CERIAS Tech Report 2005-06

VULNERABILITY LIKELIHOOD: A PROBABILISTIC APPROACH TO SOFTWARE ASSURANCE

by Rajeev Gopalakrishna, Eugene H. Spafford, and Jan Vitek

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

Vulnerability Likelihood: A Probabilistic Approach to Software Assurance

Rajeev Gopalakrishna, Eugene H. Spafford, and Jan Vitek
Department of Computer Sciences, Purdue University
250 N. University Street, West Lafayette, IN 47907-2066
Email: {rgk,spaf,jv}@cs.purdue.edu

Abstract

The importance of software security is undeniable given the impact of software on our lives. Assurance about the security properties of a software artifact should ultimately translate into a quantitative measure of vulnerabilities. In this paper, we present the idea of vulnerability likelihood as a probabilistic approach to software assurance. Gaining assurance early in the software development cycle is of immense value in directing future efforts. So we first discuss vulnerability likelihood in the context of vulnerability prediction in software artifacts. We propose four types of program properties that can be observed in software artifacts to potentially determine their vulnerability likelihood. Then we discuss vulnerability likelihood in the context of vulnerability detection. We propose a technique to quantify the assurance in the solutions of checkers for vulnerability detection that use static analysis. And finally, we illustrate the importance of vulnerability likelihood in a software development methodology to measurably increase software assurance.

1. INTRODUCTION

The present state of software development is more an art than science—therefore producing software artifacts. The subjectivity involved in art appreciation is unacceptable with software, especially when it's impact on human life is tremendous and continuously increasing. Assurance about software properties should be guaranteed by objective measures of software attributes. One such attribute that is of immense concern is software security.

To Err is Human: as long as programming involves human activity, software vulnerabilities will exist. A software vulnerability is an instance of an error in the specification, development, or configuration of software such that its execution can violate the security policy [33]. As such, any assurance about the security properties of a software artifact should ultimately translate into a quantitative assessment of the vulnerabilities in that artifact. While the areas of software engineering, software quality, and software reliability have been studied extensively over the last three decades, they are mostly concerned with assuring the usability of software under normal conditions. Accidental failures resulting from accidental faults are their subject of concern and not malicious attacks resulting from vulnerability exploits. Software security has to deal with both accidental and malicious failures resulting from faults introduced either accidentally or deliberately [35]. It is this differentiating factor of intention that makes software security a challenging task.

Boehm et al.[13] define software quality in terms of the high-level software characteristics of portability, reliability, efficiency, human engineering, testability, understandability, and modifiability. The degree to which the software has these characteristics determines the quality of the software. Software reliability, a characteristic of software quality, is the probability that software will provide failure-free operation in a fixed environment for a fixed period of time. The errors and the faults leading to failures in this context are accidental in nature and not malicious.

Deliberate attempts to cause failure by triggering faults is what software security is concerned with and has to protect against. While improving the quality of software could in general reduce the number of faults because of all the contributing characteristics, it might not eliminate faults that have a low likelihood of being triggered accidentally. However, such faults could be exploited by someone deliberately looking for flaws in the software and therefore should be avoided to secure (w.r.t a given policy) the software from malicious events. Nonetheless, a majority of the problems in software security today can be attributed to commonly repeated mistakes—an indicator of poor quality. Therefore, improving software quality in a manner such that the common mistakes are avoided would greatly improve software security.

Conte et al. [43] define *software metrics* as those metrics that are used to characterize the essential features of software quantitatively, so that classification, comparison, and mathematical analysis can be applied. Software metrics can be classified into process metrics and product metrics. While process metrics quantify attributes of the development process and of the development environment, product metrics quantify attributes of the software artifact itself. Software metrics have been proposed for software quality in general and software complexity and software reliability in particular.

The 2001 workshop on information security system rating and ranking [4] discussed different aspects of security metrics. "What should we count and what do the numbers mean?" as pertaining to software security metrics was one of the challenge problems discussed by security experts at the 2003 UW-MSR Summer Institute [7]. The development of meaningful security metrics was chosen as a grand challenge at the CRA Conference on "Grand Research Challenges" consecutively in 2002 [5] and 2003 [8]. This exemplifies the immediate need for meaningful measures of software security as perceived by experts in the field.

In this paper, we introduce the concept of *vulnerability likelihood* of a software artifact, which we define as a probabilistic assessment of vulnerabilities in the software artifact. We focus on two dimensions of the idea of quantifying vulnerability likelihood. First, we discuss the idea of quantifying the vulnerability likelihood of software artifacts. This deals with predicting the number and location of vulnerabilities in a software artifact at different phases of it's development life-cycle. We propose four types of program properties that can potentially be used in a software vulnerability prediction model and discuss the challenges in using them. We consider this as a new paradigm in the context of prediction because previous work has concentrated on predicting software faults, failures, and intrusions but not vulnerabilities. Vulnerabilities have been investigated primarily from the detection viewpoint. Second, we propose a technique based on probabilistic static analysis for quantifying the *likelihood of a vulnerability* as detected by a static security checker. Although the use of static analysis to detect vulnerabilities is not new, the concept of quantifying the approximations in static analysis to estimate the likelihood of the detected vulnerabilities is a new paradigm that we believe will significantly impact the usefulness of static security checkers by ranking vulnerabilities by their "false positiveness." And finally, we describe the role of vulnerability likelihood in a software assurance methodology.

2. VULNERABILITY LIKELIHOOD OF A SOFTWARE ARTIFACT

An error is a mistake made by a developer. It might be a typographical error, a misreading of a specification, a misunderstanding of what a subroutine does, and so on [29]. An error might lead to one or more faults. Faults (also known as defects) are located in the text of the program. More precisely, a fault is the difference between the incorrect program and the correct version [29]. The execution of faulty code may lead to zero or more failures, where a failure is the (non-empty) difference between the results of the incorrect and correct program [29].

A study released by the U.S Department of Commerce's National Institute of Standards and Technology (NIST) in 2002 estimated that software defects cost U.S economy $59.5 billion annually [6]. The study also found that over half of those defects are not found till late in the development cycle and that more than a third of the costs could have been eliminated if the defects been identified and removed earlier in the cycle. Therefore, it would be useful in a number of ways to predict the number and location of defects in a software artifact at different phases in it's life-cycle. For example, predicting the number of defects in different software modules can help prioritize testing efforts to the more defect-prone modules. Predicting the number of residual defects (post-release defects) in a software product can help guage the quality of the delivered product and determine the maintenance effort.

A software vulnerability is an instance of an error in the specification, development, or configuration of software such that its execution can violate the security policy [33]. In other words, a defect whose execution can violate the security policy is a vulnerability. So all vulnerabilties are defects but all defects are not vulnerabilties. And as mentioned earlier, triggering of defects leading to failures is accidental in nature but triggering of vulnerabilities leading to security violations or intrusions may be deliberate and malicious. A malicious intrusion can, in general, do more harm than an accidental failure. Some of the losses may also be intangible such as the reputation of both the software developer whose software had the vulnerability and the software user who experienced the intrusion. As such, it would be a higher priority for both the developer and the user to be able to predict vulnerabilities than defects.

In this section, we look at the work in the area of software defect prediction. We identify a model based on a probabilistic and holistic approach that has the potential to overcome the drawbacks of the previously proposed models. We believe that this software defect prediction model can also serve as a basis for a software vulnerability prediction model because vulnerabilities are also defects. We then propose four types of program properties that can be used within the framework of such a model towards predicting vulnerabilities.

2.1 Software Defect Prediction

There are numerous studies in this area of software defect prediction. Here we briefly summarize the different defect prediction models as discussed in Fenton and Neil [25] and describe the probabilistic model proposed by them.

1. Prediction using Complexity Metrics

   Most defect prediction research has focussed on establishing defects as a function of complexity metrics with the hypothesis that programming complexity affects the introduction and persistence of defects. Complexity metrics can be considered under four categories as proposed by Conte et al. [43]:

   (a) *Size metrics* that measure the size of software using parameters such as lines of code, token count, and function count. For example, Akiyama [10] computes the following equation relating the number of defects D and the lines of code L:

   $$D = 4.86 + 0.018L \quad (1)$$

   This equation suggests that larger software modules have lower defect densities (defects/size). Other studies that observe the same phenomenon of larger modules having lower defect densities attempt to explain it by hypothesizing that larger modules may be developed more carefully than smaller modules or that larger modules may

still have numerous residual defects because of smaller test case coverage. Compton and Withrow [16] compute the following polynomial regression equation for Ada modules:

$$D = 0.069 + 0.00156L + 0.00000047L^2 \qquad (2)$$

And based on this equation, they propose the "Goldilock's Principle," which suggests that there is an optimum module size that is neither too big nor too small with respect to defect density.

(b) *Data structure metrics* that measure the amount of data input to, processed in, and output from software such as variable count, Halstead's operand count (η_2) and total occurrences of operands (N_2), live variables, variable spans, fan-in, and fan-out.

(c) *Logic structure metrics* that measure the control flow aspects of software such as decision count, McCabe's cyclomatic complexity, Schneidewind and Hoffmann's minimum number of paths and reachability metrics, depth of nesting, and knots.

(d) *Composite metrics* that combine two or more of the above metrics such as Halstead's *Software Science* metrics. Halstead derives the following relationship between the number of defects D and the Volume metric V (which in turn is defined in terms of the number of unique operands and unique operators):

$$D = V/3000 \qquad (3)$$

To explain the negative correlation between module size (in terms of lines of code L) and defect density (D/L), Rosenberg [41] suggests that because there must be a negative correlation between values of L and 1/L, the correlation between size and defect density must also be negative whenever defects are growing at most linearly in size. The negative correlation and the Goldilock's principle challenge the fundamental concept of software decomposition in software engineering. Program complexity is certainly one of the contributing factors to defects but it is not the only cause. Models that predict the defects as a function of only program complexity do not consider the causal effects of programmers and designers, problem difficulty, and design complexity. The relationship between defects and program complexity is therefore not a straightforward one.

2. Prediction using Testing Metrics

   The defects detected during the testing phases can be used to predict residual defects using statistical extrapolation techniques. Test coverage, which is a metric of completeness according to a test selection criterion such as a branch or statement, can be used to derive the Test Effectiveness Ratio (TER) metric for a given set of test cases. The TER for a set of test cases is the proportion of the coverage achieved. The relationship between test coverage metrics and defect density has also been studied [36].

3. Prediction based on Process Quality Data

   Process metrics quantify the attributes of the software development process including process quality. The argument is that process quality is a good predictor of product quality in terms of the residual defect density. The 5-level SEI Capability Maturity Model (CMM) is an example of a process quality metric whose influence on residual defect density has been empirically shown by Diaz and Sligo [21]. Other software development process methodologies include the Cleanroom approach [22], Extreme Programming [11], and Personal/Team Software Process [27, 28].

Fenton and Neil [25] suggest that although a lot of research has been conducted in the area of software defect prediction, the problem remains largely unsolved and the solutions unsatisfactory. Univariate approaches to prediction modeling are simple, intuitive, and appealing, but inaccurate. Multivariate approaches (that use more than one predictor variable) have suffered from the problem of collinearity where two or more predictor variables capture the same underlying factor when multivariate regression depends on the assumption of zero correlation between predictor variables. They argue that the problem is mainly of understanding and representing the complex inter-relationships in the software development process and propose a holistic approach to software defect prediction modeling based on a probabilistic technique known as Bayesian Belief Networks.

2.1.1 A Model Based on Bayesian Belief Networks

A Bayesian Belief Network (BBN) is a directed graph that represents probabilistic relationships among the graph nodes. The nodes represent variables and the arcs represent the causal or influential relationships among the variables. Each node is associated with a *node probability table* (NPT) that contains the (Bayesian) conditional probability of the node given the state of it's parent nodes. The probabilities in the table are derived from previously observed statistical data or from expert opinions in its absence.

BBNs enable one to model and reason in the presence of uncertainty. The sound mathematical basis of Bayesian probability and the intuitive graphical representation make them suitable for modeling and visualizing complex relationships among variables such as those in the software development process. A prototype BBN for defect prediction (as illustrated in [25]) representing the relationships between defects and the processes of specification, design (including coding), and testing is shown in figure 1. Testing the model on data from 28 projects has shown that the predictions are reliable and encouraging [24].

2.2 Software Vulnerability Prediction

A defect prediction model such as the one based on BBN should also be able to predict vulnerabilities. Our hypothesis is that we can enhance the model towards predicting vulnerabilities by incorporating our knowledge about vulnerabilities into the model. We should consider both the general characteristics of vulnerabilities and the specific characteristics of different types of vulnerabilities in the model considering that a drawback of most empirical research on defect

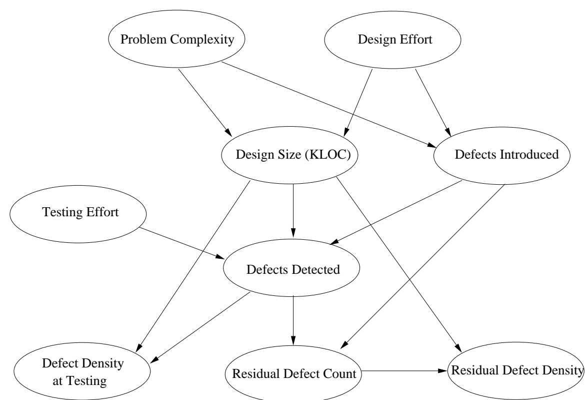

Figure 1: BBN topology for defect prediction as illustrated in [25]. It represents the causal/influential relationships between defects and the processes of specification, design (including coding), and testing. In the figure, ***Problem complexity*** **can be interpreted as the complexity inherent in the functional requirements of the specification and the term** ***Design*** **includes both design and coding. KLOC: 1000 Lines of Code.**

prediction is that they have failed to distinguish between different classes of defects [25].

In this section, we describe program properties that we believe could affect the vulnerability likelihood of a software artifact and thus influence the vulnerability prediction. We focus only on the vulnerabilities introduced in the coding phase and not on the other contributing factors such as problem complexity, design effort, or the testing effort. It is important to note that we are not interested in actually detecting vulnerabilities at this stage of predicting vulnerability likelihood. We are interested only in determining program properties that indicate the presence of vulnerabilities.

We use C as the programming language for illustrating most of our ideas. This is because, a majority of the critical and vulnerable programs continue to be written in C [3]. There is also a lot of legacy code written in C. We expect many of the observations and results to be applicable to programs written in other languages such as C++ and Java. We also conjecture that there will be language-specific features that will have influence on the vulnerability likelihood.

1. Privileged Lines of Code

   The notion of privilege is fundamental to the concept of access control where principals (processes or users) have different levels of privileges and can access only those system resources that their privilege level is allowed to access. Higher privilege levels are allowed access to the more critical system resources and are susceptible to greater abuse.

   Different modules of a software artifact may require different levels of privileges in which case they execute under the identity of users that have the necessary privileges. When different parts of a single module differ in their privilege requirement because of the nature of operations they perform, privileges can be elevated and dropped within the modules using certain mechanisms (such as `setuid` in Unix). So not all the lines of code of an artifact necessarily run with the same privileges. And those that run with greater privileges are capable of greater damage because they have access to critical system resources. Therefore, any error made by a programmer while writing code that executes with high privileges (henceforth refereed to as Privileged Lines of Code or PLOC) may result in a vulnerability. For example, an instance of a buffer overflow in a block of code that executes with the privileges of a `root` in Unix or `SYSTEM` in Windows leads to a vulnerability.

   Now if we assume that programmers have a certain error rate that can be expressed in terms of lines of code then we can hypothesize that greater the PLOC in a software artifact or module, greater it's vulnerability likelihood. Although this seems to be intuitively valid and is epitomized in the principle of granting least privilege for the least length of time, we need empirical evidence for a positive correlation between PLOC and the number of vulnerabilities in software artifacts. Then we can use PLOC as a factor for predicting vulnerability likelihood analogous to the use of LOC as a size metric of software complexity in predicting defects.

2. Error-prone Constructs

   These refer to specific constructs and certain ways of using such constructs that have a history of flaws. They may either be library functions or system calls (invoked through library stubs). They could also be user-written functions that have been error-prone in the past. The very presence of such constructs might be error-prone or might depend on the nature of their arguments or their environment of use. For instance, the length of arguments should be carefully checked while using functions such as `gets`, `strcpy`, and `strcat` to avoid buffer overflows. Format arguments of functions such as `printf` and `sprintf` should be checked, especially if they are variables and user-modifiable, to avoid format string vulnerabilities [17]. And functions that operate on file names should be avoided and alternatives that use file descriptors should be used to prevent race condition vulnerabilities [12].

   Absence of such error-prone constructs does not imply the absence of vulnerabilities resulting from their use. Programmers often write their own safe versions of error-prone C library functions and use them instead. While reproducing the functionality of the library functions, the errors might also have been duplicated in which case the use of such programmer-written functions is as error-prone as their C library counterparts.

   Presence of such error-prone constructs does not necessarily constitute vulnerabilities. They might have been used in an appropriate manner with knowledge of their error-prone variations. But this is not always

the case. Programmers often do not completely understand the dangers of using such functions. Even if they do, they might not be aware of safer alternatives. Ultimately, such functions end up getting used because of lack of awareness or lack of concern. Although programmers might get lucky in that their use of such constructs turn out to be harmless because of the nature of their arguments or the environment, making it a habit would result in their repeated use and ultimately result in vulnerabilities. However, there is no empirical evidence for this conjecture. If we can empirically demonstrate the existence of such a phenomenon then that would justify the use of error-prone constructs as a factor for predicting the vulnerability likelihood of a software artifact.

Checkers that detect occurrences of error-prone constructs include RATS [2], Flawfinder [1], and ITS4 [46]. These checkers perform lexical analysis using regular expressions and detect matches with error-prone constructs stored in their databases. Arguments are minimally checked in certain cases. They produce a list of "hits" with descriptions of the problems, their location in the code, associated level of risk, and potential safer alternatives. Such checkers are better than using the `grep` utility to search for error-prone constructs for many reasons [46]. Although they use simple pattern matching, these checkers indicate portions of the code that at least deserve a careful scrutiny. The use of only lexical analysis makes these checkers faster than those that work on parsed code. This also enables them to work on non-preprocessed code, which makes it possible to analyze all possible builds of a program.

3. Programming mistakes

   In March 2004, a critical security vulnerability was found in the Linux kernel memory management code inside the `mremap` system call because of a failure to check the return value of a function called in the system call code (identified as CAN-2004-0077 by CVE). Checking the return values of functions especially system calls has long been recognized as a good programming practice. Failure to do so may not always lead to a vulnerability. But it might when an exceptional condition occurs (such as the function fails and returns an error code) as in the case of the Linux vulnerability.

   Xie and Engler [47] check for redundant assignments, dead code, and redundant conditionals in Linux. They found strong correlation between such redundant program properties and what they refer to as "hard errors" that include use of freed memory, dereferences of null pointers, potential deadlocks, and unreleased locks. They further found that a file containing such a redundancy was roughly 45% to 100% more likely to have a hard error than a randomly chosen file.

   Tools such as `lint`, LCLint [23], and the `-Wall` option in the GNU C compiler check for program properties such as dead code, unused declarations, type inconsistencies, use before definition, ignored return values, statements with no effects, likely infinite loops, and fall through cases in `switch` statements. Such program properties are not defects or vulnerabilities in themselves but they could be indicators of potential ones. They might be the result of mistakes made by a programmer who is either unaware or unconcerned. Or the programmer might have ignored them after checking them to be harmless. Empirically observing a correlation between such program properties and vulnerabilities would justify their use in predicting the vulnerability likelihood of software artifacts. The fact that Xie and Engler [47] have found positive correlation between three such properties and hard errors is encouraging.

4. Program Style

   Programming style is concerned with the readability of programs. A program that is difficult to read will be difficult to understand, modify, test, and maintain because humans (mostly different ones in a collaborative development environment) are involved in these activities. It would therefore affect the psychological complexity of the programs. Issues of readability and complexity would either introduce new vulnerabilities (or defects) or allow the existing ones to persist by avoiding detection. In this manner, program properties concerned with style affect the vulnerability likelihood of software artifacts.

   There are several books and articles on programming style. Kernighan and Plauger [31] give 77 rules of "good" programming style. Oman and Cook [40] provide a list of 236 style rules and also propose a taxonomy [39] that has the following four categories:

   (a) *General practices* related to the programming process such as understanding and defining the problem before coding and rewriting code instead of patching or commenting bad code.
   (b) *Typographic style* related to the layout and commenting of code such as using spacing, blank lines, comments, parentheses, and meaningful identifier names to improve clarity and readability.
   (c) *Control structure style* related to the control flow aspects that affect the execution of the program such as issues of modularity, nesting, looping, and branching.
   (d) *Information structure style* related to data structure and data flow techniques such as choice of data structures, initialization of variables, and validation of inputs.

   Although these style rules are believed to have emerged from a consensus of experienced programmers and researchers, there are many contradictory guidelines as noted by Oman and Cook [39]. Lack of sufficient empirical results and the use of improper methodology in studying the benefits of programming style have also been observed [45]. There hasn't been much progress in the empirical validation of style factors. The widely-used style rules have been accepted as best practices. For example, the Motor Industry Software Reliability Association (MISRA) in its guidelines for the use of C software in automobiles [37] has several program style factors. Checking for style factors and observing how they correlate with vulnerabilities would provide a basis for including them as a factor in software vulnerability prediction.

3. LIKELIHOOD OF A VULNERABILITY

A program's compliance with a property can be checked either at run-time (dynamic checking and conventional testing) or at compile-time. Static analysis is the process of extracting semantic information about a program at compile time. The ability to check program properties without having to execute the program is especially appealing for security properties. A variety of static analysis checkers for detecting software vulnerabilities have been proposed [12, 15, 20, 44].

Intraprocedural static analysis is undecidable [9] and many interprocedural static analysis problems are NP-complete [38]. This implies that interprocedural static analysis must make approximations to be decidable and tractable (finish in polynomial time). Alias analysis, which is the problem of statically finding aliases, is a fundamental problem of static analysis because any data-flow analysis in the presence of pointers and procedures (call-by-reference kind) has to deal with aliases. What makes alias analysis particularly difficult is the fact that even with the simplifying assumption that all paths in a program are executable (necessary to make intraprocedural static analysis decidable), intraprocedural alias analysis is still undecidable for languages with conditionals, loops, dynamic storage, and recursive data structures [34].

This means that intraprocedural alias analysis for a language such as C has to make approximations to be decidable and further approximations to be tractable. These approximations result in inaccurate solutions to data-flow problems. From the viewpoint of checking program properties, these approximations affect the accuracy of our assertions. Safe approximations, those that preserve the externally observed program behavior, resulting in overestimates may lead to false positives. And unsafe approximations may lead to both false positives and false negatives.

Current static analysis based security checkers approach the detection of vulnerabilities in a binary manner—a vulnerability is either present or absent. This binary treatment coupled with the fact that the checkers make approximations to keep the analysis decidable, tractable, and scalable leads to both false negatives and false positives. Furthermore, a vulnerability detected by such checkers can be classified as a false positive only by manual inspection. Increased occurrence of false positives and the effort involved in identifying false positives make security checkers less useful.

In our research, we are investigating safe approximations related to flow-sensitivity, context-sensitivity, path-sensitivity, heap modeling, aggregate modeling (structures and arrays), and recursive data structures, and unsafe approximations such as ignoring function pointers, pointer arithmetic, and signal handlers. The hypothesis is that if instances of safe and unsafe approximations present in alias analysis and in it's client analyses such as reaching definition analysis are tracked, measured, and combined using certain heuristics, then the security checkers built on top of these analyses can use these measurements to associate a confidence value (probability) with each report of a vulnerability.

The confidence in a vulnerability report, which signifies the likelihood of the vulnerability being actually present, will be inversely proportional to the extent of approximations involved in all the analyses that lead to the detection of that vulnerability. This is because, the more the approximations made while detecting a vulnerability, the greater the probability of that detection being a false positive. This technique can be used to rank the vulnerability reports based on their "false positiveness" which we believe will significantly impact the usefulness of security checkers. The analysis of vulnerability reports from security checkers can then be prioritized by first analyzing those reports that have a greater likelihood of being actual vulnerabilities and not false positives.

4. ROLE OF VULNERABILITY LIKELIHOOD IN SOFTWARE ASSURANCE METHODOLOGY

We have described the concept of vulnerability likelihood in the two contexts of vulnerability prediction in software artifacts and vulnerability detection using static analysis checkers. In this section, we describe how a methodology based on vulnerability likelihood can help prioritize and concentrate the resources in the testing phase to a subset of the modules in a software artifact.

Let us assume that the four types of program properties described in section 2 are empirically shown to correlate with the number of vulnerabilities in the modules of a software artifact. When a software artifact is ready to be tested, the different modules of the artifact can be run through a *vulnerability predictor* which checks for the four property types. Metrics can be generated for each property type. For example, in the case of PLOC, it would be the number of lines of code in the module that run with elevated privileges. And in the other three cases, it could be the number of instances of error-prone constructs, programming mistakes, and poor programming style. Instances occurring in privileged lines of code would be considered more severe than others.

These metrics may be weighted based on the extent of their correlations with vulnerabilities during prior empirical validations. They may also be normalized based on the size of the software module and the number of properties checked under each type. The multidimensional metrics of vulnerability likelihood associated with the modules can be used to rank them. Such a ranking methodology would help identify the more vulnerability-prone software modules from the others. This can then be used to prioritize the time and effort involved in the testing phase. A validation of this methodology would require the testing phase to actually discover more vulnerabilities in modules identified as more vulnerability-prone than in the other modules and without any prior knowledge of the vulnerability-proneness of modules.

We could further incorporate into this methodology, the technique we propose to quantify the likelihood of vulnerabilities detected by static security checkers. This technique can be used to associate a likelihood metric with each vulnerability detected by static checkers. Heuristics may be used to combine such metrics for all the vulnerabilities detected in a module. These metrics can be used along with the mul-

tidimensional metrics in ranking modules to determine the extent of further testing efforts based on test coverage and manual source code audits.

Such a metric-based and focussed testing methodology would lead to increased software assurance from the security viewpoint. Furthermore, metrics based on vulnerability likelihood would give a probabilistic measure of software assurance. And on improving the metrics by addressing the contributing factors, one could also demonstrate measurable gains in assurance.

5. OTHER RELATED WORK

Browne et al. [14] study statistical trends in reported intrusions incidents involving exploited vulnerabilities and derive a regression equation that represents the cumulative count of reported incidents as a function of the time since the start of the exploit cycle.

There are a number of studies that attempt to quantify the assurance of an operational system as opposed to quantifying static attributes in the design and development of a system [18, 30, 35, 42]. Such approaches model the intrusion process analogous to modeling failure in reliability studies. However, the difficulty here lies in modeling the intentionality present in the intrusion process.

DaCosta et al. [19] attempt to identify functions in a software program that have a greater vulnerability likelihood than the other functions in the program. Their hypothesis is that a small percentage of functions near a source of input (such as those containing `read`, `getlogin`, `getenv`, `scanf`, and `getc`) is most likely to contain vulnerabilities. They provide limited empirical validation (using only four software artifacts) of their hypothesis. This approach is interesting but has to be tested with more software artifacts before being used as one of the heuristics to classify the functions in a software program into "more vulnerable" and "less vulnerable" sets.

Kremenek and Engler [32] propose a statistical technique called *z-ranking* to rank error reports generated by static program analysis tools. They classify checks for program properties into successful checks (program locations that satisfy the checked property) and failed checks (program locations that violated the checked property thereby constituting error reports). The underlying hypothesis is that "true" error reports are generally few in number. So for a particular property, they compare the number of successful checks to the number of failed checks. If there are fewer failed checks (error reports) compared to successful checks, then there is a greater chance that those failed checks are true error reports and not false positives. They use this factor to rank error reports and their measurements indicate that this ranking technique is better than a randomized ranking 98.5% of the time. This technique determines the false positiveness of an error report by comparing it with the number of other error reports and successful checks, and not by analyzing the reasons behind the generation of the report. We anticipate our approach of ranking an error report based on the extent of approximation used in its generation to do much better than their relative ranking method.

6. CONCLUSIONS

The concept of vulnerability likelihood as an approach to probabilistically measuring and demonstrably improving software assurance is a new paradigm that deviates from the traditional binary approach to vulnerability detection. And to the best of our knowledge, this is also the first holistic approach to software vulnerability prediction.

We are currently identifying software artifacts and software development environments to empirically validate our hypothesis about software vulnerability prediction. And towards validating our hypothesis on the likelihood of vulnerabilities detected by static checkers, we have implemented prototype checkers for buffer overflow and format string vulnerabilities using the PAF framework [26] and it's associated algorithms as the basis for our implementation platform. We are working on tracking and measuring the approximations to associate a probabilistic metric with vulnerabilities detected.

7. REFERENCES

[1] Flawfinder. www.dwheeler.com/flawfinder.

[2] Rough Auditing Tool for Security (RATS). www.securesoftware.com.

[3] The Twenty Most Critical Internet Security Vulnerabilities. The SANS Institute, http://www.sans.org/top20.htm/.

[4] Workshop on Information-Security-System Rating and Ranking. Williamsburg, Virginia, May 2001.

[5] CRA Conference on Grand Research Challenges in Computer Science and Engineering, June 2002. http://www.cra.org/Activities/grand.challenges/.

[6] The Economic Impacts of Inadequate Infrastructure for Software Testing, May 2002. NIST Planning Report 02-3. Available from http://www.nist.gov/director/prog-ofc/report02-3.pdf.

[7] Challenge Problems in Software Security, June 2003. http://research.microsoft.com/projects/SWSecInstitute/challenge-metric.htm.

[8] CRA Conference on Grand Research Challenges in Information Security & Assurance, November 2003. http://www.cra.org/Activities/grand.challenges/security/home.html.

[9] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools.* Addison-Wesley Longman Publishing Co., Inc., 1986.

[10] Fumio Akiyama. An example of software system debugging. *IFIP Congress*, 1:353–379, 1971.

[11] Kent Beck. *Extreme programming explained: embrace change.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[12] Matt Bishop and Michael Dilger. Checking for race conditions in file accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[13] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *Proceedings of the 2nd international conference on Software engineering*, pages 592–605, 1976.

[14] H. K. Browne, W. A. Arbaugh, J. McHugh, and W. L. Fithen. A trend analysis of exploitations. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 214–229, May 2001.

[15] Hao Chen and David Wagner. MOPS: an infrastructure for examining security properties of software. In *Proceedings of the 9th ACM conference on Computer and communications security*, pages 235–244, 2002.

[16] B. T. Compton and C. Withrow. Prediction and control of ADA software defects. *Journal of Systems Software*, 12(3):199–207, 1990.

[17] Crispin Cowan, Matt Barringer, Steve Beattie, Greg Kroah-Hartman, Mike Frantzen, and Jamie Lokier. FormatGuard: Automatic protection from printf format string vulnerabilities. In *Proceedings of 10th USENIX Security Conference*, Washington DC, August 2001.

[18] M. Dacier, Y. Deswarte, and M. Kaniche. Quantitative assessment of operational security: Models and tools. Technical Report LAAS Report 96493, May 1996.

[19] Dan DaCosta, Christopher Dahn, Spiros Mancoridis, and Vassilis Prevelakis. Characterizing the 'security vulnerability likelihood' of software functions. In *Proceedings of the International Conference on Software Maintenance (ICSM)*, September 2003.

[20] Alan DeKok. PScan: A limited problem scanner for c source files. Available at http://www.striker.ottawa.on.ca/ aland/pscan.

[21] Michael Diaz and Joseph Sligo. How software process improvement helped motorola. *IEEE Software*, 14(5):75–81, 1997.

[22] Michael Dyer. *The cleanroom approach to quality software development.* John Wiley & Sons, Inc., 1992.

[23] David Evans, John Guttag, James Horning, and Yang Meng Tan. LCLint: a tool for using specifications to check code. In *Proceedings of the 2nd ACM SIGSOFT symposium on Foundations of software engineering*, pages 87–96, 1994.

[24] Norman Fenton, Paul Krause, and Martin Neil. A probabilistic model for software defect prediction, 2001. Submitted for Publication. Available from http://www.dcs.qmul.ac.uk/ norman/papers/fenton_krause_neil_IEEE.pdf.

[25] Norman E. Fenton and Martin Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999.

[26] Programming Languages Research Group. PROLANGS Analysis Framework (PAF). http://www.prolangs.rutgers.edu/.

[27] Watts S. Humphrey. *Introduction to the personal software process.* Addison-Wesley Longman Publishing Co., Inc., 1997.

[28] Watts S. Humphrey. *Introduction to the team software process.* Addison-Wesley Longman Ltd., 2000.

[29] IEEE.1990. ANSI/IEEE standard glossary of software engineering terminology. IEEE press.

[30] Erland Jonsson and Tomas Olovsson. A quantitative model of the security intrusion process based on attacker behavior. *IEEE Transactions on Software Engineering*, 23(4), April 1997.

[31] B.W. Kernighan and P.J. Plauger. *The Elements of Programming Style.* McGraw-Hill, New York, 1974.

[32] Ted Kremenek and Dawson Engler. Z-ranking: Using statistical analysis to counter the impact of static analysis approximations. In *Proceedings of the 10th Annual International Static Analysis Symposium*, June 2003.

[33] Ivan Victor Krsul. *Software Vulnerability Analysis.* Phd thesis, Purdue University, May 1998.

[34] William Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[35] B. Littlewood, S. Brocklehurst, N. Fenton, P. Mellor, S. Page, D. Wright, J. Dobson, J. McDermid, and D. Gollmann. Towards operational measures of computer security. *Computer Security*, 2:211–229, 1993.

[36] Yashwant K. Malaiya, Naixin Li, Jim Bieman, Rick Karcich, and Bob Skibbe. The relation between software test coverage and reliability. In *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, November 1994.

[37] The Motor Industry Software Reliability Association (MISRA). *Guidelines for the Use of the C Language in Vehicle Based Software.* 1998.

[38] E. Myers. A precise inter-procedural data flow algorithm. In *Proceedings of the 8th ACM symposium on Principles of programming languages (POPL)*, January 1981.

[39] P. Oman and C. Cook. A taxonomy of programming style. In *Proceedings of the Eighteenth Annual ACM Computer Science Conference*, pages 244–247, 1990.

[40] P. Oman and C. Cook. A programming style taxonomy. *Systems Software*, 15(4):287–301, 1991.

[41] Jarrett Rosenberg. Some misconceptions about lines of code. In *Proceedings of the 4th IEEE International Software Metrics Symposium (METRICS 1997)*, November 1997.

[42] Gregg Schudel and Bradley J. Wood. Adversary work factor as a metric for information assurance. In *Proceedings of the New Security Paradigms Workshop, Cork, Ireland*, September 2000.

[43] Conte S.D., Dunsmore H.E., and Shen V.Y. *Software Engineering Metrics and Models*. The Benjamin/Cummings Company, Inc., 1986.

[44] Umesh Shankar, Kunal Talwar, Jeffrey S. Foster, and David Wagner. Detecting format string vulnerabilities with type qualifiers. In *Proceedings of the 10th Usenix Security Symposium*, Washington, D.C., August 2001.

[45] B. A. Sheil. The psychological study of programming. *ACM Computing Surveys (CSUR)*, 13(1):101–120, 1981.

[46] John Viega, J.T. Bloch, Tadayoshi Kohno, and Gary McGraw. ITS4: A static vulnerability scanner for C and C++ code. December 2000.

[47] Yichen Xie and Dawson Engler. Using redundancies to find errors. *SIGSOFT Software Engineering Notes*, 27(6):51–60, 2002.