Vulnerability Scrying Method for Software Vulnerability Discovery Prediction Without a Vulnerability Database

Sanaz Rahimi and Mehdi Zargham

Abstract—**Predicting software vulnerability discovery trends can help improve secure deployment of software applications and facilitate backup provisioning, disaster recovery, diversity planning, and maintenance scheduling. Vulnerability discovery models (VDMs) have been studied in the literature as a means to capture the underlying stochastic process. Based on the VDMs, a few vulnerability prediction schemes have been proposed. Unfortunately, all these schemes suffer from the same weaknesses: they require a large amount of historical vulnerability data from a database (hence they are not applicable to a newly released software application), their precision depends on the amount of training data, and they have significant amount of error in their estimates. In this work, we propose vulnerability scrying, a new paradigm for vulnerability discovery prediction based on code properties. Using compiler-based static analysis of a codebase, we extract code properties such as code complexity (cyclomatic complexity), and more importantly code quality (compliance with secure coding rules), from the source code of a software application. Then we propose a stochastic model which uses code properties as its parameters to predict vulnerability discovery. We have studied the impact of code properties on the vulnerability discovery trends by performing static analysis on the source code of four real-world software applications. We have used our scheme to predict vulnerability discovery in three other software applications. The results show that even though we use no historical data in our prediction, vulnerability scrying can predict vulnerability discovery with better precision and less divergence over time.**

Index Terms—**Code security, static analysis, vulnerability discovery model, vulnerability prediction.**

Notation

t	Time expressed in days
$v(t)$	The number of vulnerabilities discovered during day t
$V(t)$	The total number of vulnerabilities discovered from day 0 to day t
$\lfloor M \rfloor$	The largest integer not greater than M
$\{\xi_n\}$	A sequence of random variables
$Pr\{x\}$	The probability of x
$E[x]$	The expected value of x

Acronyms

GDP	Gross Domestic Product
VD	Vulnerability Discovery
VDM	Vulnerability Discovery Model
NVD	National Vulnerability Database
ROI	Return on Investment
AT	Anderson Thermodynamic
LP	Logarithmic Poisson
AML	Alhazmi-Malaiya Logistic
CVE	Common Vulnerability Enumeration
PGP	Pretty Good Privacy
XML	Extensible Markup Language
AST	Abstract Syntax Tree
IR	Intermediate Representation
CQ	Code Quality
CC	Code Complexity

I. Introduction

SOFTWARE vulnerabilities pose a real threat to computing systems ranging from personal computers to mobile devices and critical systems. Previously unknown vulnerabilities discovered by hackers can be used in developing zero-day exploits which can breach even highly secure systems.

According to the National Vulnerability Database [1] in 2009 alone, 6052 vulnerabilities have been discovered. Learning about these vulnerabilities, developing and installing patches, and finding a mitigation strategy require a significant amount of resources and manpower. Note that other factors such as loss of data and service, brand damage, loss of reputation, loss of revenue, and user perception damage make vulnerabilities even more costly. In fact, software vulnerability is a major threat to the U.S. economy. According to the National Institute of Standards and Technology (NIST), the cost of vulnerable software in 2002 was estimated to be $60 Billion [2]. This cost in 2009 was approximately 1% of the Gross Domestic Product (GDP), or about $140 Billion.

Manuscript received August 21, 2011; revised April 24, 2012 and December 07, 2012; accepted January 24, 2013. Date of publication April 16, 2013; date of current version May 29, 2013. Associate Editor: S. Shieh.

The authors are with the Department of Computer Science, Southern Illinois University, Carbondale, IL 62901 USA (e-mail: srahimi@cs.siu.edu; mehdi@cs.siu.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TR.2013.2257052

The rate of software vulnerability discovery is extensively increasing [2]. A significant number of these vulnerabilities, if exploited, can cause damages to educational institutions, corporations, government systems, software vendors, and customers. It may be impossible or prohibitively costly to develop systems free of vulnerabilities. However, having an estimate of the number of vulnerabilities in a software application at each time would assist us to be prepared, design contingency plans, provision backup capabilities, and allocate enough resources and man power to maintain the system's mission. Moreover, security design principles advocate diversity in mission critical systems. Predicting vulnerabilities in software applications can facilitate diversity planning. For example, if the probabilities that a new vulnerability is found in two software applications that do not share any code within the next month are 0.1 and 0.2, a diverse fail-over system that deploys both applications can be damaged with the probability of 0.02 (i.e., only when vulnerabilities are discovered in both applications).

In addition, a traditional problem in cyber security is the lack of metrics, which often makes it difficult to measure or improve the processes related to security (e.g. software engineering or configuration management) [3]. Predicting vulnerabilities in a software system can provide a metric for organizations to decide how to respond, prepare, and plan for cyber incidents. It also enables them to make informed decisions based on facts and measurements.

Vulnerability discovery can be modeled as a stochastic process [4]. Vulnerabilities are discovered at different times during the lifecycle of a software application. Many statistical models have been proposed in the literature to estimate the vulnerability discovery trends (a.k.a. vulnerability discovery models (VDMs)). They have varying degrees of accuracy in modeling. Although the original intent of VDMs is vulnerability modeling, research has shown that they can be used in predicting vulnerability discovery [5].

Unfortunately, we show in Section III that the VDMs studied in this paper have the same weaknesses. First, VDMs are parametric models that have to be fitted to real vulnerability data. Hence, VDMs need a large amount of historical vulnerability data in order to model the vulnerability discovery trend of a software application. The data often come from vulnerability databases such as the National Vulnerability Database (NVD) [1], or the Open Source Vulnerability Database [6]. As a result, to predict vulnerability discovery in a software application using VDMs, many of its vulnerabilities have to be discovered already. This requirement precludes vulnerability prediction for newly released applications. Second, the precision of VDMs depend on the number of known vulnerabilities for a software application. Early in a software's lifecycle, the precision of VDMs is very low [7]. Third, we show that, even with large amounts of training data (e.g. half of the total vulnerabilities), VDMs have significant prediction errors. For example, one of the VDMs (LP model) overestimates the number of vulnerabilities in Lynx by 77% while another one (AT model) underestimates it by 48% (see Section III-A). As another example, all three VDMs underestimate the number of vulnerabilities in Emacs, one (AT) by as much as 66% (see Section III-B).

In this work, we propose a new model for vulnerability prediction that uses a technique that we call vulnerability scrying. We use static analysis of the source code to extract certain properties of the code which we hypothesize to impact vulnerability discovery. The two properties that we analyze are code complexity (cyclomatic complexity), and code quality (compliance with secure coding practices). By gaining insight into the source code properties of a software application, we show in Section V that we can predict its vulnerability discovery trend more accurately, and without relying on a vulnerability database. Therefore, the proposed scheme can be used to predict vulnerabilities in a software application as soon as it is released.

Scrying is defined in the dictionary as "foretelling the future using a crystal ball or other reflective object or surface" [45]. We believe that code properties ignored in the VDMs are reflective windows into vulnerability prediction. Hence we call our scheme vulnerability scrying.

We analyze the impact of certain code properties on vulnerability discovery by analyzing four popular, real-world applications using compiler-based static analysis: Emacs, GNUPG, OpenSSL, and Lynx. We then test our scheme on three other popular applications to predict their vulnerabilities. We calculate the estimation error by comparing our prediction scheme with real vulnerability data for the applications extracted from the NVD. The results show that vulnerability scrying can predict vulnerability trends accurately, yet it does not rely on, nor does it use historical vulnerability data. Moreover, the prediction diverges less over time.

Our work has important managerial implications for software companies and developers. The vulnerability information predicted using our scheme can be used to optimize investment in better software development practices. For instance, given the costs of a vulnerability (reputation cost, patch development cost, patch deployment cost, etc.), companies can invest time and manpower to improve the quality of their codebases, and actually calculate its impact on future vulnerabilities. This work can justify investment in better development practices, and more comprehensive testing; and can be used to calculate the return on investment (ROI).

The rest of the paper is organized as follows. Section II provides a quick overview of the major VDMs proposed in the literature. Section III studies the application of VDMs in vulnerability prediction, and their weaknesses. Section IV describes our static analysis methodology, the code property extraction process, and the prediction scheme. We evaluate our scheme, and present the results in Section V. We provide a list of the most important observations we had during our analysis in Section VI. We review the related work in Section VII before concluding the paper in Section VIII.

II. Vulnerability Discovery Models (VDMs)

Software vulnerabilities are often discovered by chance at different times during a software lifecycle. Rescorla [4] argues that vulnerability discovery can be modeled as a random process which is slow at the beginning of a software's lifecycle because the users are not familiar with it yet. The rate grows rapidly in the middle of the lifecycle. Finally, the rate slows down significantly after a few years because the users often upgrade to newer versions, and the hackers lose interest in the old software.

It is possible to model the vulnerability discovery trends using mathematical models. The goal in this case is to minimize the overall modeling error.

Different statistical vulnerability discovery models have been proposed in the literature. They either try to capture the underlying process or apply principles used in other fields of science (e.g. thermodynamics) to vulnerability discovery. Here we provide a quick overview of the major VDMs.

A. Exponential Model

The exponential model proposed by Rescorla [4] is designed to fit real data. We refer to this model as the RE model. In this model, the number of vulnerabilities discovered at time t $(v(t))$ decays exponentially with time. Therefore, the total number of vulnerabilities discovered until time t $(V(t))$ shown in (1) will exponentially approach a constant, which shows the overall number of vulnerabilities in the system. Also, N and a are application specific constants in this model.

$$V(t) = N \times (1 - e^{-at}) \tag{1}$$

B. Logarithmic Model

The logarithmic model proposed by Poisson (LP model) [8] estimates the total number of vulnerabilities as a logarithmic growth. The model is shown in (2), in which a and b are the constants that are calculated by fitting the model to the vulnerabilities of a specific application.

$$V(t) = a \times ln(1 + b \times t) \tag{2}$$

C. Thermodynamic Model

The thermodynamic model proposed by Anderson (AT model) [9] is also another model for vulnerability discovery. Anderson argues that the time is inversely proportional to the rate of vulnerability discovery at each instant. Equation (3) describes the AT model. The cumulative number of vulnerabilities is denoted by V(t), which has a logarithmic form (4). a and k are the application specific constants in this model.

$$v(t) = \frac{\gamma}{\beta \times t} \tag{3}$$

$$V(t) = k \times ln(a \times t) \tag{4}$$

D. AML Model

The Alhazmi-Malaiya Logistic (AML) model [5] is based on capturing the underlying process of vulnerability discovery. In general, a piece of software becomes a target for vulnerability discovery when it is newly released. The interest is gradually reduced as the newer versions become available, and fewer users use the older version. Based on this assumption, (5) describes the AML model. The rate of vulnerability discovery depends on two factors. One of these factors declines as the number of remaining undetected vulnerabilities declines. The other factor increases with time to take into account the rising share of the application usage. The time domain solution to this model is expressed in (6). A, B, and C are the application specific constants.

$$\frac{dV(t)}{dt} = A \times V \times (B - V) \tag{5}$$

$$V(t) = \frac{B}{B \times C \times e^{-ABt} + 1} \tag{6}$$

III. Vulnerability Discovery Prediction

The original intent of the VDMs is to model vulnerability discovery, but research has demonstrated that VDMs can be used to predict vulnerability discovery [5]. To predict future vulnerability discoveries, a VDM has to be fitted to the vulnerability history of a software application. Vulnerability history refers to the time and the number of vulnerabilities discovered previously in the application. Fitting refers to the process of determining the VDM parameters which minimize the total amount of error (deviation of the model from the vulnerability history).

We first study the vulnerability prediction capabilities of the major VDMs. To fit the models to the vulnerability data, we choose the parameters in the model in such a way that minimizes the sum of squared errors. For instance, if a model has parameters α, β, and γ, they are optimized using (7), in which $v(t)$ is the number of real vulnerabilities discovered at time t, and $\hat{v}(t)$ is the number of vulnerabilities predicted by the model.

$$\alpha, \beta, \gamma \mid min\left(\sum_t (v(t) - \hat{v}(t))^2\right) \tag{7}$$

For our study, we analyze four popular open source applications: Lynx web browser, Emacs text editor, GNUPG PGP application, and OpenSSL security library. To compare the prediction capability of different VDMs, we fit the model to the first half of the vulnerabilities, and use it to predict the second half of the vulnerabilities. This approach means that we divide the vulnerabilities discovered in an application into two bins. Assuming that N total vulnerabilities have been discovered in an application, we put vulnerabilities 1 to $\lfloor N/2 \rfloor$ in the first bin, and vulnerabilities $\lfloor N/2 \rfloor + 1$ to N in the second bin. We use the vulnerabilities in the first bin to train the model, and optimize its parameters. We then use the model to predict the vulnerabilities in the second bin (second half of the vulnerabilities).

The half-half split of the data for training and prediction is chosen as a starting point to compare the prediction capabilities of various VDMs. Later in this section, we will illustrate that if a larger portion of data is used for training, the predictions become more accurate.

We collect the real vulnerability history from the NVD, the U.S. government's repository of vulnerability management data represented using the Security Content Automation Protocol (SCAP). These data enable the automation of vulnerability management, security measurement, and compliance [1]. Each vulnerability in NVD is described in the Common Vulnerabilities and Exposures (CVE) format. Each CVE entry has a unique identifier (a.k.a CVE name or CVE number), and a status of

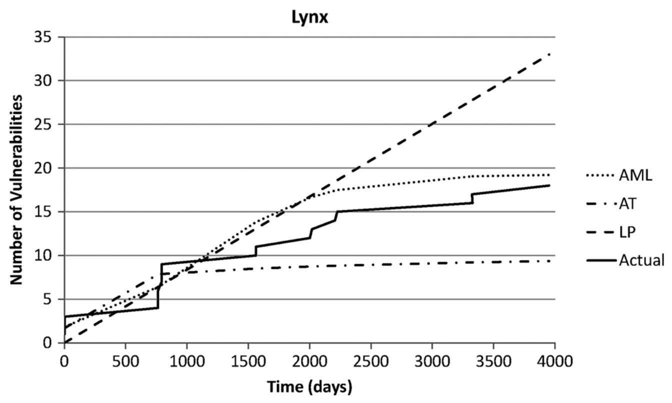

Fig. 1. Lynx vulnerability discovery and prediction.

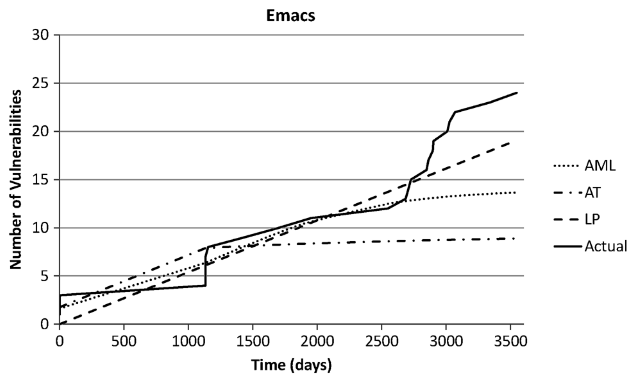

Fig. 2. Emacs vulnerability discovery and prediction.

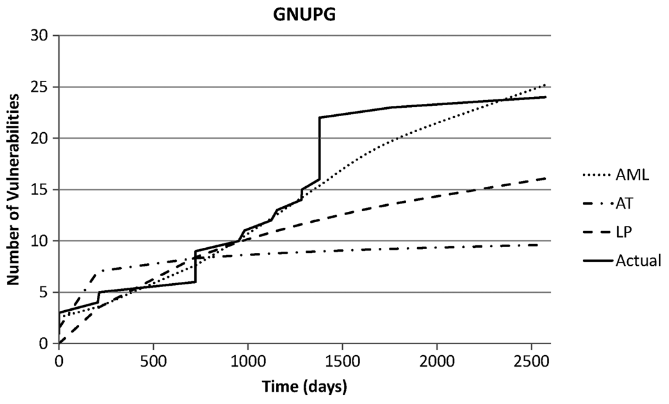

Fig. 3. GNUPG vulnerability discovery and prediction.

either entry or candidate, which denote vulnerabilities that are accepted in the list, and those that are under review for inclusion respectively. Each CVE entry also includes a list of references about the vulnerability.

To automate the data processing, we have parsed the CVEs in the Extensible Markup Language (XML) format, and populated them into a MySQL [10] database. By using a MySQL database, we can easily collect statistics about the vulnerability data by writing arbitrarily complex SQL queries.

A. Lynx

Lynx is a text-based web browser first developed for Linux-like operating systems (although it is now ported to other platforms as well). Fig. 1 illustrates the vulnerability discovery trend in Lynx. Time is expressed in the number of days past the discovery of the first vulnerability. As is evident in the figure, the vulnerability discovery is a stochastic process, and there are often large jumps in the graph due to the fact that many vulnerabilities are often released at the same time.

Moreover, the figure also illustrates the prediction of different VDMs. As mentioned earlier, each VDM is fitted to half of the vulnerabilities (in this case, vulnerability 1 to 9 of the total 18 vulnerabilities). Because of the prediction similarity of the RE and LP models, we only show the LP model in each case.

It can be observed from the figure that the models follow the real data rather closely before the half point. They, however, diverge quickly afterwards. The most accurate model in this case is the AML model. Note that the accuracy of the VDM predictions degrades with time.

B. Emacs

The next software that we analyze is the Emacs text editor. Fig. 2 illustrates the vulnerability discovery and prediction for Emacs. The total number of known vulnerabilities in Emacs at the time of writing this paper is 24 [1]. Just as with the previous software, the models follow the real vulnerability history up to the point that they were fitted to (half point), and then they diverge quickly. In this case, the most accurate prediction over the long term comes from the LP model.

C. GNUPG

We study the GNUPG open source PGP application next. The vulnerability discovery and prediction for GNUPG are illustrated in Fig. 3. The total number of known vulnerabilities in GNUPG is 24 [1]. Again, similar trends can be observed in the data as in the previous models. The AML model provides the most accurate prediction for GNUPG.

D. OpenSSL

The last software we analyze here is the OpenSSL library. Fig. 4 illustrates the vulnerability discovery and prediction. The total number of known vulnerabilities in OpenSSL is 105 [1]. Because of the large number of vulnerabilities discovered in OpenSSL, the divergence between reality and the model is even greater in this case.

We can make a few other observations by looking at the VDM predictions. First, depending on the software, different VDMs have varying degrees of accuracy. For instance, AML provides the most accurate prediction for GNUPG, whereas LP is the best model for Emacs. This variation is troublesome because no one model provides accurate prediction capability over the long term. Second, the VDMs require large amounts of vulnerability history for better prediction. Third, even with large amounts of vulnerability history, the prediction error is significant.

Note also that the prediction error greatly increases with less historical data. To illustrate this fact, we have fitted the AML model to OpenSSL using different amounts of historical data:

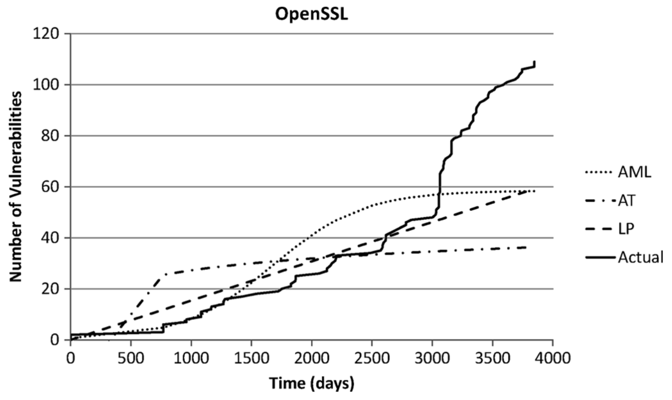

Fig. 4. OpenSSL vulnerability discovery and prediction.

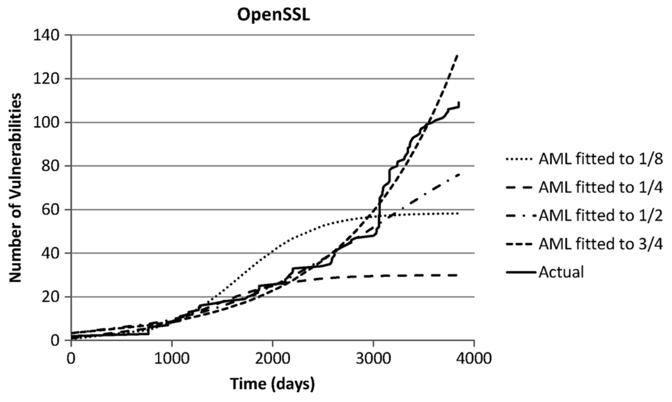

Fig. 5. Vulnerability prediction with different amount of history.

1/8, 1/4, 1/2, and 3/4 of the total vulnerabilities. As Fig. 5 illustrates, the larger portion of the vulnerability history we use to fit the model, the better the model can predict the rest of the trend.

IV. Code Properties and Vulnerability Scrying

VDMs have several shortcomings in predicting vulnerability discoveries as discussed earlier. We propose a new technique which we call *vulnerability scrying* to overcome the shortcomings of VDMs. Vulnerability scrying has three main goals.

i) Vulnerability prediction with relatively small error at the beginning of the software's lifecycle is advantageous. If VDMs are used to predict vulnerabilities, they can only do so accurately if a large amount of historical data are available. This effect can be observed in Fig. 5. If more historical data used in tuning the model, the model will predict the future trends more accurately. The effect can also be observed in Figs. 2, 3, and 4. Because the models are fitted to the first half of the historical data, they match the first half relatively well, but start to diverge as we pass the half point on the charts. However, more historical data requires waiting for new vulnerabilities to be discovered. Of course, by then the interest in vulnerability prediction may diminish because the users often move to the newer version of the software. In fact, we argue that it is more important to predict vulnerabilities as soon as a software application is released.

ii) Incorporating code properties into vulnerability discovery is a better use of available information. To the best of our knowledge, no VDM uses code properties in modeling vulnerability discovery or prediction. We hypothesize that certain code properties have significant impact on vulnerability discovery; so by using them in our prediction scheme, we seek to provide more accurate results.

iii) Vulnerability prediction without vulnerability history is necessary. When using VDMs for vulnerability prediction, one has to rely on vulnerability history. A corollary to design goal (i) is that vulnerability scrying should not rely on vulnerability history of an application. In addition to allowing the prediction as soon as the application is released, avoiding the reliance on vulnerability history has two other advantages: it works for proprietary software applications which may not exist in the vulnerability databases, and it does not rely on the accuracy of the vulnerability databases.

A. Code Properties

To incorporate the code properties into vulnerability prediction, we first have to extract these properties from the source code. The most important property for us is the security posture of the code. Security posture refers to the compliance of the code with the secure coding rules. Violations of these rules can result in security vulnerabilities in the software application. Another code property that can be linked to vulnerability discovery is the code complexity. The code complexity can be measured using various metrics [11]. In this work, we focus on the cyclomatic complexity metric [12].

Compiler-based static analysis can be used to extract the code properties from a source code. We use the Rose compiler infrastructure [13] to analyze the code. Using the Rose compiler, millions of lines can be scanned to find the security posture and complexity of the code.

Rose is a source-to-source compiler infrastructure. Using the intermediate representation (IR), Rose builds abstract syntax trees (AST) to perform a source-to-source conversion without losing any information about the structure of the original code. Thus, Rose can be used for various applications including code optimization, profiling, arbitrary program transformation, domain-specific optimizations, complex loop optimizations, performance analysis, and security evaluation.

Compass [14] is an open source tool designed to evaluate the source code for defects and coding violations using rule sets and scripts. Compass is a part of the Rose framework that uses Rose for source code to IR conversion. It performs code analysis to detect coding violation using different features in Rose such as the AST , control flow graph, system dependence graph, class hierarchy graph, call graph, etc.

Each security or coding violation must be defined as a checker in Compass. Users can either define their own checkers, or use the ones available to detect a particular violation.

Compass can check for 96 different secure coding rules. To quantify the security posture of a code, we measure the ratio of the rules not violated in a particular codebase to the number of total rules. We call this ratio the *code quality* (CQ) (See (8)).

$$CQ = \frac{\text{The number of rules not violated}}{96} \tag{8}$$

Sample secure coding rules include no binary buffer overflows, no side effect in the second term of the expressions, memory allocation must be freed in the same module, virtual methods must be protected, and no dangling pointers. A complete list of secure coding rules can be found in the Appendix. The other columns illustrate whether or not the codebases studied in this paper comply with the rule. For a complete description of each rule, the reader may refer to the Compass manual [14]. To check compliance with the coding rules, different features of a source-to-source compiler must be used. For example, to check for binary overflows, memory allocations are tracked throughout the control flow graph, and accesses beyond the allocated size are marked as overflows. As another example, to check virtual method protection, an AST traversal is performed, and public virtual methods are marked as violations.

Another important code property that can potentially impact vulnerability discovery is the code complexity. Code complexity can be measured using different metrics including deep nesting, Halstead complexity, number of lines of code, etc. [11]. In this work, we focus on cyclomatic complexity, which measures the number of linearly independent paths in the code. Cyclomatic complexity is defined in (9). In this equation, E, N, and P are the number of edges, nodes, and connected components in the control flow graph of the source code.

$$CC = E - N + 2P \tag{9}$$

B. *Vulnerability Scrying*

To use the code properties for vulnerability prediction, we have to understand their impact on vulnerability discovery. The intuition behind our method is two-fold. We hypothesize that codes that comply with secure coding rules have fewer vulnerabilities, and thus a smaller rate of vulnerability discovery. This hypothesis is consistent with the mere existence of secure coding rules (i.e. making codes less vulnerable). Moreover, we hypothesize that it takes longer to understand and find vulnerabilities in complex codebases. As a result, code complexity should be negatively correlated with vulnerability discovery rate. Although cyclomatic complexity is just one measure of code complexity, we show that it can be an effective measure for predicting vulnerability discovery rate. We start with these intuitions and first try to describe them quantitatively by analyzing four applications. We then evaluate these hypotheses on three other applications in the following section to measure the accuracy of our technique.

To quantitatively analyze the impact of code quality and cyclomatic complexity on vulnerability discovery , we analyze the source code of the four applications studied in the previous section: Lynx, Emacs, GNUPG, and OpenSSL. In each case, we compile the source code with the Rose compiler to extract the intermediate representation. We then measure the code quality and complexity by running the Compass checkers.

Table I summarizes the code quality and complexity of the codebases. Note that GNUPG has the highest code quality (least number of coding violations) while Lynx has the highest code complexity.

As mentioned earlier, vulnerability discovery can be modeled as a random process. To analyze the effect of code properties on vulnerabilities, we consider the overall rate of vulnerability discovery in the applications. The rate is the total number of vulnerabilities discovered in the codebase divided by the number of days passed since the first discovery of a vulnerability. For the codebases analyzed, the overall vulnerability discovery (VD) rate is shown in the third column of Table I.

TABLE I
CODE QUALITY, CODE COMPLEXITY, AND VULNERABILITY DISCOVERY RATE OF DIFFERENT CODEBASES

Codebase	**CQ**	**CC**	**VD rate** (vul/day)
Lynx	0.65	177	4.5e-3
Emacs	0.70	50	6.7e-3
GNUPG	0.83	58	9.3e-3
OpenSSL	0.72	44	2.6e-2

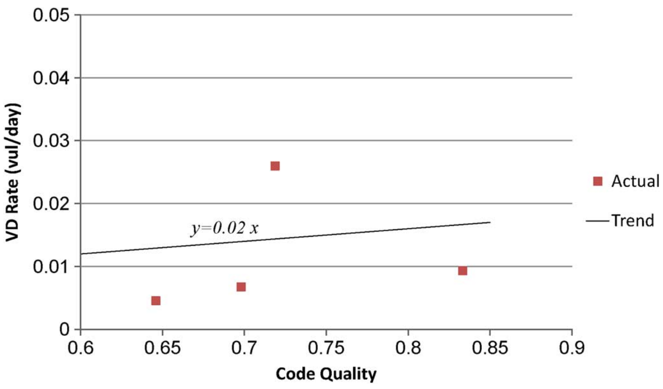

Fig. 6. Code quality impact on the overall vulnerability discovery rate.

Fig. 6 illustrates the VD rate plotted versus the code quality (as defined in (8)) for the four applications. There is no strict trend in the data, but we will show that a linear estimate of the relationship between code quality and VD rate can be used in vulnerability prediction. The figure shows the best fit linear estimate of the relationship ($y = 0.02x$). The effect is intuitive; i.e., codebases that comply with secure programming rules should contain fewer vulnerabilities over time. However, this fact has not been used previously in any vulnerability prediction model.

Fig. 7 illustrates the impact of code complexity on VD rate. Again, the trend here is not strict, but we use a linear estimate of the relationship, and we show that it can be used effectively in vulnerability prediction. The negative trend may seem counterintuitive at first because one of the principles of secure coding is simplicity. However, one hypothesis for this effect is that it takes more effort to understand, analyze, and exploit vulnerabilities in a complex code because of the inherent obscurity in the code. The best fit linear estimate of the effect is given by $y = -8 \times 10^{-5}x + 0.02$.

With vulnerability scrying, we try to build an accurate predictive trend for vulnerability discovery. Because vulnerability discovery is a stochastic process, we use a stochastic model for vulnerability scrying unlike other VDMs. In this model, vulnerability discovery is a s-independent and identically distributed (i.i.d.) sequence of random variables $\{\xi_n\}$ such that

$$Pr\{\xi_n = 2 \times r\} = Pr\{\xi_n = 0\} = \frac{1}{2} \tag{10}$$

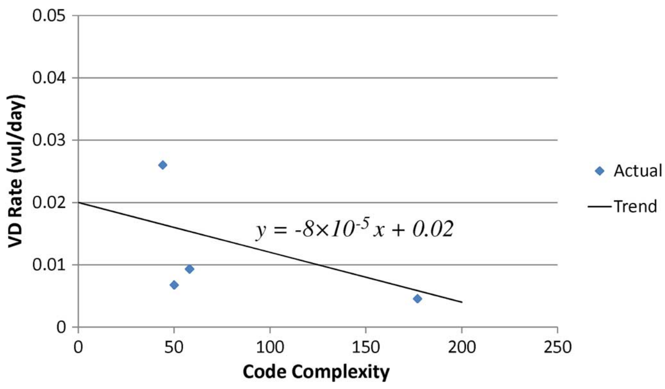

Fig. 7. Code complexity impact on the overall vulnerability discovery rate.

Consider the summation of these random variables.

$$S_0 = 0, \quad S_n = \sum_{i=1}^{n} \xi_i \tag{11}$$

S_n constitutes a random walk. We call r the rate of the random walk. Observe that the expected value of S_N is

$$E[S_N] = N \times r \tag{12}$$

We define vulnerability scrying as

$$\begin{aligned} V_r(t) &= S_t \\ r &= \frac{\big((0.02 \times CQ) - (8 \times 10^{-5} \times CC) + 0.02\big)}{2} \end{aligned} \tag{13}$$

In (13), t is time expressed in days. Vulnerability scrying describes a random walk the rate of which is determined using the code quality, and the code complexity. In (13), $V_r(t)$ is the random walk (i.e. the summation of the sequence of random variables each of which denote a vulnerability being discovered). r is the rate of the random walk (i.e. the rate of vulnerability discovery) which is the mean of the rate predicted by code quality $(0.02 \times CQ)$ and the rate predicted by code complexity $(-8 \times 10^{-5} \times CC + 0.02)$. CQ, and CC are defined in (8), and (9) respectively. The impact of code properties on vulnerability discovery rates ($0.02 \times CQ$, and $-8 \times 10^{-5} \times CC + 0.02$) are determined empirically by studying the four codebases.

In the next section, we evaluate the accuracy of vulnerability scrying on three other codebases, and demonstrate that it can outperform the VDMs in most cases without any reliance on vulnerability history.

One of the limitations of this work is that the coefficients in (13) are determined using a small sample size (four codebases). Two approaches can be used to ensure that the bias introduced due to the sample space is small. First, more codebases can be analyzed. Second, the coefficients can be verified by predicting vulnerabilities in other applications. We take the latter approach in the next section. By illustrating the fact that (13) can be used to predict vulnerabilities in three other applications with relative accuracy, we build more confidence that the bias introduced in our coefficients is limited. We leave the analysis of more codebases to future work.

TABLE II
CODE QUALITY, CODE COMPLEXITY, AND VULNERABILITY SCRYING RATE OF DIFFERENT CODEBASES

Codebase	**CQ**	**CC**	**r** (vul/day)
PostGreSQL	0.83	12	1.78e-2
OpenSSH	0.76	28	1.64e-2
Xpdf	0.66	68	1.39e-2

V. EVALUATION, AND RESULTS

Equation (13) defines our vulnerability prediction technique called vulnerability scrying. We have selected the parameters for predicting vulnerability discovery by analyzing four codebases (Lynx, Emacs, GNUPG, and OpenSSL). Now, to evaluate the accuracy of our vulnerability scrying technique, we analyze three other popular software applications. The three codebases that we analyze for our evaluation are the OpenSSH Secure Shell implementation, the PostGreSQL database management system, and the Xpdf PDF viewer.

Note that we only use the empirical rate determined by studying the initial four codebases (Lynx, Emacs, GNUPG, and OpenSSL). That is, the rate of vulnerability discovery is determined by $r = ((0.02 \times CQ) - (8 \times 10^{-5} \times CC) + 0.02)/2$ for all of the codebases. We do not consider any vulnerability history for predicting the vulnerabilities in our three evaluation codebases (OpenSSH, PostGreSQL, and Xpdf). We determine their code quality CQ and cyclomatic complexity CC using static analysis, and plug the values into (13). In other words, we have used four codebases (Lynx, Emacs, GNUPG, and OpenSSL) to train our model, which we evaluate on three other codebases (OpenSSH, PostGreSQL, and Xpdf).

However, to evaluate the accuracy of our prediction, we cross check the results using the NVD. The vulnerability history in this case is only used to measure the accuracy of vulnerability scrying, and it does not have any role in our prediction scheme. We also compare our results against three VDMs from the literature: AML, AT, and LP.

We compile the codebases with the Rose compiler, and extract the code properties using the Compass tool.

To pass the entire source code through the analyzer, we have replaced the default compiler in the main `makefile` of the codebase (typically `cc` or `gcc`) with the Compass executable. We have then configured all the secure coding rules that we need to analyze (96 different rules).

Table II summarizes the code properties of the three software applications. The vulnerability scrying rate (r) is the quantity described in (13).

Fig. 8 compares the results of vulnerability scrying with predictions using VDMs for OpenSSH . In each case, the VDM is fitted to half of the vulnerability data. The black line labeled prediction shows the vulnerability scrying result. As can be observed, vulnerability scrying provides the most accurate prediction in this case. More importantly, it does so without the need for any historical vulnerability data; i.e., the method can be deployed as soon as the software is released.

Next we analyze the Xpdf application. Fig. 9 shows the results for Xpdf. Here, vulnerability scrying provides the second best prediction (after LP). However, as seen earlier, the VDMs

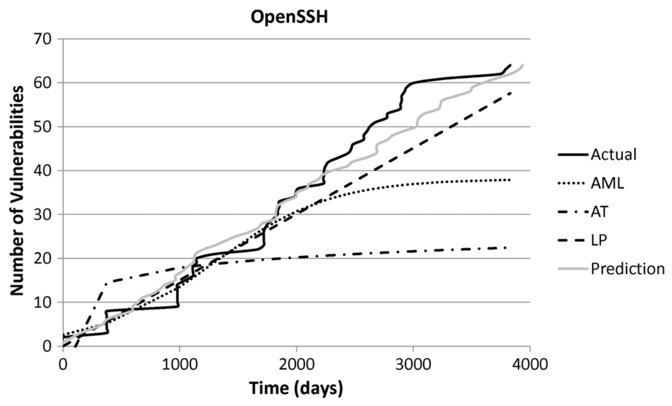

Fig. 8. OpenSSH vulnerability scrying and prediction.

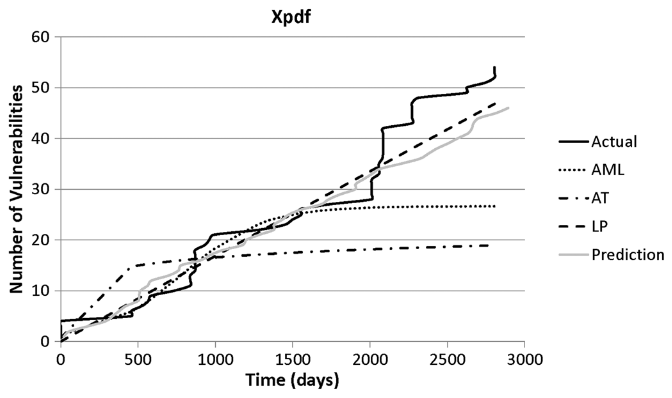

Fig. 9. Xpdf vulnerability scrying and prediction.

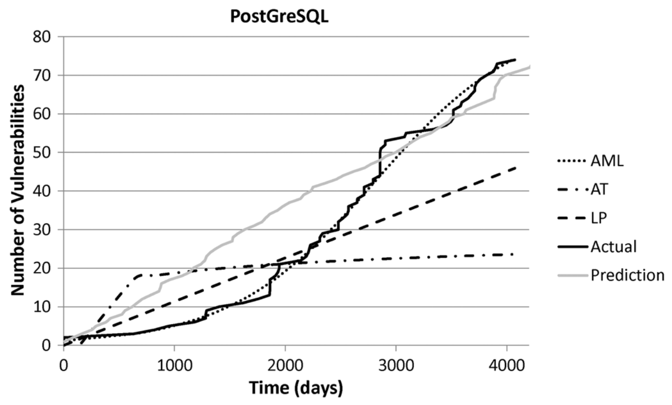

Fig. 10. PostGreSQL vulnerability scrying and prediction.

have wildly different accuracies for different applications, so it is not possible to predict which model may be suitable for a codebase. Also, the error of vulnerability scrying in this case is still relatively small.

Finally, we study PostGreSQL. As illustrated in Fig. 10, vulnerability scrying again can predict the trend with relative accuracy. Although the AML is the most accurate model in this case, it can only achieve that accuracy after 7.4 years (2711 days) when half of the vulnerabilities are discovered.

The evaluation results validate the following hypotheses.

TABLE III
AVERAGE SQUARED ERRORS OF DIFFERENT VULNERABILITY PREDICTION TECHNIQUES

Codebase	AML	AT	LP	Scrying
PostGreSQL	6	643	211	63
Xpdf	136	298	16	27
OpenSSH	119	436	45	20
OpenSSL	479	1306	574	213
Emacs	20	55	7	95
GNUPG	203	52	23	13
Lynx	6	17	33	17

i) Certain code properties studied in this paper (code quality and cyclomatic complexity) provide valuable insights into vulnerability discovery trends of the software application.
ii) Vulnerability scrying can predict vulnerability discovery trends with better accuracies than some of the VDMs fitted to half of the vulnerability history data.

Vulnerability prediction using vulnerability scrying can be performed as soon as an application is released, without the need for large amounts of vulnerability history data (or a vulnerability database).

Note that the code quality and code complexity effects on the scrying rate (13) were obtained without any knowledge of the evaluation codebases (OpenSSH, Xpdf, and PostGreSQL). We can also use the same vulnerability scrying technique to predict the vulnerability trends for the four applications that we originally studied (OpenSSL, Emacs, GNUPG, and Lynx). Because we have extracted the code property effects from these applications, we do not report the results to validate the hypotheses. They are reported here for the sake of completeness. Table III compares the accuracy of the VDMs and vulnerability scrying for all seven codebases. The table shows the average squared error for each VDM, as well as scrying. To calculate the error, we first compute the square of the difference between the number of vulnerabilities predicted by the VDM and the actual number of vulnerabilities for each day. Then we divide this value by the total number of vulnerabilities to get the average value. The average squared error is calculated using the following equation, in which N is the total number of days passed since the first vulnerability.

$$Error = \frac{\sum_{t=1}^{N} (v(t) - \hat{v}(t))^2}{N} \quad (14)$$

Fig. 11 illustrates the results for the four original codebases. As can be seen in Table III, vulnerability scrying for Emacs has a larger error compared to other VDMs. This result can also be observed in Fig. 11. In this case, the rate predicted by scrying is higher than the actual rate of vulnerability discovery; hence, the prediction scheme overestimates the number of vulnerabilities. For other cases, however, vulnerability scrying is more accurate than most of the VDMs.

A. Combined Effect

Vulnerability scrying does not rely on vulnerability history data. It can be performed as soon as a software application is released. However, it is important to consider what happens when

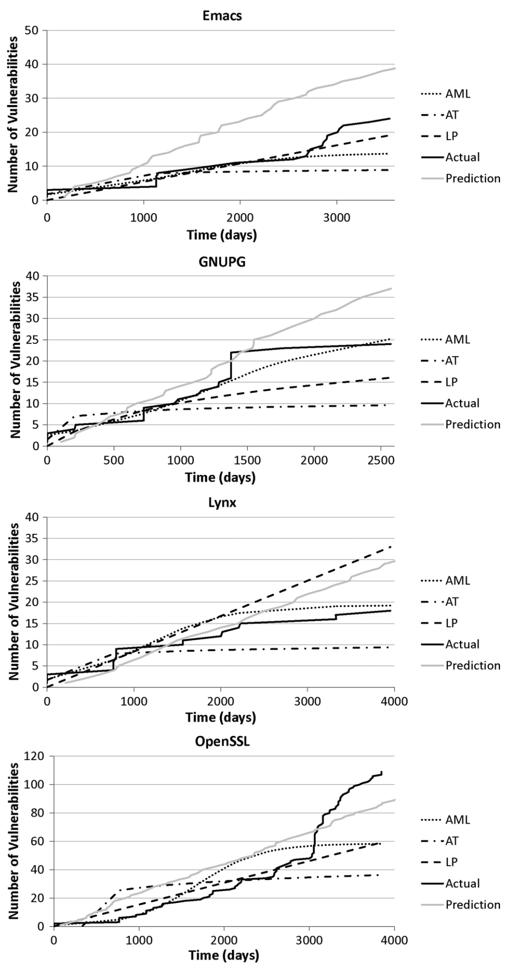

Fig. 11. Vulnerability scrying for the four codebases from which the parameters were extracted in Section IV-B.

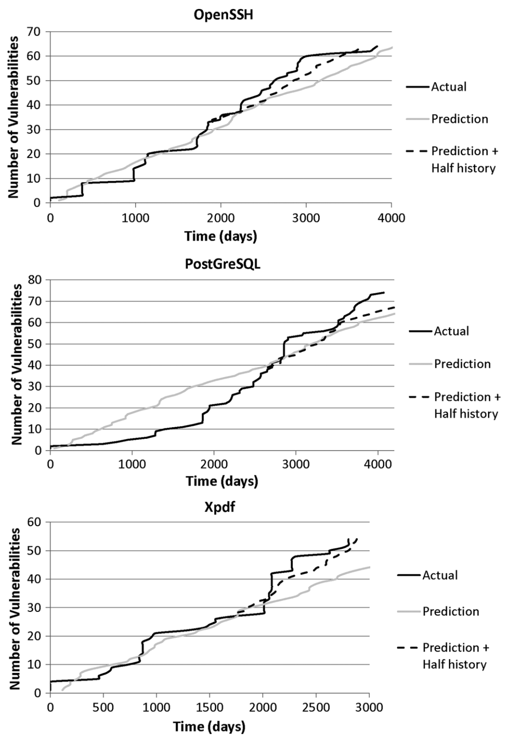

Fig. 12. Comparison of vulnerability scrying results with combined effect for OpenSSH, PostGreSQL, and Xpdf.

TABLE IV
AVERAGE SQUARED ERRORS OF VULNERABILITY SCRYING WITH AND WITHOUT HISTORICAL DATA

Codebase	Scrying	Scrying + Half History
PostGreSQL	63	55
Xpdf	27	11
OpenSSH	20	13

such data become available (i.e. vulnerabilities are discovered in the application). When vulnerability discovery data become available, they can be used to adjust vulnerability scrying in two ways:

i) it can correct our past predictions, and
ii) it can tune the parameters for future predictions.

In other words, we know the exact vulnerability discovery history for the vulnerabilities that are already discovered, so the model should match those data as close as possible. For future predictions, we can tune the prediction parameters based on past vulnerabilities.

Here we study the combined effect of vulnerability scrying and vulnerability history. Assume that the actual number of vulnerabilities discovered in a codebase at time t (discrete value in days) is given by $\hat{V}(t)$, and the number of vulnerabilities predicted by scrying is given by $V_r(t)$. Further assume that it is at time T; that is, we know the vulnerabilities 1 to $\hat{V}(T)$. We can enhance our scrying technique as follows. For $1 \le t \le T$, we exactly follow the history; i.e., $V_r(t) = \hat{V}(t)$. Note that this approach is possible because vulnerability scrying is a random walk. For every discrete value of time (day in our case), we also adjust the rate of scrying $(r = ((0.02 \times CQ) - (8 \times 10^{-5} \times CC) + 0.02)/2)$ by averaging it with the actual observed rate, as described in (15).

$$r(t) = \frac{r(t-1) + \frac{\hat{V}(t)}{t}}{2} \tag{15}$$

In (15), $\hat{V}(t)/t$ is the actual rate of vulnerability discovery. Observe that $r(1) = ((0.02\times CQ)-(8\times 10^{-5}\times CC)+0.02)/2$, and $r(\tau) = \hat{V}(\tau)/\tau, \tau \to \infty$. This means that, as we get more history, the scrying rate approaches the actual observed rate.

We have applied this scheme to our three evaluation codebases (OpenSSH, PostGreSQL, and Xpdf). Here we use half of the vulnerability data combined with vulnerability scrying. In each case, for the first half of the vulnerabilities, the scrying technique follows the exact discovery trend. For the second half, we use the adjusted rate from (15) for vulnerability scrying. The results are depicted in Fig. 12. As can be observed, combining vulnerability scrying with vulnerability history improves the accuracy of prediction in every case. The mean squared errors of scrying and scrying plus vulnerability history for the three codebases are presented in Table IV.

VI. Observations, and Limitations

During our study, we have made two observations that are worth reviewing here. First, analyzing the codebases, we noticed that the secure coding violations are often repeated many times in a codebase. If a violation is found in the first few source files, it is very likely that the same violation will be repeated in the rest of the codebase multiple times. In fact, finding a new type of violation was rare after the first few source files are analyzed. One hypothesis for this effect may be the underlying software development practices. The existence of a secure coding violation often implies that either the developers are not aware of the best practice, or the quality assurance process is unable to find that violation. As a result, chances are that the violation is repeated multiple times.

In addition, although a few common violations occur in many codebases, the exact set of violations is a highly distinctive attribute of a codebase. In fact, the sets of violations were unique in all the seven codebases that we analyzed. As such, the coding violations act as a type of finger print for the application. We do not have a firm understanding of this effect, but one hypothesis can be the developers coding habits.

Our technique has several limitations. First, it requires access to the source code of the application. In order to perform static analysis and extract code quality and cyclomatic complexity, vulnerability scrying requires source code analysis. One can imagine extending our technique to binary analysis in order to apply our technique to closed source applications, but we defer that to future work. Second, we have only considered the effect of code quality and cyclomatic complexity on the rate of vulnerability discovery. Other code characteristics may have an impact on vulnerability discovery, but they are not considered in this work. Type of a codebase (application, operating system, firmware, etc.), its vendor, its popularity, etc. are examples of characteristics not studied in this work. Third, we also ignore the changes in different versions of a code or changes applied by patching or updating an application. However, this limitation should not change the precision of our technique significantly because different studies have shown that the total amount of code changes in all versions of an application is very small compared to the code that remains unchanged. For example, Mockus and Votta [15] find that changed code lines (added, deleted, or modified) account for about 1.1% of the total lines of code. Other studies also have similar findings [16], [17].

VII. Related Work

The related work on the topic of software vulnerabilities focus on four areas: vulnerability analysis, vulnerability discovery and prediction models, vulnerability impact study, and patch and update management.

In the area of vulnerability analysis, Vache [18] studies various phases of the vulnerability lifecycle quantitatively using the OSVDB [6]. Massacci and Nguyen conducted an analysis on different vulnerability databases on Mozilla Firefox, and argues that using some of these sources might lead to loss of a large portion of vulnerabilities [19]. Clark *et al.* study the period after the release of a software product, and before the first vulnerability is discovered (honeymoon period) [20]. They also show that code re-use is a major contributor to the rate of vulnerability discovery, and the total number of vulnerabilities. Frei *et al.* study

TABLE V

Compass Rule	Emacs	GNUPG	Lynx	OpenSSH	OpenSSL	PostGreSQL	XPDF
Allocate&FreeMemoryInTheSameModule	✗	✗	✗	✓	✓	✗	✗
AssignmentOperatorCheckSelf	✓	✓	✓	✓	✓	✓	✓
AssignmentReturnConstThis	✓	✓	✓	✓	✓	✓	✓
AsynchronousSignalHandler	✓	✓	✓	✓	✓	✓	✓
AvoidSameHandlerForMultipleSignals	✓	✓	✓	✓	✓	✓	✓
BinPrintAsmFunctions	✓	✓	✓	✓	✓	✓	✓
BinPrintAsmInstruction	✓	✓	✓	✓	✓	✓	✓
BinaryBufferOverflow	✓	✓	✓	✓	✓	✓	✓
BinaryInterruptAnalysis	✓	✓	✓	✓	✓	✓	✓
BlankTestChecker	✓	✓	✓	✓	✓	✓	✓
BooleanIsHas	✓	✓	✓	✓	✓	✓	✓
BufferOverflowFunctions	✗	✗	✗	✓	✓	✓	✗
ByteByByteStructureComparison	✓	✓	✓	✓	✗	✓	✓
CharStarForString	✓	✓	✓	✓	✓	✓	✓
CommaOperator	✗	✓	✗	✓	✗	✓	✗
ConstCast	✓	✓	✗	✗	✓	✓	✓
ConstStringLiterals	✗	✗	✗	✗	✗	✗	✗
ConstructorDestructorCallsVirtualFunction	✓	✓	✓	✓	✓	✓	✓
ControlVariableTestAgainstFunction	✓	✓	✓	✓	✓	✓	✗
CopyConstructorConstArg	✓	✓	✓	✓	✓	✓	✓
CppCallsSetjmpLongjmp	✓	✓	✓	✓	✓	✓	✓
CycleDetection	✓	✓	✓	✓	✓	✓	✓
DataMemberAccess	✓	✓	✓	✓	✓	✓	✓
DefaultCase	✗	✓	✗	✗	✗	✓	✗
DefaultConstructor	✗	✗	✗	✓	✗	✗	✗
DiscardAssignment	✗	✗	✗	✗	✗	✗	✗
DoNotAssignPointerToFixedAddress	✗	✓	✗	✗	✓	✓	✗
DoNotCallPutenvWithAutoVar	✓	✓	✓	✓	✓	✓	✓
DoNotDeleteThis	✓	✓	✓	✓	✓	✓	✓
DoNotUseCstyleCasts	✗	✗	✗	✗	✗	✗	✗
DuffsDevice	✓	✓	✓	✓	✓	✓	✓
DynamicCast	✓	✓	✓	✓	✓	✓	✓
EmptyInsteadOfSize	✓	✓	✓	✓	✓	✓	✓
EnumDeclarationNamespaceClassScope	✗	✓	✓	✓	✓	✓	✗
ExplicitCharSign	✓	✓	✓	✗	✓	✓	✓
ExplicitCopy	✗	✗	✗	✓	✗	✗	✗
ExplicitTestForNonBooleanValue	✓	✓	✓	✓	✓	✓	✗
FileReadOnlyAccess	✓	✓	✓	✓	✓	✓	✓
FloatForLoopCounter	✓	✓	✓	✓	✓	✓	✓
FloatingPointExactComparison	✓	✓	✓	✓	✓	✓	✓
FopenFormatParameter	✓	✓	✓	✓	✓	✓	✓
ForLoopConstructionControlStmt	✓	✓	✗	✓	✓	✓	✗
ForLoopCppIndexVariableDeclaration	✓	✓	✓	✓	✓	✓	✓
ForbiddenFunctions	✓	✓	✓	✓	✓	✓	✓
FriendDeclarationModifier	✓	✓	✓	✓	✓	✓	✓
FunctionCallAllocatesMultipleResources	✓	✓	✓	✓	✓	✓	✓
FunctionDefinitionPrototype	✗	✗	✗	✗	✗	✗	✓
InductionVariableUpdate	✗	✗	✗	✓	✗	✗	✗
InternalDataSharing	✓	✓	✓	✓	✓	✓	✓
LocalizedVariables	✗	✗	✗	✗	✗	✗	✗
LowerRangeLimitv	✗	✓	✗	✗	✗	✓	✗
MallocReturnValueUsedInIfStmt	✓	✓	✓	✓	✓	✓	✗
MultiplePublicInheritance	✓	✓	✓	✓	✓	✓	✓
NameAllParameters	✓	✓	✓	✓	✗	✓	✓
NewDelete	✓	✓	✓	✓	✓	✓	✗
NoAsmStmtsOps	✓	✓	✓	✓	✓	✓	✓
NoExceptions	✓	✓	✓	✓	✓	✓	✓
NoExitInMpiCode	✓	✓	✓	✓	✓	✓	✓
NoGoto	✗	✓	✗	✗	✗	✓	✓
NoOverloadAmpersand	✓	✓	✓	✓	✓	✓	✓
NoRand	✓	✓	✓	✗	✓	✓	✓
NoSecondTermSideEffects	✗	✓	✗	✗	✗	✓	✗
NoSideEffectInSizeof	✓	✓	✓	✓	✓	✓	✓
NoTemplateUsage	✓	✓	✓	✓	✓	✓	✓
NoVariadicFunctions	✓	✓	✓	✓	✓	✓	✓
NoVfork	✓	✓	✓	✓	✓	✓	✓
NonAssociativeRelationalOperators	✗	✓	✗	✓	✓	✓	✗
NonStandardTypeRefArgs	✗	✓	✗	✗	✗	✓	✗
NonStandardTypeRefReturns	✗	✓	✗	✗	✗	✓	✗
NonVirtualRedefinition	✓	✓	✓	✓	✓	✓	✓
NonmemberFunctionInterfaceNamespace	✓	✓	✓	✓	✓	✓	✓
NullDeref	✗	✗	✗	✗	✗	✗	✗
OmpPrivateLock	✓	✓	✓	✓	✓	✓	✓
OneLinePerDeclaration	✗	✓	✗	✗	✗	✓	✗
OperatorOverloading	✓	✓	✓	✓	✓	✓	✓
OtherArgument	✓	✓	✓	✓	✓	✓	✓
PlaceConstantOnTheLhs	✗	✗	✗	✗	✗	✗	✗
PointerComparison	✗	✓	✗	✓	✓	✓	✓
PreferAlgorithms	✗	✗	✗	✗	✓	✗	✗
PreferFseekToRewind	✓	✓	✓	✓	✓	✓	✓
PreferSetvbufToSetbuf	✓	✓	✓	✓	✓	✓	✓
ProtectVirtualMethods	✓	✓	✓	✓	✓	✓	✓
PushBack	✓	✓	✓	✓	✓	✓	✓
RightShiftMask	✓	✓	✗	✗	✗	✓	✓
SetPointersToNull	✓	✗	✗	✓	✗	✗	✗
SingleParamConstructorExplicitModifier	✓	✓	✓	✓	✓	✓	✗
SizeOfPointer	✗	✗	✗	✓	✗	✗	✗
StringTokenToIntegerConverter	✓	✓	✗	✓	✓	✓	✓
SubExpressionEvaluationOrder	✓	✓	✓	✓	✓	✓	✓
TernaryOperator	✗	✓	✗	✗	✗	✓	✓
Time_tDirectManipulation	✓	✓	✗	✓	✗	✓	✓
UnaryMinus	✓	✓	✓	✓	✓	✓	✓
UninitializedDefinition	✗	✗	✗	✗	✗	✗	✗
UpperRangeLimit	✗	✓	✗	✗	✗	✗	✗
VariableNameEqualsDatabaseName	✓	✓	✓	✓	✓	✓	✓
VoidStar	✓	✓	✓	✓	✓	✓	✓

the vulnerability lifecycle and the performance of the security industry as a whole by analyzing all the vulnerabilities discovered during 1995 [21]. Mustafa discusses important practices (specifically design-time practices) in software design, vulnerabilities, and mitigation mechanisms [22]. Zhang *et al.* describe a five-layer model for the architecture of vulnerability discovery, analysis, and exploitation [23]. Ganapathy *et al.* present a framework to model and analyze application programming interface (API) level exploit [24]. Okun *et al.* describe an approach to evaluate the effect of using the Coverity static analysis tool to measure software security [25]. Schryen *et al.* propose metrics to quantify and compare security in closed source software vs. open source software [26]. Bozorgi *et al.* describe an approach to prioritize exploitable vulnerabilities using machine learning and data mining tools [27]. Wang *et al.* describe using Bayesian network to automate software vulnerabilities classification [28]. Tevis *et al.* describe removing software vulnerabilities in code design phase by replacing imperative techniques by functional programming approaches [29]. Nguyen *et al.* propose a model using machine learning techniques based on a set of metrics generated from the component dependency graph of software for vulnerability prediction [30]. Chowdhury *et al.* propose code level metrics to measure the security level of the source code [31].

AML [32], LP [8], RE [4], and AT [9] propose the major vulnerability discovery models. Alhazmi and Malaiya use the AML and linear models fitted to vulnerability data to predict the vulnerability discovery in three different codebases [5]. They demonstrate that by adding constraints to VDMs based on heuristics, it is possible to limit the estimation error. In another work, they use different VDMs to predict vulnerability discovery in four codebases [33]. In yet another work, they compare the accuracy of different VDMs for four codebases [7]. They observe that the prediction accuracy depends heavily on the amount of history (training data) used. Woo *et al.* use the AML VDM to model the vulnerability discovery using both time-based (number of vulnerability versus time) and effort-based (number of vulnerabilities versus number of installations) approaches [34]. They illustrate that both approaches can achieve similar accuracies. Kim *et al.* describe quantitative models for vulnerability discovery in different versions of the same application [35]. Using the AML model as their foundation, they discover that large parts of the codebase are shared across different version. Joh *et al.* show that the Weibull and AML VDMs achieve similar accuracies for the codebases studied [36]. Joh and Malaiya study the seasonal variations of vulnerability discovery [37], [38]. They find out that vulnerability discovery often peaks during the mid-end and year-end months. Chowdhury and Zulkernine propose that software metrics such as complexity, coupling, and cohesion (CCC) have strong correlation with vulnerabilities; therefore, those metrics can be used to indicate the total number of vulnerabilities in a software [11]. Ozment discusses several flawed assumptions in the VDMs (similarity in time and effort, similarity operational environments, independence, and static code) [39]. He also describes the theoretical requirements for VDMs.

On the topic of vulnerability impact analysis, Wang *et al.* develop an ontology for vulnerability management (OVM), to describe the relationships between IT products, vulnerabilities, attackers, security metrics, countermeasures, and other relevant concepts [40]. Telang and Wattal evaluate the impact of vulnerability announcements on firm stock price [41]. They show that vulnerability announcements lead to a negative, significant change in a software vendor's market value.

Finally, on patch management techniques, Okamura *et al.* describe analytical models for optimal patch release time to minimize damage cost of vulnerabilities [42]. Okhravi and Nicol use analytical and simulation techniques to evaluate the optimal patch testing time [43]. Cavusoglu *et al.* show that efficient vulnerability disclosure mechanisms can ensure the release of a patch [44].

VIII. Conclusion, and Future Work

We have proposed and evaluated a vulnerability prediction technique called vulnerability scrying. Our contributions and efforts are as follows.

- We proposed a vulnerability prediction scheme based on certain code properties (code quality and cyclomatic complexity) that does not require a vulnerability database or vulnerability history.
- We analyzed the impact of these properties (code quality and cyclomatic complexity) on vulnerability discovery trends by analyzing four popular software applications.
- We evaluated the accuracy of our approach by analyzing three other real world applications, and showed that the proposed technique can accurately predict vulnerabilities as soon as the application is released, without reliance on vulnerability history.

The future work could focus on multiple directions. Vulnerability scrying requires the source code of the application. Code quality extraction based on the binary of an application can relax this limitation. Techniques such as symbolic execution and dynamic taint analysis can be deployed for binary code analysis.

In addition, we only focused on cyclomatic complexity as a measure of code complexity. Future work can incorporate different complexity metrics to obtain the scrying rate.

Finally, performing experiments and evaluations on more codebases can result in more accurate predictions. Preparing a codebase for static analysis and collecting code violations is often a very time consuming task. For large sample sizes, automating the process can facilitate the analysis.

Appendix A
Secure Coding Rules

The table below lists the Compass secure coding rules, and whether or not the codebases studied in this paper comply with them. ✓ denotes compliance with the rule while ✗ denotes violation of the rules. See Table V on the previous page.

References

[1] National Vulnerability Database 2011 [Online]. Available: http://nvd.nist.gov/

[2] National Institute of Standards and Technology 2011 [Online]. Available: http://www.nist.gov/

[3] D. Mellado, E. Fernández-Medina, and M. Piattini, "A comparison of software design security metrics," in *Proc. 4th Eur. Conf. Software Architecture: Companion Volume*, New York, NY, USA, 2010, pp. 236–242 [Online]. Available: http://doi.acm.org/10.1145/1842752.1842797, ser. ECSA '10, ACM

[4] E. Rescorla, "Is finding security holes a good idea?," *IEEE Security Privacy*, vol. 3, no. 1, pp. 14–19, Jan./Feb. 2005.

[5] O. Alhazmi and Y. Malaiya, "Prediction capabilities of vulnerability discovery models," in *Proc. RAMS '06. Annu. Rel. Maintainability Symp.*, 2006, pp. 86–91.

[6] The Open Source Vulnerability Database 2011 [Online]. Available: http://osvdb.org/

[7] O. Alhazmi and Y. Malaiya, "Application of vulnerability discovery models to major operating systems," *IEEE Trans. Rel.*, vol. 57, no. 1, pp. 14–22, Mar. 2008.

[8] J. D. Musa and K. Okumoto, "A logarithmic poisson execution time model for software reliability measurement," in *Proc. 7th Int. Conf. Software Eng.*, 1984, pp. 230–238.

[9] R. J. Anderson, "Security in open versus closed systems—The dance of Boltzmann, Coase and Moore," in *Proc. Conf. Open Source Software Econ.omics*, 2002, p. 5.

[10] M. Widenius and D. Axmark, *Mysql Reference Manual*, P. DuBois, Ed., 1st ed. Sebastopol, CA, USA: O'Reilly Associates, Inc., 2002.

[11] I. Chowdhury and M. Zulkernine, "Can complexity, coupling, and cohesion metrics be used as early indicators of vulnerabilities?," in *Proc. 2010 ACM Symp. Appl. Comput.*, New York, NY, USA, 2010, pp. 1963–1969 [Online]. Available: http://doi.acm.org/10.1145/1774088.1774504, ser. SAC '10, ACM

[12] T. J. McCabe, "A complexity measure," in *Proc. 2nd Int. Conf. Software Eng.*, Los Alamitos, CA, USA, 1976, p. 407 [Online]. Available: http://portal.acm.org/citation.cfm?id=800253.807712, ser. ICSE '76, IEEE Comput. Soc. Press

[13] M. Schordan and D. Quinlan, "A source-to-source architecture for user-defined optimizations," in *Proc. Joint Modular Lang. Conf. (JMLC'03)*, 2003, pp. 214–223.

[14] *"Compass Manual"* Lawrence Livermore National Laboratory, 2009 [Online]. Available: http://www.rosecompiler.org/compass.pdf, Manual

[15] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Proc. Int. Conf. Software Maintenance (ICSM'00)*, Washington, DC, USA, 2000, p. 120, ser. ICSM '00.

[16] M. M. Lehman, D. E. Perry, and J. F. Ramil, "Implications of evolution metrics on software maintenance," in *Proc. Int. Conf. Software Maintenance*, Washington, DC, USA, 1998, p. 208, ser. ICSM '98.

[17] R. Purushothaman and D. E. Perry, "Toward understanding the rhetoric of small source code changes," *IEEE Trans. Software Eng.*, vol. 31, no. 6, pp. 511–526, Jun. 2005.

[18] G. Vache, "Vulnerability analysis for a quantitative security evaluation," in *Proc. 2009 3rd Int. Symp. Empirical Software Eng. Measur.*, Washington, DC, USA, 2009, pp. 526–534 [Online]. Available: http://dx.doi.org/10.1109/ESEM.2009.5315969, ser. ESEM '09, IEEE Computer Society

[19] F. Massacci and V. H. Nguyen, "Which is the right source for vulnerability studies?: An empirical analysis on mozilla firefox," in *Proc. 6th Int. Workshop Security Measur. Metrics*, New York, USA, 2010, pp. 4:1–4:8 [Online]. Available: http://doi.acm.org/10.1145/1853919.1853925, ser. MetriSec '10, ACM

[20] S. Clark, S. Frei, M. Blaze, and J. Smith, "Familiarity breeds contempt: The honeymoon effect and the role of legacy code in zero-day vulnerabilities," in *Proc. 26th Annu. Comput. Security Appl. Conf.*, New York, NY, USA, 2010, pp. 251–260 [Online]. Available: http://doi.acm.org/10.1145/1920261.1920299, ser. ACSAC '10, ACM

[21] S. Frei, M. May, U. Fiedler, and B. Plattner, "Large-scale vulnerability analysis," in *Proc. 2006 SIGCOMM Workshop Large-Scale Attack Defense*, New York, NY, USA, 2006, pp. 131–138 [Online]. Available: http://doi.acm.org/10.1145/1162666.1162671, ser. LSAD '06, ACM

[22] S. Rehman and K. Mustafa, "Research on software design level security vulnerabilities," *SIGSOFT Softw. Eng. Notes* vol. 34, pp. 1–5, December 2009 [Online]. Available: http://doi.acm.org/10.1145/1640162.1640171

[23] Y.-C. Zhang, Q. Wei, Z.-L. Liu, and Y. Zhou, "Research on the architecture of vulnerability discovery technology," in *Proc. Int. Conf. Mach. Learn. Cybern. (ICMLC)* , 2010, vol. 6, pp. 2854–2859.

[24] V. Ganapathy, S. Seshia, S. Jha, T. Reps, and R. Bryant, "Automatic discovery of api-level exploits," in *Proc. 27th Int. Conf. Software Eng. (ICSE 2005)* , May 2005, pp. 312–321.

[25] F. Massacci and V. H. Nguyen, "Which is the right source for vulnerability studies?: An empirical analysis on mozilla firefox," in *Proc. 6th Int. Workshop Security Measur. Metrics*, New York, USA, 2010, pp. 4:1–4:8 [Online]. Available: http://doi.acm.org/10.1145/1853919.1853925, ser. MetriSec '10, ACM

[26] G. Schryen and R. Kadura, "Open source vs. closed source software: Towards measuring security," in *Proc. 2009 ACM Symp. Appl. Comput.*, New York, USA, 2009, pp. 2016–2023 [Online]. Available: http://doi.acm.org/10.1145/1529282.1529731, ser. SAC '09, ACM

[27] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond heuristics: Learning to classify vulnerabilities and predict exploits," in *Proc. 16th ACM SIGKDD Int. Conf. Knowledge Discovery Data Mining*, New York, NY, USA, 2010, pp. 105–114 [Online]. Available: http://doi.acm.org/10.1145/1835804.1835821, ser. KDD '10, ACM

[28] J. A. Wang and M. Guo, "Vulnerability categorization using bayesian networks," in *Proc. 6th Annu. Workshop Cyber Security Inf. Intell. Res.*, New York, USA, 2010, pp. 29:1–29:4 [Online]. Available: http://doi.acm.org/10.1145/1852666.1852699, ser. CSIIRW '10, ACM

[29] J.-E. J. Tevis and J. A. Hamilton, "Methods for the prevention, detection and removal of software security vulnerabilities," in *Proc. 42nd Annu. Southeast Regional Conf.*, New York, NY, USA, 2004, pp. 197–202 [Online]. Available: http://doi.acm.org/10.1145/986537.986583, ser. ACM-SE 42, ACM

[30] V. H. Nguyen and L. M. S. Tran, "Predicting vulnerable software components with dependency graphs," in *Proc. 6th Int. Workshop Security Measur. Metrics*, New York, NY, USA, 2010, pp. 3:1–3:8 [Online]. Available: http://doi.acm.org/10.1145/1853919.1853923, ser. MetriSec '10, ACM

[31] I. Chowdhury, B. Chan, and M. Zulkernine, "Security metrics for source code structures," in *Proc. 4th Int. Workshop Software Eng. Secure Syst.*, New York, NY, USA, 2008, pp. 57–64 [Online]. Available: http://doi.acm.org/10.1145/1370905.1370913, ser. SESS '08, ACM

[32] O. Alhazmi and Y. Malaiya, "Prediction capabilities of vulnerability discovery models," in *Proc. RAMS '06. Annu. Rel. Maintainability Symp.*, 2006, pp. 86–91.

[33] O. Alhazmi and Y. Malaiya, "Measuring and enhancing prediction capabilities of vulnerability discovery models for apache and iis http servers," in *Proc. ISSRE '06. 17th Int. Symp. Software Rel. Eng.*, 2006, pp. 343–352.

[34] S.-W. Woo, O. Alhazmi, and Y. Malaiya, "Assessing vulnerabilities in apache and iis http servers," in *Proc. 2nd IEEE Int. Symp. Dependable, Autonomic Secure Comput.*, 2006, 29.

[35] J. Kim, Y. Malaiya, and I. Ray, "Vulnerability discovery in multi-version software systems," in *Proc. 10th IEEE High Assurance Syst. Eng. Symp.*, 2007, pp. 141–148.

[36] H. Joh, J. Kim, and Y. Malaiya, "Vulnerability discovery modeling using weibull distribution," in *Proc. 19th Int. Symp. Software Rel. Eng.*, 2008, pp. 299–300.

[37] H.-C. Joh and Y. Malaiya, "Seasonal variation in the vulnerability discovery process," in *Proc. ICST '09. Int. Conf. Software Testing Verification Validation*, 2009, pp. 191–200.

[38] H. Joh and Y. Malaiya, "Seasonality in vulnerability discovery in major software systems," in *Proc. 19th Int. Symp. Software Rel. Eng.*, 2008, pp. 297–298.

[39] A. Ozment, "Improving vulnerability discovery models," in *Proc. 2007 ACM Workshop Qual. Protection*, New York, USA, 2007, pp. 6–11 [Online]. Available: http://doi.acm.org/10.1145/1314257.1314261, ser. QoP '07, ACM

[40] J. A. Wang and M. Guo, "Ovm: An ontology for vulnerability management," in *Proc. 5th Annual Workshop Cyber Security Inf. Intell. Res.*, New York, NY, USA, 2009, pp. 34:1–34:4 [Online]. Available: http://doi.acm.org/10.1145/1558607.1558646, ser. CSIIRW '09, ACM

[41] R. Telang and S. Wattal, "An empirical analysis of the impact of software vulnerability announcements on firm stock price," *IEEE Trans. Softw. Eng.* vol. 33, pp. 544–557, Aug. 2007 [Online]. Available: http://portal.acm.org/citation.cfm?id=1435729.1437850

[42] H. Okamura, M. Tokuzane, and T. Dohi, "Optimal security patch release timing under non-homogeneous vulnerability-discovery processes," in *Proc. 20th Int. Symp. Software Rel. Eng..* , 2009, pp. 120–128.

[43] H. Okhravi and D. Nicol, "Evaluation of patch management strategies," *Int. J. Comput. Intell. Theory Practice*, vol. 3, no. 2, pp. 109–117, Dec. 2008.

[44] H. Cavusoglu, H. Cavusoglu, and S. Raghunathan, "Efficiency of vulnerability disclosure mechanisms to disseminate vulnerability knowledge," *IEEE Trans. Softw. Eng.* vol. 33, pp. 171–185, Mar. 2007 [Online]. Available: http://portal.acm.org/citation.cfm?id=1263149.1263517

[45] "Scry.," Def. 1. Google Dictionary Online, Google, n.d. Web. 17 Mar. 2013.

Sanaz Rahimi received her Ph.D. in computer science from Southern Illinois University Carbondale in 2013. She also received her M.S., and B.S. in computer science from Southern Illinois University Carbondale, and University of Southern Mississippi respectively. Her research interests include cyber security, vulnerability analysis, and software reliability.

Mehdi Zargham is a Professor and the Chair of the Department of Computer Science, Southern Illinois University Carbondale. He received his M.S., and Ph.D. degrees in computer science from Michigan State University in 1980, and 1983, respectively. His research interests include mobile learning, pattern recognition, and data mining.