An Initial Study on the Use of Execution Complexity Metrics as Indicators of Software Vulnerabilities

Yonghee Shin
DePaul University
243 S. Wabash Avenue
Chicago, IL, 60604
1-312-362-6209
yshin@cdm.depaul.edu

Laurie Williams
North Carolina State University
890 Oval Drive, EBII
Raleigh, NC, 27695
1-919-513-4151
williams@csc.ncsu.edu

ABSTRACT
Allocating code inspection and testing resources to the most problematic code areas is important to reduce development time and cost. While complexity metrics collected statically from software artifacts are known to be helpful in finding vulnerable code locations, some complex code is rarely executed in practice and has less chance of its vulnerabilities being detected. To augment the use of static complexity metrics, this study examines execution complexity metrics that are collected during code execution as indicators of vulnerable code locations. We conducted case studies on two large size, widely-used open source projects, the Mozilla Firefox web browser and the Wireshark network protocol analyzer. Our results indicate that execution complexity metrics are better indicators of vulnerable code locations than the most commonly-used static complexity metric, lines of source code. The ability of execution complexity metrics to discriminate vulnerable code locations from neutral code locations and to predict vulnerable code locations vary depending on projects. However, the vulnerability prediction models using execution complexity metrics are superior to the models using static complexity metrics in reducing inspection effort.

Categories and Subject Descriptors
D.2.8 [**Software Engineering**]: Metrics – *complexity measures, product metrics.* D.2.4 [**Software Engineering**]: Software/Program Verification – *reliability, statistical methods.*

General Terms
Reliability, Security.

Keywords
Software security, software vulnerability prediction, complexity metrics, execution metrics.

1. INTRODUCTION
A single exploited software vulnerability can cause severe damage to an organization legally and financially [3]. Hence, detecting vulnerabilities before software release is essential. With limited time and budget in development teams, efficient allocation of inspection and testing effort is also critical. Therefore, finding software characteristics that can indicate the code locations that are likely to have vulnerabilities is important to prioritize inspection and testing effort.

Security experts often hypothesize that software complexity is the enemy of software security [8]. The experts' claims are intuitive because complex software systems are difficult to understand, test, and maintain [4,7]. Therefore, the more complex a software system is, the more chance that software engineers cannot fathom the security problems in the system and the more chance for software engineers to ignore security concerns. Hence, finding complexity metrics that can indicate software complexity may help organizations to find software vulnerabilities before software release and to efficiently allocate security inspection and testing effort. To this end, the usefulness of complexity metrics that can be collected statically from software artifacts as indicators of vulnerabilities has been investigated and empirically validated in our previous study [12,13]. However, although static complexity metrics were able to detect over 79% of vulnerable files, the result warrants improvement because of the large number of false positives [12,13]. One of the reasons that static complexity metrics alone cannot find well vulnerable code locations is that static complexity metrics are collected from the whole code base while some of the complex code may be rarely executed and hence rarely exposes vulnerabilities [5]. In such a case, metrics that reflect runtime behavior of software can be complementary.

A previous study by Khoshgoftaar et al. [6] showed that execution metrics that are obtained during software execution are effective in predicting fault-prone code locations. However, execution metrics have not been investigated in the context of security. Software vulnerabilities are exposed by security researchers and attackers. Therefore, metrics that are collected during software execution using the usage patterns of software by attackers may be effective for identifying vulnerable code locations. However, considering the difficulty in obtaining the usage patterns of software by attackers, collecting execution metrics that reflect the usage patterns of software by normal users first and examining the relationships between execution metrics and vulnerabilities can still provide benefits. If the execution metrics obtained from the normal user's usage patterns can indicate vulnerable code locations, organizations need to focus their security inspection and testing effort to the commonly used code areas. Otherwise, the software usage patterns by attackers are clearly different from

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SESS'11, May 22, 2011, Waikiki, Honolulu, HI, USA.
Copyright 2010 ACM 978-1-4503-0581-5/11/05…$10.00.

the usage patterns by normal users and operational profiles[1] based on attacker's behavior should be sought. In this study we are especially interested in the execution metrics that are collected from the usage patterns by normal users and that reflect complexity at runtime measured by frequency and duration of execution time.

The goal of this research is to investigate execution complexity metrics as indicators of vulnerable code locations to improve the efficiency of security inspection and testing. For this purpose, we performed empirical case studies on two widely used open source projects: the Mozilla Firefox web browser and the Wireshark network protocol analyzer. We collected execution complexity metrics from Firefox 3.0 and Wireshark 1.2.0 and examined whether execution complexity metrics can be used to discriminate vulnerable code locations from neutral code locations (code locations whose vulnerabilities have not been discovered yet) and whether execution complexity metrics can be used as predictors of vulnerable code locations. We also compared the effectiveness of execution complexity metrics and static complexity metrics in detecting vulnerable code locations.

The rest of this paper is organized as follows: Section 2 introduces the static complexity metrics and execution complexity metrics we collected for this study. Section 3 describes the case study design and evaluation criteria. Section 4 provides the results and discusses our findings and limitations. Section 5 provides related work and Section 6 summarizes our study.

2. METRICS

This section explains the metrics we collected for this study.

2.1 Static Complexity Metrics

Complex is defined as "1. composed of many interconnected parts; compound; composite, 2. characterized by a very complicated or involved arrangement of parts, units, etc. 3. so complicated or intricate as to be hard to understand or deal with" [1]. This definition tells us that complexity arises from the amount of entities and the degree of interaction between entities and those complexity results in difficulty in understanding when the amount of entities and the degree of interaction reach a certain level.

In the view of software, an entity can be a function, a file, or a component. The first definition of complexity tells that an entity with many sub-entities is more complex than an entity with a few sub-entities. For example, a file with many lines of code is more complex than a file with a few lines of code. The second definition of complexity tells us that an entity with high interaction with other entities is more complex than an entity with low interaction. Software entities can have interactions via function calls or data references. The third definition of complexity tells us that these compositions of entities and interactions between entities result in complex software and make software engineers difficult to understand code.

In our study, we collected two types of static complexity metrics: code complexity metrics and dependency network complexity metrics. Code complexity metrics are obtained from source code within a file. Dependency network complexity metrics are

[1] Operational profile is "a quantitative characterization of how a system will be used" depending on the environment that a system will be operated. [10]

Table 1. Static Complexity Metrics

Name	Definition
Code complexity metrics	
LOC	The number of lines of code in a file.
LOCVarDecl	The number of lines of code with variable declarations in a file.
NumFunctions	The number of functions defined in a file.
NumLinePreprocessor	The number of lines of code in a file devoted to preprocessing.
EssentialComplexity	The number of branches in a function after reducing all the programming primitives such as a `for` loop in a function's control flow graph into a node iteratively until the graph cannot be reduced any further. Completely well-structured code has essential complexity 1. Sum and maximum values of EssentialComplexity in each function aggregated at file level were used in this study.
CyclomaticStrictComplexity	The number of conditional statements in a function. Sum and maximum values of CyclomaticStrictComplexity in each function aggregated at file level were used in this study.
MaxNesting	The maximum nesting level of control constructs such as `if` or `while` statements in a procedure. Sum and maximum values of MaxNesting in each function aggregated at file level were used in this study.
CommentDensity	The ratio of lines of comments to the lines of code in a file.
FanIn	The number of inputs that a function uses such as parameters and global variables. Sum and maximum values of FanIn in each function aggregated at file level were used in this study.
FanOut	The number of outputs that are set in a function to parameters or global variables. Sum and maximum values of FanOut in each function aggregated at file level were used in this study.
Dependency network complexity metrics	
InDegree	The number of unique incoming edges to a node in a dependency network graph.
InDegree_w	The number of incoming edges to a node in a dependency network graph.
OutDegree	The number of unique outgoing edges from a node in a dependency network graph.
OutDegree_w	The number of outgoing edges from a node in a dependency network graph.
Betweenness	The number of times that a node sits on the shortest paths of other node pairs.
EvCent	(Eigenvector centrality). The value computed from an eigenvalue of the adjacency matrix of a dependency network graph in a way that a node connected to nodes with high eigenvector centralities has higher eigenvector centrality than a node connected to nodes with lower eigenvector centralities.

obtained from a software dependency network graph [16], where each node represents a file and each edge represents a dependency relationship that comes from function calls and data references between files. While code complexity metrics can be obtained at coding phase, dependency network complexity metrics can be obtained at design phase. Table 1 provides the definitions of the static complexity metrics used in this study.

2.2 Execution Complexity Metrics

In this study, execution complexity metrics measure the frequency of function calls and the duration of execution of functions. Execution time also has been known to be effective in fault prediction [6]. This study is interested in whether execution complexity metrics collected from common usage patterns can indicate vulnerable code locations. Specific use cases used in this study to represent common usage patterns are explained in Section 3. NumCalls measures the frequency of function calls. InclusiveExeTime and ExclusiveExeTime measure the duration of execution. While InclusiveExeTime measures the total execution time that is spent by a function and all the functions called by the function, ExclusiveExeTime measures the time spent within a function excluding the execution time spent by the called functions. Table 2 provides the definitions of execution complexity metrics.

Table 2. Execution Complexity Metrics

Name	Definition
NumCalls	The number of calls to the functions defined in a file.
InclusiveExeTime	Execution time for the set of functions, S, defined in a file including all the execution time spent by the functions called directly or indirectly by the functions in S.
ExclusiveExeTime	Execution time for the set of functions, S, defined in a file excluding the execution time spent by the functions called by the functions in S.

3. CASE STUDIES

This section describes our case study projects, data collection methods, and evaluation methods of discriminative power and predictive power of static complexity metrics and execution complexity metrics.

3.1 Projects under Study

Firefox is a widely-used open source web browser written in C/C++ and has been developed and maintained for over seven years. Firefox 3.0 consists of 7,895 C/C++ files with over 1,800 thousand lines of code (KLOC). Among those files, 301 files (3.8% of the files) had one or more discovered vulnerabilities at the time of data collection.

Wireshark is a popular open source network protocol analyzer written in C and has been developed and maintained for over ten years. Wireshark is used to find network issues, to locate bottlenecks, and to detect network intrusion. Wireshark can capture and analyze over 1,000 network protocols that arrive to the network interface. Vulnerabilities in the protocol analyzer can allow security breaches when the packets are manipulated to include malicious data to exploit vulnerabilities in Wireshark. Wrireshark 1.2.0 consists of 2,330 files with over 1,600 KLOC. Among those files, 181 files (7.8% of the files) had one or more discovered vulnerabilities at the time of data collection. Table 3 summarizes the two projects.

Table 3. Project statistics for Firefox 3.0 and Wireshark 1.2.0

	Language	LOC	# of Files	# of Vuln. Files	% of Vuln. Files
Firefox	C/C++	1,854,877	7,895	301	3.8%
Wireshark	C	1,648,589	2,330	181	7.8%

3.2 Data Collection

This subsection explains the methods we used to collect static complexity metrics, execution complexity metrics, and vulnerability data.

3.2.1 Static Complexity Metrics

Code complexity metrics were collected using a commercial code analysis tool, Understand C++ [2]. For dependency network complexity metrics, the file dependency graph was obtained from Understand C++. Then the dependency network complexity metrics were computed using a network analysis tool, igraph[3] package in R.

3.2.2 Execution Complexity Metrics

As mentioned in Section 1, our execution complexity metrics are intended to capture the software's runtime complexity from common usage patterns by normal users. For this purpose, execution complexity metrics were collected from the execution profile obtained at runtime by using an open source profiling tool, Callgrind[4]. Callgrind gathers the execution profile while software is executed on a simulated machine. The execution profile includes the number of function calls, caller/callee relationship, and the number of events such as data read at instruction level and cache miss. We used the number of events as a surrogate measure of execution time.

For Firefox, the execution complexity metrics were collected by performing the following usual tasks that the authors perform frequently using Firefox:

- Enter a search query from www.google.com and browse retrieved results.
- Login to www.gmail.com email account and read an email.
- Login to www.facebook.com and browse the posted messages.
- Login to a bank account and check the transactions in the current month.

This study excluded the execution profile during initialization of the web browser.

For Wireshark, the execution complexity metrics were collected by performing the following tasks:

- Capture the network packets while performing the above four operations for Firefox.
- Browse the detailed information of a packet from the captured packet list.

[2] http://www.scitools.com/

[3] http://igraph.sourceforge.net/

[4] http://valgrind.org/info/tools.html#callgrind/

As with Firefox, execution profile during initialization of the GUI was excluded. Most frequently captured protocols were TCP, HTTP, and DNS protocols.

3.2.3 Vulnerability Data

We collected vulnerabilities reported for Firefox 3.0 and its minor releases from the Mozilla Foundation Security Advisories (MFSAs)[5]. Each MFSA includes bug IDs for the bug reports in the bug database. From these bug reports, we identified the files that had changed to mitigate vulnerabilities. We counted the number of the bug reports that had bug patches for a file as a surrogate measure of the number of vulnerabilities in a file.

Each Wireshark security advisory[6] includes SVN revision number that we can identify the files that have been changed to fix vulnerabilities. We counted the number of file changes as a surrogate measure of the number of vulnerabilities in a file.

3.3 Discriminative Power and Mean Ratio

To evaluate the discriminative power of the metrics, this study tests the null hypothesis that the means of metric values for vulnerable files and neutral files are equal. For this purpose, we used the Welch's t-test to compare the means of metric values between vulnerable files and neutral files. The Welch's t-test is used to test the equality of means of two groups when data have unequal variance. A preliminary analysis of data showed that the data used in our study had unequal variance. In this study, a hypothesis for discriminative power is considered to be supported when the result from the Welch's t-test is statistically significant at the $p<0.05$ level. When multiple hypotheses are tested simultaneously, the probability of falsely rejecting at least one of the null hypothesis increases. To deal with this problem, Bonferroni correction was performed. That is, the number of the hypotheses that have been tested is multiplied to the p-value of the Welch's t-test before it is compared with 0.05. To see the magnitudes of the differences in metric values between vulnerable files and neutral files, we additionally measured the *mean ratio* defined as the ratio of the mean of metric values for vulnerable files to the mean of metric values for neutral files.

3.4 Predictive Power

To predict files that are likely to have vulnerabilities, we performed binary classification using logistic regression. In binary classification, a file is classified as vulnerable when the estimated probability of a file having a vulnerability is over a certain classification threshold (0.5 in our study). Otherwise, a file is classified as neutral. In Firefox 3.0, over 94% of files with vulnerabilities have less than three vulnerabilities. Therefore, we performed binary classification rather than ranking the files based on the number of predicted vulnerabilities.

A binary classifier can make two possible errors: *false positives (FP)* and *false negatives (FN)*. In this study, a FP is the classification of a neutral file as a vulnerable file, and a FN is the classification of a vulnerable file as neutral file. False positives can lead excessive files to be inspected or tested, and false negatives increase the chance of software being released with undetected vulnerabilities. A correctly classified vulnerable file is a *true positive (TP),* and a correctly classified neutral is a *true negative (TN).*

[5] http://www.mozilla.org/security/known-vulnerabilities/

[6] http://www.wireshark.org/security/

Among the many classification performance measures that can be derived from these four classification results, *recall*, *precision*, and *probability of false alarm* are most widely used to measure the performance of fault prediction or vulnerability prediction.

Recall is the ratio of correctly predicted vulnerable files to actual vulnerable files as defined in Equation (1) :

$$recall = \frac{TP}{TP + FN} \tag{1}$$

Precision is the ratio of correctly predicted vulnerable files to all predicted vulnerable files as defined in Equation (2) :

$$precision = \frac{TP}{TP + FP} \tag{2}$$

Probability of False alarm (PF) is the ratio of files incorrectly predicted as vulnerable to actual neutral files as in Equation (3) :

$$PF = \frac{FP}{FP + TN} \tag{3}$$

The desired result is to have a high recall, a high precision, and a low PF to find as many vulnerabilities as possible without wasting inspection or testing effort.

We are especially interested in whether complexity metrics are effective in reducing inspection effort. To this end, we additionally measured the reduction of file inspection that can be expected by using prediction models with complexity metrics compared to random file selection. For this purpose, we defined *File Inspection ratio* and *File Inspection Reduction* [12].

File Inspection (FI) ratio is the ratio of files predicted as vulnerable (that is, the number of files to be inspected) to the total number of files as defined in Equation (4):

$$FI = \frac{TP + FP}{TP + FP + TN + FN} \tag{4}$$

For example, recall=0.8 and FI=0.2 mean that within the 20% of files inspected based on the prediction results, 80% of vulnerable files can be found.

File Inspection Reduction (FIR) is the ratio of the reduced number of files to be inspected by using the model with complexity metrics compared to random file selection to achieve the same recall as defined in Equation (5):

$$FIR = \frac{recall - FI}{recall} \tag{5}$$

We similarly defined and measured the line inspection reduction (LIR) compared to random line selection.

LOC Inspection Reduction (LIR) is the ratio of reduced lines of code to be inspected by using a prediction model compared to a random file selection to detect the same number of vulnerabilities as defined in Equation (6):

$$LIR = \frac{PV - LI}{PV} \tag{6}$$

where *LI* is the ratio of lines of code to be inspected to the total lines of code, and *PV* is the ratio of the number of vulnerabilities in the files predicted as vulnerable to the total number of vulnerabilities.

To build a prediction model and to perform predictions on a data set that has not been used to train the model, we performed 10x10 cross-validation. For 10x10 cross-validation, we randomly split the data set into ten folds and used one fold for testing and the remaining folds for training, rotating each fold as the test fold. Each fold was stratified to properly distribute vulnerable files to both the training data set and the test data set. The entire process was then repeated ten times to account for possible sampling bias in random splits. Overall, 100 predictions were performed for 10x10 cross-validation.

Using many metrics in a model does not always improve the prediction performance since metrics can provide redundant information [9]. Therefore, we selected three metrics that provides highest information gain using the information gain ranking method to create prediction models [14]. The data used in this study is heavily unbalanced between majority class (neutral files) and minority class (vulnerable files). Prior studies have shown that the performance is improved by "balancing" the data. To balance the data, we performed under-sampling by removing randomly chosen majority class data until the numbers of data instances in the majority class and the minority class become equal.

4. RESULTS

By using the usage patterns by a normal user provided in Section 3, 24% of files for Firefox were executed and 11% of the executed files were vulnerable. For Wireshark, only 4% of files were executed and 19% of the executed files were vulnerable. Here an interesting observation is that the percentages of vulnerable files in the executed files (11% and 19%) are noticeably higher than the percentages of vulnerable files in total files (3.8% and 7.8%) for both projects. This result indicates that execution complexity metrics are good candidates of vulnerability indicators. Table 5 summarizes the execution statistics.

Table 5. Execution statistics

	% of Vuln. Files	# of Executed Files	% of Executed Files	# of Vuln. Files in Executed Files	% of Vuln. Files in Executed Files
Firefox	3.8%	1,909	24%	207	11%
Wireshark	7.8%	94	4%	18	19%

4.1 Discriminative Power and Mean Ratio

Table 4 shows the results of Welch's t-test and mean ratios. In Table 4, 20 of 23 metrics collected for Firefox show statistically significant discriminative power including two execution complexity metrics. However, for Wireshark, only around half of the metrics show statistically significant discriminative power and none of the execution complexity metrics provide statistically significant difference in the mean values between vulnerable and neutral files. Nonetheless, the means of metrics values for vulnerable files are higher than the ones for neutral files in all metrics regardless of statistical significance except for CommentDensity. The fact that the files with more vulnerabilities have lower comment density than the files with fewer vulnerabilities reinforce our belief on the importance of well commented code.

In Table 4, metrics that represent incoming information flow (SumFanIn, MaxFanIn, InDegree, InDegree_w) tend to not be discriminative of vulnerable files, while metrics that represent outgoing information flow (SumFanOut, MaxFanOut, OutDegree, and OutDegree_w) are discriminative of vulnerable files.

To summarize, there are subsets of complexity metrics that are discriminative of vulnerable files in each set of code complexity and dependency network complexity metrics for both projects. However, none of the execution complexity metrics collected for Wireshark shows significant discriminative power. Nonetheless, the means of the execution complexity metrics from vulnerable files are at least four times and up to 26 times higher than the means of the execution complexity metrics from neutral files for both projects.

Table 4. Discriminative power

	Firefox		Wireshark	
	Welch's t-test	Mean Ratio	Welch's t-test	Mean Ratio
Code complexity metrics				
LOC	√	4.3	X	1.8
LOCVarDecl	√	3.8	X	1.5
NumFunctions	√	4.3	X	1.8
NumLinePreprocessor	√	3.4	√	1.7
SumEssential	√	4.8	X	1.9
SumCyclomaticStrict	√	4.9	√	2.0
MaxCyclomaticStrict	√	3.1	√	1.7
SumMaxNesting	√	2.5	√	2.1
MaxMaxNesting	√	1.9	√	1.6
SumFanIn	X	1.8	X	1.8
MaxFanIn	X	1.1	X	2.5
SumFanOut	√	2.2	√	2.1
MaxFanOut	√	1.8	√	2.3
CommentDensity	√	0.3	X	0.5
Dependency network complexity metrics				
InDegree	√	2.3	X	2.2
InDegree_w	√	2.3	X	1.7
OutDegree	√	3.2	√	2.3
OutDegree_w	√	5.2	√	2.1
Betweenness	√	7.8	X	2.7
EvCent	√	3.2	√	1.5
Execution complexity metrics				
NumCalls	√	7.0	X	8.2
InclusiveExeTime	√	26.7	X	4.4
ExclusiveExeTime	X	7.4	X	8.6

4.2 Predictive Power

To analyze predictive power of code complexity, dependency network complexity, and execution complexity metrics, 10x10 cross-validation was performed. For each project, a prediction model using each type of metrics and a prediction model using all types of metrics were built. Table 6 shows recall, precision, PF, FIR, and LIR averaged over 100 runs of 10x10 cross-validation.

In Table 6, for the models with combined set of metrics, code complexity metrics, and dependency network complexity metrics, recall is medium to high in general (0.67 to 0.81); precision is very low (0.08 and 0.12); PF is between 0.23 and 0.52 for both projects. The model with execution complexity metrics provided similar recall, precision, and PF to the model with combined set of metrics for Firefox even though only 24% of the files have been executed. However, the model with execution complexity

metrics for Wireshark provided radically different results from the models with other types of metrics.

Note that compared to Firefox, much smaller percentage of files were executed for Wireshark (only 4%). This low coverage of files during code execution explains the low recall from the model with execution complexity metrics alone for Wireshark. However, the model with execution complexity metrics provides over twice as high precision as other models and also provides significantly higher file and line inspection reduction than other models.

Table 6. Predictive power

	Recall	Precision	PF	FIR	LIR
Firefox					
all	0.67	0.11	0.23	0.63	0.37
code	0.71	0.09	0.31	0.54	0.04
dependency	0.79	0.08	0.38	0.49	0.09
execution	0.67	0.11	0.21	**0.65**	**0.41**
Wireshark					
all	0.80	0.12	0.52	0.31	-0.04
code	0.78	0.11	0.52	0.31	-0.17
dependency	0.81	0.12	0.51	0.33	0.04
execution	0.10	0.28	0.03	**0.62**	**0.67**

4.3 Discussion

The prediction results from Firefox are impressive because the model with execution complexity metrics alone provides similar prediction results to the model with all types of metrics, and provides higher file and line inspection reduction. This result indicates that execution complexity metrics are superior to static complexity metrics.

However, the results from Firefox and Wireshark differ: Firefox provided better discriminative power than Wireshark in general; the recall from the model using execution complexity metrics for Firefox is much higher than the one for Wireshark. We surmise the difference is mainly because that the large differences in the percentages of the executed files for the two projects. A more fair comparison of the discriminative power and the predictive power between the two projects can be made by adding usage patterns to provide higher code coverage of Wireshark. In this study, we intend to examine whether the execution complexity metrics obtained from normal user's usage patterns are useful to detect vulnerable code locations to guide inspection and testing effort. Although we employed the four representative use cases in this study, we feel we need a more systematic way to identify the most common normal user's behaviors. While common usage patterns of the Firefox web browser are obvious, the usage patterns for Wireshark require further analysis because Wireshark can analyze over 1,000 network protocols while our usage patterns involve mainly TCP, HTTP, and DNS protocols.

Because not all files were executed, we further compared the number of vulnerable files in the 1,909 executed files and the number of vulnerable files in the 1,909 largest files. This way we are essentially comparing the prediction performance when each of the three execution complexity metrics and LOC is used as a single predictor of a vulnerability prediction model. Figure 1 shows the cumulated number of vulnerable files in the 1,909 largest files when the files are ordered in descending order of LOC and the cumulated numbers of vulnerable files when the files are ordered in descending orders of NumCalls, InclusiveExeTime, and ExclusiveExeTime, respectively. In Figure 1, the executed files include a larger number of vulnerable files than the largest

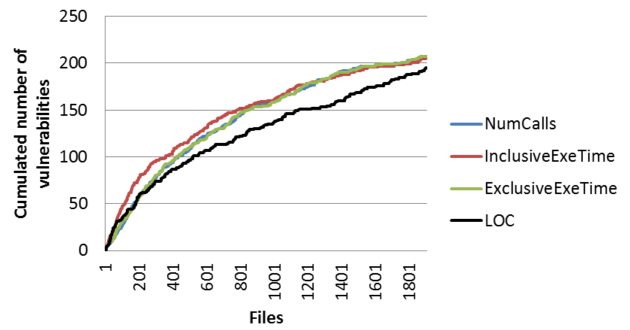

Figure 1. Cumulative number of vulnerable files for the top most complex files of Firefox

1,909 files. Figure 2 shows the similar graph but for the 94 executed files and the 94 largest files in LOC. Similar to Firefox, the executed files include a larger number of vulnerable files than the largest 94 files.

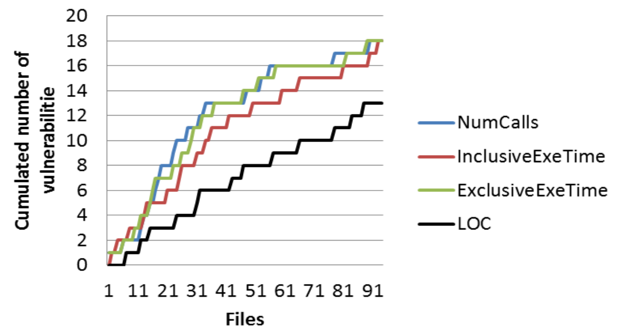

Figure 2. Cumulative number of vulnerable files for the top most complex files of Wireshark

Note that among those top largest files, 41% of files were executed for Firefox and only 3% of files were executed for Wireshark, showing that large files are not necessary executed and can have less chance of its vulnerabilities being exposed than executed files. We also analyzed other static complexity metrics in a similar way. For Firefox, the executed files included a larger number of vulnerable files than the 1,909 top most complex files for all other static complexity metrics defined in Table 1. For Wireshark, the 94 top most complex files for nine static complexity metrics included a larger number of vulnerable files than the executed files. However, this result may change if we add more use cases.

4.4 Threats to Validity

We collected vulnerability data from the publicly available security advisories. Although the long and wide usage history and mature development process of the two projects hint that the security advisories comprehensively include the existing vulnerabilities, there may be latent vulnerabilities that have not been discovered yet. Therefore, we believe the overall comparative tendency that can be observed from our study is more important than specific numbers.

Our results are limited to the two project used in this study. Therefore, more repeated studies are required and encouraged to generalize our observation.

Finally, the execution metrics depend on the use cases. Although we used representative use cases that we deem to be performed by

normal users, further analysis of usage behavior will improve our understanding on the relationship between execution complexity and vulnerabilities.

5. RELATED WORK

Complexity metrics have been effective to find fault-prone code locations [2,9,11]. General observations from these studies are that complexity metrics are useful to find fault-prone code locations, but most effective metrics vary depending on projects. Some recent studies have investigated complexity metrics as indicators of vulnerable code locations [12,15] together with other metrics such as process metrics. Although these studies provide the empirical evidence that complexity metrics are effective indicators of vulnerable code locations, vulnerability prediction suffer from a large number of false positives. Most prior studies used metrics that can be statically collected from software artifacts or development history. Execution metrics have rarely been used in fault prediction studies and none in vulnerability prediction to our knowledge. The most relevant work to ours was performed by Khoshgoftaar et al. [6] for fault prediction. Their study measured software execution time of four releases of a large telecommunications system together with code complexity and process metrics. Their classification tree model predicted fault-prone modules with less than 30% false positive rate and execution metrics were selected as effective variables in their model. Another study by Khoshgoftaar et al. [5] proposes dynamic complexity metrics that apply weights to static complexity metrics based on the probability that a module will be executed. However, the study does not mention about discriminative power or predictive power of those metrics for faults or vulnerabilities. Investigating the dynamic complexity metrics as indicators of vulnerabilities is our future work.

6. SUMMARY AND FUTURE WORK

This study investigated whether execution complexity metrics that were obtained from common usage patterns performed by normal users can be used as indicators of vulnerable code locations. The results of our case study on the Mozilla Firefox web browser show that execution complexity metrics can statistically significantly discriminative vulnerable code locations from neutral code locations. The model with execution complexity metrics also can predict vulnerable code locations with similar prediction performance to the model using the combined set of complexity metrics, but with lower inspection effort. Although the results from the Wireshark network protocol analyzer show that execution complexity metrics do not provide statistically significant discriminative power and provide very low recall of prediction mainly because of the low coverage of executed code, development teams can effectively reduce the inspection effort by using execution complexity metrics.

To generalize our observation, we plan to replicate our study with additional projects using more formal analyses of software usage patterns than the current method used in this study by consulting development teams or performing surveys. We also plan to investigate whether the execution complexity metrics are good indicators of faults to provide a baseline to measure the effectiveness of execution complexity metrics as indicators of vulnerabilities. Another interesting future work is to analyze the differences, if any, in the types of vulnerabilities that can be detected by using static complexity metrics and execution complexity metrics to further guide security inspection and testing effort.

7. ACKNOWLEDGMENTS

This work is supported in part by the National Science Foundation Cybertrust Grant No. 0716176 and the CAREER Grant No. 0346903. Any opinions expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

8. REFERENCES

[1] complexity. *Dictionary.Com unabridged*. http://dictionary.reference.com/browse/complexity.

[2] Basili, V.R., Briand, L.C., and Melo, W.L. A validation of object-oriented design metrics as quality indicators. *IEEE Trans. Software Eng. 22*, 10 (1996), 751-761.

[3] Cashell, B., Jackson, W.D., Jickling, M., and Webel, B. *CRS Report for Congress: The Economic Impact of Cyber-Attacks*. Congressional Research Service, 2004.

[4] Curtis, B., Sheppard, S.B., Milliman, P., Borst, M.A., and Love, T. Measuring the Psychological Complexity of Software Maintenance Tasks with the Halstead and McCabe Metrics. *IEEE Trans. Software Eng. SE-5*, 2 (1979), 96-104.

[5] Khoshgoftaar, T.M., Munson, J.C., and Lanning, D.L. Dynamic system complexity. *[1993] Proceedings First International Software Metrics Symposium*, 129-140.

[6] Khoshgoftaar, T.M., Shan, R., and Allen, E.B. Using product, process, and execution metrics to predict fault-prone software modules with classification trees. *Proc. Fifth Int'l Symp. on High Assurance Systems Engineering*, 301-310.

[7] McCabe, T.J. A Complexity Measure. *IEEE Transactions on Software Engineering SE-2*, 4 (1976), 308-320.

[8] McGraw, G. *Software Security: Building Security In*. Addison-Wesley Professional, 2006.

[9] Menzies, T., Greenwald, J., and Frank, A. Data Mining Static Code Attributes to Learn Defect Predictors. *IEEE Transactions on Software Engineering 33*, 1 (2007), 2-13.

[10] Musa, J.D. Operational profiles in software-reliability engineering. *IEEE Software 10*, 2 (1993), 14-32.

[11] Nagappan, N., Ball, T., and Zeller, A. Mining metrics to predict component failures. *Proceeding of the 28th Int'l Conf. on Software Engineering*, (2006), 452.

[12] Shin, Y., Meneely, A., Williams, L., and Osborne, J.A. Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities. *IEEE Transactions on Software Engineering (to appear)*, (2010).

[13] Shin, Y. Investigating Complexity Metrics as Indicators of Software Vulnerability. Ph.D. Dissertation. 2010.

[14] Witten, I.H. and Frank, E. *Data Mining: Practical Machine Learning Tools and Techniques, Second Edition*. Morgan Kaufmann, 2005.

[15] Zimmermann, T., Nagappan, N., and Williams, L. Searching for a Needle in a Haystack: Predicting Security Vulnerabilities for Windows Vista. *Third International Conference on Software Testing, Verification and Validation*, (2010), 421-428.

[16] Zimmermann, T. and Nagappan, N. Predicting defects using network analysis on dependency graphs. *Proceedings of the 13th Int'l Conf. on Software Engineering*, (2008), 531.