Improving Vulnerability Discovery Models

Problems with Definitions and Assumptions

Andy Ozment*
MIT Lincoln Laboratory & University of Cambridge
Cambridge, United Kingdom

ABSTRACT

Security researchers are applying software reliability models to vulnerability data, in an attempt to model the vulnerability discovery process. I show that most current work on these vulnerability discovery models (VDMs) is theoretically unsound. I propose a standard set of definitions relevant to measuring characteristics of vulnerabilities and their discovery process. I then describe the theoretical requirements of VDMs and highlight the shortcomings of existing work, particularly the assumption that vulnerability discovery is an independent process.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Metrics—*product metrics, security metrics*; G.3 [**Probability and Statistics**]: [reliability and life testing]

General Terms

Security, Reliability, Measurement

Keywords

security metrics, vulnerability discovery models, measuring software security, measuring vulnerabilities

*This work is sponsored by the I3P under Air Force Contract FA8721-05-0002. Opinions, interpretations, conclusions and recommendations are those of the author(s) and are not necessarily endorsed by the United States Government.

This work was produced under the auspices of the Institute for Information Infrastructure Protection (I3P) research program. The I3P is managed by Dartmouth College, and supported under Award number 2003-TK-TX-0003 from the U.S. Department of Homeland Security, Science and Technology Directorate. Points of view in this document are those of the authors and do not necessarily represent the official position of the U.S. Department of Homeland Security, the Science and Technology Directorate, the I3P, or Dartmouth College.

QoP'07, October 29, 2007, Alexandria, Virginia, USA.

1. INTRODUCTION

Vulnerability discovery models (VDMs) are probabilistic methods for modeling the discovery of software vulnerabilities in a system. They operate on historical system and vulnerability data: *e.g.* the system's release date, system usage information, and the date on which a vulnerability is discovered. The models can be used to estimate characteristics of the vulnerability discovery process for that system.

Existing VDMs are based upon previous work on software reliability models, a.k.a. software reliability growth models. "A **software reliability model (SRM)** specifies the general form of the dependence of the failure process on the principal factors that affect it: fault introduction, fault removal, and the operational environment" [16].

SRMs are based upon the assumption that the reliability of a program is a function of the number of faults that it contains. As faults are detected and removed, the system will fail less frequently and hence be more reliable. SRMs thus "apply statistical techniques to the observed failures during software testing and operation to forecast the product's reliability" [1].

In general, recent work has assumed that SRMs can fruitfully be applied as VDMs directly to vulnerability data garnered from public databases. Some researchers create a new statistical model, but they still apply the VDM to the public database vulnerability information exactly as if they were applying an SRM to internal test failure data.

Vulnerability discovery models may prove to be a useful tool for estimating and predicting vulnerability characteristics in software. However, the current literature almost universally ignores significant ambiguity and theoretical concerns. Before we can effectively discuss VDMs, we require a standardized terminology. Before we can apply VDMs, we need to understand their theoretical foundation and to detect the situations for which they are appropriate.

2. BASIC DEFINITIONS

The field of software engineering benefits from standardized terminology: *e.g.* the "IEEE Standard Glossary of Software Engineering Terminology" [13]. Unfortunately, computer security lacks a similar standard and the disparity of usage in the field is a source of confusion. One way to remedy this shortcoming in computer security is to utilize, whenever possible, the standard terminology of fields like software engineering.

2.1 Software Engineering Terms

In this work, I use standard software engineering terms when they are available; otherwise, I define my terms so that they are consistent with these standard definitions. The fundamental software engineering terms upon which I will rely are failure, fault, mistake, and error.

A **failure** is the "inability of a system or component to perform its required functions within specified performance requirements" [13]. A more intuitive description used in software reliability is that "a failure occurs when the user perceives that the program ceases to deliver the expected service" [16].

A **fault** is an "incorrect step, process, or data definition in a computer program" [13]. All failures are caused by faults, but not all faults lead to a failure. Faults are also known as 'bugs' or 'flaws.'

A **mistake** is "a human action that produces an incorrect result" [13]. An **error** is "the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition. For example, a difference of 30 meters between a computed result and the correct result" [13].

A nice summary of key software engineering terminology is that the field "distinguishes between a human action (a mistake), its manifestation (a hardware or software fault), the result of the fault (a failure), and the amount by which the result is incorrect (the error)" [13].

Rajeev Gopalakrishna, Eugene Spafford, & Jan Vitek assert that the goal of software security is to prevent "deliberate attempts to cause failure by triggering faults" [11]. Correspondingly, Sarah Brocklehurst & Bev Littlewood assert that a vulnerability is the security field's equivalent to a 'fault' [8].

2.2 Software vulnerability

The vulnerability discovery process cannot be modeled unless we know precisely what a vulnerability *is*. Unfortunately, we lack a widely accepted definition.

Among the many proposed definitions of 'vulnerability', there are:

1. "A weakness in a system that can be exploited to violate the system's intended behavior. There may be security, integrity, availability, and other vulnerabilities" [9].

2. An "internal fault that enables an external fault to harm the system" [7].

3. A"flaw or defect in a technology or its deployment that produces an exploitable weakness in a system, resulting in behavior that has security or survivability implications" [6, p. 54].

The first definition does not address whether or not similar instances of a security fault should be considered the same or different vulnerabilities. Its use of 'intended behavior' is vague.

The second definition uses fault in a nonstandard way: a person can be a fault. This usage seems to completely devalue the word 'fault.' Furthermore the definition relies upon the (undefined) meaning of 'harm.' In general, this definition is abstract, opaque, and counterintuitive to any reader not intimately familiar with the paper in which it was proposed.

The third definition is one of two that William Arbaugh *et al.* propose in the same work. This definition is excellent, but too broad: it includes hardware vulnerabilities, while in this work I am concerned only with *software* vulnerabilities.

The definition that I prefer and that I shall use here was proposed by Ivan Krsul: a **software vulnerability** is "an instance of [a mistake] in the specification, development, or configuration of software such that its execution can violate the [explicit or implicit] security policy" [15].

I have made two important changes to Krsul's definition. First, he originally used 'error' where I have written 'a mistake.' His use of 'error' is counter to the definition used in software engineering, where it describes "the amount by which the result is incorrect" [13]. I have also added 'explicit or implicit' to highlight the fact that all systems have a security policy, even if the designers have not formally written it down.

I prefer Krsul's definition because it highlights different areas in which a software vulnerability can originate: specification, development, or configuration. It emphasizes the security policy rather than the security system.

Finally, it notes that a vulnerability is a single instance of a mistake, so there is no confusion about whether or not different instances of the same mistake constitute different vulnerabilities. The literature has sometimes used vulnerability to refer to a single instance of a mistake and other times to refer to all instances of the same mistake. If a developer writes code with an integer overflow and then copies that code in another area of the system, is that one vulnerability or two? With the definition above, it is two vulnerabilities.

According to this definition, a vulnerability that results from a development mistake is a fault. However, vulnerabilities that result from design or configuration mistakes are not faults. Not all vulnerabilities are faults, and not all faults are vulnerabilities.

Software vulnerabilities are a subset of vulnerabilities: the term 'vulnerability' can encompass susceptibility to hardware manipulation, social engineering, *etc.* However, in this paper I will use 'vulnerability' as shorthand for 'software vulnerability.'

2.3 Actors

The vulnerability discovery process necessarily includes actors, whether they are people, groups, or companies.

A **detector** finds instances of vulnerabilities in software systems. In previous work, I used the term 'vulnerability hunter' instead. However, detector better encompasses the situation in which a vulnerability is unintentionally discovered. A **vendor** is any producer of software, regardless of whether or not that software is sold commercially.

A **vendor detector** is an employee of a vendor whose job responsibilities include searching for vulnerabilities. An **external detector** searches for vulnerabilities in systems whose vendors do not directly employ him.

In the context of vulnerabilities, **public fora** are the means by which vulnerability information is widely disseminated. Examples of public fora are: the Bugtraq mailing list, the Full Disclosure mailing list, and US/CERT announcements.

A **disclosure institution** is any benign organization that receives vulnerability reports and forwards them to the appropriate vendor(s). The term includes CERT/CC and white vulnerability markets like TippingPoint and iDefense.

The adjectives 'benign' and 'malicious' lie at the heart of

many distinctions made in the vulnerability detection process. There is and will continue to be ambiguity in the definitions of these terms, because the different actors in the vulnerability life cycle have different goals and philosophies. Here, I use 'benign' to indicate an actor that follows a **responsible disclosure** policy: vendors are informed about a vulnerability and given time to create a patch before the detector makes the vulnerability public.

3. VULNERABILITY LIFE CYCLE

Vulnerabilities can be characterized according to a hypothetical life cycle, in which certain events are important to vulnerability discovery modeling.[1]

3.1 Events

- **Injection Date**: the date on which the vulnerable code is first checked into the developer's source code repository. If a repository is not in use, it is the first date on which the vulnerable code is added to the build or compiled.
- **Release Date**: the date of public release for the system that first contains the vulnerability.
- **Discovery Date**: the date on which the vulnerability is first detected.
- **Disclosure Date**: the date on which the detector first notifies the vendor or a disclosure institution.
- **Public Date**: the date on which the existence of the vulnerability is made publicly known (*e.g.* via a public fora or the release of a patch). The public date is often the same as the patch date.
- **Patch Date**: the date on which the first correction for the vulnerability is released, regardless of whether the correction is official or correct. A workaround does not constitute a correction: the correction must actually remove the vulnerability from the software system.
- **Scripting Date**: the date on which the first automated exploit for the vulnerability is released.

The events in this life cycle can occur in many different orders. For example, a vulnerability may be detected during testing and fixed prior to the product's release: in that case, it is discovered before it is born.

In practice, not all of the above dates will be known for each vulnerability. The **date known** is the earliest confirmed date on which someone is aware of the vulnerability's existence. Ideally, this is the discovery date; in practice, it may be the disclosure date or the public date.

[1]Some of these events are taken from Arbaugh *et al.* [6], although I define them differently: discovery, disclosure, publication, and scripting. Arbaugh *et al.* use 'birth date' instead of 'release date.' I believe the latter is more clear, because the former could also apply to the injection date. They also use 'correction date' instead of 'patch date.' Again, I find the latter term more clear, because the former could be misinterpreted as the date on which a correction is completed internally at the vendor. Their definition of disclosure includes acts like posting the vulnerability to the Bugtraq mailing list: I believe that posting to fora like Bugtraq results in making the vulnerability public.

3.2 Status

A vulnerability's status depends on which of the events in the life cycle have occurred:

- **Unknown Vulnerability**: exists in the software but has not yet been detected.
- **Secret Vulnerability**: has been detected, but the detector has not informed the vendor, the public, or a disclosure institution. If the detector is malicious, she may be exploiting the vulnerability.
- **Disclosed Vulnerability**: has been discovered, and the detector has disclosed it to the vendor or a disclosure institution.
- **Public Vulnerability**: has been detected and made public through either a patch, a public forum, or the media.
- **Scripted Vulnerability**: one for which automated exploits exist.

4. VULNERABILITY DISCOVERY MODELS

Although SRMs have been in use for almost three decades, the security field has only begun to apply these models to vulnerability data in the past few years. As a result, this subfield lacks a standard terminology. Omar Alhazmi & Yashwant Malaiya proposed the application of SRMs to vulnerabilities as 'vulnerability discovery models' (VDMs) [2]. I previously proposed the term 'software security growth models,' acknowledging the relationship between this technique and the term 'software reliability growth model' (a synonym for SRMs) [19].

Both proposed names have shortcomings. The term 'vulnerability discovery model' hides the fact that these models have so far only been applied to vulnerability *reporting* data, rather than *discovery* data. The term 'software security growth model' ignores the fact that the rate of vulnerability reporting may increase, and thus software security may not always *grow*. It also suffers from the need to define 'software security.' An alternative, but less precise, approach is used by Gopalakrishna *et al.*: they consider the 'vulnerability likelihood' of a system, but their term encompasses a broader range of probabilistic methods than VDMs [11].

The term 'vulnerability discovery model' appears to have the most widespread traction in the literature, and I will use it here. However, it does not yet appear to have been formally defined, so I propose a definition based upon Lyu's definition of SRMs [16]. A **vulnerability discovery model (VDM)** specifies the general form of the dependence of the vulnerability discovery process on the principal factors that affect it: *e.g.* vulnerability introduction, vulnerability removal, detector effort, and the operational environment.

Among the outputs of VDMs are two particularly useful estimates. First, the estimate of the total number of vulnerabilities. Second, the **mean time to next vulnerability (MTTNV)**: the mean time until another vulnerability is detected in the software system [10]. The MTTNV is analogous to the software engineering term 'mean time between failures.'[2]

[2]Gopalakrishna & Spafford state that a breach or an intrusion can be considered the security field's equivalent to the

4.1 The benefits of using VDMs

VDMs may provide useful quantitative insight to supplement the current approaches to assessing software security. In particular, VDMs can be used for both prediction and comparison. Some of the possible uses of VDMs are:

1. Helping vendors and users to allocate and schedule their resources.
2. Estimating the time to achieve an assurance goal.
3. Quantifying the impact of design and implementation methodologies.
4. Comparing similar software systems.

The estimated total number of vulnerabilities and the MTTNV can be used by vendors to allocate and schedule developer resources. The estimate of the total number of vulnerabilities can provide insight into the total resources and time necessary for the maintenance of a system. The MTTNV enables vendors to schedule developer time for the creation of patches, quality-assurance time for the testing of those patches, and possibly a regular schedule for their release (*e.g.* Microsoft releases security patches on the second Tuesday of every month).

The MTTNV is also useful to users: system administrators can use it to estimate how frequently they will have to test and apply security patches for a particular system.

Finally, vendors can use VDMs to estimate the time necessary to achieve an assurance goal. For example, this goal might be in terms of the number of remaining undetected vulnerabilities or in terms of a desired MTTNV. Vendors can thus create a release schedule based upon the expected date on which this goal will be achieved.

Both vendors and customers want to quantify and compare software security. Vendors are interested in comparing similar projects to see whether their internal development processes are improving and to predict whether the projects will have similar behavior in the field. Customers may want to identify which of two software systems has fewer remaining vulnerabilities or a lower rate of vulnerability detection.

In previous work, I proposed the use of VDMs as a means for customers to compare the security of two different systems [19]. This proposal was premature. While it is worthy of investigation, we need a great deal more experience with VDMs before we can be confident in their efficacy for cross-project comparison—much less comparison across both vendors and projects. In particular, the assumptions upon which VDMs rely may prevent effective comparisons of this sort. These assumptions are discussed in Section 6, but first I consider the literature on VDMs.

5. VDM LITERATURE

The existing literature on VDMs can be divided according to the four research groups most active in this area. I will introduce the literature here; in later sections, I will note the shortcomings in this literature.

reliability field's 'failure' [10]. That assertion is logical from the standpoint of a system's user: the system has failed to provide the expected service. However, I am more interested in the standpoint of the developer. From that perspective, a better analogy compares a failure to a vulnerability detection event, defined below in Section 6.3. The developer considers a failure to be the event that enables him to find and correct a vulnerability—and intrusions may not result in the detection of a new vulnerability.

5.1 Rescorla

Eric Rescorla was unable to fit a linear model or an exponential Goel-Okumoto SRM to NVD[3] vulnerability data on three operating systems: WinNT4, Solaris 2.5.1, and FreeBSD 4.0. He was able to fit both models to data for RedHat 6.2 data. However, he does not test either model's predictive accuracy against the Redhat 6.2 data, so neither can be considered proven [22].

5.2 Purdue University

Gopalakrishna & Spafford consider NVD data on vulnerabilities in IIS, BIND, Lpd, Sendmail, and RPC. They chose these five software systems because they have a focused purpose (unlike operating systems), have been deployed at least two years, are widely deployed, and have significant numbers of serious vulnerabilities. They do not apply specific models to the data: rather, they discuss the theoretical and practical requirements for doing so [10]. Gopalakrishna *et al.* later assess these requirements for a number of different approaches to measuring vulnerabilities and vulnerability discovery [11].

5.3 Colorado State University

Alhazmi & Malaiya propose two models specifically for vulnerability discovery: an S-shaped, time-based logistic model of vulnerability discovery (AML) and an effort-based VDM, which approximates effort with the number of users of a system (AME) [3]. They compare the goodness-of-fit of a number of different models to NVD vulnerability data on Win95, WinXP, and RedHat 6.2. They find that their AML model provides the best overall fit [2]. However, goodness-of-fit is only a prerequisite for testing models: predictive accuracy is the most important criterion. The authors do test the predictive accuracy of their AML model on Win2000, WinXP, and RedHat 7.1 NVD data; they are satisfied with the result, particularly when the model is constrained with vulnerability density information obtained from previous versions of the software being examined (the constrained model is called AML-C) [5].

Sung-Whan Woo, Alhazmi, & Malaiya test the goodness-of-fit of the AML and AME models against NVD vulnerability data on Apache and IIS; they also categorize the vulnerabilities by type and apply the models to each category of vulnerability [24]. In a later work, Alhazmi & Malaiya test the predictive accuracy of the AML, AML-C and a linear model against NVD Apache and IIS data [4] Finally, Woo *et al.* assesses the goodness-of-fit of the AML model on NVD vulnerability data, overall and categorized by type, from IE and Firefox [23].

5.4 MIT Lincoln Laboratory

In prior work, I create and analyze a data set of individually examined vulnerabilities for the OpenBSD operating system [18, 19]. I test the fit and predictive accuracy of more than a dozen SRMs against the data. Stuart E. Schechter and I assess the complete data set and conclude that the rate of vulnerability discovery in OpenBSD is declining—for vulnerabilities introduced prior to a cutoff date [21].

[3]The National Vulnerability Database (NVD) contains information on all vulnerabilities with Common Vulnerabilities and Exposures (CVE) identifiers. The NVD was formerly known as ICAT.

6. VDM ASSUMPTIONS

Because VDMs are probabilistic tools, their usage is based on assumptions about the data to which they are applied. These assumptions are often the same as those made by SRMs, so researchers have assumed that they are also satisfied for VDMs. Unfortunately, most of the existing work has failed to satisfy all of the necessary assumptions. As a result, the validity of this work is uncertain. VDMs face particular challenges in satisfying four assumptions: time, operational environment, independence, and static code.

6.1 Time and effort

The accuracy of both SRMs and VDMs reflects the accuracy of the data to which they are applied. For example, software engineers usually prefer execution time to calendar time for use with SRMs. However, obtaining accurate chronological data for VDMs is made difficult by the nature of the detection process: many detectors are external (not employed by the vendor). As a result, it is difficult to accurately quantify the effort expended by detectors, their number, and their knowledge.

SRM best practices state that calendar time should be normalized for the number of individuals testing the software system [16]. For example, Jack and Jill work together to find a vulnerability in one week of searching. If calendar time is normalized for the number of individuals, then this vulnerability was detected after two weeks of work. Two detectors working for one week is equivalent to one detector working for two weeks.

Reality may be even more complicated: Jack, a novice detector, and Jill, a highly skilled detector, worked varying hours, part time, to discovery a vulnerability.

The **effort** expended to discover a vulnerability is composed of the number of detectors, their skill, and the number of hours they worked. Ideally, vulnerability databases would include effort information with which to normalize the chronological information.

Unfortunately, I am aware of no vulnerability database that includes such accurate information about the effort expended to find each vulnerability. Alhazmi & Malaiya propose an effort-based model and argue that system usage figures should be used as a proxy for effort [3]. However, there is no evidence that usage data is a suitable proxy for the effort expended by detectors. The fraction of users who are looking for vulnerabilities is not necessarily a fixed proportion of the total user population. Furthermore, vulnerability detectors may choose to examine software that is popular in their community or currently prominent in the media, so the ratio of detectors to users may differ between programs and also across time.

This problem is unlikely to be solved. Vulnerabilities are often reported by external detectors, so gathering detailed and accurate information on the effort they expend is probably not possible. The best that VDMs can therefore achieve is to model the discovery process given the vulnerability detection environment that existed during the time modeled. If that environment changes, then we cannot rely upon the model.

6.2 Operational environment

In order to effectively represent post-release reliability, SRMs require that the environment from which the data are obtained (usually the testing environment) must be equivalent to the environment in which the software will be utilized after deployment. The **operational profile**: is "the set of run types that the program can execute along with the probabilities with which they will occur" [17].

However, many vulnerabilities rely upon the adversary intentionally inputting abnormal data—data outside the bounds of a normal operational profile. This characteristic of vulnerabilities highlights one of the shortcomings of operational profiles noted by Barbara Kitchenham & Steve Linkman: faults may be most prevalent in infrequent operations. First, because developers have often thought less about infrequent operations than frequent operations. A rare transition state may not be as well understood as the regular operating state. Second, because operational profiles are self-obsolescent in terms of testing efficacy. If you invest a lot of testing effort in the most frequently used operations, most of the faults in those operations will be found. The majority of the faults that remain are likely to be in the least used—and thus least tested—portions of the code [14].

As a result, VDMs applied to pre-release test data may not be indicative of post-release vulnerability discovery. The operational profile used in pre-release testing may not anticipate the testing approaches used by external detectors.

6.3 Independence

Sometimes the detector community discovers an entirely new class of vulnerabilities, *e.g.* integer overflows. This discovery may lead to chronologically local spikes in the number of vulnerabilities found, and it is conceptually equivalent to an expansion of the operational profile. It also violates an additional assumption that underlies all of the existing stochastic VDMs: that vulnerability discoveries occur independently.

If vulnerability data is dependent, it may also be because developers may repeat the same mistake in different places in the source code. For example, a developer may always perform a copy of size LENGTH into a buffer when that buffer is actually one byte smaller than LENGTH. Detectors are thus often rewarded for looking for other or similar instances of the same type of vulnerability. Gopalakrishna & Spafford apply a run test to determine independence on data for vulnerabilities in IIS, BIND, Lpd, Sendmail, and RPC. They found mixed results: for each different product, vulnerability discovery for some types of vulnerabilities is dependent; in others it is independent [10]. This finding indicates that any modeling effort should first test to see whether the vulnerability data is independent. If it is not, then VDMs are not an appropriate tool.

It may instead be possible to apply VDMs to **detection events**: an independent act of detection that may result in the discovery of multiple instances of dependent vulnerabilities. The previous paragraph describes a hypothetical program in which a copy of size LENGTH is always and erroneously performed. In this example, each time this copy is performed in the source code it constitutes a vulnerability, but the discovery of these multiple vulnerabilities can be lumped into a single detection event.

The VDM literature has inadequately considered this problem. Rescorla [22] states his assumption of independence, while Ozment & Schechter [18, 19, 21] try to construct a data set of detection events. Although those works acknowledge the problem of independence, they do not test for it. In more recent work, I have tested my data set and found

that vulnerability discovery events of some types of vulnerabilities are dependent [20]. The Colorado State University literature does not consider that the NVD data points they use may not be independent.

This problem is not limited to the VDM literature. Katerina Goševa-Popstojanova & Kishore Trivedi note that many SRMs are applied to fault data that may not be independent, and they argue that models are needed that do not rely upon this assumption [12].

6.4 Static code

Most SRMs and VDMs assume a static code base. However, software rarely remains static for long: patches are applied to fix faults, remediate vulnerabilities, and add features. If the data to which the VDM is applied doesn't contain adequate release or patch information, then the model can be confounded by changes to the code base.

For example, Woo *et al.* initially lump together the vulnerabilities released in IIS 4 & 5 into one data set and vulnerabilities released in Apache 1 & 2 into another [24]. This work thus fails to consider whether vulnerabilities are being introduced into the software even as they are being removed. In a later work, some of the same authors divide those data sets by version, into IIS 4, IIS 5, Apache 1, and Apache 2 [4]. Again, however, this granularity is insufficient. IIS has had patches and service patches added to it. 'Dot' revisions to Apache, such as from 2.0 to 2.1, are equivalent to integer revisions to IIS: they introduce significant new features and changes. Both of these works thus apply VDMs to a changing code base.

The remainder of the literature is mixed with respect to this assumption. Rescorla considers operating systems by version but ignores patches and service patches [22]. Most of the other work from Colorado State University does the same [2, 3, 5]. However, my previous work does take the changing code base of modern systems into consideration: I examine each vulnerability in the data set to ascertain exactly when it was introduced [18, 19, 21].

7. CONCLUSION

VDMs hold promise for providing useful information to vendors, users, and customers. However, this research is unlikely to provide robust results unless the shortcomings described above are remedied.

Future work with VDMs should test the data to ascertain whether or not it is dependent: existing VDMs and SRMs are inappropriate for dependent data. Furthermore, most VDMs cannot reliably be applied to data that mixes vulnerabilities from different releases: only vulnerabilities from a specific release of a program should analyzed with these models. Such work should also indicate the accuracy of the dates used and whether or not any adjustments were made for the effort/attention of detectors.

Most importantly, researchers should clearly state the assumptions upon which their models rely and define the terms that they use.

8. ACKNOWLEDGMENTS

I am indebted to Shari Lawrence Pfleeger for many useful discussions on the definitions in this work. My thanks as well to Robert Cunningham, for his feedback and advice.

9. REFERENCES

[1] AIAA/ANSI. *Recommended Practice: Software Reliability*. ANSI, 1993. R-013-1992.

[2] O. H. Alhazmi and Y. K. Malaiya. Modeling the vulnerability discovery process. In *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)*, pages 129–138, Washington, DC, USA, 2005. IEEE Computer Society.

[3] O. H. Alhazmi and Y. K. Malaiya. Quantitative vulnerability assessment of systems software. In *Proceedings of the IEEE Reliability and Maintainability Symposium (RAMS'05)*, pages 615–620, Alexandria, VA, USA, 2005.

[4] O. H. Alhazmi and Y. K. Malaiya. Measuring and enhancing prediction capabilities of vulnerability discovery models for Apache and IIS HTTP servers. In *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE'06)*, pages 343–352, Washington, DC, USA, 2006. IEEE Computer Society.

[5] O. H. Alhazmi and Y. K. Malaiya. Prediction capabilities of vulnerability discovery models. In *Proceedings of the IEEE Reliability and Maintainability Symposium (RAMS '06)*, pages 86–91, Jan. 2006.

[6] W. A. Arbaugh, W. L. Fithen, and J. McHugh. Windows of vulnerability: A case study analysis. *IEEE Computer*, 33(12):52–59, Dec. 2000.

[7] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions On Dependable And Secure Computing*, 1(1):11–33, Jan-Mar 2004.

[8] S. Brocklehurst, B. Littlewood, T. Olovsson, and E. Jonsson. On measurement of operational security. Technical Report 160, Predictably Dependable Computing Systems, Apr. 1994.

[9] Computer Science and Telecommunications Board. *Computers at Risk: Safe Computing In the Information Age*. National Academy Press, Washington, DC, 2001.

[10] R. Gopalakrishna and E. H. Spafford. A trend analysis of vulnerabilities. Technical Report 2005-05, CERIAS, Purdue University, May 2005.

[11] R. Gopalakrishna, E. H. Spafford, and J. Vitek. Vulnerability likelihood: A probabilistic approach to software assurance. Technical Report 2005-06, CERIAS, Purdue University, 2005. 2005-06.

[12] K. Gošva-Popstonjanova and K. S. Trivedi. Failure correlation in software reliability models. Technical Report 00/04, Center for Advanced Computing and Communication (CACC), 2000.

[13] IEEE. IEEE standard glossary of software engineering terminology, Sept. 1990.

[14] B. Kitchenham and S. Linkman. Validation, verification, and testing: Diversity rules. *IEEE Software*, 15(4):46–49, 1998.

[15] I. V. Krsul. *Software Vulnerability Analysis*. PhD thesis, Purdue University, May 1998.

[16] M. R. Lyu. Introduction. In M. R. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 1, pages 3–22. McGraw-Hill, 1996.

[17] J. D. Musa, A. Iannino, and K. Okumoto. *Software Reliability: Measurement, Prediction, Application*. McGraw-Hill Book Company, New York, 1987.

[18] A. Ozment. The likelihood of vulnerability rediscovery and the social utility of vulnerability hunting. In *Workshop on the Economics of Information Security (WEIS)*, June 2005. Cambridge, MA, USA.

[19] A. Ozment. Software security growth modeling: Examining vulnerabilities with reliability growth models. In D. Gollmann, F. Massacci, and A. Yautsiukhin, editors, *Quality Of Protection: Security Measurements and Metrics*, Milan, Italy, 2006. Springer.

[20] A. Ozment. *Vulnerability Discovery & Software Security*. PhD thesis, University of Cambridge, 2007.

[21] A. Ozment and S. E. Schechter. Milk or wine: Does software security improve with age? In *Proceedings of the 15th Usenix Security Symposium*, pages 93–104, 2006.

[22] E. Rescorla. Is finding security holes a good idea? *IEEE Security & Privacy*, 3(1):14–19, Jan-Feb 2005.

[23] S.-W. Woo, O. H. Alhazmi, and Y. K. Malaiya. An analysis of the vulnerability discovery process in web browsers, Nov. 2006.

[24] S.-W. Woo, O. H. Alhazmi, and Y. K. Malaiya. Assessing vulnerabilities in Apache and IIS HTTP servers. In *IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pages 103–110, Sept. 2006.