DECISION SCIENCES INSTITUTE

Examining the relationship between external software characteristics and vulnerabilities

Yaman Roumani
Eastern Michigan University
Email: yroumani@emich.edu

Joseph K. Nwankpa
University of Texas Rio Grande Valley
Email: joseph.nwankpa@utrgv.edu

Yazan F. Roumani
Oakland University
Email: roumani@oakland.edu

ABSTRACT

This study examines the relationship between external software characteristics and vulnerabilities. Based on the analysis of 227 vulnerable software products, our results showed that the number of compatible operating systems has a positive effect on vulnerability severity and vulnerability frequency.

KEYWORDS: vulnerability, severity, frequency, software, external characteristics

INTRODUCTION

Software vulnerabilities have been regarded as one of the key reasons for computer security breaches which result in enormous losses (Kirk 2013). With the growth of the software industry and the Internet, the number of vulnerability attacks and the ease with which an attack can be made have increased. Furthermore, since software developers are not always aware of security holes in their code, vulnerabilities will inevitably continue to be discovered after the software is released.

Most of the studies concerning software vulnerabilities have been exploratory in nature, with the majority dedicated to vulnerability detection and prevention and a few devoted to vulnerability predictions. Moreover, there has been a lack of quantitative studies in this area as most vulnerability studies rely on qualitative methods. Despite the existence of few predictive studies (Alhazmi and Malaiya 2005a 2005b 2006; Murtaza et al. 2016; Scandariato et al. 2014; Shin and Williams 2013), some of these studies have been known to have issues with their assumptions (Ozment 2007), while others are not generalizable and do not consider the different characteristics of the software products.

The research undertaken in this study proposes a novel vulnerability prediction model using software characteristics. This study will show that software characteristics can reveal patterns and trends that will offer new insights into the nature of vulnerabilities. The prediction capabilities of the proposed model will be examined to predict the vulnerability severity and vulnerability frequency. The proposed model will be empirically tested using vulnerability data collected from the National Vulnerability Database (NVD). The data analysis of this study will add to the understanding of vulnerability prediction models and will suggest directions for future research in this area. . Based on the analysis, we will be able to make some important research and practical contributions. The results are foreseen to help software companies and IT practitioners in analyzing their own vulnerability risks assessment and implementing any needed changes which can increase their capabilities in providing more secure software products. It is our belief that such prediction capabilities will expand the understanding of vulnerability prediction beyond the existing literature.

This paper is organized as follows. We first review the background literature on vulnerabilities and existing prediction models. We then present the research model. Next, we report an empirical study based on data collected from NVD, followed by a presentation of the data results. We conclude with a discussion of research findings, limitations and implications.

LITERATURE REVIEW

Vulnerabilities

According to Arbaugh et al. (2000), a vulnerability is a “technological flaw in an information technology product that has security or survivability implications”. Further studies defined software vulnerability as “a bug, flaw, behavior, output, outcome or event within an application, system, device, or service that could lead to an implicit or explicit failure of confidentiality, integrity, or availability” (Schiffman 2007).

Existing Vulnerability Prediction Models

Existing studies on software security measured security of a system from an operational security perspective (Littlewood et al. 1993). In their paper, Littlewood et al. (1993) analyzed the similarities between security and reliability. It was concluded that the use of effort of the attacker would be a suitable measure of software vulnerabilities. Alves-Foss and Barbosa (1995) proposed Security Vulnerability Index (SVI) to assess the vulnerability of computer systems to common intrusion methods. Based on predefined sets of SVI rules, this model would reveal system vulnerabilities. The SVI model used three factors to calculate SVI value: system characteristics, neglectful acts and malevolent acts. However, one of the drawbacks of SVI was the lack of measure of system factors. Moreover, the SVI model was designed to focus on operating systems only.

The next wave of vulnerability studies focused on predicting vulnerabilities using attack trees, privilege graphs and vulnerability density functions. For example, Dacier et al. (1996) and Ortaelo et al. (1999) proposed to model vulnerabilities using a privilege graph where every node represents user privileges and every edge represents a single vulnerability. The model can be used to construct a variety of ways where the attacker can exploit vulnerabilities and gain user privileges. The privilege graph is then transformed to a Markov chain based on successful attack patterns. Browne et al. (2001) used vulnerability data of Computer Emergency Readiness Team (CERT) to analyze vulnerability incidents trends. The authors concluded that the number of vulnerability incidents which are reported to CERT is related to square root of time. Their proposed model was able to predict the severity of future vulnerability exploitations based on earlier vulnerability reports. Other studies have used vulnerability density analogies. Vulnerability density relies on the number of vulnerabilities found per x lines of code to predict the future number of undiscovered vulnerabilities based on the maturity of the software product. Furthermore, vulnerability density assumes that different software versions are comparable to each other and have static code. Although vulnerability density models have been used to predict vulnerability, however, such models have failed to satisfy major assumptions about their data such as time and effort, operational environment, independence, and static code. Thus, the validity of such models has been questioned (Ozment 2007).

Recent studies on vulnerability prediction used different data mining techniques and mathematical models. Shin and William (2008) performed statistical analysis on nine complexity metrics and showed that complexity metrics can be used to predict vulnerabilities. However, their analysis was limited to JavaScript Engine of Mozilla application framework, thus, the authors concluded that their results may not be generalized to other software products. In their research, Murtaza et al. (2016) mined all vulnerabilities between 2009 and 2014. The authors found that the sequential patterns of vulnerabilities follow a first order Markov property. Moreover, Murtaza et al. (2016) concluded that it is possible to predict the next vulnerability by using the previous vulnerability with a recall of 80% and precision of 90%. However, the study did not take into consideration the different categories or the characteristics of software products.

RESEARCH MODEL

Krusl (1998) classifies software characteristics into internal and external ones. Internal characteristics such as size (lines of code), complexity, coupling…etc. are of interest mostly to software developers and engineers as they are related to the development of software. On the other hand, external characteristics such as compatibility, license, price…etc. are the main properties and features which are relevant to customers and adopters. External software characteristics are always used by customers to evaluate software products in order to make the adoption decision. Furthermore, for customers, external software characteristics are the only measure they can use to evaluate software quality. Based on the relevance of external software characteristics, our proposed model (figure 1) focuses on examining the relationship between such characteristics and vulnerabilities.

For our model, we chose four external software characteristics: software price, software programming language, software source code availability, and software compatibility. Our choice of these four characteristics was based on the availability of information for all software products. Moreover, for our dependent variables, we chose vulnerability severity and vulnerability frequency. Further information about the dependent and independent variables is discussed in the next section.

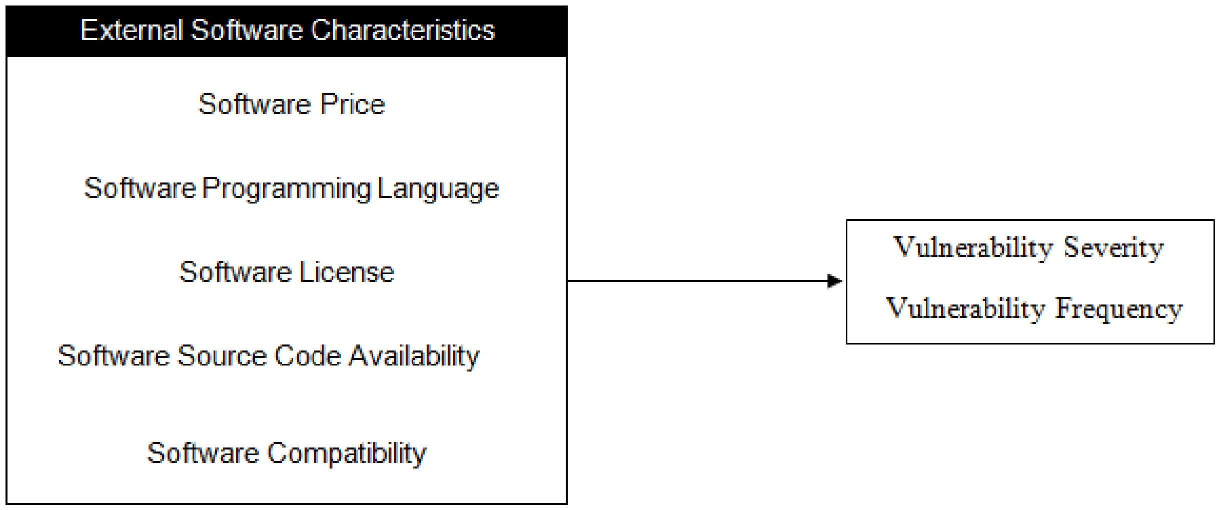

Figure 1: Proposed model

Vulnerability Severity

Vulnerability severity refers to the extent to which an intruder gains unauthorized privilege on various system resources. Oftentimes, vulnerabilities are classified according to their severity levels where each classification is assigned a numerical value. For example, gaining control of the vulnerable host and compromising the entire system is considered critical-level severity vulnerability while being able to collect information about the vulnerable host is considered low-level severity vulnerability.

There have been various tools and measures that were developed to evaluate and measure severity of vulnerabilities. The most accepted measure is the Common Vulnerability Scoring System (CVSS). CVSS is "a universal language to convey vulnerability severity and help determine urgency and priority of response" (Schiffman 2007). It's a universal language and a standard for assessing the severity of vulnerabilities in operating systems and software products (Mell et al. 2007). CVSS offers a way to consistently report on infrastructure vulnerabilities and it solves the issue of multiple, incompatible scoring systems in use today.

CVSS consists of three sets of measures: base measures, temporal measures and environmental measures. Base measures represent the characteristics of a vulnerability which are constant over time and over user environments. Temporal measures, on the other hand, are the characteristics of a vulnerability which change over time but not among user environments. Environmental measures are the characteristics of a vulnerability which are relevant and unique to a particular user's environment. Each set of measures has a different equation that results in a score which in turn is the essence of CVSS metric.

Vulnerability Frequency

The frequency of vulnerability refers to the rate at which vulnerabilities occurs over a unit of time. According to reports by Computer Emergency Readiness Team (CERT), there has been an increase in frequency of vulnerabilities coupled with an increase in speed, automation and sophistication of attacks which makes it more difficult for software vendors and system administrators to keep up to date with security patches and they experience more disruptive security attacks. While frequency of attacks and vulnerability disclosure have received wide interest among researchers, frequency of vulnerabilities remains a lesser explored topic. In this study, we will use vulnerability frequency to measure the number of vulnerability occurrences within a given time period without looking at a specific type or distribution of a vulnerability.

External Software Characteristics

Software Price:

Software price plays an important role in modifying the individual's attitude toward the software in many ways. For instance, research studies which looked at software piracy found that software price to be a significant factor (incentive) which influences the intention to pirate (Gopal and Sanders 2000). In their work, Peace et al. (2003) conducted a survey of 201 respondents and found that software price was among the major reasons for illegally copying software.

Following the same analogy, studies have shown that attackers' attitudes and hackers' motivations for finding vulnerabilities are associated with several factors such as: peer approval, self-esteem, politics, publicity, financial gains, curiosity and sabotage (Shaw et al. 1999). Within the hackers' community, hacking achievements typically help individuals gain higher and more respectable status as it refers to the persons' skills and mastery level. Reaching a higher status is oftentimes associated with noteworthy achievements such as hacking popular software. For those hackers who seek publicity or peer approval, they tend to target software with large user base due to their significant reach. So despite software price, hackers look for vulnerabilities in open source and proprietary software as long as there is a significant user base. Similarly, infamous social networking sites such as Facebook and Myspace are constant targets of vulnerabilities regardless of their service cost. Outside the hacker's community, hackers' incentives tend to vary among political reasons, financial gains, self-esteem and sabotage.

Software Programming Language:

Selecting a suitable programming language is one of the most important decisions which is made during software development cycle. A chosen programming language has direct effect on how software ought to be built and what means must be used to guarantee that the software functions properly and securely. Software programs which are written using an insecure language may cause system dependent errors which are known to be difficult to find and fix (Hoare 1973). For instace, buffer overflows vulnerabilities and other low-level errors are well known issues which caused major problems in C and C++ languages (Cowan 1999).

As of today, there exist numerous programming languages but the topic of security in programming languages has been widely disregarded as it's believed that programming errors and flaws are caused by numerous factors and cannot be easily eliminated. Current approaches to this issue are essentially ad hoc where best programming practices and secure programming techniques are implemented during or after the design stage. Although this approach helps in preventing coding errors and flaws by relying on programmers' skills and experience, it is hard to say with any certainty what vulnerabilities are prevented and to what extent. More importantly, the ad hoc approach does not protect against new and evolving vulnerabilities as it only handles known vulnerabilities and specific coding flaws.

An evolving trend in secure programming has been the use of formal language semantics. Formal language semantics try to reason with and prove security properties of the code. For example, in their paper, Leroy and Rouaix (1998) developed a formal technique to validate a typed functional language to ensure that memory locations always contain appropriate values to avoid buffer overflow vulnerabilities. Although the use of formal language semantics has been advocated (Meseguer and Talcott 1997, Volpano 1997), it has not been adopted as a standard by programmers.

Software Source Code Availability:

Security of open source software (OSS) and closed source software has been a hot topic with many arguments repeatedly presented. Advocates of OSS argue that more reviewers strengthen the security of the software as it eases the process of finding bugs and speeds it up "given enough eyeballs, all bugs are shallow" (Raymond and Young 2000). Opponents of this idea disagree and claim that not all code reviewers and testers have enough skills and experience compared to code reviewers at companies who are more skilled at finding flaws.

The argument is that oftentimes code reviewers and testers need to have further skills other than programming such as cryptography, stenography and networking. Moreover, proponents of closed source software claim that security by obscurity is the main strength of closed source software since it’s harder to find vulnerabilities when the code is not accessible. However, proponents of OSS argue that it’s possible to gain access to closed source code through publicly available patches and disassembling software (Tevis 2005).

It’s important to note that the impact of source code availability on security depends on the open source development model. For instance, the open source cathedral model allows everyone to view the source code, detect flaws/bugs/vulnerabilities and open reports; but they are not permitted to release patches unless they are approved by project owners. OSS projects are typically regulated by project administrators who require some time to review and approve patches. Attackers can take advantage of the availability of source code and published vulnerability reports to exploit them (Payne 2002). However, proponents of OSS argue that vulnerabilities in OSS projects can be fixed faster than those in closed source software because the OSS community is not dependent on a company’s schedule to release a patch. Moreover opponents of this idea argue that the lack of incentives in OSS projects can diminish contributors’ motivation to fix vulnerabilities.

Despite the continuous debate on OSS security, advocates from both sides agree that having access to the code makes it easier to locate vulnerabilities but they differ about the impact of vulnerabilities on software security. First of all, keeping the source code open provides attackers with easy access to information which might be helpful to successfully exploit the code. Publically available source code gives attackers the ability to search for vulnerabilities and flaws and thus increase the exposure of the system. Second, making the source code publicly available does not guarantee that a qualified person will look at the source and evaluate it. In the bazaar style environment, malicious code such as backdoors may be sneaked into the source by attackers posing as trustful contributors. For instance, in 2003 Linux kernel developers discovered an attempt to include a backdoor in the kernel code (Poulsen 2003). Finally, for many OSS projects there is no selection criterion of programmers based on their skills; project owners tend to accept any help without checking for qualifications or coding skills.

Software Compatibility:

Software producers often create applications to run on a single or a combination of operating systems (OS). From a software viewpoint, maintaining security is the obligation of both the OS and the software program. But since computer hardware such as the CPU, memory and input/output channels are accessible to a software programs only by making calls to the OS, therefore, the OS bears a tremendous burden in achieving system security by allocating, controlling and supervising all system resources.

For the most part, each of today’s streamlines OSs has a main weakness. For instance, earlier OSs such as Windows NT, UNIX and Macintosh had a weakness in their access control policies (Krsul 1998). Such OSs did not specify access control policies very clearly which meant that applications that ran by users inherited all the privileges that the access control mechanisms of the OS provided to those users (Wurster 2010). An access control policy requires an OS to give a program or a user the minimum set of access rights necessary to perform a task. In his work, Denning (1983) illustrated the working of an access control policy which typically consists of three entities namely, subjects, objects and access rights matrix. Subjects refer to users or domains whereas objects are files, services, or other resources and access rights matrix specifies different kinds of privileges including read, write and execute which are assigned to subjects over objects. A configuration of the access matrix describes what subjects are authorized to do. Vulnerabilities in OSs tend to rely on weaknesses in configuration of access control matrices to gain access to software applications and system software. This creates a serious problem since vulnerabilities can exploit software applications through the OS, gain access and ultimately take over the system.

Moreover, even with an access control policy in place, consideration must be given to system design. The OSs which are in use today have different architectures and are designed with different kernels without considering security and controlled accessibility as significant design criteria. For instance, a large portion of UNIX and Linux vulnerabilities result from boundary condition errors which are commonly known as buffer overflow (Lee and Davis 2003). These boundary conditions result from a failure to check the bound size of arrays, buffers and strings. Attackers tend to exploit this weakness in UNIX and Linux systems to gain access to system software and software applications. Vulnerabilities in Windows OS on the other hand tend to be evenly divided among exceptional conditions, boundary conditions and access control validations (Lee and Davis 2003). With these types of vulnerabilities root break-ins and execution of arbitrary code are common types of attacks.

METHODOLOGY

Data Collection

In order to obtain data for the dependent variables, we used vulnerability data available in the National Vulnerability Database (NVD) between 2008 and 2014. NVD is considered one of the most reliable and comprehensive vulnerability databases on the web. NVD contains publicly known security vulnerabilities of software and hardware products aggregated from security firms, organizations, forums and advisory groups. Despite the large amount of information in NVD, for the purpose of this study, we collected the following information: the unique name and ID of the vulnerable software, the publication day of the vulnerability and the vulnerability severity score (CVSS). CVSS score was used for vulnerability severity (severity scores ranged from 0 to 10, where 0 refers to least severe vulnerability and 10 is most severe vulnerability). The publication day of the vulnerability was used to calculate vulnerability frequency. To do so, we computed the rate at which vulnerabilities occurred starting from the date of first published vulnerability until the date of the last published one. Vulnerability frequency was determined based on a monthly basis. Given the wide variety and large number of vulnerable software products on NVD, we used SPSS to choose a random sample of 227 software products.

The next step involved data gathering for the independent variables for each vulnerable software product. For each software product, we referred to software vendor’s website and publically available information on the web to obtain data regarding the following software characteristics: software price, software programming language, software source code availability, and software compatibility.

Multiple Linear Regression

Multiple linear regression analysis is a mathematical maximization method which predicts the dependent variable based on the association between a dependent variable and two or more independent variables (Ryan 2009). In multiple linear regression, each variable has its own regression coefficient that gives its relative importance in the relationship outcome. According to

Brace et al. (2002), multiple linear regression is appropriate for exploring linear relationships between dependent and independent variables. Also, the dependent variable being predicted should be measured on a continuous scale and the independent variables should be measured on a ratio, interval, ordinal, or nominal scale (expressed using dummy variables). Multiple linear regression analysis involves the creation of a mathematical equation which describes the relationship between the dependent and independent variables (Keller 1997). The prediction of dependent variable (Y) is accomplished by the following equation (Levine et al. 1995):

$$Y = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_i X_i + \varepsilon$$

Where:
Y = the predicted value of Y for observation
X_i = the matrix of covariates
β_0 = Y intercept when all Xi are set to be zero
β_i = the regression coefficient for variable X_i
ε = error term

In order to examine the relationship between vulnerabilities and software characteristics, two regression models were built: a vulnerability severity model and a vulnerability frequency model. Each model included the four independent variables. We also used the proper coding for each independent variable as shown in Table 1 and Table 2. Also, prior to running the analysis, all the required assumptions were tested for each model.

Table 1: Dependent variables

Variable Name	Variable Type	Variable Descriptions & Coding
Vulnerability Severity	Ratio	The average severity level of all vulnerabilities related to a software product.
Vulnerability Frequency	Ratio	The rate at which vulnerabilities occurred over a month starting from the date of first published vulnerability until the date of the last published one.

Table 2: Independent variables

Variable	Variable Type	Variable Description and Coding
Software Price	Interval	This variable considers the price of a single software license
Software Programming Language	Nominal	Indicates the type of programming language used to develop the software. Software programming languages were divided into two groups according to the programming paradigm used. (static language=0, dynamic language=1)
Software Source Code Availability	Nominal	This variable indicates whether software source code is open, closed or mixed. Dummy variables were used in the model.
Software Compatibility	Ratio	Indicates the total number of operating systems which are compatible with the software product.

The multiple linear regression models are shown below:

$$VUL\_SEV = \beta_0 + \beta_1 PRICE + \beta_2 LANGUAGE + \beta_3 SOURCE\_DUMMY1 + \beta_4 SOURCE\_DUMMY2 + \beta_5 COMP + \varepsilon$$

$$VUL\_FRQ = \beta_0 + \beta_1 PRICE + \beta_2 LANGUAGE + \beta_3 SOURCE\_DUMMY1 + \beta_4 SOURCE\_DUMMY2 + \beta_5 COMP + \varepsilon$$

Where: VUL_SEV = Vulnerability severity
VUL_FRQ = Vulnerability frequency
PRICE = Software Price
LANGUAGE = Software programming language
SOURCE_DUMMY1, SOURCE_DUMMY2 = Software source code availability
COMP = Software compatibility
ε = random error term

RESULTS

Multiple Linear Regression Model: Vulnerability Severity

Table 3 shows that 29.1% of the variance in vulnerability severity is explained by the model. The overall F-test for the model indicates that multiple linear regression is statistically significant at $p < 0.05$.

Table 3: Model summary for vulnerability severity

R Square	Std. Error of the Estimate
.291	1.188

Table 4 evaluated each of the predictor variables individually with respect to the dependent variable. In assessing the effect of the other predictor variables on vulnerability severity, the results showed that software compatibility (number of compatible operating systems) was the only predictor variable that had significant effect on vulnerability severity at a 5% significance level. It's important to note that including and excluding outliers made no difference to the results.

Table 4: Coefficients for the dependent variable: vulnerability severity

	Unstandardized Coefficients		Standardized Coefficients		
	B	Std. Error	Beta	t	Sig.
(Constant)	7.326	.461		13.346	.000*
Software Price	-4.983e-4	.002	-.045	-.035	.284
Software Programming Language	-.245	.567	-.064	-.935	.719
Software Source Code Availability1	.098	.473	.031	.456	.833
Software Source Code Availability2	-.674	.386	-.042	-.834	.453

	Unstandardized Coefficients		Standardized Coefficients	t	Sig.
Software Compatibility	.245	.043	.672	2.092	.004*

* p < 0.05

The final multiple linear regression equation for vulnerability severity model is:

$$VUL\_SEV = 7.326 + 0.245 (COMP)$$

The equation can be described as follows. As the number of compatible operating systems increases by 1, the severity level of a vulnerability increases by 0.245

Multiple Linear Regression Model: Vulnerability Frequency

The model, in Table 5, explained 28.1% of the variance in the dependent variable (vulnerability frequency). The overall F-test for the model indicates that multiple linear regression is statistically significant at $p < 0.05$.

Table 5: Model summary for vulnerability frequency

R Square	Std. Error of the Estimate
.281	1.303

Table 6 evaluated each of the predictor variables individually with respect to the dependent variable. In assessing the effect of the other predictor variables on vulnerability frequency, results showed that software compatibility (number of compatible operating systems) was the only predictor variables that had significant effect on vulnerability frequency at a 5% significant level.

Table 5: Coefficients for the dependent variable: vulnerability frequency

	Unstandardized Coefficients		Standardized Coefficients		
	B	Std. Error	Beta	t	Sig.
(Constant)	.452	.085		6.083	.000*
Software Price	-1.150e-4	.000	-.062	-.752	.842
Software Programming Language	.060	.027	.997	1.593	.393
Public Source Code Availability1	-.055	.049	-.038	-.411	.971
Public Source Code Availability2	-.041	.081	-.098	-.293	.830
Software Compatibility	.072	.043	.232	2.759	.002*

* p < 0.05

Multiple linear regression for the vulnerability frequency model produced the following equation:

$$VUL\_FRQ = 0.452 + 0.072(COMP)$$

The equation can be described as follows. As the number of compatible operating systems increases by 1, the frequency of a vulnerability increases by 0.072 per month.

DISCUSSION AND IMPLICATIONS

This paper extends previous research by examining multiple aspects of vulnerability risks including: severity level, and frequency of occurrence. Two novel vulnerability prediction models were proposed and explored for their applicability of using software characteristics to predict vulnerability risks. Both models were found to have significant associations with software characteristics. The vulnerability severity model was fitted to vulnerability data obtained from NVD and the fit was found to be statistically significant. This model observed software compatibility to have significant positive association with vulnerability severity. This result confirms with Krsul (1998) and Wurster (2010) findings that software inherit vulnerabilities from compatible operating systems. The second proposed model was based on vulnerability frequency. The model was fitted to vulnerability frequency data and the fit was found to be statistically significant. This model also observed software compatibility to have a significant positive association with vulnerability frequency.

Our study offers new insights into the area of vulnerability prediction and software security. In this research, the focus was on the external aspects of software products rather than the internal ones. Predicting vulnerabilities based on this approach has not been paid enough attention especially in the software security field. This paper tackles this limitation in research and explores the importance of external software characteristics and their effects on vulnerabilities. A research implication is the discovery of the significant role of software characteristics in predicting vulnerability severity and frequency. More specifically, the results demonstrated that software compatibility (number of compatible operating systems) has significant effect on vulnerability severity and frequency. The other research implication is the creation of two vulnerability prediction models. Such models answered the call of Alhazmi and Malaiya (2006) for the need of more quantitative vulnerability predictive models. Finally, an obvious research implication of the obtained result is the necessity of extending vulnerability risks and software characteristics to encompass other variables in order to improve predictability.

LIMITATIONS

The results presented in this study must be considered in view of certain limitations. First, the data used in the analysis comes from the National Vulnerability Database (NVD). NVD is known as the most reliable vulnerability database but it still suffers from inaccuracies and missing information. In future research, it would be more appropriate to combine NVD with other vulnerability databases such as Open Source Vulnerability database (OSVD) to overcome this limitation. Second, it's important to note that this research analyzed publically reported vulnerabilities without considering unreported/undisclosed data. While this drawback is common among vulnerability research, overcoming this limitation can be achieved through direct contact with software vendors which can be an extension for future research. Third, given the limited information available on software products, the proposed models only considered four external software characteristics. Future research may benefit from considering more software characteristics.

CONCLUSION

This study examined the effects of external software characteristics on vulnerability severity and vulnerability frequency. Our findings offer new insights into the area of vulnerability and software security. The results showed that the number of compatible operating systems has a positive effect on the severity level and frequency of vulnerabilities. Also, this study showed that there is no relationship between each of software price, software programming language, software source availability and vulnerability severity. Similarly, there was no relationship between these characteristics and software frequency. Future research should examine other software characteristics in order to better understand their effect on vulnerabilities.

REFERENCES

Alhazmi, O. H., & Malaiya, Y. K. (2005a). Quantitative vulnerability assessment of systems software. In Proc. annual reliability and maintainability symposium (pp. 615-620).

Alhazmi, O. H., & Malaiya, Y. K. (2005b). Modeling the vulnerability discovery process. In Software Reliability Engineering, 2005. ISSRE 2005. 16th IEEE International Symposium on (pp. 10-pp). IEEE.

Alhazmi, O. H., & Malaiya, Y. K. (2006). Measuring and enhancing prediction capabilities of vulnerability discovery models for Apache and IIS HTTP servers. In Software Reliability Engineering, 2006. ISSRE'06. 17th International Symposium on (pp. 343-352). IEEE.

Arbaugh, W.A., Fithen, W. L. and McHugh, J. (2000), Windows of Vulnerability: A Case Study Analysis, IEEE Computer.

Brown, W. A., and Booch, G. (2002). Reusing Open-Source Software and Practices: The Impact of Open-Source on Commercial Vendors, in C. Gacek (ed.), Software Reuse: Methods, Techniques, and Tools, New York: Springer-Verlag, 123-13.

Browne, H., McHugh, J., Arbaugh, W., and Fithen, W. (2001). A Trend Analysis of Exploitations. IEEE Symposium on Security and Privacy.

Cowan, C., Wagle, P., and Pu, C. (1999). Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade, DARPA Information Survivability Conference and Expo.

De Ru, W.G., Eloff, J.H.P. (1996), "Risk analysis modelling with the use of fuzzy logic", Computers and Security, Vol. 15 No.3, pp.239-48

Dillman, D. A. (2000). Mail and Internet Surveys: The Tailored Design Method, Wiley, New York.

Dr Dobbs Journal (2009). Open Source Study Reveals High Level of Code Reuse. Retrieved on Jan 25th 2011 at http://www.drdobbs.com/open-source/216401796

Gopal, R., and Sanders, G. (2000). Global software piracy: You can't get blood out of a turnip. Communications of the ACM, 43(9), 83–89.

Hoare, C. A. R. (1974). Hints on Programming Language Design. Computer Systems Reliability: State of the Art Report, Vol. 20, pp. 505-34.

Kirk, J. (2013). Deep cyberattacks cause millions in losses for U.S. banks. Retrieved from http://www.computerworld.com/article/2484007/malware-vulnerabilities/deep-cyberattacks-cause-millions-in-losses-for-u-s--banks.html

Krsul I. V. (1998). Software Vulnerability Analysis. PhD thesis, Purdue University.

Lee, S. C., and Davis, L. B. (2003). Learning from experience: Operating system vulnerability trends. IT Professional, 5(1).

Littlewood, B., Brocklehurst, S., Fenton, N., Mellor, P., Page, S., Wright, D., & Gollmann, D. (1993). Towards operational measures of computer security. Journal of computer security, 2(3), 211-229.

Mell P., Scarfone K., and Romanosky S., "A Complete Guide to the Common Vulnerability Scoring System Version 2.0," Forum of Incident Response and Security Teams, June 2007, http://www.first.org/cvss/cvssEguide.html.

Mercuri, R. T. (2003). Analyzing security costs. Comm. ACM 46(6) 15–18

Murtaza, S. S., Khreich, W., Hamou-Lhadj, A., & Bener, A. (2016). Mining Trends and Patterns of Software Vulnerabilities. Journal of Systems and Software.

Ozment, A. "Improving Vulnerability Discovery Models," Proc. 2007 ACM Workshop on Quality of Protection, ACM Press, 2007, pp. 6-11.

Parrend, P., Frenot, S. (2008). Classification of Component Vulnerabilities in Java Service Oriented Programming (SOP) Platforms. In: CBSE 2008. LNCS, p80-96, Springer Berlin/Heidelberg.

Payne, C. (2002). On the security of open source software. Information Systems Journal, 12(1), 61-78.

Peace, A. G., Galletta, D. F., & Thong, J. Y. L. (2003). Software piracy in the workplace: A model and empirical test. Journal of Management Information Systems, 20, 153E177.

Pham, N. H., Nguyen, T. T., Nguyen, H. A., Wang, X., Nguyen, A. T., and Nguyen, T. N. (2010). Detecting Recurring and Similar Software Vulnerabilities. Proceedings of the 32nd International Conference on Software Engineering (ICSE 2010).

Poulsen, K. (2003). Thwarted Linux backdoor hints at smarter hacks. Retrieved from at: http://www.securityfocus.com/news/7388

Raymond, E.S., and B. Young. (2001). The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O'Reilly, Sebastopol, CA.

Ryan T. (2009). Modern Regression Methods. New York, NY: Wiley Interscience.

Scandariato, R., Walden, J., Hovsepyan, A., & Joosen, W. (2014). Predicting vulnerable software components via text mining. Software Engineering, IEEE Transactions on, 40(10), 993-1006.

Schiffman, M. (2007), A Complete Guide to the Common Vulnerability Scoring System (CVSS). http://www.first.org/cvss/cvss-guide.html

Schmidt, A.D., Schmidt, H.G., Batyuk, L., Clausen, J.H., Camtepe, S.A., Al-bayrak, S. (2009). Smartphone Malware Evolution Revisited: Android Next Target? In: Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE).

Shaw, D. S.; Post, J. M. and Ruby, K. G. (1999). Inside the minds of the insider, Security Management, 43(12) 34E44.

Shin, Y., & Williams, L. (2013). Can traditional fault prediction models be used for vulnerability prediction?. Empirical Software Engineering, 18(1), 25-59.

Tevis, J. (2005). Automatic Detection of Software Security Vulnerabilities in Executable Program Files. Dissertation submitted to the Department of Computer Science. Auburn University. Auburn, AL.

Wurster, G. (2010). Security Mechanisms and Policy for Mandatory Access Control in Computer Systems. PhD thesis, Carleton University.