.

VII. Related Work

To the best of our knowledge, only few empirical studies exist for software vulnerabilities. There exists a large body of work on defect prediction for which we refer to a survey by Catal and Diri [3].

Shin and Williams [27] correlated several complexity measures with the number of security problems, for the JavaScript Engine of Mozilla, but found only a weak correlation. This result indicates that there are further factors that influence vulnerabilities, and that there is a need for new metrics for prediction of vulnerabilities.

Gegick et al. used code-level metrics such as lines of code, code churn, and number of static tool alerts [8] as well as past non-security faults [7] to predict security faults. In the most recent work, Gegick et al. achieved a precision of 0.52 and a recall of 0.57. Gegick et al. created a vulnerability prediction model using security-related static analysis alerts, code churn, size and inspection faults. The model was used to rank the components of a large Cisco system based upon the likelihood the component contained a vulnerability. The model predicted that 75.6% of the vulnerabilities could be found in the top 18.6% of the components [9].

Neuhaus et al. [22] investigated the Mozilla project and found a correlation between vulnerabilities and imports (that is, the *include* directives in source files). They used the imports to predict vulnerabilities with SVMs. We replicated their study as part of Section 5.3. Our median precision of 0.60 and recall of 0.40 is roughly comparable to the study by Neuhaus et al. who reported average precision of 0.70 and recall of 0.45.

Neuhaus and Zimmermann [23] analyzed RedHat packages and their dependencies and build a model to predict packages with vulnerabilities. While the goal is similar to our study, the level of analysis is completely different. They focus on application level, while our study focused on component level within one single application.

Ozment at al. [24] and Li et al. [11] studied how the number of defects and security issues evolve over time. Di Penta et al. [5] tracked vulnerabilities across versions in order to investigate how different kinds of vulnerabilities evolve and decay over time.

Meneely and Williams [15] performed an empirical case study by examining correlations between the known security vulnerabilities in the open source Red Hat Enterprise Linux 4 kernel and developer activity metrics. Files developed by otherwise- independent developer groups were more likely to have a vulnerability. However, files with changes from nine or more developers were 16 times more likely to have a vulnerability than files changed by fewer than nine developers, indicating that many developers changing code may have a detrimental effect on the system's security.

VIII. Conclusion and Consequences

In this paper, we present the results of an empirical case study on the ability of classical metrics that have been used for defect prediction for vulnerability prediction. To the best of our knowledge this is the first large scale study carried out on a widely-deployed commercial OS like Windows wherein we explore the ability of a significant variety of measures ranging from code churn, complexity, dependencies to organization structure of the company building the software system.

Our results indicate that there is no one universal set of metrics that work efficiently for predicting vulnerabilities. Churn, complexity, coverage predict vulnerabilities with high precision but low recall values. Alternatively code dependencies predict vulnerabilities with low precision and high recall. Hence is possible to use a combination of metrics to obtain reasonable precision and recall while predicting defects.

Our results motivate future work in three areas:

(i) Vulnerabilities are not as simple to predict as defects. The "needle in the haystack" problem challenges standard statistical prediction methods. This to some degree explains that the precision and recall values for predicting vulnerabilities is not in the comparable range for predicting defects. Further, the risk of vulnerabilities depends also on usage and domain of the components.

(ii) To better predict vulnerabilities we therefore need **new measures that capture domain and usage**, (for example attack surface measurement [12] and better statistical techniques to deal with sparse data. As pointed out earlier for example Vista has 66 advisories have been recorded in the NVD database, and only few of the Windows binaries are affected by security updates.

(iii) We plan to leverage specific metrics related to software security like buffer overflows, integer overruns, arithmetic errors, spoofing attack bugs, repudiation bugs, denial of service attack bugs found during software development as a predictor of post-release vulnerabilities.

IX. Acknowledgements

Laurie Williams was a visiting researcher in Microsoft Research (MSR) Redmond when this work was performed. We would like to thank the Windows team at Microsoft and Brendan Murphy of MSR Cambridge for his help in understanding the data sources. We would also like to thank the anonymous ICST reviewers for valuable feedback on an earlier revision of this paper.

X. References

[1] V. R. Basili, F. Shull, and F. Lanubile, Building Knowledge Through Families of Experiments, IEEE Transactions on Software Engineering, vol. 25, pp. 456-473, 1999.

[2] B. E. Boser, I. Guyon, and V. Vapnik. A Training Algorithm for Optimal Margin Classifiers. In *Proceedings of the Fifth Annual ACM Conference on Computational Learning Theory*. COLT '92. 144-152.

[3] C. Catal and B. Diri. A systematic review of software fault prediction studies. *Expert Systems With Applications,* 2008.

[4] J. Cohen, Statistical Power Analysis for the Behavioral Sciences, second ed. Lawrence Erlbaum Assoc., 1988.

[5] M. Di Penta, L. Cerulo, and L. Aversano. The evolution and decay of statically detected source code vulnerabilities. *In Proc. Int'l. Working Conf. on Source Code Analysis and Manipulation* (SCAM), 2008.

[6] N. E. Fenton and S. L. Pfleeger, Software Metrics: A Rigorous and Practical Approach: Brooks/Cole, 1998.

[7] M. Gegick, P. Rotella, and L. Williams. Toward non-security failures as a predictor of security faults and failures. In *Proc. Int'l. Symposium on Engineering Secure Software and Systems (ESSoS)*, 2009.

[8] M. Gegick, L. Williams, J. Osborne, and M. Vouk. Prioritizing software security fortification throughcode-level metrics. In *QoP '08: Proc. of the 4th ACM workshop on Quality of protection*, pages 31–38. 2008.

[9] M. Gegick, P. Rotella, and L. Williams. Predicting Attack-Prone Components, In *International Conference on Software Testing, Verification, and Validation (ICST) 2009*, Denver, CO, pp. 181-190.

[10] Michael Howard and David LeBlanc. Writing Secure Code, Second Edition. 2002.

[11] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now? An empirical study of bug characteristics in modern open source software. In *Proc. Workshop on Architectural and System Support for Improving Software Dependability 2006*, pages 25–33, October 2006.

[12] P. K. Manadhata, J. M. Wing, M. Flynn, and M. McQueen: Measuring the attack surfaces of two FTP daemons. Proceedings of the 2nd ACM Workshop on Quality of Protection, QoP 2006. 3-10

[13] T. J. McCabe: A Complexity Measure. IEEE Trans. Software Eng. 2(4): 308-320 (1976)

[14] P. Mell, K. Scarfone, and S. Romanosky. CVSS. A Complete Guide to the Common Vulnerability Scoring System. Version 2.0. http://www.first.org/cvss/cvss-guide.html

[15] A. Meneely and L. Williams. Secure Open Source Collaboration: An Empirical Study of Linus' Law, ACM Computers and Communication Security (CCS), 2009.

[16] J. Munson and T. Khoshgoftaar. The Detection of Fault-Prone Programs, IEEE Transactions on Software Engineering, vol. 18, pp. 423-433, 1992.

[17] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05. 284-292.

[18] N. Nagappan and T. Ball. Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study. In *Proceedings of the First International Symposium on Empirical Software Engineering and Measurement*, ESEM '07. 364-373

[19] N. Nagappan, T. Ball, and A. Zeller. Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06. 452-461.

[20] N. Nagappan, B. Murphy, and V. Basili. The influence of organizational structure on software quality: an empirical case study. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08. 521-530.

[21] National Vulnerability Database. http://nvd.nist.gov/

[22] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller. Predicting vulnerable software components. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07. 529-540.

[23] S. Neuhaus, T. Zimmermann. The Beauty and the Beast: Vulnerabilities in Red Hat's Packages. In Proceedings of the 2009 USENIX Annual Technical Conference (USENIX ATC), June 2009.

[24] A. Ozment and S. E. Schechter. Milk or wine: Does software security improve with age? In *Proc. 15th Usenix Security Symposium*, August 2006.

[25] J. C. Platt, Fast Training of Support Vector Machines using Sequential Minimal Optimization. In *Advances in Kernel Methods - Support Vector Learning*, B. Schölkopf, C. J. C. Burges, and A. J. Smola, Eds.: MIT Press, 1998, pp. 185-208.

[26] A. Schröter, T. Zimmermann, and A. Zeller. Predicting component failures at design time. In *Proceedings of the 2006 ACM/IEEE international Symposium on Empirical Software Engineering*, ISESE '06. 18-27.

[27] Y. Shin and L. Williams. Is complexity really the enemy of software security? In *QoP '08: Proc. 4th ACM workshop on Quality of protection*, pages 31–38. ACM, 2008.

[28] T. Zimmermann and N. Nagappan. Predicting Defects with Program Dependencies (Short Paper). In *Proceedings of the Third International Symposium on Empirical Software Engineering and Measurement*, ESEM '09.