Vulnerability Discovery & Software Security

Andy Ozment

University of Cambridge
Computer Laboratory
Computer Security Group
&
Magdalene College

August 31, 2007

This dissertation is submitted for
the degree of Doctor of Philosophy

Declaration

This dissertation is the result of my own work and includes nothing which is the outcome of work done in collaboration except where specifically indicated in the text.

This dissertation does not exceed the regulation length of 60,000 words, including tables and footnotes, but excluding the bibliography and appendix.

To my parents, Buck and Susan Ozment

Acknowledgements

My work has been supported at various times by a Marshall Scholarship; Magdalene College, Cambridge; the Cambridge Trust; a UK Overseas Research Scholarship; the Institute for Information Infrastructure Protection (The I3P); and MIT Lincoln Laboratory. I am thankful for this support and for the wonderful individuals at these institutions who work tirelessly to promote good research.

I gratefuly acknowledge the support of my supervisor, Ross Anderson, in this endeavor. His insight and provocative questions have helped me to become a better researcher and a better computer scientist.

Stuart E. Schechter has been a mentor and a friend. He has taught me the value of collaboration, and I have been lucky enough to work directly with him. His strong sense of academic ethics and his research 'manners' will continue to guide me throughout my career.

Robert Cunningham has been a stalwart friend and supporter. His suggestions have often caused me to swear—because they usually meant I'd missed something important!

Shari Lawrence Pfleeger has encouraged me and challenged me at just the right times. The quality of this dissertation has been significantly improved by her questions concerning my definitions and her suggestions about important literature.

Richard P. Lippman has provided patient tutoring in statistics. All of my statistics that are right are due to him; all that are wrong are, unfortunately, my own fault.

I am extraordinarily fortunate to have the support, humor, adventurousness, and eccentricity of Ragnhild Handagard upon which to rely. My sanity at the end of this process is due entirely to her. (Any insanity is only partially her fault.)

Finally, this work could not have happened without the loving support of my parents, Buck and Susan Ozment. I am incredibly lucky to have been raised by such smart, hard-working, and caring individuals: all that I have accomplished in my life is due to their efforts. Their example continues to inspire and inform my actions.

Vulnerability Discovery & Software Security

Andy Ozment

Summary

An effective means of measuring software security—and the likelihood of vulnerability discovery—would be a significant aid in increasing that security. One proposed technique for better understanding software security is to model vulnerability discovery. I examine the existing work on vulnerability discovery models (VDMs) and find a number of shortcomings. First, many terms are undefined or lack widely accepted definitions: as a result, the models are not always applied as intended. Second, the models' assumptions are not always stated, understood, or fulfilled: particularly the assumption that vulnerability discovery is an independent process. Third, the models are usually applied to unsuitable and inaccurate data—caused in part by the failure to definite the term 'vulnerability.'

I create a data set of eight years of vulnerabilities in the OpenBSD operating system; for each vulnerability, I ascertain the exact date on which it was first injected into the source code and, as accurately as possible, the date on which it was detected. I combine clearly dependent vulnerability discoveries into a single 'vulnerability detection event,' and I categorize the data set using two different taxonomies. Using the categorized data, I test the independence of the vulnerability detection process: for this data set, some types of vulnerability discoveries are dependent.

Because this data set contains dependent events, I do not apply a vulnerability discovery model. Instead, I look at other approaches of analyzing the data. I analyze the evolution of the source code in OpenBSD: the degree to which the source code changes in each release. I then examine, for each version, the density of vulnerability detection events per unit size of code added/altered.

I also examine whether or not the rate of vulnerability detection is decreasing over time for OpenBSD. I find evidence that it is decreasing. I then consider a question that underlies the debate on vulnerability disclosure policies: are individuals working independently likely to discover the same vulnerability? I find strong anecdotal evidence that this independent rediscovery occurs.

Finally, if software engineering approaches to measuring software security prove inadequate, an economic approach may be more suitable. I consider an auction-based method to measure the difficulty of discovering a vulnerability in a system.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Most software contains design and implementation vulnerabilities. The ubiquity of such vulnerabilities can be traced to two primary problems: complexity and motivation. Software developers push to create ever more complex products and work constantly on the boundary of manageable complexity. However, even taking this difficulty into account, most software contains vulnerabilities that its creators were readily capable of preventing. The second cause of software insecurity is a lack of motivation: although vendors are capable of creating more secure software, the economics of the software industry provide them with little incentive. Both of these problems are caused, in part, by our inability to measure software security. **Software security** is the ability of a system to perform its required functions without software-caused violations of its explicit or implicit security policy. More casually, software security is the degree to which software is free of vulnerabilities.

1.1 The need for software security metrics

Although some software security metrics exist, they are inadequate. For example, estimates of software size or complexity could give an indication of the probable number of vulnerabilities in a product, but software complexity is itself difficult to measure. The number of vulnerabilities patched during a given time period is quantifiable and readily observable; however, this number depends in part on the level of usage for the software and is also apparent only after the software has been in use for a period of time. Finally, one could assess the amount of effort put into security when designing and implementing the software, but such effort does not always translate to results.

The software engineering importance of security metrics is intuitive: we cannot consistently improve what we cannot measure. Software engineers are faced with a steady stream of development methodologies and tools that promise to help developers improve the security and reliability of their products. Without a metric for security, there is no good way to choose between these competing tools and methodologies. Product managers also need metrics to tell them when a product has reached an acceptable level of quality.

The economic motivations for a security metric are perhaps less obvious. Consumers generally reward vendors for adding features and for being first to market. These two motivations are in direct tension with the goal of writing more secure software, which requires time consuming testing and a focus on simplicity. Nonetheless, the problems of software insecurity, viruses, and worms are frequently in the headlines; why does the potential damage to vendors' reputations not motivate them to invest in more secure software?

Vendors' lack of motivation is readily explained: the software market is a 'market for lemons' [And01]. In a Nobel prize-winning work, economist George Akerlof employed the used-car market as a metaphor for a market with asymmetric information [Ake70]. Imagine a city in which 20 good used cars and 20 bad used cars (a.k.a. 'lemons') are for sale. The good cars are worth $2,000 and the lemons are worth $1,000. The sellers know whether or not they are offering good cars, but the buyers cannot tell which cars are good and which cars are bad. The market-clearing price is perhaps surprising: buyers are unwilling to pay more than $1,000, because they could be buying a lemon. However, at that price, no owner of a good car will sell it; as a result, only lemons are offered for sale.

In short, buyers cannot ascertain the quality of the used cars on the market, so they are unwilling to pay a premium to obtain an (ostensibly) higher quality car. Owners of high quality cars thus become unwilling to sell them, because they cannot obtain a reasonable premium.

The software market suffers from the same asymmetry of information. Even if vendors have some intuition as to the security of their products, buyers have no reason to trust the vendors' assertions. As a result, buyers have no reason to pay the premium required to obtain more secure software, and vendors are disinclined to invest in securing their products.

An effective means of measuring software security could decrease the asymmetry of information and ameliorate the 'market for lemons' effect.

1.2 Software security and risk

Software security is only a subset of information security, which also includes hardware security, environmental security, *etc.* A vulnerability in a single software system thus may or may not affect the overall risk faced by an organization: that vulnerable system may be protected via other controls, such as a firewall. Nonetheless, software security is a critical component of both an organization's security and how it measures that security.

If systems can be reached from the internet, then vulnerabilities in those systems present a real risk: the vulnerability is almost always accompanied by a threat. Automated exploits—and worms—are now often created for vulnerabilities in widely used software systems within days of that vulnerability becoming public. For example, the Zotob worm was released five days after a vulnerability became public in Microsoft Windows [Tre05]. Systems that become infected by the worms must be repaired, even if the vulnerable system does not protect mission-critical data. Moreover, an organization cannot predict which vulnerabilities will be incorporated into worms: as a result, it must treat vulnerabilities in all internet-reachable systems as a potential risk.

Moreover, no matter how an organization chooses to model its risk, the security of individual software systems is a component in that model. Until we can better measure software security, the values we assign to those components of the overall model will be little more than guesses. The ability to measure software security is thus an important part of the ability to measure organizational risk.

Unfortunately, the current 'measures' of software security are a consideration of the process by which the product was made, a superficial security review of the product, or a gross consideration of its vulnerability history. In addition to being imprecise, none of these techniques are consistent, reliable, or particularly useful in cross-product comparison.

1.3 Understanding vulnerabilities

In this work, I do not propose a silver-bullet security metric nor do I propose a system of measurement. Our understanding of software security and vulnerabilities has not progressed to the point where we can construct measurements that satisfy measurement theory. Instead, in this work I propose and evaluate two imperfect yet nonetheless useful approaches to better understanding vulnerabilities and software security: one that is 'engineering' in nature and another that is 'economic' in nature. Both approaches focus on vulnerabilities and the process by which they are discovered.

The first approach uses information that might be characterized as 'engineering' data: when was a vulnerability introduced, when was it discovered, how is the source code of a system changing, *etc.* This approach employs statistical analysis of vulnerabilities that have already been discovered and characteristics of the systems in which they were discovered.

The second approach uses 'economic' data: what is the auction-ascertained price of a previously-unreported vulnerability in a specific system. Entities offer a steadily increasing reward for a vulnerability in a system: the first person to report such a vulnerability receives the reward. The reward serves as both an incentive to find a vulnerability and a measurement of the perceived value of that vulnerability.

Both of these approaches provide insight into the number of vulnerabilities in a system, the rate at which they are detected, and the difficulty in doing so. Neither approach can provide perfect information on how many vulnerabilities exist in a system: that goal may well be impossible. The estimates provided by both approaches can be rapidly made obsolete by the discovery of a new type of vulnerability or a new tool for detecting vulnerabilities.

1.3.1 Users of these techniques

The engineering approach is probably most useful to software vendors. It will enable them to compare products, teams, and development methodologies. Most importantly, it will enable vendors to better allocate and schedule resources: to anticipate how frequently they will need to patch vulnerabilities and to plan accordingly.

Customers and vendors will both value the information provided by the economic approach. It will provide them with a monetary value that approximates the cost of discovering a vulnerability in a system. Customers can use the information provided by this approach to compare competing systems: the system in which vulnerabilities cost the most is more secure. For vendors, this cost is another means of comparing products, teams, and development methodologies. The comparison is made simple by the fact that the unit of measurement is monetary.

The two approaches are complementary. The economic approach provides more readily understandable and comparable information: the price of a vulnerability in a system. It is also self-correcting with respect to the number of detectors investigating a product. If the price has risen to a high level for a system, more detectors will be motivated to test that system. A price that remains high is thus a reliable indicator. The engineering approach is more susceptible to errors caused by variability in the number of detectors investigating a system. On the other hand, only the engineering approach provides vendors with information that assists in planning and resource allocation. Moreover, the engineering approach does not require the cultural shift necessary for the implementation of the economic approach: many vendors are currently leery of offering money for vulnerabilities.

1.3.2 Applicable systems

The engineering approach utilizes an analysis of vulnerabilities that have been reported in a system, so it can only be performed on systems in which enough vulnerabilities have been reported. Currently, this requirement is met for systems for which:

1. Security is an important characteristic of the system and it goes through a rigorous testing process before it is deployed. In this case, the engineering approach would be applied internally, to internally collected vulnerability information.
2. The system is publicly available and of interest to vulnerability detectors. In this case, the engineering approach could be applied internally or externally, using public vulnerability databases.

This dissertation has focused on the second type of system. Examples of that type include operating systems (MS Windows, Gnu/Linux, *BSD, OS X); database management systems (Oracle); web browsers (Firefox, IE, Safari); web servers (Apache, IIS); and other popular servers (sendmail, lpd, BIND).

The economic approach is more versatile: the interested entity offers a reward for vulnerabilities and thus generates the necessary interest in discovering those vulnerabilities. As a result, the economic approach may be tried for any system.

1.4 Thesis

A better understanding of vulnerabilities and the nature of vulnerability discovery can provide useful insight into software security.

Vendors and customers can use this information for system comparison and resource allocation. Software engineering tools, such as vulnerability discovery models, are a potentially useful approach to estimating characteristics of software security. However, previous work in this area fails to fully account for the assumptions of the models and the shortcomings of public databases. Vulnerability databases provide a potentially rich source of information, but the existing public databases provide information that is inadequately precise and at too low a level of detail. Nonetheless, a careful examination of the vulnerability history and source code of OpenBSD provides useful information, *e.g.* the rate of independent vulnerability detection appears to be decreasing. The success of this analysis suggests that the software engineering approach could be usefully employed by vendors. More information about the discovery process, such as the likelihood of independent discovery of the same vulnerability, is also critical for the formation of effective vulnerability disclosure policies. Economic approaches, such as a bug auction, provide a better means of measuring software security but are less useful for estimating other characteristics of the vulnerability discovery process.

1.5 Outline

I first describe in Chapter 2 the vulnerability lifecycle from the developer's perspective, and I present definitions for the terms I will use throughout this dissertation. In Chapter 3, I introduce vulnerability discovery models (VDMs). After presenting the existing literature, I identify several assumptions made by that literature that may not be valid.

Other than my own work, the VDM literature relies upon data from the National Vulnerability Database (NVD), which I describe in Chapter 4. I note that the NVD was not designed with the purpose of VDMs in mind, and I highlight the problems in applying VDMs to its data.

Realizing the shortcomings of the NVD for the purpose of modeling vulnerability discovery, I create my own database: I investigate every vulnerability reported in OpenBSD over a period of eight years. Chapter 5 describes the data set created through this efforts.

Unfortunately, the security and software engineering communities do not have a standard taxonomy for vulnerabilities. I discuss the literature on vulnerability taxonomies in Chapter 6. I then apply both the NVD taxonomy and a custom taxonomy to the OpenBSD dataset. Using the taxonomized OpenBSD data, I consider in Chapter 7 whether vulnerability detection is an independent process.

In Chapter 8, I analyze the evolution of the OpenBSD source code. I then consider the vulnerability density: the number of detection events for vulnerabilities introduced in a version per unit size of source code added/altered during that version. In Chapter 9, I use the OpenBSD data sets to consider whether or not the rate of vulnerability reporting is declining in OpenBSD.

The likelihood that independently working detectors will discover the same vulnerability has critical import for vulnerability reporting policies. In Chapter 10, I provide evidence that independent rediscovery occurs with some non-trivial frequency. I then consider the implications of this result on vulnerability reporting policies.

Software engineering approaches may not be the best way to measure software security. In Chapter 11, I consider market-based proposals, and I describe an improvement to these proposals: the bug auction.

I then propose areas for future work in Chapter 12, and in Chapter 13, I summarize my results and draw conclusions.

1.6 Publication history

Much of the material in this dissertation has previously been presented at peer-reviewed workshops or conferences: [Ozm04; Ozm05; Ozm06; OS06b; Ozm07]. During my time at Cambridge, I have also published work unrelated to this dissertation: [GLO+04; OSD06; OS06a; SDOF07; AMNO07].

1.7 A note on style

In "The Hedgehog, the Fox, and the Magister's Pox," Stephen Jay Gould discusses different approaches to writing within the academic community. In particular, he criticizes the use of the passive tense and the avoidance of personal pronouns in scientific writing [Gou03]. I find his arguments persuasive and have thus chosen to use personal pronouns in this work. I use the pronoun 'we' to refer to the community of individuals directly interested in software security. When referring to a paper with multiple authors, I use an & rather than a written 'and' inside the list of the coauthors names.

Chapter 2

Concepts and Definitions

The information security field lacks widely accepted, standard, definitions; as a result, the disparity of usage in the field is a source of confusion. In this chapter, I define the terms I will use. I use these definitions to describe vulnerability models in the next chapter. Then, in Chapter 4, I show that confusion about the definition of a vulnerability has resulted in inconsistent data in public vulnerability databases.

One of the challenges when defining terms in information security is the perspective to which the definition applies. For example, William Arbaugh *et al.* note that 'attack' can have different meanings for a system's defender and an intruder [AFM00]. The definitions here are from the perspective of a developer or a detector whose goal is to improve the security of software by decreasing the number of vulnerabilities it contains.

2.1 Software reliability engineering

Many software vulnerabilities are caused by implementation mistakes. There is thus an inherent overlap between software security and software reliability engineering, because the latter field is concerned with reducing the number and impact of such implementation mistakes.

The field of software engineering benefits from standardized terminology: *e.g.* the "IEEE Standard Glossary of Software Engineering Terminology" [IEE90]. One way to remedy lack of standard definitions in software security is to utilize, whenever possible, the standard terminology of fields like software engineering.

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" [IEE90]. 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" [Lyu96, p. 12].

A **fault** is an "incorrect step, process, or data definition in a computer program" [IEE90]. 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" [IEE90]. 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" [IEE90].

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)" [IEE90].

Rajeev Gopalakrishna *et al.* assert that the goal of software security is to prevent "deliberate attempts to cause failure by triggering faults" [GSV05]. Correspondingly, Sarah Brocklehurst & Bev Littlewood assert that a vulnerability is the security field's equivalent to a 'fault' [BLOJ94].

2.2 Software vulnerability

The computer security field lacks a widely accepted definition of vulnerability.

2.2.1 Definition

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" [Krs98].

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" [IEE90]. 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, vulnerabilities are a subset of faults. Not all faults are vulnerabilities, but all vulnerabilities are faults. Both terms can encompass the results of mistakes made in the design phase.

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

2.2.2 Alternative definitions

A number of other authors have proposed definitions for 'vulnerability,' but all of these alternatives have significant shortcomings. For example, some definitions assume that a vulnerability results in a violation of the security system or security measures:

> A "weakness in the security system, for example, in procedures, design, or implementation, that might be exploited to cause loss or harm" [PP07, p. 6].

However, the focus on a security system or security measures excludes a number of cases that I believe are vulnerabilities. For example, consider a multi-user system that contains private data but that lacks access control. When a user reads another user's private data, he is not violating the security system (it doesn't exist). Nonetheless, I believe that this system suffers from a design vulnerability. One solution to this shortcoming is to focus on the security *policy* rather than the security *system*.

Other definitions are ambiguous. For example, the following definition uses qualifiers such as "usually" or "often:"

> "The CERT/CC has an internal understanding that a vulnerability:
> 1) Violates an explicit or implicit security policy
> 2) Is usually caused by a software defect
> 3) That similar defects are the same vulnerability (*e.g.* SNMP was 2 vulnerabilities)
> 4) Often causes unexpected behavior" [Lon03].

Perhaps the most canonical definition was produced by the Computer Science and Telecommunications Board of the US National Research Council:

> "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" [CSTB01].

Unfortunately, that definition does not address whether or not similar instances of a security fault should be considered the same or different vulnerabilities.

A more recent definition uses fault in a nonstandard way and furthermore relies upon the (undefined) meaning of 'harm.' This definition resulted from a useful but abstract exploration of software engineering and survivability definitions:

> An "internal fault that enables an external fault to harm the system" [ALRL04].

Arbaugh *et al.* propose two different definitions. These definitions address many of the concerns mentioned above, but they are too broad: they include hardware vulnerabilities, while in this work I am concerned only with *software* vulnerabilities.

> "A flaw in an information technology product that could allow violations of security policy" [AFM00, p. 52].

> "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" [AFM00, p. 54].

2.3 Actors and entities

The vulnerability discovery process necessarily includes various actors and entities; those important to this work are defined below.

A **detector** finds instances of vulnerabilities in software systems. I have previously used the term 'vulnerability hunter' to describe detectors [Ozm05; Ozm06; OS06b]. I believe that 'detector' is superior because it also encompasses the situation in which a vulnerability is unintentionally discovered. Other works have used vulnerability 'identifier' [AKN$^+$04]. However, related bodies of literature (*e.g.* on intrusion detection or biometrics) use 'detect' to refer to the discovery of an instance of something and 'identify' to refer to the act of categorizing/matching an instance. In this work, I will use the terms 'detect' or 'discover' interchangeably with respect to vulnerabilities.

A **vendor** is any producer of software, regardless of whether or not that software is sold commercially. It is equivalent to the term 'producer.'

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. An **accidental detector** has unintentionally discovered a vulnerability.

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

A **vulnerability market** pays detectors for vulnerability information. 'White' markets like TippingPoint or iDefense forward the information to the vendor and work to keep malicious entities from learning about the vulnerability. 'Black' markets are illegal and are generally unconcerned with whether or not the vulnerability is used for exploitation.

iDefense's business model is to sell subscriptions to their advance warning service. They thus purchase vulnerability information and then notify both the vendor and their own customers about the vulnerability. They may not tell their customers the exact nature of the vulnerability; instead, they may simply warn customers to, for example, use their firewall to block connections of a certain type [iDe07].

TippingPoint sells an intrusion prevention system: a combination of hardware and software that proactively blocks connections it determines are attacks. Their 'Zero Day Initiative' buys vulnerabilities and then both informs the vendor and updates their customers' intrusion prevention systems [3Co05].

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.

2.4 Disclosure policies

The adjectives 'benign' and 'malicious' lie at the heart of many distinctions made in the vulnerability detection process. For example vendors may consider detectors benign if those detectors follow a specific vulnerability disclosure policy. 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 the principles of 'responsible disclosure.'

A **responsible disclosure** policy specifies that vendors must be informed about a vulnerability and given time to create a patch before the detector makes the vulnerability public. One example of a responsible disclosure policy is that of CERT/CC:

> All vulnerabilities reported to the CERT/CC will be disclosed to the public 45 days after the initial report, regardless of the existence or availability of patches or workarounds from affected vendors. Extenuating circumstances, such as active exploitation, threats of an especially serious (or trivial) nature, or situations that require changes to an established standard may result in earlier or later disclosure. Disclosures made by the CERT/CC will include credit to the reporter unless otherwise requested by the reporter. We will apprise any affected vendors of our publication plans, and negotiate alternate publication schedules with the affected vendors when required.

> It is the goal of this policy to balance the need of the public to be informed of security vulnerabilities with the vendors' need for time to respond effectively. The final determination of a publication schedule will be based on the best interests of the community overall [CER05].

A vulnerability detector follows the practice of **instant disclosure** if he makes public a vulnerability without previously notifying the vendor and giving it adequate time to prepare a patch. (This definition is derived from but more explicit than that in [AKN$^+$04].) The definition of 'adequate time' is part of the responsible disclosure debate, which is discussed in Chapter 10.

A **benign detector** wants to prevent the exploitation of a vulnerability. For example, a benign detector may: chose not to inform anybody of the vulnerability, inform the vendor, inform a disclosure institution, or make public the vulnerability. Benign detectors are sometimes referred to as 'white hats.'

A **malicious detector** is unconcerned about whether or not a vulnerability is exploited. For example, a malicious detector may sell a vulnerability in the black market or publish the vulnerability without having informed the vendor. Malicious detectors are sometimes referred to as 'black hats.'

2.5 Vulnerability life cycle

Vulnerabilities can be characterized according to a hypothetical life cycle. This life cycle is defined by specific events.

2.5.1 Events and data points

- **Injection Date:** The injection date is 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 release date is the date of public release for the system that first contains the vulnerability.
- **Discovery Date:** The discovery date is the date on which the vulnerability is first detected.
- **Disclosure Date:** The disclosure date is the date on which the detector first notifies the vendor or a disclosure institution.
- **Public Date:** The public date is 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 patch date is the date on which the first correction for the vulnerability is released, regardless of whether the correction is official (from the vendor) or correct (lacking faults).
- **Scripting Date:** The scripting date is the date on which the first automated exploit for the vulnerability is released. (One example of an automated exploit is a worm. Another example is a program that exploits the vulnerability to provide its user with administrative control on a remote machine.)[1]

The events in this life cycle can occur in many different orders. For example, a vulnerability may be discovered before it is born: *i.e.* it may be detected during testing and fixed prior to the product's release. Hilary Browne *et al.* argue that these vulnerabilities should be disregarded [BAMF00, p. 2]; I disagree. For large software products, there may be enough of these cases to enable useful statistical analysis. Moreover, these cases can be the best documented.

In practice, not all of the above dates will be known for each vulnerability. Vulnerability data sets may instead record the date known. 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.

For the remainder of this work, I will say that vulnerabilities were detected at a certain date or in a certain version of the program. I am actually referring to the date known: unless I say otherwise, I have no information on the exact date on which a vulnerability was detected.

2.5.2 Status

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

- **Unknown Vulnerability:** An unknown vulnerability exists in the software but has not yet been detected.
- **Secret Vulnerability:** A 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.

[1]Some of these events are borrowed from Arbaugh *et al.* [AFM00], 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.

- **Disclosed Vulnerability:** A disclosed vulnerability has been discovered, and the detector has disclosed it to the vendor or a disclosure institution.
- **Public Vulnerability:** A public vulnerability has been detected and made public through either a patch, a public forum, or the media.
- **Scripted Vulnerability:** A scripted vulnerability is one for which automated exploits exist.

As example, consider the life cycle and status of a single vulnerability. Software vendor DiligentCompany is developing SecureProduct. A developer for DiligentCompany writes code for SecureProduct 2007 that contains a buffer overflow. He submits the code into a CVS repository, at which point the vulnerability has been injected. The date on which SecureOS is first released is also the release date for the vulnerability. The vulnerability is unknown until it is discovered by a detector, Susan. It is secret until Susan discloses it to CERT/CC. Eventually, DiligentCompany releases a patch; on the same day, Susan posts a notice to Bugtraq about the vulnerability. That day is thus both the patch date and the public date for the vulnerability. An automated exploit is later released and the vulnerability is then considered scripted.

Chapter 3

Vulnerability Discovery Models

Vulnerability discovery models (VDMs) are probabilistic methods for modeling the discovery of software vulnerabilities in a single software 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.

VDMs are a potentially useful tool for understanding vulnerabilities and estimating characteristics of software systems. In this chapter, I propose standard definitions for VDMs, and I discuss the different means of assessing their efficacy. I then provide an overview of the existing VDM literature. Finally, I consider the assumptions upon which VDMs rely and consider whether the existing literature has met those assumptions. In Chapter 4, I examine the suitability of the databased used by this literature.

For an example of VDMs and their use, consider our hypothetical system, SecureProduct.

> The number of security patches each month in the first year after SecureProduct is released is: 0, 6, 10, 18, 17, 15, 14, 11, 10, 8, 7, 6. A VDM that predicts an S-shaped curve might be able to accurately model this data. We could extend the curve to estimate how many patches will be released for each of the next six months. DiligentCompany could use this information to schedule time for its engineers to work on those future patches.

However, even our simple example demonstrates some of the challenges of modeling vulnerability discovery. Is DiligentCompany fixing only one vulnerability per patch or do their patches fix multiple vulnerabilities? What *is* a vulnerability? If a source code file contains buffer overflows in two similar functions, is that one vulnerability or two? According to the definition of vulnerabilities that I introduced in Section 2.2, two instances of a buffer overflow are two different vulnerabilities. However, most prior work in this area has not defined vulnerability, and thus this ambiguity is most often unresolved.

Worse, this hypothetical model would not necessarily say anything about vulnerability discovery: it models the rate at which patches are released. But what causes DiligentCompany to release a patch—and is the time it takes to develop a patch always the same?

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, theoretical concerns, and data shortcomings. 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 identify the situations for which they are appropriate. Before we can successfully test VDMs, we need reasonably precise data whose limitations we understand.

3.1 SRMs and VDMs

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" [Lyu96].

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" [AIA93].

These models can be utilized to estimate characteristics about the number of faults remaining in the system and when those faults may cause failures. These estimates can be used to gauge the amount of further testing required to meet reliability requirements.

3.1.1 Definition of a VDM

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 new subfield lacks a standard terminology. Omar Alhazmi & Yashwant Malaiya proposed the application of SRMs to vulnerabilities as 'vulnerability discovery models' (VDMs) [AM05a]. I have 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) [Ozm06].

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 for a standard definition of '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 [GSV05].

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 this definition: 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. (This term was introduced by Alhazmi & Malaiya [AM05a] but not defined; I have defined it using a variation of Lyu *et al.*'s definition of SRMs [Lyu96].)

In this work, I shall reserve the term 'SRM' to refer to models that have been applied to failures and the term 'VDM' to refer to models that have been applied to vulnerability discoveries. However, it is important to note that the same model can be used as both an SRM and a VDM: most existing work takes existing SRMs and applies them as VDMs.

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 [GS05]. The MTTNV is analogous to the software engineering term 'mean time between failures.'[1]

3.1.2 Common SRMs

Only two VDMs have been designed specifically as such: the Alhazmi & Malaiya models discussed in Section 3.4.4, below. The remainder of the models that have been applied to vulnerability data were originally created to model fault discovery. Most of these models are constructed with the belief that the time between failures follows a specific distribution: *e.g.* exponential, Weibull, or gamma. A more detailed description of common SRMs and their functional forms is available elsewhere [MIO87; AIA93; Far96]. However, some models were designed with specific assumptions about faults or the means by which software is designed, and these assumptions are worth highlighting.

[1]Rajeev Gopalakrishna & Eugene Spafford state that a breach or an intrusion can be considered the security field's equivalent to the reliability field's 'failure' [GS05]. 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 3.5.3. 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.

The Schneidewind model assumes that the rate of fault detection changes (possibly dramatically) over time. As a result, estimates of the future rate of detection should be based upon recent data. This approach is unusual: most models treat all historical data equally. Brooks & Motley constructed a model that takes into account that new faults may be introduced even as others are removed. Yamada's S-shaped model assumes a learning curve: faults are discovered slowly while the testers first learn the system. Then, fault discovery accelerates. Eventually, it levels off as the obvious faults are removed and only the more difficult to find faults remain [Far96]. The AML model described in Section 3.4.4 makes a similar assumption.

These 'traditional' models only change the estimated reliability of a system after faults are removed. In contrast, Bayesian models allow for an increase in estimated reliability as the system is used and no failures occur. A system's calculated reliability is thus "a reflection of both the number of faults that have been detected and the amount of failure-free operation" [Far96, p. 104]. The Littlewood-Verrall model is a commonly used Bayesian SRM.

3.2 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 to allocate and schedule their resources.

2. Helping users to allocate and schedule their resources.

3. Estimating the time necessary to achieve an assurance goal.

4. Quantifying the impact of design and implementation methodologies.

5. 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. For example, in 2003 Microsoft instituted a schedule of releasing security patches on the second Tuesday of every month, which was dubbed 'patch Tuesday' [Mic05].

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. Indeed, one author notes that Windows administrators should "devote a certain number of person-days per month to test and distribute whatever critical patches may come out" [Liv03].

Finally, vendors can also 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 [Ozm06]. 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 below in Section 3.5, but first I consider the means by which VDMs are assessed.

3.3 Assessing VDMs

Software reliability practitioners note that not every SRM is suited to every project, but most projects can be modeled accurately by at least one SRM [Far96]. Intuitively, the same is likely to be true of VDMs.

However, before we can be confident that a VDM is a useful model for the discovery process in a specific project, we must test both its absolute and relative accuracy: is the VDM accurate and is it the *most* accurate such model for this project?

A preliminary step in testing the accuracy of a VDM is to ensure that the model has acceptable goodness-of-fit, which is a necessary but not sufficient prerequisite. The most common means of testing a model's goodness-of-fit is the chi square test. Models are first fitted to the data, and parameters that provide the best fit are found.

However, the critical test of a model is its predictive accuracy. There are several common ways of assessing a model's absolute and relative predictive accuracy.

3.3.1 Absolute predictive accuracy

The VDM literature has relied upon two different approaches to assessing absolute predictive accuracy. In previous work, I used u- and y-plots. These test one-step-ahead predictive accuracy, which is useful for scheduling patches. Other work has employed average error and average bias: these test how effective the model is at predicting the final vulnerability count. Future work in this area should apply both tests, to enable comparison within the literature and to assess both aspects of models' predictions.

One-step-ahead *u*-plot & *y*-plot

One approach to absolute predictive accuracy is to assess a model's ability to make accurate one-step-ahead predictions.

> "The purpose of the u-plot is to determine whether the predictions, $\hat{F}_j(t)$, are on average close to the true distributions, $F_j(t)$.... If we were to observe the realization t_j of T_j, and calculate $u_j = \hat{F}_j(t_j)$, the number u_j will be a realization of a uniform random variable. When we do this for a sequence of predictions, we get a sequence u_j, which should look like a random sample from a uniform distribution. Any departure from such uniformity will indicate some kind of deviation between the sequence of predictions... and the truth" [BL96, p. 139].

Sarah Brocklehurst & Bev Littlewood suggest considering the results graphically as the u-plot: plotting the sample distribution function of the u_j sequence and comparing it with the line of unit slope. They suggest the graphical approach because it can make clear the nature of the prediction's deviations from the true function, but a non-graphical consideration is also possible [BL96].

However, the u-plot can present an overly optimistic picture of bias if there are trends in the errors that average out. (For example, if pessimistic early bias is cancelled by an optimistic later bias.) In order to detect this situation, the u-plot should be paired with the y-plot test. The latter test transforms the sequence u_j into one that looks like "the realization of the successive inter event times of a homogeneous Poisson process; any trend in the u_j's will show itself as a nonconstant rate for this process" [BL96, p. 140].

Final count average error & bias

Instead of considering one-step-ahead predictions, another approach ascertains the accuracy of the model's continuous predictions of the total number of vulnerabilities in the product.

For example, Alhazmi & Malaiya apply VDMs to data divided into n equal-length calendar time intervals $t_1, t_2, t_3, ..., t_n$. A best-fit model is used to predict the estimated total number of vulnerabilities (Ω_i) based on the information available at time t_i. The total number of vulnerabilities known to exist in the product is denoted by Ω_{known}. This information is used to calculate the average error (AE) and average bias (AB) [AM06b]:

$$AE = \frac{1}{n}\sum_{i=n}^{n}\left|\frac{\Omega_i - \Omega_{known}}{\Omega_{known}}\right| \tag{3.1}$$

$$AB = \frac{1}{n}\sum_{i=n}^{n}\frac{\Omega_i - \Omega_{known}}{\Omega_{known}} \tag{3.2}$$

3.3.2 Relative predictive accuracy

Two tests of the relative predictive accuracy of a model are the Akaike Information Criteria (AIC) and the Prequential Likelihood Ratio (PLR). These tests cannot provide information on the absolute accuracy of the model; instead, they indicate which of several competing models is the most accurate. The former is an entropy-based approach used to find the model that best explains the data with a minimum of free parameters. Alhazmi & Malaiya use the AIC to compare different VDMs [AM05a]. In previous work, I have used the PLR: it examines whether one model's probability density function is everywhere closer to the real probability density function than the other model being tested [BL96, p. 131].

3.4 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 assess it with respect to common problems in modeling.

Most of the literature described here uses vulnerability data from the National Vulnerability Database (NVD), which is described in more detail in Chapter 4.

3.4.1 Rescorla

The VDM literature began in 2004 with a workshop paper by Eric Rescorla [Res04] (a revised version was later published as [Res05]). Rescorla was investigating whether or not the social value of vulnerability discovery by external detectors is positive (this question is further discussed in Chapter 9). He applied two models to vulnerability data from the NVD: a linear vulnerability discovery model (LVD) and an exponential Goel-Okumoto model. The former is often used in the literature as a baseline with which to compare more complex models. The latter was developed as an SRM [GO79].

Rescorla was unable to fit either the LVD or the exponential Goel-Okumoto model to data on three operating systems: WinNT4, Solaris 2.5.1, and FreeBSD 4.0. He was able to fit both models to RedHat 6.2 data. However, as I note in Section 3.3, goodness-of-fit is a necessary but not sufficient requirement for a model. More important is the model's predictive accuracy. Rescorla did not test either model's predictive accuracy against the Redhat 6.2 data, so neither can be considered proven [Res05].

3.4.2 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 2 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 [GS05]. Gopalakrishna *et al.* later assess these requirements for a number of different approaches to measuring vulnerabilities and vulnerability discovery [GSV05].

3.4.3 Ozment and Schechter

In earlier work, I created and analyzed a data set of individually examined vulnerabilities for the OpenBSD operating system [Ozm05; Ozm06]. I tested the fit and predictive accuracy of more than a dozen SRMs against the data. With Stuart E. Schechter, I analyzed a larger data set and concluded that the rate of vulnerability discovery in OpenBSD is declining—for vulnerabilities introduced prior to a cut-off date. In that same work, I looked at the rate of change in the source code of OpenBSD [OS06b].

3.4.4 Colorado State University

Since 2005, researchers at Colorado State University have been investigating the use of VDMs.

Alhazmi & Malaiya propose two models specifically for vulnerability discovery: the Alhazmi-Malaiya effort-based model (AME) and the Alhazmi-Malaiya Logistic model (AML) [AM05b].

Alhazmi-Malaiya effort-based model (AME)

The AME is an effort-based model, which approximates effort with the number of users of a system. Alhazmi & Malaiya argue that the effort of vulnerability detectors is related to the number of installations of the targeted software. However, they provide no evidence for this assumption, and in the next section (3.5.1), I note some reasons that it may not be true.

They find that this model has acceptable chi square goodness-of-fit to WinNT4 and Win98 data from the NVD [AM05b]. Furthermore, Sung-Whan Woo *et al.* successfully fit this model to NVD data on all vulnerabilities found in Apache and IIS. They then use the NVD taxonomy to break the vulnerabilities down into categories and apply the models to the three largest categories; again, they find a good fit [WAM06b]. However, none of this work has tested the model's predictive accuracy. Without such tests, this model must be considered unproven.

Alhazmi & Malaiya logistic model (AML)

The AML is an S-shaped, time-based logistic model of vulnerability discovery. This model assumes that vulnerability discovery occurs in three phases: learning, linear, and saturation. The first phase is the initial, flat portion of the S-shape. In this phase, detectors are still learning about the newly released software, so they do not report many vulnerabilities. The linear phase is the steep portion of the S-shape: detectors are familiar with the product and the number of vulnerabilities reported grows linearly. The saturation phase is the final, flat portion of the S-shape: the easiest vulnerabilities have been found, users are migrating to a newer version of the product, detectors are less interested in the product, and the rate of vulnerability reporting declines [AM05b].

Alhazmi & Malaiya find that the fitted model has acceptable chi square goodness-of-fit for NVD data on five Windows operating systems (95, 98, XP, NT 4.0, and 2000) and two RedHat Linux operating systems (6.2 and 7.1) [AMR05]. Woo *et al.* then find that the AML has acceptable goodness-of-fit to their NVD Apache and IIS data. Using the NVD taxonomy, they break these two web servers' vulnerabilities down by category and apply the models to the three largest categories for each server; again, they find a good fit [WAM06b]. These papers did not include results on the model's predictive accuracy.

Alhazmi & Malaiya do use average error and average bias to test the predictive accuracy of the AML model on vulnerability data for Win98, Win2000, and RedHat Linux 7.1 [AM06b]. They then propose a combined approach, the AML-C, that uses known vulnerability densities from previous releases of the systems to discard some of the extreme estimates produced by the AML. They find that the AML-C is superior to the AML. However, they also found that the LVD model had the best average error and average bias for the Win2000 data. It was a close second to the AML-C for the RedHat 7.1 data and second to the AML-C for the Win98 data. They argue that the LVD is a good fit for systems in which 'saturation point' in vulnerability discovery has not yet occurred. The saturation point occurs when the easiest vulnerabilities in a system have been discovered and the rate of discovery thus decreases. Their AML model assumes the existence of such a point [AM06b].

They also test the predictive accuracy of the AML and AML-C on NVD data from Apache 1 & 2 and IIS 4 & 5. They find that an adaptive approach improves their results: they observe the errors made in past estimations and adjust their current predictions accordingly [AM06a]. (Adaptive techniques were previously proposed for SRMs, where they were also found to be an improvement over non-adaptive techniques [BCLS90].) Finally, Woo *et al.* assesses the goodness-of-fit of the AML model on NVD vulnerability data, overall and categorized by type, for IE and Firefox [WAM06a].

Table 3.1 summarizes the results of the Colorado State University work on VDMs. It shows the predictive accuracy, as assessed by average error and aver-

	LVD		AML		AML-C	
Data Set	AE	AB	AE	AB	AE	AB
Win 98†	29.1%	29.1%	43.8%	-43.8%	**20.1%**	**-20.1%**
Win 2k†	**4.6%**	**4.6%**	29.2%	-29.8%	16.8%	-15.5%
RH 7.1†	19.2%	**-4.0%**	37.9%	-5.3%	**15.4%**	6.3%
Apache 1.0 ‡	29.9%	-29.1%	52.5%	**8.3%**	**29.8%**	-29.8%
Apache 2.0 ‡	**8.2%**	**8.2%**	25.2%	-25.2%	16.3%	-10.7%
IIS 4.0 ‡	60.0%	60.0%	20.3%	-20.3%	**18.9%**	**-18.9%**
IIS 5.0 ‡	47.0%	47.0%	**17.4%**	-17.1%	**17.4%**	**-17.0%**

† Data in this row is from [AM06b]. ‡ Data in this row is from [AM06a].

Table 3.1: *The accuracy of the predictions made by three VDMs, assessed by Average Error and Average Bias. The prediction in bold is best for its data set.*

age bias, of successfully fitted LVD, AML, and AML-C models applied to NVD data on seven different software systems.[2] The AML-C has the best average error for five of the seven data sets. However, the best predictive accuracy it had was 15.4%, on RedHat 7.1 data. In contrast, the simple linear model, LVD, had an average error of 4.6% and 8.2% for the two models for which it was most accurate.

3.4.5 Other literature

Robert Brady *et al.* proposed a theoretical model of fault discovery that is based on thermodynamics [BAB99]. In a later work, Ross Anderson used the model to provide a theoretical bound of the effects of vulnerability discovery and remediation on security [And02]. Anderson did not apply the model to any data. However, Alhazmi & Malaiya find that it cannot be fitted to their Win 95, Win XP, or RedHat 6.2 data sets [AM05a].

3.5 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.

[2]When applied to the Windows and RedHat data, the AML-C was restricted such that the duration of the linear phase was $\epsilon_1 < 2.63/\mathrm{AB} < \epsilon_2$. The ϵ values were chosen based on data from previous versions of the system [AM06b]. The AML-C was restricted to $21 < 2.63/\mathrm{AB} < 42$ for the Apache and IIS datasets [AM06a].

3.5.1 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 [Lyu96]. 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 [AM05b]. 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.

3.5.2 Operational environment

In order to be effective, SRMs require that the environment from which the data is obtained (usually the testing environment) must be equivalent to the environment in which the software will be utilized after deployment. However, many vulnerabilities rely upon the adversary intentionally inputting abnormal data—data outside the bounds of a normal operational profile.

Theoretically, over a long period of time and the wide range of real world environments, we can consider that the operational profile includes *all* possible input. This perspective justifies the application of these models to vulnerabilities, but it does imply that vulnerabilities may be detected more slowly than faults would be detected. Furthermore, pre-release VDMs will thus likely be less accurate than pre-release SRMs.

In addition, chronologically local 'spikes' in the number of vulnerabilities found may indicate the discovery of a new technique for detecting vulnerabilities. The discovery of a new technique is conceptually equivalent to an expansion of the operational profile.

3.5.3 Independence

An additional assumption underlies almost all VDMs: that vulnerability discoveries occur independently. However, there is reason to doubt that they do.

Gopalakrishna & Spafford consider NVD data on vulnerabilities. They classified these vulnerabilities using both the NVD and the Taimur Aslam *et al.* [AKS96] classification systems. Interestingly, they find that in one-third of the instances in which "two successive vulnerabilities are considered in a software product, they are of the same type." This result lead them to believe that vulnerability discovery is not independent, although the result may be skewed by the fact that half of the vulnerabilities are in the same category: input validation errors. They next apply a run test to determine data independence and find mixed results: for each different product, vulnerability discovery in some categories is dependent; in others it is independent [GS05]. This finding indicates that any modeling effort should first test to see whether the vulnerability data is independent.

If vulnerability data is dependent, it may be because vulnerability hunters are looking for other instances of the same type of vulnerability that has just been discovered—and developers may repeat the same type of mistake. 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 rewarded for looking for other or similar instances of the same type of vulnerability. Other, non-security related, faults may be introduced in the same way. However, SRMs are usually applied to failure data obtained during automated testing: as a result, the discovery of these faults is independent [MIO87].

It may instead be possible to apply VDMs to **vulnerability 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 [Res05] states his assumption of independence but does not test it. In my previous work, I acknowledge the problem and attempt to construct a data set of vulnerability detection events [Ozm05; Ozm06; OS06b]. In that work, however, I did not test for independence. (I do so in Chapter 7, below.) The Colorado State University literature does not address the question of independence at all.

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 [GPT00].

3.5.4 Static code

Most SRMs and VDMs assume a static code base.[3] 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 [WAM06b]. 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 [AM06a]. 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 [Res05]. Most of the other work from Colorado State University does the same [AM05a; AM05b; AM06b]. 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 [Ozm05; Ozm06; OS06b].

[3]One exception is the B&M Binomial SRM, which assumes that patches are applied and may introduce new faults at a fixed rate [AIA93].

3.6 Guidelines for VDM research

One way to address the problems of existing work is to compile a list of best practices to be used in VDM research. I propose that future work should, at a minimum:

1. Define its terms, including: vulnerability, VDM, and detection event.
2. Clearly state the assumptions on which the model depends.
3. Ensure that the data examined is for a single version of the system.
4. Test the data for independence and then choose a model based upon that result.
5. Acknowledge those assumptions that may not be perfectly met, like constant effort, and consider how those assumptions may drive the results.
6. Assess both one-step-ahead and final-count predictive accuracy.
7. Publish the data, at least online, so that others can replicate the work and use it to test their own models.

3.7 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 here are remedied.

Almost all of the existing work on VDMs, including my own, has inadequately noted and abided by the models' assumptions. Research efforts have failed to: acknowledge the possibility of dependence in vulnerability discovery [AM05a; AM05b; AMR05; AM06a; AM06b; WAM06a; WAM06b], test for dependence [Ozm05; Ozm06; OS06b], and only apply the models to static code [AMR05; WAM06b]. Other efforts have failed to test models' predictive accuracy [Res04; AM05a; AM05b; AMR05; WAM06b].

The next chapter highlights another important consideration for VDM research: ensuring that the data used are accurately and consistently gathered.

Chapter 4

Collecting Vulnerability Data

The accuracy of VDMs—and any other statistical analysis of vulnerabilities—relies upon the accuracy of the vulnerability data to which they are applied.

The reliability engineering literature generally assumes that SRMs have been applied during pre-release testing and in settings where the collection of failure data is an integral part of the testing environment. Unfortunately, vulnerabilities are unlikely to be identified as such in that stage of software development: if they are found at all, they will probably be perceived simply as quality faults. As a result, vulnerabilities may be most often detected as such after the product is released, when the collection of precise data is much more difficult.

Vulnerability research is frequently based upon public vulnerability databases. The National Vulnerability Database (NVD)[1] is the database used for almost all of the research on VDMs: for example, it was used by Rescorla [Res05], the Purdue University work [GS05; GSV05], and the Colorado State University work [AM05b; AMR05; AM06a; AM06b; WAM06a; WAM06b].

However, the NVD was not designed with vulnerability discovery modeling in mind, and it therefore has four important shortcomings for this purpose: chronological inconsistency, incomplete inclusion, multiple entries for a single detection event, and lack of documentation.

4.1 Problems with using the NVD for VDMs

The NVD was not designed for the purpose of employing VDMs; it was not even designed with the goal of high-quality statistical analysis. Instead, the NVD helps accomplish "the mandate to warn the public about vulnerabilities in computer systems;" it does so "by offering official vulnerability information on all known computer vulnerabilities" [NISa]. The problems I describe below have arisen because the NVD is being used for a purpose outside of its mandate.

[1]The NVD was formerly known as ICAT.

4.1.1 Chronological inconsistency

The NVD has inadequate information on the chronological information most important to the use of VDMs: when the vulnerable code was first released and when the vulnerability was discovered.

Version information

The NVD currently has a data field called 'vulnerable software and versions.' From 2001 to 2005, information in this field was "obtained from various public and private sources. Much of this information is obtained (with permission) from CERT, Security Focus, and ISS X-Force" [ICA02]. The NVD does not describe how they currently obtain this information.

The goal of this field is to notify users when they are relying upon vulnerable software [ICA02]. It may thus omit obsolete versions of the software that are nonetheless vulnerable. For example, the detector of a vulnerability may test it against only the most current version of a program. Detectors and vendors are rarely interested in whether or not a vulnerability applies to old, no longer supported versions of the software.

As a result, this field is not an accurate source of information on when a vulnerability was released. Unfortunately, VDM research has often used it as such: researchers identify the earliest listed version of the software and assume that the vulnerability was introduced in that version. The release date for the vulnerability is then given as the date on which that version of the software was introduced.

This inaccuracy can result in two types of errors. First, vulnerabilities can appear to be younger than they actually are: their release date is based on a more recent version than the one in which they were truly released. This error can cause models to indicate a more rapid time to discovery than is actually the case. Second, software versions can seem to be more secure than they actually are: it appears that no recent vulnerabilities have been found in obsolete versions of the software, when in reality those versions were just not tested when vulnerabilities were discovered. This error can cause models to indicate that the version was more rapidly depleted or contained fewer vulnerabilities than is actually the case. Alhazmi & Malaiya have applied VDMs to NVD data on a number of operating systems [AM05b; AM05a; AM06b]. Unfortunately, they do not mention the chronological inconsistencies in the NVD that were described in Section 4.1.1, above. As a result, we cannot ascertain whether the satiation phase they find is due to a decrease in the number of vulnerabilities reported or to a failure of the NVD to include the pertinent operating system as affected.

Discovery date

The date on which a vulnerability is discovered is also only approximately recorded in the NVD as the 'original release date.' For vulnerabilities added before 2001, the NVD reports a "published before" date; after 2001, it reports the date on which the vulnerability was added to the NVD [ICA02]. Both approaches may be inaccurate by several months. Of the literature that relies upon the NVD, only the work by Rescorla [Res05] and Gopalakrishna & Spafford [GS05] notes this change.

Furthermore, even if the vulnerability is added to the NVD on the same day that it became public, that date may greatly differ from the actual discovery or disclosure date. Some vendors work on vulnerabilities for months or even years before they release a patch and the vulnerability becomes public [Ozm05]. As a result, the "published before" date may be inaccurate by over a year.

4.1.2 Inclusion

Another drawback of the NVD is that it does not necessarily contain every vulnerability detected in a software system. It contains only those vulnerabilities that have been assigned Common Vulnerability and Exposure (CVE) identifiers; many vulnerabilities that predated the creation of the CVE list in September 1999 have never been assigned CVE identifiers.[2] Even after that date, not all vulnerabilities recorded by the vendor or other databases, such as Bugtraq, are included.

4.1.3 Separation of events

For the purposes of vulnerability discovery modeling, perhaps the most significant shortcoming of the NVD is that its entries are sometimes *vulnerabilities* and sometimes *vulnerability detection events*. This inconsistency is actually in the CVE list. The NVD simply has one entry for every CVE identifier.

This shortcoming is perhaps due to the field's lack of a widely accepted definition for 'vulnerability.' The definition proposed in Section 2.2 clarifies this situation by specifically stating that a vulnerability is a single instance of a mistake.

If multiple vulnerabilities are discovered at the same time, by the same individual, then their discovery is unlikely to be independent. The detector probably discovered one vulnerability and then looked for similar instances of it in similar areas. If the CVE assigns the different vulnerabilities different CVE identifiers, than any VDM using the NVD data will be modeling dependent events.

[2]The CVE list is an initiative to provide a single, widely accepted name to vulnerabilities. Rather than a database of information about a vulnerability, it simply provides a universal identifier for that vulnerability [MIT05]. CVE identifiers are an important tool for ascertaining when entries in different databases refer to the same vulnerability.

4.1.4 Documentation

Another challenge with using the NVD for vulnerability modeling is its lack of documentation. Other than a short list of frequently asked questions, there is very little information available about how the data is collected, what information is recorded, and how the process has changed over time. The database has indeed changed over the past decade, and these changes are inadequately documented for the purposes of VDMs.

For example, the current site does not contain a description about the categorization scheme for vulnerabilities: there is not even a listing of each category. Furthermore, there is no mention of how the date recorded as the 'original release date' or 'published before' date has changed over time. The only way to discover some of this information is through the Internet Archive's stored historical snapshots of the site [ICA02].

Using the NVD for a long-term analysis of data is thus problematic: the meaning of the fields may change without notice.

4.2 Other public vulnerability databases

The NVD is not the only public vulnerability database, although it is the database used by the remainder of the VDM literature. In addition to the NVD, some of the more prominent databases include: Bugtraq (run by the same organization as the Bugtraq mailing list) [Sec05], ISS X-Force [XFed], and the Open Source Vulnerability Database (OSVDB) [OSVed]. These diverse databases often, but not always, have overlapping data. They may suffer the same shortcomings described above.

4.3 Next generation vulnerability database

Before we can assess the utility of VDMs in measuring software security, we need an accurate source of data: we need a next generation vulnerability database. In this section, I propose some requirements for that database.

Any next generation vulnerability database should include as much of the event data listed in Section 2.5 as possible: the injection, release, detection, disclosure, public, patch, and scripting dates. Each field should also contain a note indicating the precision of the date: *e.g.* approximate to within three weeks. Because most of this information will have been found on the internet, each date should also have any evidentiary URLs and the dates of that evidence associated with the entry.

One important act for such a database is to distinguish between vulnerabilities and vulnerability detection events: it should list both. It should also contain links to pertinent entries in other databases (as is commonly done in the databases discussed here). It should include a 'different from' field to help distinguish similar yet different vulnerabilities and detection events. In addition, the database should include copies of the moderators' discussions. Understanding how and why moderators decided to separate a vulnerability detection event is often useful.

Any public database of vulnerabilities must be maintained for years before it can be useful for VDMs. Such a database is likely to evolve over time. As a result, any public database should also document:

- Each data field: how the data is obtained and how accurate it is.
- Changes to the process, information, or fields that may occur during the lifetime of the database.
- The date on which versions of software are no longer tested to see if they contain a vulnerability.

This information will enable the research community to gain a better understanding of vulnerabilities, how they are found, and how their prevalence changes over time.

4.4 Conclusion

The existing VDM literature almost universally relies upon inconsistent and inaccurate data from public vulnerability databases, which were never intended to be used for this purpose. That data does not necessarily represent *vulnerability detection events*, which are what current VDMs are capable of modeling (the VDMs used in the literature are not appropriate for modeling vulnerabilities whose discovery is dependent). The accuracy of VDMs depends on the data to which they were applied: we cannot better model vulnerability discovery until we have a database designed with this usage in mind. The security community should design and implement a next generation vulnerability database to provide the high-quality data that we need.

Chapter 5

The OpenBSD Data Sets and Methodology

The VDM work by other research groups has relied upon data from the NVD. However, Chapter 4 described the problems with this data set. In order to ascertain whether VDMs could be accurately applied to vulnerability data, I first had to obtain an accurate data set. Because no such set could be found, I decided to collect accurate vulnerability data for a single system.

5.1 Selecting a system to study

I chose to study OpenBSD because it is open source, long-running, consolidated, and security-focused.

In order to obtain precise, accurate information on vulnerabilities, I needed access to the system's source code. As a result, I only considered open source software as potential sources of data. OpenBSD fit this requirement: its source code and change history are readily accessible via a publicly accessible CVS repository.[1]

OpenBSD is also consolidated: unlike Gnu/Linux, there is a reasonably clear delineation between the portions of the system that are OpenBSD and the portions of the system that are external contributions. I refer to the former as 'OpenBSD' and, following common usage, the latter as the 'ports' collection. Gnu/Linux, while more widely used, also has the shortcoming of being fragmented into a large number of distributions.

Finally, OpenBSD is security-focused: its developers started the project with the goal of creating a more secure operating system. They have invested significant effort into code audits, secure practices, and security features [Ope98]. Prompted by Rescorla's intriguing work on the social utility of non-vendor vulnerability detection

[1]CVS is an example of a version control system. Such a system enables multiple individuals to change a single set of files without over-writing each other's work. It also tracks and credits the changes to each file, so the evolution of the source code can be ascertained.

[Res04], I wanted to know whether the rate of vulnerability discovery was declining in a widely used operating system. I therefore sought to test a system whose developers focused on finding and removing vulnerabilities: if I had found no decline in the rate of vulnerability reporting for this system, then less security-focused systems would presumably have the same outcome. My analysis of the detection event rate in OpenBSD is presented in Chapter 9.

5.2 Compiling the OpenBSD vulnerability data set

The initial release of OpenBSD was version 2.0; this version was forked from NetBSD 1.1 in late 1996. Prior to version 2.2, the OpenBSD developers performed an extensive security audit and repaired numerous vulnerabilities without reporting them. As a result, I could not obtain good vulnerability data for the releases prior to 2.2. In addition, in version 2.3 the OpenBSD team changed the way they integrated X11 into the code base.

I therefore selected version 2.3, released on 19 May 1998, as the earliest version for the data set: it was the first truly stable release in which vulnerabilities were consistently documented. I refer to this version as the **foundation version**, and I refer to code and vulnerabilities present before the release of this version as **foundational code** and **foundational vulnerabilities**.

The OpenBSD project releases a new version approximately every six months, incrementing the version number by 0.1. This data set incorporates the eighteen versions of OpenBSD from 2.3 to 4.0, inclusive.

5.2.1 Vulnerability data collection

The OpenBSD vulnerability data set was created through the following process:

1. I compiled a database of vulnerabilities detected in the 8 years between 19 May 1998 and 19 May 2006. I examined every vulnerability listed on the OpenBSD security web page; I used four public vulnerability databases to obtain additional information on those vulnerabilities: NVD [NISb], Bugtraq [Sec05], OSVDB [OSVed], and ISS X-Force [XFed].

2. I examined CVS records and the source code to detect the date on which a correction for the vulnerability was first checked into the CVS repository. If the fix was itself faulty, the date of the first repair effort is used because it most closely tracks the date of discovery. I also examined the public vulnerability databases for information on when the vulnerability became public. I then selected the vulnerability's date known: the first date for which evidence of the vulnerability's existence is available.[2]

[2]The release of a public report and the repair of the vulnerability do not always occur in the same order. When a vulnerability is reported to an entity other than the OpenBSD development

3. I manually examined prior versions of the source code to detect the vulnerability's injection date. If there was any doubt, the earliest possible date was chosen. Using that information and other CVS records, I was then able to ascertain the vulnerability's release date.

Not all vulnerabilities could be easily and precisely categorized: the process was manual, time-consuming, and laborious. Below, I describe the decisions I made with respect to inclusion and uniqueness.

5.2.2 Which vulnerabilities are included?

I included vulnerabilities that I believed to be applicable to the bulk of OpenBSD's installed base. I therefore excluded vulnerabilities that were specific to processor architectures other than the x86. I also excluded vulnerabilities that were location/country dependent. In addition, I excluded reports of vulnerabilities in historical versions of OpenBSD if the release that was current at the time of the report was not vulnerable.

My analysis covers all portions of the OpenBSD code in the primary CVS repository. This includes the X-windowing system, the Apache web server, and many additional services not traditionally considered to be part of the core operating system. However, this repository excludes the 'ports' collection of third-party software that is not officially part of OpenBSD. I included vulnerabilities regardless of whether or not they applied to the default configuration of OpenBSD. (A default configuration in which most services are disabled is another commendable aspect of OpenBSD's security policy; however, in practice, many of those services will be enabled by the users.)

5.2.3 Lack of information on third-party subsystems

OpenBSD includes some software that is maintained by third parties (*e.g.* sendmail). Those third parties often release new versions of their software that bundle together fixes for multiple (previously secret) vulnerabilities. Unfortunately, the third party vendors do not always make available the information necessary to detect the release and discovery date of the component vulnerabilities. I was thus sometimes unable to obtain access to the data necessary to establish the relevant dates for each vulnerability and detection event. As a result, some detection events in the data set have the same release date or date known, even though in reality the vulnerabilities in these detection events may have been injected or discovered at different times.

team, the date of the public report often precedes the date on which a repair is committed to the CVS. When a vulnerability is reported directly to the OpenBSD development team, they usually commit a repair into the CVS repository prior to publicly announcing the vulnerability. I utilize the earlier of the two dates so that I most closely approximate the actual discovery date.

5.2.4 What constitutes a unique vulnerability?

I used the data available in the public vulnerability databases and in other public sources to ascertain whether or not a group of vulnerabilities were discovered in the same vulnerability detection event. For example, I sometimes found announcement emails by the detector stating the process by which the vulnerabilities were discovered. If the vendor credited the same individual for discovering the vulnerabilities and reporting them at once, I considered each of those vulnerabilities to have been in the same detection event *if they are of the same type* (using the custom taxonomy described in Section 6.3). If the detection event contains multiple vulnerabilities, then the detection event's release date is that of the earliest released vulnerability. Only a handful of detection events contained vulnerabilities with different release dates. By definition, all of the vulnerabilities in a detection event share the same date known.

5.3 Results

Table 5.1 shows the number of vulnerabilities that were released and became known in each version of OpenBSD. The version in which the vulnerability was released is specified by the column. The version in which the vulnerability became known is specified by the row. The first column contains a total of 97 vulnerabilities that are foundational: they were introduced before the start of the study and were thus present in the code of the foundation version, 2.3. The top entry in that column indicates that 7 vulnerabilities became known during the six months between the release of version 2.3 and the release of 2.4.

The bottom row of Table 5.1 also shows the number of lines of code, in millions, that were altered/introduced in each release until version 3.7 (see Chapter 8 for the methodology used to obtain this information).

The most startling result shown in Table 5.1 is that 63% of the 155 detection events that occurred during the study contained foundational vulnerabilities. This result does not necessarily imply that the code added in OpenBSD 2.3 was particularly insecure. It may instead mean that the historical code base, which could date back to the early 1980s, still dominates the overall code base.

5.4 Conclusion

The OpenBSD vulnerability data set contains accurate information on the exact dates on which vulnerabilities were injected, released, and became known. The next step in understanding the security of the OpenBSD software is to taxonomize the detection event data in order to test for dependence in the vulnerability hunting process.

Version current when vulnerability was patched	Version in which the vulnerability was first released																		
	2.3	2.4	2.5	2.6	2.7	2.8	2.9	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	4.0	Total
2.3	7																		7
2.4	11	0																	11
2.5	6	0	1																7
2.6	5	1	0	0															6
2.7	12	2	2	2	2														20
2.8	11	1	0	1	3	0													16
2.9	3	0	0	2	0	0	0												5
3.0	2	1	0	0	1	0	2	0											6
3.1	9	2	1	2	0	0	0	1	1										16
3.2	7	3	0	0	0	0	1	2	0	1									14
3.3	2	2	0	1	0	0	0	0	0	1	2								8
3.4	3	0	0	0	1	0	1	0	1	0	0	0							6
3.5	13	2	1	0	0	0	0	2	0	1	0	0	1						20
3.6	2	0	1	0	0	0	0	0	0	0	0	0	0	0					3
3.7	1	1	0	0	1	0	0	0	0	0	0	0	2	0	0				5
3.8	2	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0			3
3.9	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0		2
4.0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
Total	97	15	6	9	8	0	4	5	2	3	2	1	3	0	0	0	0	0	155
MLOC	10.1	0.4	0.3	1.1	0.8	0.4	2.2	0.6	0.8	0.3	0.3	0.8	1.4	0.7	0.9				

Table 5.1: *The number of OpenBSD vulnerabilities according to the version in which each was released and the version current at the time each was patched. The final row, at the bottom of the table, shows the count in millions of lines of code altered/introduced in each version prior to 3.8.*

Chapter 6

Vulnerability Types and Taxonomies

A key consideration for vulnerability discovery modeling is whether or not vulnerability discovery is an independent process. However, we cannot test for independence unless we have somehow categorized the vulnerabilities. A dependent process is one in which detecting a vulnerability somehow triggers the discovery of another vulnerability—and the sequential discovery of *similar* vulnerabilities is one clue that the discovery of the first led to the discovery of the second. In order to ascertain 'similarity,' I categorize the vulnerabilities according to two taxonomies.

Here, I discuss the literature on vulnerability taxonomies. I then describe the taxonomy used by the NVD and thus used by most VDM research. I apply the NVD taxonomy to the OpenBSD dataset. Next, I describe a custom taxonomy created specifically for this dataset and apply it to the OpenBSD data.

6.1 Existing vulnerability taxonomies

Selecting a taxonomy to use when analyzing vulnerabilities is a difficult task. A number of different taxonomies exist; indeed, numerous philosophies of taxonomy are competing in this area. At the most comprehensive end of the spectrum, taxonomies attempt to place vulnerabilities in categories that are: mutually exclusive, exhaustive, unambiguous, repeatable, accepted, and useful [Amo94, 34]. Requirements like this can lead to a taxonomy with only three or four categories: formally correct but not very useful. At the other end of the spectrum are those taxonomies based on categories that seem to arise naturally from the databases themselves. These taxonomies tend to be formally weak: often, a vulnerability can fit into multiple categories. Their focus is usually simply on being useful for a narrow analysis or task. Existing databases tend to be categorized according to this latter type of weak taxonomy.

Carl Landwehr *et al.* create a taxonomy of vulnerabilities with the three high level categories of how, when, and where - genesis, time of introduction, and location. Each category is further subdivided multiple times. For example, genesis, is further divided into intentional and inadvertent. The most fine-grained categories within genesis include trojan horse, trapdoor, logic/time bomb, covert channel, and 'other.' The purpose of this taxonomy is to aid in the automation of vulnerability detection and the general improvement of the software engineering process. The authors also include a sample of fifty vulnerabilities, categorized according to their taxonomy. Their examples are from a wide variety of operating systems and are thus interesting for their diversity when compared to more current examinations of vulnerabilities [LBMC94].

Matt Bishop creates a taxonomy with the goal of helping software engineers fix existing and avoid creating new vulnerabilities. His taxonomy has six separate axes: the nature of the vulnerability (using the categories from a previous system, the Protection Analysis project), time of introduction, exploitation domain (user domain in conjunction with network and/or physical resources), effect domain, minimum number of components necessary to exploit it, and the source of the vulnerability information. He discusses broad classes of vulnerabilities which his taxonomy can help detect and then applies this taxonomy to a number of UNIX vulnerabilities [Bis95].

Taimur Aslam *et al.* create a taxonomy with only two high-level categories: coding faults (synchronization errors or condition validation errors) and emergent faults (configuration errors or environment faults). They derive their taxonomy from software testing tools and techniques and the results that are typically produced. They created a database of faults, which was used in the development of an intrusion detection system. Unfortunately, the minimal number of categories seems to limit the utility this taxonomy provides for software engineering education or process improvement [AKS96].

Matt Bishop & David Bailey survey three existing taxonomies of vulnerabilities, including Aslam [AKS96]. Each of these taxonomies is criticized for failing to define a unique category for each vulnerability. Using as examples the `xterm` and `finger` vulnerabilities, they demonstrate that multiple categories in each taxonomy could be assigned based on the level of abstraction and the perspective (attacker, process, operating system, *etc.*) from which the vulnerability is considered [BB96].

John Howard & Thomas Longstaff develop a high level 'common language' of definitions and taxonomies to better enable the collection and exchange of information about computer security incidents. They focus on a high level set of terms, believing that they can better find and create agreement with these terms than with more specific, lower level terms. Their hope is that the lower level terms will then be defined with the higher level terms thus created [HL98].

David Baker *et al.* describe the motivation for creating the Common Vulnerabilities and Exposures (CVE) list. They then elaborate on the lessons learned in its initial years. They particularly highlight the challenges of developing a full taxonomy as a reason for focusing on assigning canonical names alone in the list [BCHM01]. In another practical paper, L. Ma *et al.* describe their motivation for creating a collaborative vulnerability database for use internally at CERT/CC and describe some of the lessons they have learned from the experience [MMSM03].

Accepting that no single taxonomy will be sufficient for all uses, Frank Piessens designs one that highlights the cause of the vulnerability in order to help software engineers prevent future instances of it. The higher level of his taxonomy thus corresponds to the phase of the software engineering process in which the vulnerability was first introduced: analysis/specification, design, implementation, deployment, or maintenance. Each higher level category has a number of lower level categories for further refining the classification (each also has a lower level category that is essentially for 'miscellaneous' vulnerabilities). He concludes that his taxonomy is useful for educating developers, for testing, and for audit [Pie02]. However, Piessens' taxonomy does not seem to be used in any large database. It suffers from the problem that real-world vulnerabilities will be highly concentrated in only one of his nineteen low-level categories. That category, 'insufficiently defensive input checking,' will contain both buffer overflows and input string vulnerabilities.

Shuo Chen *et al.* analyze the Bugtraq database on security vulnerabilities and reach three conclusions: an exploit is composed of multiple 'elementary activities' and can be foiled at any of these, exploits require multiple operations on multiple vulnerable objects, and analyzing vulnerabilities can result in detecting predicates necessary to achieve security. They then develop a finite state machine modeling technique by which vulnerabilities are composed of a number of primitive finite state machines. Finally, they use these primitive finite state machines to model existing vulnerabilities and detect a new vulnerability. For their Bugtraq analysis, they use the taxonomy that is part of the Bugtraq database. Unfortunately, this taxonomy is not well applied: they note several instances when the same coding mistake is classified in a number of different taxonomy categories [CKXI03]. In a separate, although related work, Chen *et al.* present a more detailed analysis of the vulnerabilities recorded in the Bugtraq database. They combine this information with an examination of source code to detect and analyze the prevalence of vulnerabilities in application code versus operating system code. They also draw inferences as to the trustworthiness of various operating systems. Finally, they examine SNORT intrusion alert databases to detect the most common alerts raised by that IDS in real world situations [CKI03]. The Bugtraq taxonomy is similar to that of the NVD.

6.2 The NVD classification

The NVD's system is not an attempt at a formal taxonomy: it allows vulnerabilities to be in more than one category and some vulnerabilities may not fit in any category.

Despite these shortcomings, this system is widely used in the literature: the VDM research both from Purdue University and from Colorado State University has relied upon the NVD categories [GS05; WAM06b; WAM06a]. I therefore applied this system to the OpenBSD dataset in order to more readily compare it with this other literature.

The NVD classifies each vulnerability within the categories shown below [ICA02]. Unfortunately, the NVD system uses the term 'error' in the title of most of its categories. This usage is contrary to the standard definition used in software engineering; a better term would have been 'fault' (see Section 2.1).

- **Access Validation Error (AVE)**: The access control system is faulty, not just misconfigured.
- **Configuration Error (CE)**: Caused not by "how the system was designed but on how the end user configures the system. We consider it a configuration error when a system ships from a developer with a weak configuration."
- **Design Error (DE)**: "there exists no errors in the implementation or configuration of a system, but the initial design causes a vulnerability to exist."
- **Exceptional Condition Handling (ECH)**: The system fails to safely handle an exceptional condition.
- **Environmental Error (EE)**: Caused by the system's environment. An example is "an unexpected interaction between an application and the operating system or between two applications on the same host." The "installation environment somehow violates the developer's security assumptions."
- **Input Validation Error (IVE)**: A system does not properly check its input and thus can be exploited by malicious/malformed input. This is the catch-all category for input validation vulnerabilities that do not fit either of the next two categories.
  - **Boundary Condition Error (BCE)**: Input can cause the system to "exceed an assumed boundary." "For example, the system may run out of memory, disk space, or network bandwidth." Integer overflows and divide by zero are also included in this category.
  - **Buffer Overflow (BO)**: Stack and heap buffer overflows.
- **Not Classified (NC)**: For vulnerabilities that were either not classified or were considered to not fall into any category.
- **Race (R)**: A race condition involving a security check. *E.g.*, changing the environment in the time between a security check and the approved operation.

I have applied the NVD classification system to the OpenBSD dataset. When possible, I have used the category provided by the NVD itself. If a detection event is not in the NVD, then I have categorized it according to the descriptions above.

	Version in which the vulnerability was first released													
Type	2.3	2.4	2.5	2.6	2.7	2.8	2.9	3.0	3.1	3.2	3.3	3.4	3.5	All
NC	12	2	1	1										16
AVE	9			1										10
BCE	8	2		2					1		1		1	15
BO	31	3	4	3	1		1	1		1	1	1	1	48
CE	3	1		2										6
DE	11	7	1	1	3		1	2		1			1	28
ECH	8	1			2		1	1		1				14
EE	3						1		1					5
IVE	10			1	1		1	1	1	1				16
R	10				1									11
Total	105	16	6	11	8	0	5	5	3	4	2	1	3	169

Table 6.1: *The number of OpenBSD detection events of each type in the NVD classification system, by version. A single detection event may be counted under multiple types, so the totals may differ from those in Table 5.1.*

However, I have left those entries that the NVD categorized as not classified (NC) unchanged.

Table 6.1 shows the breakdown of the OpenBSD vulnerability detection events into the NVD categories, by version. Each entry indicates the number of detection events of that type and release version. For example, the fifth row of the second column indicates that there were 31 detection events containing buffer overflow (BO) vulnerabilities with a release version of 2.3. The total number of vulnerability detection events in each version differs from that shown in Table 5.1, because detection events may be classified in multiple categories. The table does not include versions more recent that 3.5: all of the vulnerabilities detected during the study were released in version 3.5 or earlier. I have thus excluded from the table the blank columns for later versions.

The most frequent type of vulnerability detection event was buffer overflows (BO): 48 of the total of 169 classifications. The second most frequent type was design errors (DE), with 28 detection events classified in that category. Several types apply to 14–16 detection events: not classified (NC), boundary condition errors (BCE), error condition handling (ECH), and input validation errors (IVE). The least frequently assigned categories were environmental errors (EE) and configuration errors, with five and six respective detection events.

Table 6.1 shows the number of vulnerability detection events of each type by version. Another way of considering this data is by the date known. Are there trends in the type of vulnerabilities detected over time?

Figure 6.1 shows the OpenBSD vulnerability detection events according to both

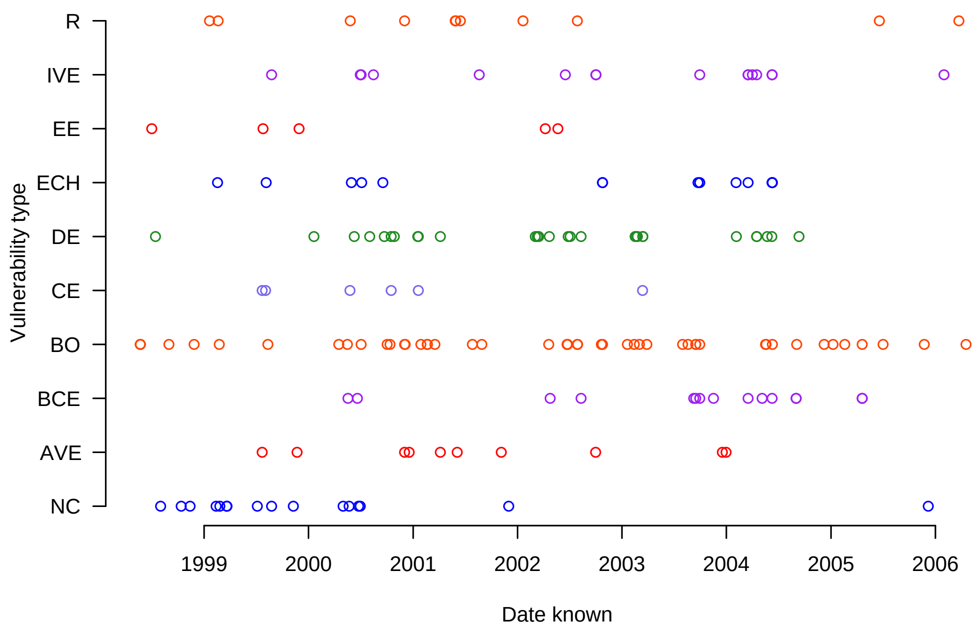


Figure 6.1: *Vulnerability detection events' types in the NVD classification system* vs. *their date known. A single detection event may be shown as having multiple types.*

category and the date known. Circles at the same position on the vertical axis represent detection events with vulnerabilities of the same type. The location of the categories on the vertical axis is alphabetical, with those not classified (NC) at the bottom. For example, the first detection event in the study, in mid-1998, contained vulnerabilities categorized as buffer overflows (BO). It is shown as the leftmost circle on the figure, vertically aligned with the label BO on the left.

The figure shows that buffer overflows (BO) were detected regularly throughout the study. In contrast, environmental errors (EE) and configuration errors (CE) were detected only occasionally and mostly in the first half of the study. The 'not classified' (NC) category was assigned mostly to vulnerabilities detected early in the study. This usage is probably because those entries were added to the CVE list retrospectively: knowledge of their existence predated the existence of the list. Boundary condition errors (BCE) were detected more frequently in the latter half of the study, because integer over/underflows became known and more prominent during this period.

6.3 The custom OpenBSD dataset classification

I have also created a classification system specifically for the OpenBSD dataset. Starting with the NVD system, I created additional categories whenever there were three or more similar vulnerabilities. I do not propose this classification system for general use; it is intended only to reveal possible patterns in vulnerability discovery in OpenBSD.

The OpenBSD vulnerability classification uses all of the categories from the NVD database. In addition, it includes the following categories:

- **Coding Mistakes (CM)**: a catch-all category encompassing mistakes not at the design level but at the implementation level.
- **Design Error, Interaction (DEI)**: a design mistake caused by the unanticipated interaction of different programs. It is a subset of Environmental Errors (EE); in my OpenBSD classification system, EE is reserved for vulnerabilities caused by environmental variables.
- **File Handling / File System (F)**: related to the use of temporary files or the program's interaction with the file system.
- **File Descriptor (FD)**: related to or caused by the use of file descriptors.
- **Format String (FS)**: caused by using unvalidated user input as a format string parameter.
- **Heap Corruption (HC)**: caused by a double free, memory leak, or some other mistake related to the (mis)allocation of memory on the heap.
- **Integer Overflow/Underflow (IOU)**: caused by an integer over/underflow.
- **Null Pointer De-reference (NPD)**: caused by an attempt to de-reference a null pointer.

Table 6.2 shows the breakdown of the OpenBSD vulnerability detection events into these custom categories, by version. This table is the same as Table 6.1, except the categories used are from my custom OpenBSD taxonomy. Each entry indicates the number of detection events of that type and release version. The total number of detection events in each version differs from that shown in Table 5.1, because detection events may be classified in multiple categories. The total number differs from that shown in Table 6.1 because detection events may be classified in only a single category in one taxonomy but in multiple categories in the other taxonomy. Again, the table does not include versions more recent that 3.5: all of the vulnerabilities detected during the study were released in version 3.5 or earlier.

As with the NVD system, the most frequent type of vulnerability detection event was buffer overflows (BO): 50 of the total of 169 classifications. Unlike the NVD system, the other categories contain relatively few detection events (fifteen or fewer). This more even distribution reflects the means by which the taxonomy was designed. I created categories if there were a minimum number of vulnerabilities of a similar nature. As a result, there are seven categories into which only four or five detection events are classified.

For example, there were a handful of detection events classified as either file/file

	Version in which the vulnerability was first released													
Type	2.3	2.4	2.5	2.6	2.7	2.8	2.9	3.0	3.1	3.2	3.3	3.4	3.5	All
AVE	3	1												4
BCE	4				1									5
BO	35	3	3	2	1		1	2			1	1	1	50
CE		1												1
CM	4		2		1		2							9
DE	5	4		1	2			1		1			1	15
DEI	4													4
ECH	2	1			1									4
EE	2		1	1					1					5
F	8	1	1	2	1									13
FD	5													5
FS	4													4
HC	2	2	1	2				1	1	2				11
IOU	7	1			1			1	1		1			12
IVE	5	2		2				1	1					11
NPD	1						1						1	3
R	12				1									13
Total	103	16	8	10	9	0	4	6	4	3	2	1	3	169

Table 6.2: *The number of OpenBSD detection events of each type in the custom classification system, by version. A single detection event may be counted under multiple types, so the totals may differ from those in Table 5.1.*

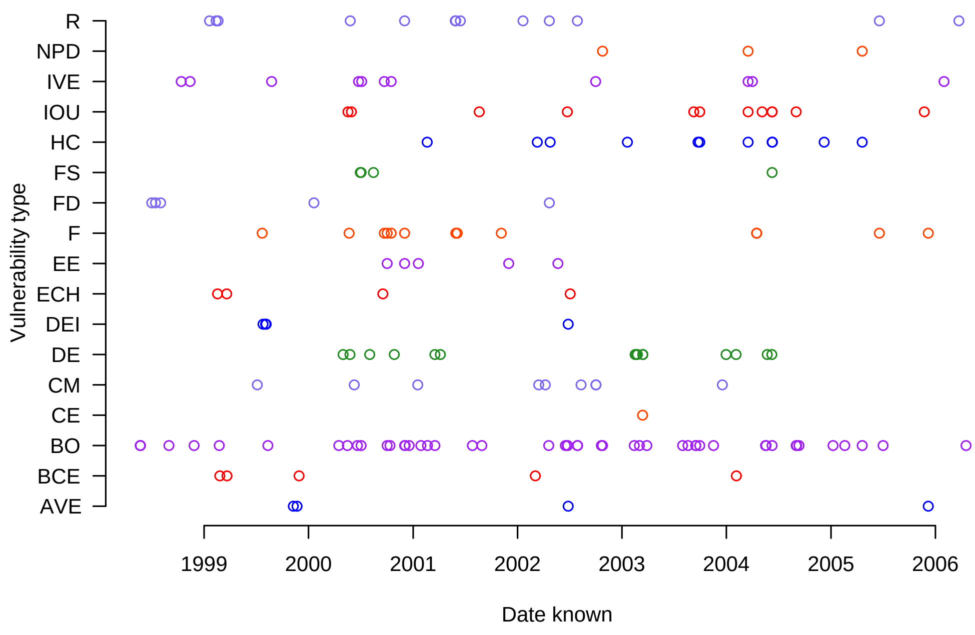


Figure 6.2: *Detection events' types in the custom OpenBSD classification system* vs. *their date known. (Detection events may be categorized under multiple types.)*

system (F) faults or as file descriptor (FD) faults. The NVD would not necessarily create these two categories because the number of vulnerabilities in the CVE list to which this category would apply is small. However, they seem to be over-represented in this study, and thus I thought it appropriate to create these two categories.

Figure 6.2 shows the OpenBSD vulnerability detection events according to both category and the date known. (It is the same as Figure 6.1, except that it uses the custom taxonomy rather than the NVD taxonomy.) Circles at the same position on the vertical axis represent detection events with vulnerabilities of the same type. The location of the categories on the vertical axis is alphabetical.

In this taxonomy, integer over/underflows (IOU) are separated from the remainder of boundary condition errors (BCE). As a result, this figure shows that IOUs became more common towards the end of the study, while BCEs became less common. This distinction was concealed by the NVD taxonomy used in Figure 6.1.

6.4 Conclusion

The research on vulnerability taxonomies has yet to produce a single, widely accepted, useful system. The existing vulnerability databases, such as the NVD, have thus relied upon makeshift systems that were probably created based upon an analysis of the data they contained at a given time. The resulting systems are hard to compare. Furthermore, a few categories such as buffer overflows or input validation errors are over-represented.

I have not constructed a better universal taxonomy. Instead, I have applied the widely used NVD taxonomy to the OpenBSD data set. I have also created and applied a related custom taxonomy based upon the similarity of different vulnerabilities in the data set.

Work on a next generation vulnerability database, as proposed in Section 3.7, should include work on an accompanying, developer-perspective taxonomy. That taxonomy should strive to achieve the goals enumerated by Edward Amoroso, including categories that are: mutually exclusive, exhaustive, unambiguous, repeatable, accepted, and useful [Amo94, p. 34]. A good taxonomy is critical to ascertaining whether or not the vulnerability detection is an independent process.

Chapter 7

The Independence of Vulnerability Detection

Vulnerability discovery models assume, among other things, that the data to which they are applied represents independent events. Unfortunately, vulnerability detection is frequently a dependent process. In Chapter 6, I categorized the OpenBSD data according to both the NVD and a custom taxonomy. In this chapter, I test the categorized OpenBSD data to see whether the detection of each type of vulnerability is a dependent process. If vulnerability discovery is dependent, existing SRMs may provide inconsistent results when they are applied to vulnerability data—and new VDMs that do not assume independence should then be developed.

7.1 Reasons for detection dependence

There are three primary causes for dependency in the detection of vulnerabilities: the discovery of a new type of vulnerability, a previously unconsidered location, or a new tool for finding vulnerabilities. Each of these causes can result in the discovery of large numbers of similar vulnerabilities in a short time span.

7.1.1 A new type of vulnerability

The first cause for dependency is the discovery/popularization of a new type of vulnerability. For example, three different file descriptor vulnerability detection events occurred in OpenBSD during one month in 1998. The vulnerabilities were in different areas of the operating system, were detected by different people, and were reported at different times. During the entire eight years of the study, only one other file descriptor vulnerability was detected. Although I have no documentary evidence, I suspect that the discovery of the first file descriptor vulnerability prompted detectors to look for similar vulnerabilities.

The first public post [Twi99] asserting the exploitability of format string faults was made in September, 1999 [Wik07]. These faults were thus now considered a vulnerability. Within a year, nine of these vulnerabilities were reported in OpenBSD.[1]

7.1.2 A new location in which to search for vulnerabilities

The second cause for dependency in the vulnerability discovery process is the discovery/popularization of a new target: either a subsystem or an entirely new system. For example, after five years of no vulnerabilities within CVS in OpenBSD, nine vulnerabilities were discovered in the CVS program within a year.[2] A detector reported one CVS vulnerability in May of 2004 [Ess04a]. Two weeks later, he and another detector reported another six vulnerability discovery events in CVS *while they were investigating the first vulnerability* [Ess04b].

The latter six were vulnerability detection events. Following the process described in Section 5.2, I combined several vulnerabilities into one of those detection events: *i.e.* that detection event contains multiple same-type vulnerabilities discovered by the same person in a short time period.

This example illustrates the difficulty in constructing a dataset of independent vulnerability detection events. Where do you draw the line? This problem is insurmountable when the data collection is distantly retrospective: sometimes the only way to achieve accuracy is to query the detectors.

7.1.3 A new tool with which to detect vulnerabilities

A third cause of dependence is when a tool is developed that enables detectors to discover vulnerabilities. Such tools can lead to the detection of vulnerabilities of different types, in different locations in a single program, or in different programs. The execution of such a tool is likely to lead to a cluster of vulnerabilities and thus may break the independence assumption made by VDMs.

For example, in 2002 researchers at the University of Oulu designed the Protos SNMPv1 test suite [Uni02]. This program is an example of a 'fuzzer': it generates sample inputs to programs in an attempt to find specific inputs that cause failures. This tool found at least four vulnerabilities of at least two different types.[3] Moreover, these vulnerabilities were present in a wide variety of different programs that implement the SNMPv1 protocol.

7.2 The independence of OpenBSD vulnerabilities

One way to assess the independence of the vulnerability discovery process is a Wald-Wolfowitz runs test [GS05]. This test considers a sequence of two values and indi-

[1]Including CVE identifiers: 2000-0751, 2000-0763, 2000-0993, 2000-0994, 2000-0995, 2000-0996, 2000-0998, 2000-0999, 2000-1010

[2]Including CVE identifiers: 2004-0180, 2004-0396, 2004-0405, 2004-0416, 2004-0417, 2004-0418, 2004-1471

[3]CVE identifiers: 2002-0012 and 2002-0013

		OpenBSD vulnerabilities			
		Foundational		All	
Type	Full name of type	#	p-value	#	p-value
NC	Not Classified	12	0.148	16	**0.000**
AVE	Access Validation Error	9	0.838	10	0.629
BCE	Boundary Condition Error	8	0.638	15	0.148
BO	Buffer Overflow	31	**0.031**	48	0.053
CE	Configuration Error	3	0.731	6	0.601
DE	Design Error	11	0.438	28	**0.007**
ECH	Error Condition Handling	8	0.064	14	0.084
EE	Environmental Error	3	0.731	5	0.662
IVE	Input Validation Error	10	0.972	16	0.235
R	Race condition	10	0.099	11	0.785

Table 7.1: *The number of detection events in the OpenBSD data set, according to the NVD taxonomy and the version released (foundation version or all versions combined). The p-values indicate the result of a Wald-Wolfowitz runs test. Those values less than 0.05 are in **bold**, indicating 95% certainty that detection events of this type were not independent.*

cates the likelihood that the sequence was generated randomly. If there are too many or too few 'runs,' in which the same value is repeated, then the sequence is unlikely to have been generated randomly.

To apply this test, the vulnerabilities are classified according to their type. The test is performed on each type of vulnerability: the two values are whether or not the vulnerability was of the type being considered. For example, consider a sequence of five vulnerabilities of the following types: BO, IOU, BO, EE, IOU. These vulnerabilities were discovered in the order in which they are listed. To test whether buffer overflows were discovered independently, the runs test is performed on the sequence: Yes, No, Yes, No, No. The 'yes' values indicate buffer overflows.

Table 7.1 shows the number of vulnerabilities of each category of the NVD classification system described in Section 6.2. Table 7.2 shows the number of vulnerabilities of each category of the OpenBSD classification system described in Section 6.3.

The two Tables are constructed the same. Each row represents vulnerabilities of a single category. The columns are divided into data for version 2.3 and all versions combined. (The data contains an insufficient number of detection events for statistical analysis of single versions other than the foundation version.) The 'all' column includes every detection event in every version. The main columns are divided in two: one sub-column to indicate the number of vulnerabilities of that type and the other to indicate the p-value produced by a Walf-Wolfowitz runs test. The p-values less than 0.05 are in bold: the detection events in that category occur

		OpenBSD vulnerabilities			
		Foundational		All	
Type	Full name of type	#	p-value	#	p-value
AVE	Access Validation Error	3	0.731	4	**0.002**
BCE	Boundary Condition Error	4	0.651	5	0.662
BO	Buffer Overflow	35	0.407	50	0.490
CE	Configuration Error	0	NA	1	NA
CM	Coding Mistake	4	**0.022**	9	**0.026**
DE	Design Error	5	0.105	15	**0.000**
DEI	Design Error, Interation	4	**0.000**	4	**0.000**
ECH	Error Condition Handling	2	NA	4	0.725
EE	Environmental Error	2	NA	5	0.662
F	File handling	8	0.638	13	**0.042**
FD	File Descriptor	5	**0.000**	5	**0.000**
FS	Format String	4	0.651	4	0.725
HC	Heap Corruption	2	NA	11	0.331
IOU	Integer Over/Underflow	7	0.428	12	0.287
IVE	Input Validation Error	5	0.105	11	0.129
NPD	Null Pointer Dereference	1	NA	3	0.788
R	Race condition	12	**0.004**	13	0.333

Table 7.2: *The number of detection events in the OpenBSD data set, according to the custom taxonomy and the version released (foundation version or all versions combined). The p-values indicate the result of a Wald-Wolfowitz runs test. Those values less than 0.05 are in* ***bold****, indicating 95% certainty that detection events of this type were not independent.*

in a dependent fashion, with a 95% confidence level. Large values do *not* necessarily indicate independence: the test can indicate one type of dependence, but no test can prove independence.[4]

Table 7.1 shows the results of the Walf-Wolfowitz runs test for vulnerability detection events categorized according to the NVD system. When applied to all detection events in the study, regardless of the version in which the vulnerabilities were introduced, it indicates that detection events of type NC and DE occur dependently. NC stands for 'not classified.' Many of the earlier vulnerabilities are not classified, because they predated the NVD. As a result, their dependence may be caused by the fact that most of them occurred early in the study. The table also shows that detection events vulnerabilities of type BO introduced in the foundation version occur dependently.

[4]The Walf-Wolfowitz tests in this chapter were performed using the R statistics program [R] and the function 'runs.test' within the package 'tseries.'

These results are less extreme than those found by Gopalakrishna & Spafford in their analysis of NVD data on vulnerabilities in IIS, BIND, Lpd, Sendmail, and RPC. They find that in one-third of the instances in which “two successive vulnerabilities are considered in a software product, they are of the same type.” Their total results for all of the categories of all of these products were: 24 dependent, 8 independent, and 8 unknown [GS05].

Table 7.2 shows the results of the Walf-Wolfowitz runs test for vulnerability detection events categorized according to the seventeen types in the custom classification system. When applied to all detection events in the study, regardless of the version in which the vulnerabilities were introduced, it indicates that detection events of type AVE, CM, DE, DEI, F, and FD occur dependently. When applied to detection events with vulnerabilities introduced in the foundation version, it indicates that types CM, DEI, FD, and R occur dependently.

7.3 Caveats

There are three potential concerns with the dependence results presented here.

First, we must be careful when considering the results for foundational vulnerabilities. Detectors do not look for vulnerabilities based upon the version in which the vulnerability was first released. Logically, then, some vulnerabilities whose discovery was dependent may have been introduced in different versions of OpenBSD: this dependence will not be visible when individual versions are considered. The results for all of the vulnerabilities, regardless of the version in which they were first introduced, are the best representation of the dependence of the process.

Second, these results depend upon the taxonomies used. Both taxonomies have shortcomings. The NVD taxonomy has too little granularity and thus is overly reliant on a few categories. The custom taxonomy was designed specifically for this data set, and it may therefore be excessively fine-grained.

Finally, dependency may be exhibited in multiple ways. It would be useful to test for other types of dependence, but first we need more information about what other forms dependence could take. The Wald-Wolfowitz test considers whether the data contains too many (or too lengthy) ‘runs’ of consecutive detection events of the same type. This test can indicate the dependence exhibited when the discovery of a vulnerability prompts the immediate discovery of a vulnerability of the same type.

However, the dependency of vulnerability discovery could be indicated in other ways than the sequential discovery of identical types. For example, it is possible that dependency in detection is not related to a vulnerability’s type but rather by its location. Unfortunately, the collection process for the OpenBSD data set has resulted in clusters of vulnerabilities from the same location. These clusters occur when a third-party vendor like the X.Org group releases a patch that bundles together fixes for several vulnerabilities, as discussed in Section 5.2.3. As a result, testing for locational dependency would be strongly biased by the collection methodology.

To effectively test each of the different aspects in which dependency might be exhibited, we need a better understanding of those aspects. A good next step for learning about dependence is to go to the source: a fruitful topic for future work may be to survey external detectors about their motivations and processes.

7.4 Effects of dependence

This analysis of detection events indicates that discovery is a dependent process, at least for some categorizes of vulnerability—even though the clearly dependent vulnerabilities were already consolidated into detection events, so the data analyzed above already had any obvious dependencies removed.

One way to illustrate the effects of dependence is to compare the foundational detection events with foundational vulnerabilities. This comparison will not reveal the dependence discovered above, but it will highlight how important dependence is for VDMs.

7.4.1 Comparing vulnerabilities with detection events

Each vulnerability is associated with a time-to-next-vulnerability (TTNV): the number of days that elapsed between the discovery of the prior vulnerability and the discovery of this one. Figure 7.1 depicts foundational vulnerabilities (not detection events) according to their date known. The vertical axis indicates the TTNV for each vulnerability. The blue circles are vulnerabilities that appear to be independent: they are not included in any detection events. The red 'x' marks indicate vulnerabilities that are clearly dependent: these vulnerabilities have been consolidated into detection events in the foundational data set. Most of the dependent vulnerabilities have small TTNVs. Indeed, one reason they are identified as dependent is that they are detected *soon* after another vulnerability of the same type.

Figure 7.2 provides another illustration of the effect of dependence. Again, the horizontal axis indicates the date known for each vulnerability, but this vertical axis indicates the cumulative number of foundational vulnerabilities or detection events. The top curve represents individual vulnerabilities. Once again, independent vulnerabilities are shown as blue circles and dependent vulnerabilities are shown as red 'x' marks. The bottom curve shows detection events, with each detection indicated by a green '+' mark.

The bottom curve thus incorporates all of the vulnerabilities in the top curve: it simply consolidates the dependent vulnerabilities into single detection events. A visual examination suggests that the variability in the top curve is higher than in the bottom curve. The bottom curve also appears to exhibit some kind of 'software security' growth. In other words, the rate of detection appears to be declining over time.

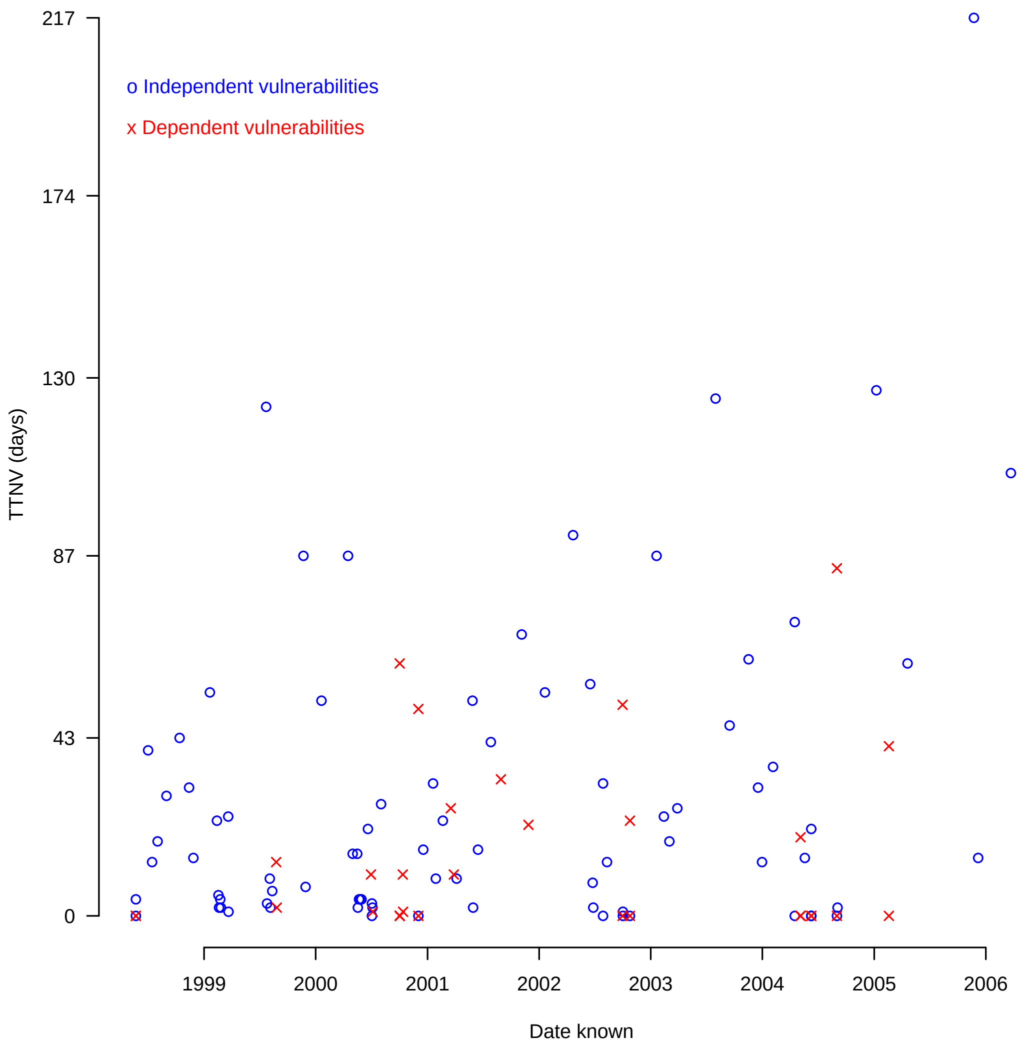


Figure 7.1: *The TTNV of foundational vulnerabilities, with detection events broken down into their constituent dependent vulnerabilities.*

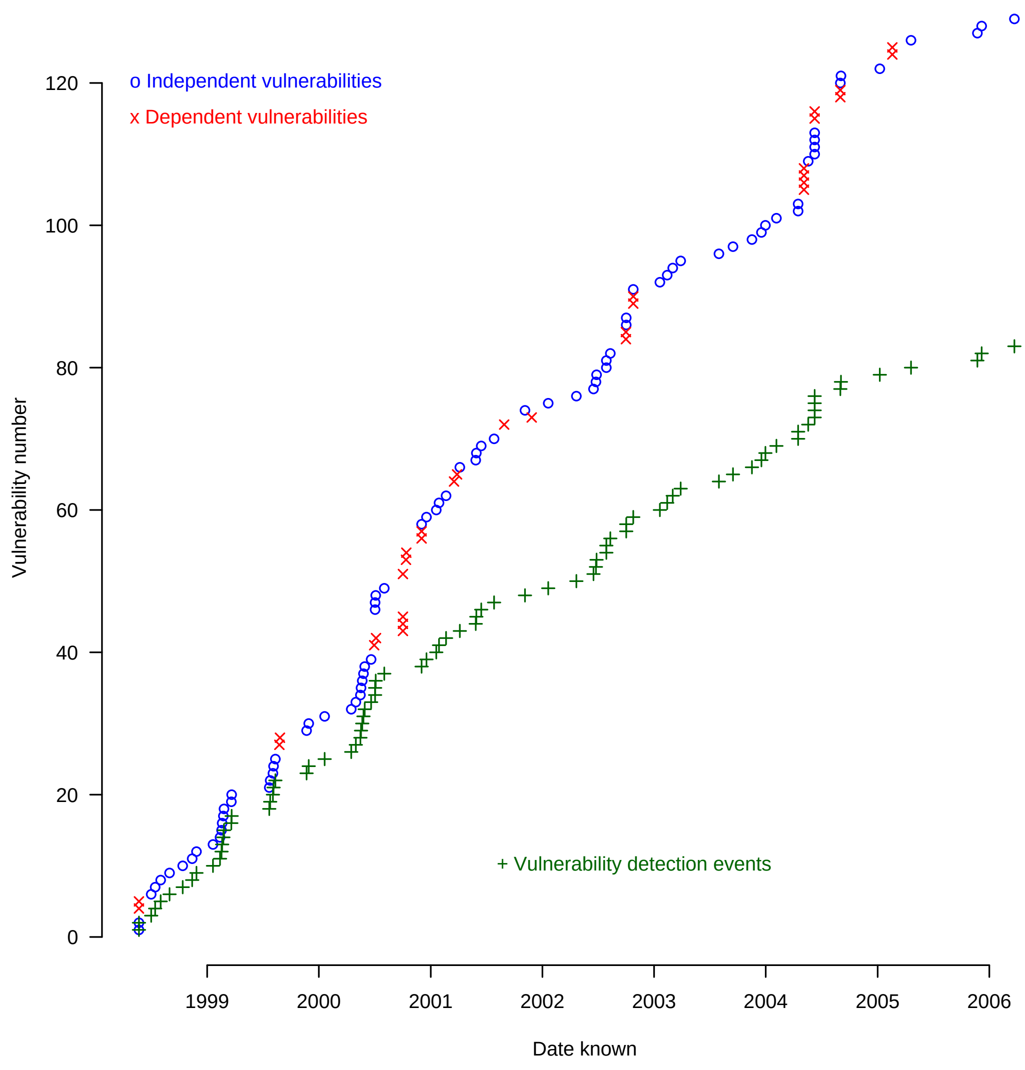


Figure 7.2: *The cumulative number of foundational vulnerabilities and detection events over time. The two curves are different representations of the same data. The top curve depicts vulnerabilities, while the bottom curve groups dependent vulnerabilities into detection events.*

	Successfully fitted models			
	Littlewood/Verrall (Q)		Littlewood/Verrall (L)	
Statistic	Value	(Rank)	Value	(Rank)
Bias (u-plot)	0.18	(1)	0.18	(1)
Trend (y-plot)	0.12	(1)	0.12	(2)
Relative accuracy (PLR)	228.67	(1)	228.86	(2)
Overall Rank		(1)		(2)

Table 7.3: *Applicability results for the two successfully fitted VDMs applied to foundational detection events.*

7.4.2 Applying VDMs

Dependence in the vulnerability discovery process violates an assumption common to most VDMs. As a result, these models cannot be trusted to consistently produce accurate results. The application of these models to first the foundational vulnerability data and then to the foundational detection event data is illustrative of this problem.

I took seven commonly used SRMs and applied them, unchanged, as VDMs to the foundational data. None of the seven could be successfully fitted to the foundational vulnerabilities. However, two of the seven were successfully fitted to the foundational vulnerability detection event data: Littlewood & Verrall (linear), and Littlewood & Verrall (quadradic). The consolidation of thirty-two dependent vulnerabilities into fourteen detection events was significant enough to enable two models to fit the data.[5]

Table 7.3 provides applicability results for these two successfully fitted models (the applicability tests were discussed in Section 3.3). They estimate an end-of-study mean TTNV of 44.8 and 52.0, respectively. They also estimate that an additional 67.6 and 51.8 detection events will occur in the next ten years, respectively.

The successful fit and one-step-ahead prediction of these two models may indicate that the remaining dependency in the data is not a problem. However, in an earlier work, these two successful models had *not* fit an incomplete version of this dataset, which covered less time and contained seven fewer detection events. For that subset, a different model was successful: Musa's logarithmic [OS06b]. The fact that no model has been consistently accurate and successfully fitted to the detection events could be caused by residual dependency in the data; alternatively it is conceivable that other assumptions, such as time and effort, are not adequately met.

[5]The models were applied using the SMERFSˆ3 reliability modeling tool [Sto03]. The seven models tested were: Moranda's geometric, Jelinski-Moranda's de-eutrophication, Musa's logarithmic, Musa's basic, Non-homogeneous Poisson, Littlewood & Verrall (linear), and Littlewood & Verrall (quadradic). See [MIO87; AIA93; Far96] for descriptions of these models.

7.5 Conclusion

The OpenBSD vulnerability data set was carefully constructed in an attempt to identify and eliminate dependence: when vulnerability discovery was clearly dependent, those discoveries were combined into a single detection event in the data set. Nonetheless, the application of the runs test indicates that some of the detection events in the data set still occur dependently. Even the categories whose detection events were not found to be dependent may still be so: other tests may find different types of dependence. This result violates the basic assumption of data independence that underlines VDMs.

The application of VDMs to the foundational vulnerability data—as I have done in previous work [Ozm05; Ozm06; OS06b]—is thus theoretically unsound. We cannot rely upon the predictions of models applied to this data, as long as those models require independent events. This result also calls into question the other literature on VDMs: none of the existing work that utilizes VDMs has tested its data for independence.

At the same time, this work highlights the need for a better database and a better taxonomy, as discussed in Section 4.3. Both of the taxonomies used here have shortcomings that may affect these results.

Chapter 8

Source Code Evolution & Vulnerability Density

Existing VDMs presume that vulnerability discovery is an independent process. The evidence presented in Chapter 7 contradicts this assumption and suggests that we need new models that do not assume independence. However, even without such models, a careful examination of existing vulnerability data and other characteristics of the software system can provide useful information for efforts to secure that system.

In this chapter, I analyze the evolution of OpenBSD's source code during the study, and I consider the results of that analysis in combination with vulnerability data. In Chapter 9, I take this information and consider whether or not the rate of vulnerability detection in OpenBSD is decreasing over time.

A startlingly high 63% of the 155 detection events in the OpenBSD data set were foundational vulnerabilities. Why have so few vulnerabilities been reported that were introduced in later versions? Alternatively, why were so *many* of the reported vulnerabilities introduced in the foundational version? One possible explanation is that, even after eight years and seventeen new versions, the foundational code still dominates the source code.

In order to better understand how the source code has evolved over time, I analyzed the collective changes to the OpenBSD code repository. I thus establish how much code was altered/introduced in each version.

8.1 Methodology

I first pre-processed each version of the source code. Only files with the suffix `.c` or `.h` were retained, and all comments were stripped. Furthermore, files whose name included keywords indicating that they belonged to an architecture other than x86 were removed.

After pre-processing was completed, each version was compared with each successive version. I used `diff` to compare files with the same path and filename. The `diff` tool was instructed to ignore changes in whitespace or the location of line breaks.

The OpenBSD development team sometimes moved or copied files, which is difficult to track via CVS. To detect copies and moves, files with the same name but different paths were also compared. If they were found to be identical, I copied the file in the earlier version to the directory in which it was found in the later version. (These copies were used only to determine if code in future versions derived from earlier versions: they were not used to calculate the total line count.)

The estimate of code commonality is highly conservative. The `diff` tool marked code lines as changed even for trivial alterations like global variable renaming and some types of reformatting—and the OpenBSD team has been reformatting the code base. In addition, this process will indicate that all of the code in a file is new if that file was moved/copied and then had just one line altered between versions. (Recall that the automated comparison process only understands that a file was moved if the file in the new location is an exact copy of the file in the old location.) Furthermore, if the name of a file is changed then all of the code in that file is considered to be new. The comparison data will thus understate the degree to which later releases are composed of substantively unchanged code from earlier releases.

This analysis was performed for a previous work [OS06b], so it only encompasses fifteen versions of OpenBSD: 2.3 – 3.7. The vulnerability data set described in Chapter 5 includes vulnerability information on all of those versions plus an additional three versions.

8.2 Source code composition

Table 8.1 illustrates the proportion of each version of OpenBSD that is derived from earlier versions. Each column represents a composite version; each row represents a source version that contributes code to the composite. Values represent the percentage of the lines of code in the composite version that originate in the source version.[1] A line of code in a composite version of OpenBSD is said to originate in a source version if the line was last modified in that source version.

For example, the fifth column breaks down the composition of OpenBSD version 2.7. The top row of the column indicates that 6% of the lines of code originate in that version: they were either altered since the prior version or have been newly introduced. The second row from the top shows that 9% of the source code was altered/introduced in the prior version, 2.6, and was not changed after that version. The bottom row indicates that the bulk of the code in version 2.7 (79%) was both present in and remains unchanged since the foundation version.

[1] Because the percentages were rounded, the total percentage for each version may not exactly equal one hundred.

Source version	2.3	2.4	2.5	2.6	2.7	2.8	2.9	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
3.7															4
3.6														3	3
3.5													7	6	6
3.4												4	4	3	3
3.3											2	2	1	1	1
3.2										2	1	2	1	1	1
3.1									5	5	5	4	3	3	1
3.0								4	3	3	3	2	2	2	2
2.9							14	13	12	11	11	10	8	8	7
2.8						3	2	3	2	2	2	2	2	1	1
2.7					6	6	4	4	3	3	3	2	2	2	2
2.6				9	9	9	7	6	6	6	6	5	5	5	5
2.5			3	2	2	2	1	1	1	1	1	1	1	1	1
2.4		4	4	3	3	3	2	2	2	2	2	1	1	1	1
2.3	100	96	94	86	79	77	70	67	66	66	65	64	62	61	61

Composite version

Table 8.1: *The percentage of each version of OpenBSD that is composed from earlier versions. Columns represent* composite *versions of OpenBSD, whereas rows represent the* source *versions of OpenBSD from which they are composed. Each value in the table is the percentage of lines in the composite version that were last modified in the source version.*

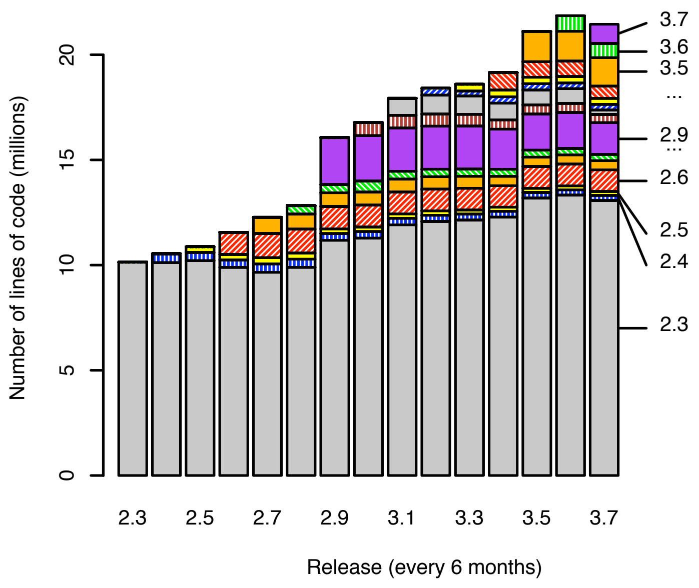


Figure 8.1: *The composition of the full source code. The composition of each version is broken-down into the lines of code originating from that version and from each prior version. (This is a graphical representation of Table 8.1.)*

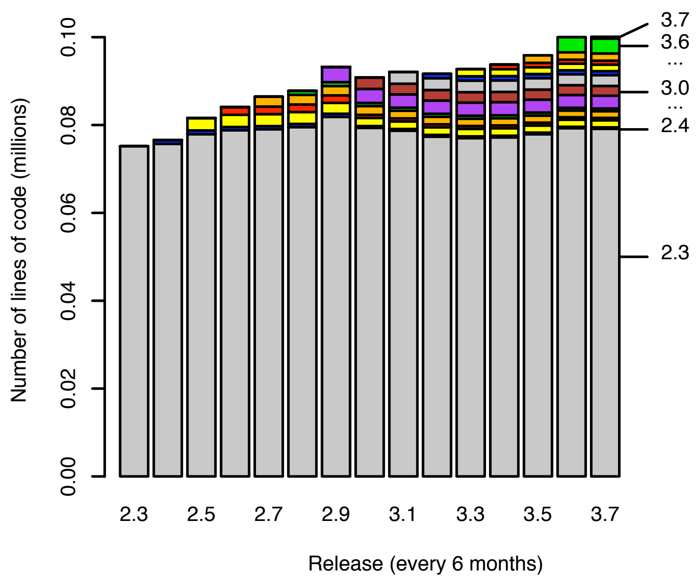


Figure 8.2: *The composition of the source code directory within the kernel (sys/kern) that had the most vulnerabilities. The composition of each version is broken-down into the lines of code originating from that version and from each prior version.*

Figure 8.1 is a graphical representation of Table 8.1: it shows the composition of each version, using the number of lines of code rather than percentages. Version 2.3 is composed of a single bar: by definition, all code in this foundation version is said to originate in it. For each successive version, a new bar is added to represent the number of lines of code that were altered/introduced in that release. The entries in Table 8.1 correspond to the colored bars that appear above the gray version 2.3 bar in each successive version shown in Figure 8.1.

Several large alterations/introductions of code stand out in Figure 8.1: in versions 2.6, 2.9, and 3.5. The magnitude of the changes in versions 2.6 and 3.5 is primarily due to a large number of files being renamed and slightly altered. This comparison methodology thus overstates the number of new lines of code and understates the contribution of code derived from earlier versions. The changes in version 2.9 are caused in part by the renaming of files; however, they were also the result of a major upgrade of the XFree86 package.

Another surprise is that the number of lines of foundational code fluctuates both downwards and *upwards*. However, increases in the number of lines of foundational code are readily explained: source files unaltered since the foundation version were copied and used in other areas of the code.

Of all the second-level source code directories, the sys/kern directory contains the largest number of detection events. Seventeen of the nineteen detection events in this portion of the kernel were introduced in the foundation version. Figure 8.2 shows the evolving composition of the source code in the sys/kern directory. Many of the vulnerabilities in this subsystem have been in code related to the processing of signals. Although this subsystem is part of the kernel, it does not include networking, file system, or virtual memory code. The code in one of the networking portions of the kernel (sys/netinet) has contributed thirteen detection events during the course of the study, seven of which are foundational.

The most startling result presented here is that the foundational code comprises a majority of the OpenBSD operating system, even after fourteen newer versions have been released—and this estimate is highly conservative.

8.3 Vulnerability density

Reliability engineers have investigated the possibility of a relationship between 'software size metrics' (*e.g.* the number of lines of code) and the number of faults in a system. The **fault density** is the number of faults per some measure of software size. For example, the number of faults per one thousand lines of non-comment source code (KLOC). Some have argued that any well-written code can be expected to have a fault density that falls within a certain range, *e.g.* 3–6 faults per thousand lines of code (KLOC) [Hat97].

The related term for vulnerabilities is vulnerability density: The **vulnerability density** is "the number of vulnerabilities per unit size of code" [AM05b]. Note that calculated fault or vulnerability densities are estimates: we do not know the real number of faults or vulnerabilities in a system. In this section, I consider the density of vulnerability detection events instead of vulnerabilities. As in prior chapters, a vulnerability detection event is assigned the release date of the earliest injected vulnerability that it includes.[2]

8.3.1 Linear relationship

Is there a linear relationship between the number of lines of code altered/introduced in a version of OpenBSD and number of vulnerabilities introduced in that version?

As I do not know the actual number of vulnerabilities present, I consider the number of detection events that occurred within five years of release for each version that is at least five years old. The number of detection events reported during this period is shown in the third column of Table 8.2. The fourth column contains the vulnerability detection event density for events in the five year period after each version's release. In this instance, densities are reported in units of detection events per millions of lines of code (MLOC).

Figure 8.3 illustrates the relationship between the number of vulnerability detection events and the number of lines of altered/introduced code. The standard correlation test (Pearson's ρ) is not applicable because I do not have enough data points. A non-parametric correlation test, Spearman's ρ, is unable to reject the null

[2]An earlier version of the work performed in this section was done collaboratively, with Stuart E. Schechter, and published as [OS06b].

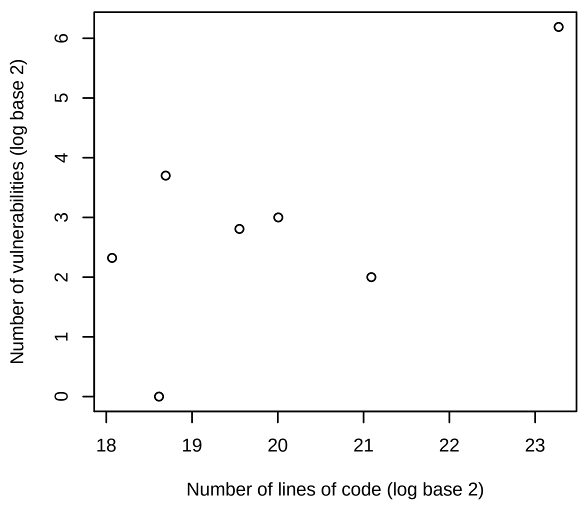


Figure 8.3: *The number of vulnerability detection events introduced and reported within five years of release compared to the number of lines of code altered/introduced, by version.*

hypothesis that there is no correlation: it calculates a correlation coefficient of 0.46 and a p-value of 0.30.[3]

I am thus unable to find a significant correlation between the number of lines of altered/introduced code and the number of vulnerability detection events.

8.3.2 Trends over time

The vulnerability detection event density of code added in new OpenBSD releases could provide an indication of the success of their efforts to produce secure code. On the other hand, code added by the OpenBSD team often provides security functionality: *e.g.* OpenSSH. As a result, that code is likely to attract a disproportionate share of attention from individuals searching for vulnerabilities; this extra attention may account for any differences between the versions' detection event densities.

For each release, Table 8.2 shows the number of vulnerability detection events, the number of lines of code altered/introduced (in millions), and the detection event density. The third column shows the number of vulnerability detection events within five years of each version's release, and the fourth column shows the corresponding detection event density. The fifth column shows the number of vulnerability detection events during the entire study, and the sixth column shows the corresponding detection event density.

The detection event density of the foundation version is in the middle of the pack. Versions 2.4 and 2.5 stand out for having the highest detection event densities (35.7 and 21.4 reported per million lines of code at the end of the study, respectively).

[3]A correlation coefficient of 1 indicates a positive linear correlation, -1 a negative correlation, and 0 indicates no correlation. This test was performed using the function 'cor.test' in R [R].

		Within 5 yrs of release		By end of study	
Vers.	MLOC	Events	$\frac{\text{Events}}{\text{MLOC}}$	Events	$\frac{\text{Events}}{\text{MLOC}}$
2.3	10.14	73	7.2	97	9.6
2.4	.42	13	31.0	15	35.7
2.5	.28	5	17.0	6	21.4
2.6	1.05	8	7.6	9	8.6
2.7	.77	7	9.1	8	10.4
2.8	.40	0	0.0	0	0.0
2.9	2.23	4	1.8	4	1.8
3.0	.63			5	7.9
3.1	.81			2	2.5
3.2	.33			3	9.1
3.3	.32			2	6.3
3.4	.83			1	0.0
3.5	1.44			3	2.1
3.6	.74			0	0.0
3.7	.91			0	0.0
Total	21.30	110	5.2	155	7.3

Table 8.2: *Vulnerability and code modification statistics for each version of OpenBSD. MLOC is the number of lines of code (in millions) altered/introduced in each version. Events are vulnerability detection events.*

The large ratio of vulnerability detection events per line of code in version 2.4 seems to support the intuition that code providing security functionality is more likely to contain vulnerabilities. Version 2.4 saw the introduction of the Internet Key Exchange (IKE) key management daemon (isakmpd, two vulnerabilities introduced) and OpenSSL (six vulnerabilities introduced). As a result, the new code added in that release may have drawn particular attention from vulnerability hunters.

The density of vulnerability detection events for code originating in versions 2.6, 2.9, and 3.5 are lower in part because of the inflated new-code counts for those versions (see Section 8.2).

8.4 Conclusion

When calculated per *thousand* lines of code, rather than per million, the density of *all* detection events ranged from 0–0.0357 and averaged 0.0073. As noted above, some software engineers estimate the fault density of well-written code to be 3–6 per thousand lines of code [Hat97]; these detection event densities are three orders of magnitude less than that amount. The two figures are not necessarily contradictory:

'faults' include both quality faults and vulnerabilities, and a handful of detection events included multiple vulnerabilities.

The majority (97 of 155, or 63%) of the vulnerabilities found during the study are foundational; that is, they were born prior to the release of the foundation version. I considered two hypotheses to explain why reported vulnerabilities were so often foundational: foundational code might be of lower quality than more recent code, or foundational code may constitute the bulk of the total code base.

The source code history data supports the latter hypothesis. Even after 7.5 years and 14 newer versions, the foundation version dominates the overall source code: at least 61% of the lines of code in version 3.7 are foundational, unchanged since the release of version 2.3. The foundational version thus contributes 61% of the source code and 63% of the detection events in the study. As a result, the security of the foundation version may still be driving the overall security of OpenBSD.

Chapter 9

Are Vulnerability Reporting Rates Declining?

Dependence in vulnerability detection—and in the existing data on detection—prevents the use of most current VDMs. However, even without explicit models, can we learn anything important from the OpenBSD data set? One of the questions we can consider is whether or not the rate of vulnerability detection in OpenBSD is declining over time.[1]

Many in the security research community have criticized both the insecurity of software products and developers' perceived inattention to security. However, we have lacked quantitative evidence that such attention can improve a product's security over time. Seeking such evidence, I investigate whether efforts by the OpenBSD development team to secure their product have decreased the rate at which vulnerability detection events occur.

9.1 Social utility of vulnerability detection

In an original and intriguing work, Rescorla challenged a common assumption held by the security community: he argues that the post-release detection of vulnerabilities results in an overall loss to society [Res04]. To reach this conclusion, Rescorla models both the costs suffered by users after a disclosure (either responsible or instantaneous) and the benefits it provides.

9.1.1 Cost-benefit analysis

He first notes that users patch their systems slowly (even when it is known that a vulnerability is being exploited), and some may not patch at all [Res03]. At the same time, a vulnerability report or a patch will inform attackers of a vulnerability and help them to create an exploit for that vulnerability. By forcing vendors to

[1]An earlier version of the work performed in this chapter was done collaboratively, with Stuart E. Schechter, and published as [OS06b].

release patches, external detectors ensure that attackers will also be aware of the vulnerability. Since many systems remain unpatched for some time after the patch is released, their users will likely suffer the effects of attacks (although the unpatched systems may be those least valued). All disclosure thus leads to a significant social cost.

Rescorla also questions the argument that benign detectors should seek to find vulnerabilities because those vulnerabilities may *already* have been found and utilized by attackers. He believes that an attacker cannot exploit a vulnerability many times before the vendor learns about the vulnerability: the attacker will eventually attack a system that is being carefully watched, thus alerting the system administrators to the vulnerability.

In addition, Rescorla was unable to fit VDMs to NVD data on three of the four operating systems he considered. He also applied a Laplace test to his data, which reveals that there is no trend towards a decreasing rate of reporting for the same three out of four operating systems (Section 9.4 describes this test). He thus concluded that there is no decrease in the rate of vulnerability reporting: the pool of vulnerabilities in a product is essentially infinite with respect to the lifespan of the product. Therefore, an attacker is unlikely to independently rediscover a vulnerability that has been previously discovered by a benign detector—the pool of vulnerabilities is too large. This result indicates that benign detectors provide no real benefit: they are not racing to find and fix vulnerabilities before those same vulnerabilities are found by malicious detectors. Instead, detectors are all finding different vulnerabilities.

Using these assumptions, Rescorla modeled the social utility of vulnerability hunting by benign external detectors and concludes that it is not socially beneficial.

9.1.2 Analysis

However, there are three important sources of potential error in Rescorla's work: independence, rediscovery, and noisy data.

First, Rescorla assumes that vulnerability detection is independent. Chapter 7 presents evidence that vulnerability detection in OpenBSD is not independent. If Rescorla's data is not independent, then the tests that Rescorla used are inapplicable and no conclusion can be drawn from his data.

Second, he assumes that independent rediscovery, when independently working detectors discover the same vulnerability, is unlikely. In reality, Rescorla notes that independent rediscovery happens on occasion, and he thus concludes that some detection is not stochastic. However, he finds this situation rare enough to exclude it from the cost-benefit analysis. Chapter 10, provides evidence that independent rediscovery occurs too frequently to be dismissed and discusses this phenomenon in more depth.

Third, Rescorla notes that his NVD data set is noisy and problematic, and he recommends the collection of better data. This chapter answers his challenge by utilizing the OpenBSD dataset and investigating whether the rate of detection events in OpenBSD is declining. I consider the median lifetime of vulnerabilities in different versions of OpenBSD, look at overall detection event trends, and finally employ a Laplace test on data for foundational vulnerabilities.

9.2 What is the median lifetime of a vulnerability?

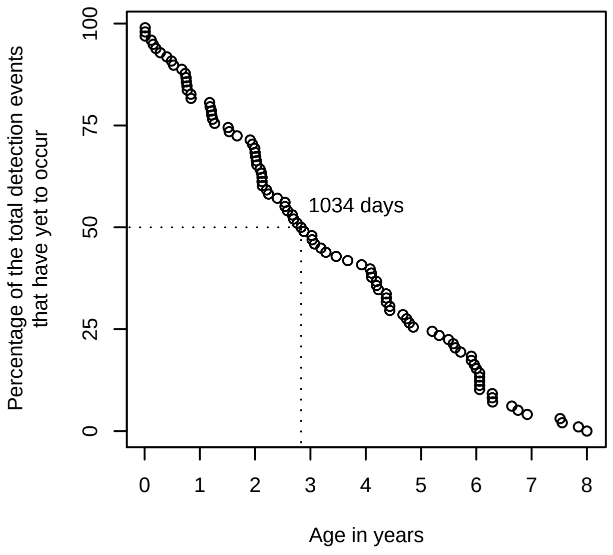


Figure 9.1: *The lifetime of foundational detection events during the study.*

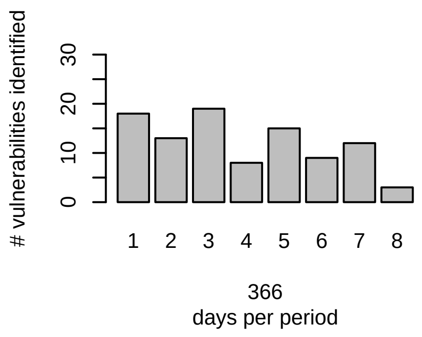
Version	Median Lifetime
2.3	877
2.4	1337
2.5	502
2.6	551
2.7	216

Table 9.1: *Median lifetime in days of vulnerabilities detected within the first six years of a version's release.*

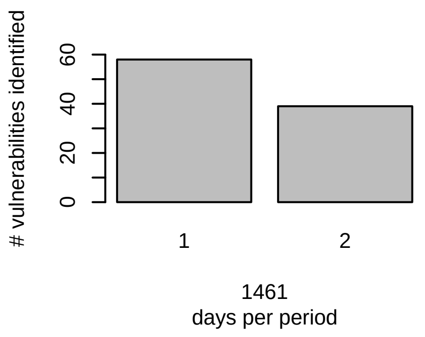
Rescorla [Res04] applies an exponential model to his data, so he is able to ascertain the half-life of the vulnerability sets he considers: those half-lives range from 10 months to 3.5 years. Without an exponential model of my data, I am not able to ascertain, in a formal sense, the half-life of vulnerabilities in OpenBSD. Instead, I calculate the median lifetime of vulnerabilities in vulnerability detection events: the time elapsed between the release of a version and the detection of half of the vulnerabilities detected in that version.

Figure 9.1 plots the age, at detection event, of foundational vulnerabilities. The data is necessarily right censored: I do not know that I have found all of the vulnerabilities in the foundation version. This data provides a *lower-bound* of 2.8 years (1034 days) on the median lifetime of foundational vulnerabilities. More foundational vulnerabilities will undoubtedly be found, so this value may increase but will never decrease.

Is the median lifetime of vulnerabilities decreasing in newer versions? Table 9.1 depicts this time for those vulnerabilities detected within six years of the release of versions 2.3 through 2.8; this data relies upon the gross simplifying assumption that



(a) The number of foundational detection events during each eighth of the study.



(b) The number of foundational detection events during each half of the study.

Figure 9.2: *Foundational detection events divided into periods according to their date known*

all vulnerabilities present were found within six years of each version's release. (I make this assumption in order to include the same time span after release for each version.) The results do not indicate a trend.

The most striking part of this analysis is that the median lifetime of vulnerabilities is so long. Vulnerabilities have remained in the source code, unnoticed, for years—despite code reviews.

9.3 Illustrating detection event trends

We can only reliably analyze the rate of detection events if we have sufficient data points. In this section, I consider the foundation version, for which I do have adequate data.

9.3.1 Vulnerability detection events per interval

A simple means of examining the detection event rate is to categorize detection events by the time period in which they were occurred. To do so, I divide the study into periods of equal length.

The columns in Figure 9.2(a) represent the number of vulnerabilities detected in each of eight, equal-length periods of approximately a year (366 days). Visual inspection does not indicate a convincing decrease in the rate of detection events. Somewhat more convincing results can be obtained by dividing the study period into halves, as shown in Figure 9.2(b). The number of detection events does declines from the first period (58 vulnerabilities) to the second (39 vulnerabilities).

Neither figure provides significant evidence that the rate of reporting has declined. However, the latter figure does suggest it. More convincing results would rely upon confidence intervals. Unfortunately, the evidence for dependence presented

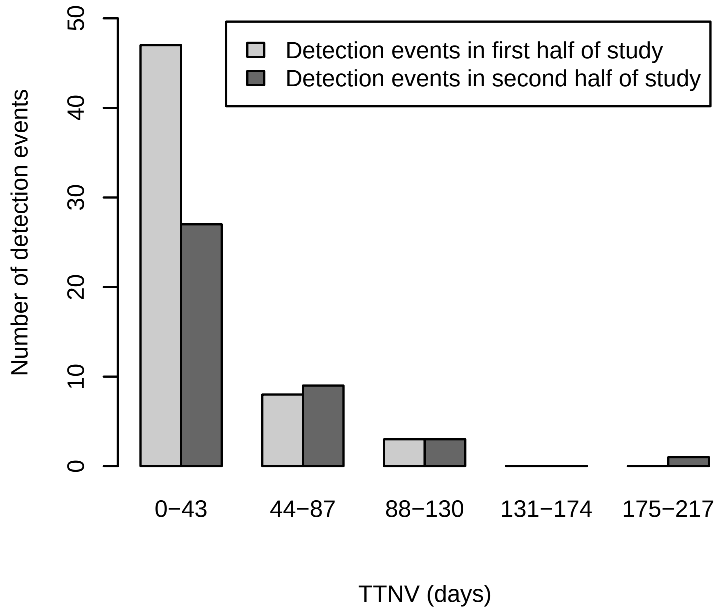


Figure 9.3: *The number of days between foundational vulnerability detection events in the first half of the study compared with events from the second half.*

in Chapter 7 prevents the ready assumption of an underlying statistical model from which to obtain such confidence intervals.

9.3.2 Time to next vulnerability

Another way to examine the frequency of vulnerability reports is to measure the time that passes until the discovery of the next vulnerability: the time to next vulnerability (TTNV). As the foundation version aged, did the TTNV change?

Figure 9.3 groups foundational vulnerability reports by their TTNV. Each group appears as a pair of columns. The light grey columns, the first column in each pair, represent detection events that occurred during the first half of the study. The dark grey columns, the second column in each pair, represent detection events that occurred in the last half of the study. The TTNV ranges were chosen by dividing by five the maximum TTNV of 217 and rounding the results.

Figure 9.3 shows that the second half of the study had fewer foundational vulnerabilities with TTNV of 43 or less than the first half of the study (38 in the first half vs. 27 in the second half). The number of vulnerabilities with TTNV greater than 43 also decreased between the two halves (21 in the first half vs. 12 in the second half).

Figure 9.4 presents another perspective on the TTNV. The vertical axis indicates the TTNV of each vulnerability: the number of days after this detection event before the next detection event occurred. The horizontal axis indicates the date known for each detection event. This figure shows that the number of detection events

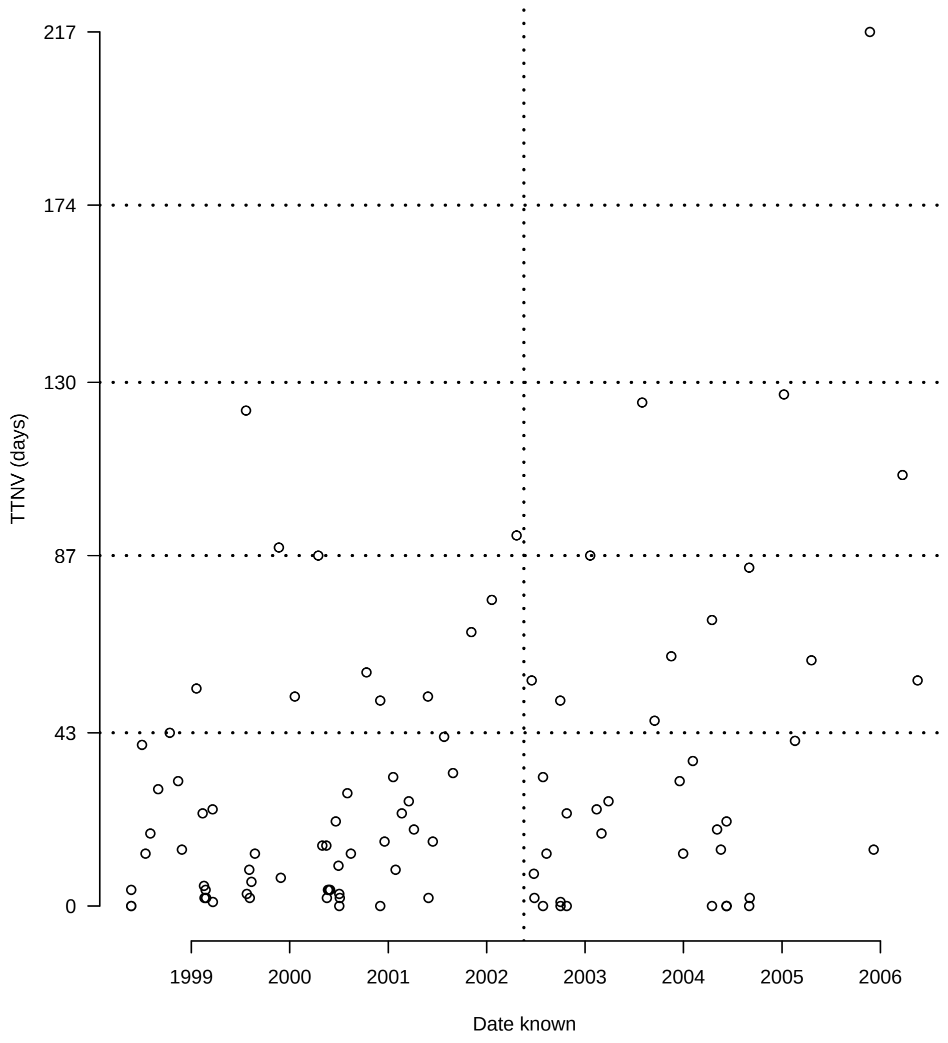


Figure 9.4: *The TTNV, in days, for each vulnerability detection event. The vertical dotted line separates the study into halves. Taken together, the dotted lines mirror the delineations in Figure 9.3.*

with short TTNVs particularly decreased in the last quarter of the study. This figure differs from Figure 7.1 by showing the TTNV of detection events rather than individual vulnerabilities.

9.3.3 Summary

For the foundation version, the analysis does not conclusively indicate a declining trend in the rate of vulnerability reporting, but it suggests such a trend. This finding contradicts that of Rescorla [Res04], whose analysis failed to reject the hypothesis that the rate of vulnerability reporting has remained constant in three of the four operating systems he evaluated.

9.4 Laplace test for trend

The analysis above indicates a decrease in the rate of reporting of foundational vulnerabilities. In this section, I apply a Laplace test to make the data more directly comparable to the work of Rescorla.

The Laplace test is used to ascertain whether or not there exists a trend in the rate of the events being studied (vulnerability detection events, in this case). This test is commonly used in the reliability engineering field [KL96]: it is always appropriate if the underlying process is a non-homogenous Poisson process (NHPP). However, if the process is not NHPP, then its appropriateness cannot be assumed [Gau92]. In Chapter 7, I presented evidence for dependence in vulnerability discovery. That evidence implies that a Poisson process is not appropriate for this data. Nonetheless, I include the results of the Laplace test here for correspondence with Rescorla's work.

Table 9.2 summarizes the interpretation of the Laplace factor. The H_0 is that the occurrence of the events (*i.e.* vulnerability detection events) is a homogenous Poisson process (HPP): there is no change in the rate at which the events occur. It is tested against the following alternative hypotheses:

- H_{1dec} the event intensity is decreasing
- H_{1inc} the event intensity is increasing
- H_{1trend} there is a trend in the event intensity

For our purposes, the intensity is the number of vulnerabilities expected to be reported on a given day.

Here, I analyze the TTNV data for the foundation version. Figure 9.5 shows the calculated Laplace factors, $u(T)$, for each vulnerability report. The lowest horizontal dotted line is at -1.96. When the calculated Laplace factors are less than that amount, the data indicate with a two-tailed confidence level of 95% that the rate of vulnerability reporting is not constant. The test finds significant evidence for

Laplace factor	Hypotheses	Conclusion
$u(T) < -1.645$	Reject H_0 *vs.* H_{1dec}	Indicates a decreasing rate
$u(T) > 1.645$	Reject H_0 *vs.* H_{1inc}	Indicates an increasing rate
$\|u(T)\| < 1.96$	Accept H_0 *vs.* H_{1trend}	Indicates stable rate (no trend)

Table 9.2: *The Laplace factor is used to ascertain the existence and direction of a trend in the rate of reporting.*

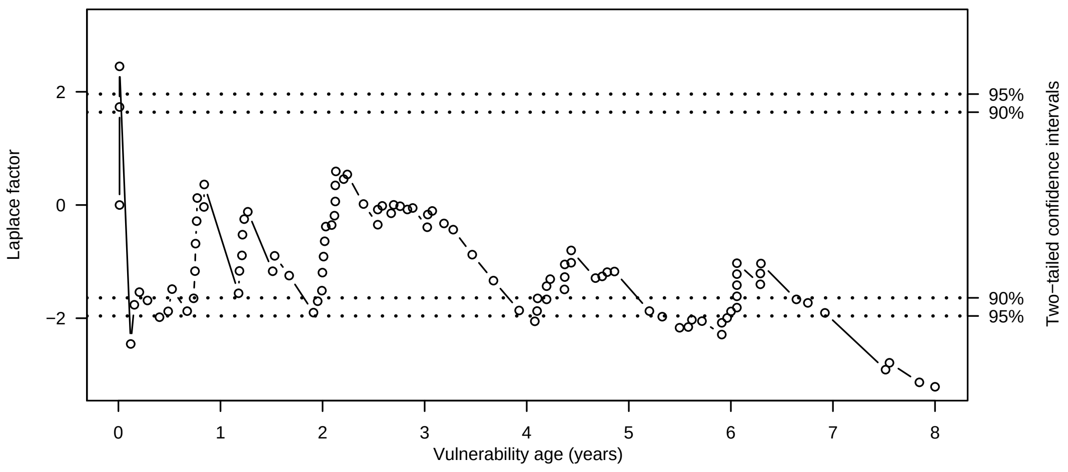


Figure 9.5: *Laplace test for the existence and direction of a trend in the rate of vulnerability reporting.*

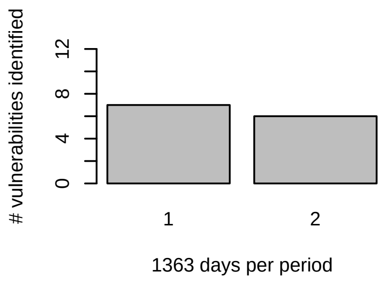
a decrease in the rate of vulnerability reporting by the end of year six (when the Laplace factor is below –1.96).

This test thus provides some evidence that the rate at which foundational vulnerabilities are reported is declining, although, this decrease did not begin until almost seven years after the foundation version was released. However, until the Laplace test is shown to have significance for dependent processes, it cannot confidently be used with this data.

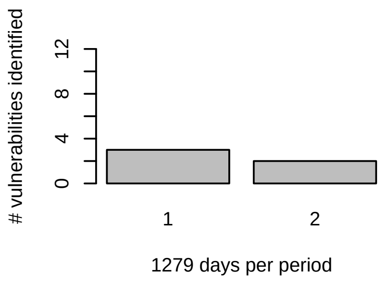
9.5 Other versions

The foundation version is the only version with enough vulnerabilities for a statistical analysis. However, we can examine the other versions using the simple methodology employed in Section 9.3: examining the number of vulnerabilities from each version that were detected in different periods of the study.

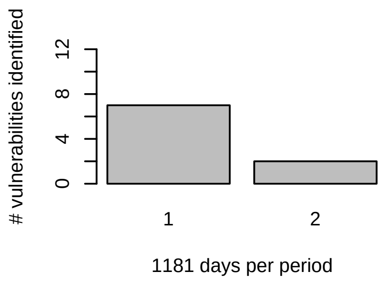
Figure 9.6 shows the number of vulnerabilities discovered in each half of the study for OpenBSD versions 2.4–3.1. The first bar represents the number of vulnerabilities discovered in the first half of the study; the second bar represents the number of vulnerabilities discovered in the second half of the study. The figures use all of the



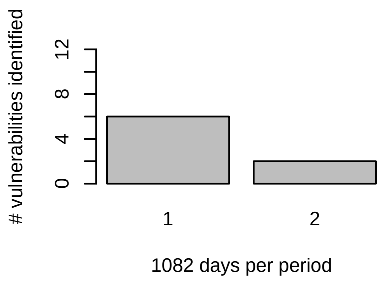
(a) Vulnerabilities introduced in 2.4



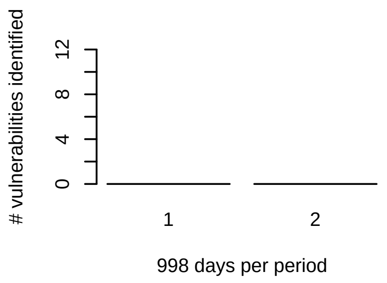
(b) Vulnerabilities introduced in 2.5



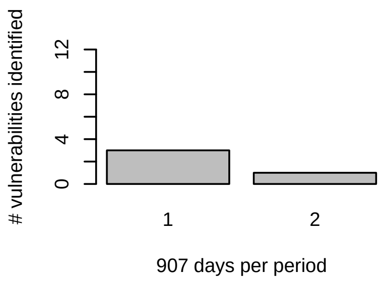
(c) Vulnerabilities introduced in 2.6



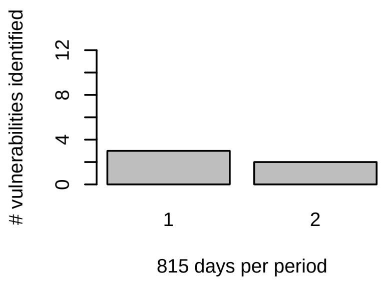
(d) Vulnerabilities introduced in 2.7



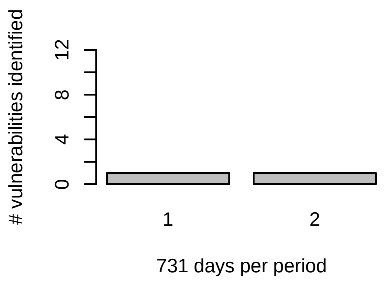
(e) Vulnerabilities introduced in 2.8



(f) Vulnerabilities introduced in 2.9



(g) Vulnerabilities introduced in 3.0



(h) Vulnerabilities introduced in 3.1

Figure 9.6: *The number of vulnerabilities detected in the first and second halves of the study. The study is of different length for each version, so the number of days in each half is noted on the figures.*

available study data, so the length of the study differed by each version (*i.e.* version 2.4 was released six months prior to version 2.5, so we have more days of information on version 2.4). The length of the study halves, in days, is noted on each figure.

For every version, the number of detection events either remained stable or decreased from the first half of the study to the second. Note that no version 2.8 vulnerabilities were detected during the study: none of the detected vulnerabilities was first released in version 2.8.

9.6 Limitations of vulnerability analyses

My analysis examines the rate at which vulnerabilities became known in OpenBSD. However, this information is only one aspect of the security of OpenBSD. The OpenBSD development team has not only worked to increase the security of the system's code base; it has also worked to improve its overall security architecture. These improvements include new security functionality and safeguards that reduce the severity of vulnerabilities.

9.6.1 New security functionality

The addition of valuable new security functionality, like the OpenSSH encrypted network program, increases the amount of code that is deemed security-critical and may thus increase the pool of reportable vulnerabilities. This increase does not necessarily imply that the code is less secure: it may only mean that the operating system has assumed new security responsibilities.

9.6.2 Reductions in vulnerability severity

Architectural improvements that reduce the severity of a vulnerability—but do not eliminate it entirely—can improve security without reducing the rate at which vulnerabilities are discovered and reported. For example, the OpenBSD team improved the security architecture of OpenBSD by adding stack-guarding tools and randomized memory allocation [dR05], both of which reduce the severity of vulnerabilities within the code base.

These security improvements are not accounted for in this study, because I lack an accurate and unbiased methodology with which to assess the severity of vulnerabilities. Simply measuring reductions in the total pool of vulnerabilities is thus likely to underestimate improvements to the security of the overall system.

9.6.3 The lack of effort-normalization

As discussed in Section 3.5.1, we currently have no way to accurately normalize the chronological data according to the effort expended by detectors. The rate of vulnerability discovery is thus dependent upon the effort invested by detectors. The whims and interests of those detectors also determine which areas of the operating system are most closely examined for vulnerabilities. These data are thus unlikely to be a perfect indicator of the underlying number of vulnerabilities in the source code; however, they are the best information available and provide useful insight into the rate of vulnerability detection in the current detection environment.

9.7 Conclusion

Because the OpenBSD vulnerability detection events are dependent, most common VDMs are inapplicable. Some simple analysis indicates that the rate at which detection events occur in OpenBSD is declining. However, this evidence is not statistically significant.

Chapter 10

Independent Rediscovery and Vulnerability Disclosure Policies

The previous chapter demonstrated that vulnerability data can provide useful information for developers, even if appropriate VDMs are not available. In this chapter, I look more broadly at how vulnerability data is gathered. In particular, I review vulnerability disclosure policies and consider an assumption debated in the literature on such policies: vulnerability rediscovery. The likelihood that a single vulnerability will be detected by multiple individuals, working independently, is a critical factor in ascertaining the most appropriate vulnerability disclosure policy.

The means by which an independent, benign detector informs the vendor and the public of a vulnerability is a topic of dispute. In the past, a benign detector might have notified the vendor of the vulnerability and then waited an indeterminate length of time until the vendor released a patch that fixed it. In the 1990s, discontent with the length of time some vendors were taking to release patches led to the creation of public 'full-disclosure' fora (*e.g.* the Full Disclosure [Seca] and Bugtraq [Secb] mailing lists). In these fora, benign detectors publish vulnerability reports that detail the existence of the vulnerability and sometimes go so far as to give instructions on its exploitation.

The creation of the full-disclosure fora provoked a policy debate: was public welfare served better by releasing a vulnerability report before the vendor's patch (*instantaneous disclosure*) or from delaying the report until the patch was ready (*responsible disclosure*)?[1] Advocates of instantaneous disclosure argue that it enables users to mitigate the impact of the vulnerability and pressures software vendors to provide patches more rapidly. Advocates of responsible disclosure argue that notifying attackers of the existence of the vulnerability without simultaneously providing a patch results in increased social cost due to attacks. Implicit in both arguments

[1] A term like 'responsible disclosure' is not value free and is hotly debated. I use it out of convenience and in accordance with its usage in the popular media. When possible, I employ the terminology of [AKN+04] to facilitate the creation of a standard nomenclature.

is the assumption that vulnerabilities should be found and fixed because they are likely to be rediscovered by malicious actors: if a vulnerability has been found by a benign detector, it should be fixed before an attacker independently rediscovers and exploits it.

10.1 Related work

Advocates of responsible disclosure do acknowledge that vendors may not release patches promptly or at all without the benign detector threatening to make the vulnerability public; many solve this dilemma by giving vendors a deadline. Three prominent responsible disclosure policies emphasize timely and continuing contact between the benign detector, the coordinating center (*e.g.* CERT/CC[2]) and the vendor in order to ensure that the patch is released promptly: [CER05], [Rai01], and [CW02].[3] The CERT/CC policy is perhaps the most important, as CERT/CC is often used as a mediator between benign detectors and vendors. Its policy gives a forty-five day deadline for the vendor to resolve the problem, although extensions may be granted if the vulnerability is a particularly difficult one to remediate. The latter two policies emphasize that the reporter and vendor should negotiate to ensure that the vendor has a reasonable length of time to create a patch while not unnecessarily dragging its feet.

Ashish Arora *et al.* empirically compare the results of the instantaneous disclosure policy with those of the responsible disclosure policy. They find that instantaneous disclosure does force vendors to provide patches more rapidly than otherwise, and that it also increases the probability that the vendor will provide a patch *at all.* They note, of course, that the number of attacks against that vulnerability is increased (possibly from a starting point of zero) by instantaneous disclosure and thus that overall social utility may decline [AKN+04].

In a separate work, Arora *et al.* model a policy for disclosure in which a social planner (*e.g.* a publicly funded disclosure institution like CERT/CC) threatens to publicly disclose a vulnerability if the vendor does not provide a patch before a deadline. Vendors are averse to bearing the increased cost of developing a patch rapidly rather than more leisurely. In Arora *et al.*'s model, the social planner is explicitly motivated by the fear of attacker independent rediscovery. **Independent rediscovery** occurs when a detector discoveries a vulnerability that has previously and independently been discovered (but not made public) by a different detector. Their model includes the time that an attacker independently rediscovers the vulnerability as stochastic, with a likelihood that increases as time passes after the vulnerability was found by the benign detector.

[2]CERT/CC was previously the Computer Emergency Response Team Coordinating Center. However, now that institution is known only as CERT/CC.

[3]The [CW02] policy has been retracted by its authors, but it remains influential in the detection community.

They find that neither instantaneous disclosure nor non-disclosure is socially optimal. Furthermore, without a deadline, vendors do not provide patches rapidly enough. In their model, the social planner discloses after a period less than that desired by the vendor, so that vendors are motivated to release patches more quickly. Early disclosure (if it does not result in a hurried patch release by the vendor) thus accepts some losses due to exploitation against a longer delay in patch release, which may have resulted in attackers independently rediscovering the vulnerability. Their results are not altered when they expand the model to include incomplete compliance in users' implementing patches [ATX04]. Unfortunately, they fail to weigh the costs of early disclosure against the risk of attacker independent rediscovery.

In a later work, Hasan Cavusoglu *et al.* also create a game theoretic model of the vulnerability disclosure process [CCR05]. They consider the amount of time vendors are given to create a patch before the vulnerability information is made public—including the extremes of instantaneous public disclosure or never disclosing the vulnerability. They model the benefits and costs of vulnerability disclosure policies to vendors, social coordinators, vulnerability detectors, and users. They find that no single policy is always optimal: instead, the social coordinator should alter the amount of time available to the vendor in accordance with the cost of creating a patch (*e.g.* its difficulty), the vulnerability's risk, and the user population. Although no single policy is optimal, an optimal policy does always exist. However, for vulnerabilities that affect multiple vendors, no policy can guarantee that all vendors will release a patch.

In both Arora *et al.* [ATX04] and Cavusoglu *et al.* [CCR05], the impetus for releasing patches quickly is the fear of attacker independent rediscovery. This assumption is the opposite of Rescorla's (see Section 9.1), who argues that the likelihood of such rediscovery is small because the pool of vulnerabilities is so large. However, the results below demonstrate that the likelihood of independent rediscovery is non-negligible and should not lightly be dismissed.

10.2 Possible reasons for rediscovery

Vulnerabilities are likely to be independently rediscovered for three primary reasons: dependence in the detection process, usage patterns, and attack surface.

10.2.1 Dependence in the detection process

Dependence in the detection process could cause multiple detectors to independently discovery the same vulnerability. This dependence was discussed in Chapter 7, but it has a different implication here. The previous discussion referred to the likelihood that the discovery of one vulnerability leads to the discovery of another. This discussion focuses on the likelihood that the discovery process might lead multiple individuals, working separately, to discover the same vulnerability.

First, the use of similar tools to find vulnerabilities is an obvious source of potential independent discoveries of the same vulnerability. Second, when a new class of vulnerability (*e.g.* integer overflows) is discovered, individuals will search for instances of that vulnerability in the products in which they are interested. For a popular product, the more obvious examples of this vulnerability may thus be discovered by multiple independent individuals.

Third, a new class of 'target' may be found: for example, a rash of vulnerabilities were recently found in the file processing portions of graphics libraries. Between August and October in 2004, twenty-two different vulnerabilities were published in various graphics libraries used by the various POSIX (*e.g.* Linux) and Microsoft operating systems.[4] Because graphics libraries became a popular target, many detectors may have looked in the obvious examples of these libraries and thus discovered the same vulnerabilities.

10.2.2 Usage patterns

When a product is newly released, the most obvious vulnerabilities (*e.g.* those that arise from a reasonably frequent pattern of usage) seem intuitively likely to be discovered by multiple users. Brady *et al.* add that when products are initially released a surge of flaws will be detected as the product is used in a more wide range of environments than it was possible for the vendor to test [BAB99]. However, Arora *et al.* reason differently; they claim that the socially optimal patching/disclosure times are more lengthy when a product is newer, because attackers have not yet learned the product [ATX04]. The S-shaped VDM proposed by Alhazmi & Malaiya combines these two beliefs: immediately after the release of the product, few vulnerabilities are discovered while detectors learn the product. Then, after they are knowledgeable about the system, detectors rapidly uncover a large number of vulnerabilities [AM05b].

10.2.3 Attack surface

A limited attack surface is another cause of independent rediscovery. If the key functionality or security critical portion of a product is small, then interested detectors will have only a limited area in which to search for vulnerabilities. Although the vendor's detectors will also examine this area, the number of independent detectors is likely more than the number that work for the vendor. As a result, the external detectors may discover vulnerabilities missed by the vendor's detectors.

[4]CVE identifiers: 2004-0200, 2004-0597, 2004-0802, 2004-0803, 2004-0817, 2004-0827, 2004-0929 (all buffer overflows), 2004-0599, 2004-0886, 2004-0888, 2004-0889 (all integer overflows), 2004-0691 (heap overflow), 2004-0804 (division by zero), 2004-0598, 2004-0692, 2004-0693 (Null pointer dereferences), 2004-0687, 2004-0688, 2004-0753, 2004-0782, 2004-0783, 2004-0788 (misc.)

10.3 Evidence of independent rediscovery

Examples of independent rediscovery of vulnerabilities are by their nature hard to find. If the first benign detector of a vulnerability follows a full-disclosure policy, then it is difficult for a future benign detector to claim that she detected that vulnerability independently. The best source of such information thus seems to be software vendors who receive more than one report of a vulnerability while they are working on the patch. Unfortunately, not all vendors record the additional reports. However, Microsoft security patch bulletins do at times credit multiple external detectors for reporting vulnerabilities.

As a source of information, these bulletins are very limited. First, the multiple detectors credited may have collaborated on finding the vulnerability, rather than detecting it independently. Furthermore, the creation of a patch seems usually to require less than three months and only rarely takes more than seven months: the window of time for recording independent rediscoveries is thus fairly short. In addition, benign detectors credited with independently detecting the same vulnerability may have actually found two different vulnerabilities that were similar and thus fixed with one patch and assigned the same CVE identifier. Finally, Microsoft may or may not have a policy to acknowledge multiple independent reporters. Although they have at times done so, they may have omitted the repeated discoveries at other times. Nonetheless, the security bulletins provide a means for ascertaining at least some portion of the times when a vulnerability has been independently reported to Microsoft.

Fortunately, security professionals often release their own vulnerability report to the public on the date that the vendor releases its patch, and that report may be used to ascertain when individuals worked together. I examined three years of Microsoft vulnerability bulletins: 2002–2004. I started with 2002 because that was the first year in which Microsoft committed to crediting detectors who followed the Microsoft disclosure policy. I examined the bulletins through the end of 2004, the last complete year available when I undertook this effort. For each bulletin that credited multiple detectors, I tried to discovery whether or not the detections were independent. For some vulnerabilities, the detectors had posted their own reports on the web that confirmed whether or not they had worked independently. For the majority, I contacted the detectors via email. For the Microsoft bulletin data shown in Tables 10.1–10.3, vulnerabilities are credited as having been detected independently if at least one of the two reporters (or, correspondingly, two of the three reporters) asserted that he independently detected the vulnerability.[5]

[5] I am thankful for the assistance of the individuals who provided me with information about their vulnerability finding efforts: Peter Winter-Smith, Brett Moore, Dustin Schneider, Jouko Pynnon, Jelmer, SPILabs, Renaud Deraison, Cesar Cerrudo, Greg Jones, Ophir Polotsky, Joseph Steinbberg, Noam Rathaus, Michel Trepanier, zenomorph, qFox, Mark Litchfield, David Litchfield, Rodrigo Gutierrez, Roozbeh Afrasiabi, Yorick Koster, and Mike Price.

Year	Not credited	Confirmed independent reporters			Total
		1	2	3	
2002	62	71	5	0	138
2003	22	43	4	0	69
2004	22	54	3	2	81
Total	106	168	12	2	288

Table 10.1: *The number of vulnerabilities listed in Microsoft security bulletins without any credited detectors, with a single credited detector, with two confirmed independent detectors, and with three confirmed independent detectors.*

Year	Multiple reports*	All credited reports	Multiple/Total
2002	5	76	6.58%
2003	4	47	8.51%
2004	5	59	8.47%
Total	14	182	7.69%

*Reports credited to either two or three independent reporters.

Table 10.2: *The total number of vulnerabilities credited and the number credited to multiple independently working detectors in Microsoft security bulletins.*

Table 10.1 shows the number of individual vulnerabilities announced by Microsoft via security bulletins. In the second column is the number of vulnerabilities for which Microsoft provided no reporting credit: presumably these vulnerabilities were either discovered internally or publicly announced before they were reported to Microsoft (*i.e.* instantaneous disclosure). The third column reports the number of vulnerabilities for which Microsoft identified exactly one detector. The fourth column lists the number of those vulnerabilities that had exactly two confirmed independent reporters; the fifth column lists the number of those vulnerabilities that had exactly three confirmed independent reporters. The final column is total number of vulnerabilities noted by Microsoft in security bulletins.

Table 10.2 summarizes the total number of confirmed multiple reports in the second column (*i.e.* those vulnerabilities for which two or three individuals independently reported the vulnerability to Microsoft). The third column is the total number of vulnerabilities for which anybody is credited: both single reports and multiple reports. The final column is the percentage of vulnerabilities for whom multiple individuals are credited with the report. The final column thus represents the percentage of vulnerabilities that were independently rediscovered in the reasonably short time frame that Microsoft was working to create a patch.

Table 10.3 lists each vulnerability that was confirmed to have been discovered by multiple independent detectors and notes the dates it was disclosed to Microsoft (as remembered or recorded by the reporter), the patch date, and the vulnerability's

CVE	Disclosure dates			Date public	Patch
	1st*	2nd	3rd		creation
2002-0018	2000-10-31	?		2002-01-30	456 days
2002-0074	2001-12-03	?		2002-04-10	128 days
2002-0641	2002-05-28	?		2002-07-10	43 days
2002-0693	2002-07-31	?		2002-10-02	63 days
2002-1145	2002-08-23	?		2002-10-16	54 days
2003-0226	c.2003-01-15	?		2003-05-28	133 days
2003-0228	2003-03-14	2003-03-23		2003-05-07	54 days
2003-0528	c.2003-07-23	2003-07-29		2003-09-10	49 days
2003-0662	c.2003-04-15	?		2003-10-15	183 days
2003-0908	c.2003-10-15	?	?	2004-04-13	181 days
2004-0123	c.2004-01-13	?		2004-04-13	91 days
2004-0212	2004-05-06	2004-06-20	2004-07-07	2004-07-13	68 days
2004-0214	c.2002-04-15	2004-08-03		2004-10-12	758 days
2004-0216	c.2003-04-15	2004-07-12		2004-10-12	546 days

*Dates preceded by a 'c.' are approximate. When the reporter remembered only the week or month, the middle of that period was chosen.

Table 10.3: *The dates on which the independent reporters disclosed each vulnerability to Microsoft. Some entries have question marks, indicating that the reporters either did not know the date or did not respond to requests for information. The final column shows the days that elapsed between the first known disclosure date and the patch date.*

age (the days between the first known disclosure date and the patch date). If not every date on which the vulnerability was disclosed to Microsoft is known, then the earliest known date is used: this practice may thus underreport the age.

These instances of multiple reporting suggest that vulnerability rediscovery does occur. The most interesting information would be to look for a relationship between the length of time the vendor works on a patch and the likelihood of rediscovery. Unfortunately, complete information on the dates when vulnerabilities were reported is unavailable: not every reporter could be reached or could remember the date on which she reported the vulnerability.

Nonetheless, the information that these bulletins provide is an indicator that vulnerability rediscovery occurs 'in the wild' and that it may be common. Again, however, it is important to note that the evidence presented above assumes that related but different vulnerabilities were not assigned the same CVE and patched at the same time.

10.4 Conclusion

The evidence presented here suggests that multiple, independent discoveries of the same vulnerability do occur—and they occur in time frames that are often less than a year. This result provides support for the literature that assumes independent rediscovery. It thus suggests that forcing vendors to patch their vulnerabilities rapidly is indeed socially beneficial.

Chapter 11

Economic Approaches

In the previous chapters, I have considered the application of software engineering tools to the problem of measuring software security. The existing work on VDMs and vulnerability databases has shortcomings; nonetheless, these tools have the potential to provide real value to vendors. In particular, they will assist vendors in planning, resource allocation, and the evaluation of different development strategies. However, one shortcoming of these models is difficult or impossible to overcome: the time and effort invested in vulnerability discovery in a given product will likely change over time. The engineering approach assumes a constant (or slowly changing) vulnerability detection environment. If that environment changes, then the engineering approach will be rendered inaccurate, at least until new information is gathered.

Another, complementary, approach is economic: it looks at measuring software security through market-mechanisms. As a means of measurement, this approach is superior to the engineering approach. The metric is monetary and thus readily understandable. Furthermore, the existence of an increasing reward will serve to normalize the effort invested in finding vulnerabilities: if not enough detectors are looking for vulnerabilities, the reward will increase in value and thus induce more participation by detectors. A market-based approach might therefore provide a more accurate—and useful—reflection of the true software security of a system than a software engineering approach.

A market-based metric would quantify software security such that buyers could effectively assess their risk. It might also allow vendors to provide a quantitative assurance for their product. One such metric would be the 'cost to break' a system. A number of security firms have contests in which they offer prizes for 'breaking' their own product or the mathematical foundation upon which their product is built [Gol04; RSA04].

In 2002, Stuart E. Schechter proposed creating markets for reports of previously undiscovered vulnerabilities. He argued that the bid, ask, and most recent sale prices in such a market approximate the labor cost to find a vulnerability. He further argued that these prices can establish which of two products the market deems to have vulnerabilities that are less expensive to find [Sch02a; Sch02b; Sch04].

Schechter characterizes his idea as a vulnerability market (VM). However, this concept can also be characterized as an auction. When considered this way, auction theory provides a number of useful tools for assessing its effectiveness. First, in Section 11.1, I consider the literature on economic approaches to measuring software security. In Section 11.2 I describe the current situation with respect to testing and vulnerability detection. Section 11.3 reviews Schechter's vulnerability market and notes some attacks that it anticipates and against which it defends. In Section 11.4, I review the pertinent literature in auction theory and consider Schechter's vulnerability market as a 'bug auction.' In Section 11.5, this perspective is used to identify a number of improvements to the original design. Section 11.6 identifies two attacks by detectors that are applicable to Schechter's vulnerability market and articulates alterations to the bug auction to defend against these attacks. Attacks by the vendor are considered in Section 11.7. In Section 11.8, I note some fundamental problems with both the vulnerability market and the bug auction. Section 12.2 discusses some areas of future work and Section 11.10 summarizes the results.

Throughout the paper, a vulnerability market as characterized by Schechter will be referred to as a *VM*. A *bug auction* will refer to a VM that has been modified as suggested in this chapter. This chapter is based on work I have previously published [Ozm04]. I will thus retain the nomenclature 'bug auction,' although this term is not in accord with the definitions I introduced in Chapter 2.

11.1 Literature review

L. Jean Camp & Catherine Wolfram proposed a market through which vulnerability credits could be traded; such markets have worked previously to create incentives for the reduction of negative externalities like environmental pollutants [CW00].

Given the emergence of a black market for reports of undiscovered vulnerabilities, metrics that estimate the cost to discover a vulnerability may be more valuable than those that quantify the reporting rate. Several organizations are now actively purchasing vulnerabilities, so an open market or auction as described in this literature is not infeasible.

However, the market that does exist is problematic. The existing purchasers, such as iDefense [iDe07] and TippingPoint [3Co05], are not sharing pricing information, hindering the movement toward an open market or auction. Charlie Miller describes the difficulty for a seller to ascertain an appropriate price and find a benign purchaser [Mil07]. Another organization, SNOsoft Research, has offered its services as an intermediary between buyers and sellers [SNO07].

Furthermore, the business models of the existing organizations are not socially optimal: they always have an incentive to leak information about the vulnerability [KT04].

Until an open market with visible pricing arises—and until that market has resulted in several years of data—other means of measuring software security are necessary.

11.2 Software life cycle

The life cycle of software systems can be divided into three distinct phases. In the *pre-release* phase, the product is still under development and testing; it has not yet been commercially released. *Post-release*, the product is commercially available and used by customers. *Post-depreciation*, the vendor is no longer interested in actively improving the product or its security, usually because a successor product has become available. The VM is a tool for finding vulnerabilities during the pre-release and post-release phase.

Currently, vendors' pre-release testing occurs both in-house and using external volunteers (beta testers). During the post-release phase, vendors learn about vulnerabilities in four general ways (in order of decreasing preference on the part of the vendor): vendor detectors, directly from benign external detectors, public fora, or exploits 'in the wild.'

The first of these, vendor detectors, is the most attractive to vendors. Unfortunately, for any reasonably complex software product, internal testing may find some vulnerabilities but is unlikely to find all of them [BAB99].

Second, benign external detectors identify vulnerabilities after the product is in production and available to consumers.

The third source of vulnerability reports is public fora. The vendor learns about the vulnerability at the same time as the general public and must then race to create a patch before the vulnerability can be exploited.

Fourth, the vendor can learn about a vulnerability when it is exploited. For example, an exploit circulating in the black market might be detected when it is used against a system. This scenario is generally the worst for the vendor, as it can result in prominent negative publicity.

11.3 The vulnerability market

Schechter envisages the VM as a tool both to improve a product's security and to quantitatively assess this security. Vendors would ideally recoup the expenses they incur from the VM by the increased sales they achieve from having a more secure product. (Or, conversely, vendors would recoup their expenses by retaining those customers they would otherwise lose due to the negative publicity and other effects of security failures.) He also notes that the VM can be extended to improve a product's quality, in addition to its security.

Schechter argues that the vendor is most interested in three properties: the cost of finding vulnerabilities ('value'), the speed with which they are found ('speed'), and the order in which they are found ('order'). The last of these goals is based on the assumption that vulnerabilities are not stochastically distributed; rather, the most common and obvious vulnerabilities should be found first, because these are the vulnerabilities most likely to become evident when the product is used [BAB99]. This belief is supported by the evidence for multiple independent discoveries of the same vulnerability that was presented in Chapter 10. By offering a monetary reward for vulnerabilities, Schechter argues that vendors will find them more rapidly (*e.g.* earlier in the production process, when they are less expensive to fix) and that the most obvious vulnerabilities will be identified first.

Schechter refines his initial proposal in a number of ways, by including: a continuously increasing reward, a reward spectrum, and a trusted third party.

Pre-release, the reward R is small when first offered; it then grows over time until it is claimed. After each new vulnerability is reported and verified, the reward amount is reset to R_0, the minimum reward value. If a vulnerability is reported more than once, only the first reporter receives the reward. The continuously increasing reward scheme maximizes value at the expense of speed: it trades potential delays in the reporting of vulnerabilities in return for monetary savings. A detector can choose to report a vulnerability at any time; waiting longer increases the reward she will receive, but it also increases the probability that another detector will report the same vulnerability and thus deprive her of the reward. (The post-release phase may differ and is described below.) If the vulnerability is genuine and unique, the reward will be reset [Sch02a].

A vendor may also create a spectrum of rewards, so it does not pay a large reward for an unimportant vulnerability. To reduce complexity, only one reward is published: the category of the vulnerability ascertains the fraction of the reward that the detector receives. During the pre-release phase, this refinement can also be used to improve the product's quality: the reward can also be offered for non-security related faults. For example, a report of an important vulnerability (*e.g.* a remote root vulnerability) might have a multiple of one and thus earn R, a less important vulnerability (*e.g.* a local privilege escalation vulnerability) might earn $\frac{4}{5}R$, while a minor quality fault (*e.g.* a misspelled dialog box) might earn $\frac{1}{10}R$. Because faults can be related (see Section 10.2), the reward is reset after each report—regardless of the fault's classification [Sch02a].

Vendors and detectors interact through a trusted third party (TTP) who ensures that the vendor pays the reward when appropriate and that the detectors may remain anonymous. Involving the TTP prevents one form of vendor cheating: the vendor cannot accept vulnerability reports and falsely claim that the reports are non-unique (which would enable it to avoid paying the reward). The (pseudo)anonymity it provides detectors allows them to submit a report without fearing retaliation by either the vendor or the black market. If a reward spectrum is used, the TTP will rank the submitted vulnerabilities according to severity and thus determine the proportion of the reward that the detector receives; using the TTP prevents the vendor from minimizing the severity of a vulnerability in order to pay a smaller proportion of the reward. (Other attacks by the vendor are considered in Section 11.7.) However, the TTP will have less knowledge of the product than the vendor; as a result, verifying reports will be more difficult for it. In order to ease the burden of verification, all vulnerability reports must include an example program that exploits the vulnerability. This requirement will result in rapid verification and minimize the expense of employing a TTP. Additionally, to reduce the frequency of 'frivolous' reports, detectors may be charged the transaction cost of submitting a report; the minimum reward (R_0) will always be equal or greater to that transaction cost [Sch02a].

The VM is thus first implemented pre-release, using the continuously increasing reward; both the vendor's detectors and some number of external detectors participate (Section 11.8.4 explains how detectors obtain the product). If this approach is taken, the vendor needs to ensure that the existence of the market is incentive compatible with respect to the vendor detectors (those detectors directly employed by the vendor). The goal of this phase is to identify and fix those vulnerabilities that are easy to find. The product is not commercially released until the reward for finding a vulnerability has gone unclaimed for some period of time.

Perhaps the best way to describe the VM is to use a (best case) hypothetical example:

> Software vendor DiligentCompany builds the closed-source SecureProduct. After SecureProduct is finished, the quality assurance division of DiligentCompany tests the product and identifies a number of vulnerabilities. These vulnerabilities are fixed and SecureProduct is deemed ready for wider testing. So, DiligentCompany engages TTP, a trusted third party, to oversee a vulnerability market. DiligentCompany sets $R_0 = \$100$ and decides that R will increase at a rate of \$1/hour. Quality faults are worth $\frac{1}{10}R$, while vulnerabilities are worth R. The TTP then distributes beta (pre-release) copies of SecureProduct to external detectors Alice and Bob.
>
> After ten hours of testing, Alice identifies a quality fault. She submits the fault to TTP. TTP verifies the fault and pays Alice $R = (\frac{1}{10})[100 + (1)(10)] =$ \$11. TTP then resets the reward to $R_0 = \$100$. This process continues for two months, with a number of quality faults and vulnerabilities identified and fixed. At the end of two months, the reward has climbed to $R = 100 + (1)(24)(14) =$ \$436 because no faults have been found in the past fourteen days.

The product is then made commercially available, and the post-release phase of the VM begins. R is reset to a new R_0, the value of the security assurance the vendor is offering its customers (the reward is now offered only for security vulnerabilities).[1]

[1]The security assurance is not a guarantee: the vendor is not offering to recompense customers if they suffer losses due to a security breach.

Because the reward continues to grow and then be reset when vulnerabilities are reported, the security assurance is dynamic but provides a constant assurance at the level of R_0. Alternately, R can be set to a constant level (R does not increase) in order to provide a stable security assurance. The choice to use a continuously increasing reward or a stable reward can be made based on the vendor's perception of the product's quality and the perceived reputation benefits of a continuously increasing reward. Regardless of whether or not the reward is increasing, its magnitude when unclaimed is the lower bound on the product's cost to break.[2] The product can safely be used to protect information whose total worth, summed across each customer and installation, is less than or equal to the magnitude of the unclaimed reward. Any rational, risk-neutral criminal who discovers a vulnerability would rather report it to the vendor and receive the reward than risk the legal consequences of an attack that could at most reap the same monetary benefit.

Of course, for a widely used product, the cumulative value of information that relies on that product will almost certainly be greater than the assurance provided by the vendor. However, a potential attacker faces the difficulty of converting access to that information into monetary gain: the reward may still be the most attractive source of remuneration for locating a vulnerability. Perhaps more importantly, the real value of the reward is comparative: it highlights the security of the product relative to its competitors.

> DiligentCompany decides that SecureProduct is now secure and commercially releases it. DiligentCompany decides to also use a VM in the post-release phase. It again contracts with TTP. Now DiligentCompany sets $R_0 = \$250$ and advertises that SecureProduct has a \$250 security assurance. R still increases at a rate of \$1/hour, but the reward is now offered only for security vulnerabilities.
>
> SecureProduct is widely adopted. It is used in a variety of environments, and as a result four new vulnerabilities are identified and reported. Then, no vulnerabilities are identified for thirty-two days. DiligentCompany sends press releases to industry journals to highlight the fact that the reward stands at \$1018 and remains unclaimed. Sales of SecureProduct increase dramatically.

However, the total number of vulnerabilities found in a system is sometimes used as a rough measure of software security. As a result, a vendor that employs a VM may be vulnerable to public relations attacks from a competitor who does not.

> Now, BigCompetitor enters the market. It advertises that its CompetingProduct is more secure than DiligentCompany's SecureProduct. BigCompetitor notes that four different vulnerabilities have been identified in SecureProduct. Although those vulnerabilities have been fixed, BigCompetitor argues that these

[2]In actuality, the magnitude of the unclaimed reward only approximates the cost to break: it also includes transaction costs, risk, *etc.* However, for the purposes of the VM, this approximate cost is sufficient.

vulnerabilities indicate the poor quality of programming in SecureProduct. In contrast, no vulnerabilities have been identified in CompetingProduct! Sales of SecureProduct plummet.

Schechter provides a defense against this public relations attack: he notes that any vendor can bootstrap the use of the VM in order to differentiate itself from competitors. The vendor offers at least R_0 for vulnerabilities in its product, thus establishing a lower bound on the cost to break its own product. It then employs a TTP to purchase, at a lower price, a single vulnerability for its competitor's product (establishing an upper bound on the cost to break that competitor's product). The vendor has thus shown that its product is more secure than the competitor's: the lower bound on the cost to break of the vendor's own product is higher than the upper bound on the cost to break of its competitor's product [Sch02b]. Although the value of the reward and the quantitative assurance may thus be low when compared to the total value of information that relies on the product, the use of the VM enables the vendor to quantitatively assert that its product is *relatively* secure. That is, the product is more secure than those offered by competitors.

DiligentCompany decides that it must respond to BigCompetitor. So, DiligentCompany pays TTP to offer a $100 reward for any vulnerabilities in CompetingProduct. A vulnerability is soon reported to TTP and the reward is paid. Because it is employed by DiligentCompany, TTP does not report the vulnerability to BigCompetitor (or to DiligentCompany).

Now DiligentCompany advertises that it offers a security assurance of $250 on SecureProduct, while vulnerabilities for CompetingProduct can be purchased for just $100! SecureProduct is thus more secure than CompetingProduct!

BigCompetitor has no choice but to respond. It employs TTP to run a vulnerability market for CompetingProduct. It matches the $250 security assurance provided by DiligentCompany. However, BigCompetitor decides that the reputation advantage of an increasing reward is not worth the cost. So, its $250 security assurance is constant, rather than continuously increasing.

11.4 The vulnerability market as an auction

Schechter's VM proposal is innovative, but it would be more effectively structured as an auction, for which the economics literature can provide useful insight. Although a bug auction would be monopsonistic, I will follow convention in this section by using monopolistic auctions for my general examples. In a monopolistic auction, many buyers compete to purchase from just one seller, while in a monopsonistic auction many sellers compete to sell to just one buyer.

Auctions can largely be divided into two types: first price and second price. The most common example of a first-price auction is the Dutch, or open first-price descending, auction. In a Dutch auction the seller sets an exceptionally high price for the good being sold. That price then begins to descend automatically. When a the price reaches a point at which a buyer considers it reasonable, she signals the seller that she will purchase the good at that price; the first buyer to signal the seller thus obtains the good at the first price she is willing to pay for it.

The commonly used example of a second-price auction is the English, or open second-price ascending, auction. In an English auction, buyers compete with each other by offering increasingly higher bids for the good being sold. When only two buyers remain in competition, the bids will increase until one bid is equal to the lower of the two remaining bidders' valuations.[3] The winning buyer will then place a bid one unit higher than the losing buyer's valuation and thus obtains the good at the second price: the price at (or just above) the valuation of the item by the *second* highest bidder. The winning buyer's valuation may be much greater than the price he pays.

First-price and second-price auctions can also be held as closed auctions. In this instance, the bids would be sealed when submitted and then opened simultaneously by the seller. The winner would then pay either the price of her offer, or the price of the second-highest offer, for the respective types of auctions.

Although these four auctions are different in important ways, they are *revenue equivalent*: under standard conditions they provide on average the same revenue to the seller of the item, because rational bidders adjust their behavior according to the auction's structure [Vic61; RS81]. Those conditions are [MM87, p. 706]:

1. All bidders are risk neutral.
2. Each bidder's valuation of the good is private and independent of the good's valuation by other bidders.
3. Bidders are symmetric: they draw their values from the same probability distribution.
4. The seller's revenue comes entirely from the bids themselves; it receives no payments from the bidders for the privilege of bidding and no revenue from the use of the good being sold.[4]

Schechter constructs a vulnerability *market*, perhaps because bootstrapping this market may require vendors to offer rewards for finding vulnerabilities in *competitors'* products. However, he notes that when the vendor is willing to reward anonymous detectors, that vendor rapidly becomes the only important buyer because of its

[3]That is, the price reaches the value that he places upon the item. Below this value, he would purchase the item and earn a profit. Above this value, if he purchases the item he suffers a loss. At this value, he is indifferent as to whether or not he obtains the item.

[4]This condition does not imply that bidders have no costs for participating in the auction. Bidders can pay both entry costs and bidding costs in a standard auction; the condition only requires that these payments are not received by the seller. For example, bidders typically pay some transaction cost to enter an auction, *e.g.* traveling to the auction site.

unique ability to remediate the vulnerability with an update for the product [Sch02a]. Any vulnerability available on the black market will eventually be reported to the vendor. Because the reward is offered to anybody, either the black market buyer or seller of a vulnerability would be tempted to engage in arbitrage and increase their profit by also selling the vulnerability to the vendor. Indeed, since "the only way to ensure the detectors won't resell is to take them out of the game, detectors who wish to avoid sleeping with the fishes will be wary of selling vulnerabilities to anyone other than the [vendor]" [Sch02a, p. 11]. The only vulnerabilities a vendor will not learn about are those a detector identifies and then exploits himself (because he values the exploitation of the vulnerability more highly than the reward) or those vulnerabilities with black market value greater than the reward.

Thus, when a vulnerability is widely available on the black market or has been identified by an individual not interested in illegal gains, it will be sold to the vendor at a price established by the continuously increasing reward function. When considered this way, the situation can be accurately described as an *auction*. This auction has one buyer, the vendor (through its proxy, the trusted third-party auctioneer) and a potentially unlimited number of sellers, the detectors. If modeled after the VM, it is a reverse Dutch auction, or open first-price ascending auction, in which the price is set to be exceptionally low and then rises continuously until it is accepted by a seller.

Before a more complex analysis of this auction can be performed, it is first necessary to compare it with the standard auction described above. Requirement B is fulfilled: detectors' valuations will in part depend upon the amount of work they put into identifying the vulnerability (their costs) and will thus be both private and independent (unless detectors collude, an occurrence that will be discussed in Section 11.6.2). Requirement A is also fulfilled: no information extant suggests that detectors will be risk averse or risk prone. Requirement D is met, as well: although the bidders must pay entry fees (elaborated upon below), those fees are not paid to the vendor.

Requirement C, symmetry, is more interesting. If the auction is implemented according to the rules of the VM, bidders are asymmetric because detectors from different countries will draw their valuations from different probability functions. Preston McAfee & John McMillan note that a typical example of asymmetry is a procurement auction in which both domestic and foreign bidders participate [MM87, p. 706]. In first-price auctions, asymmetry of values can result in the seller capturing less than all of the available value because the bidder with the highest valuation may not place the highest bid [MR85; MM87].

Given this shortcoming, it is worth considering the value of using the current broad format of the VM (Dutch) as opposed to the other main formats (English, sealed first or second price). For this environment, the Dutch auction has one overriding structural advantage: a reward is always offered, ensuring that vulnerabilities are reported immediately if they are being traded on the black market. Sealed auctions must be periodic or the bidders can assume that they are the only bidder or one among few bidders (and thus bid more conservatively, causing the vendor to overpay). English auctions require a waiting period after each bid to ensure that other bidders have the opportunity to counter. The delays necessary in these alternative models could cause a detector to first sell a vulnerability on the black market. The vulnerability could then be in use for the entire interval required by that auction format before the detector has the opportunity to sell it to the vendor. Vendors concerned about security may be risk averse; if so, then they also prefer first-price auctions to second-price auctions (although they would prefer a first-price closed auction over one that is open) [WHR98]. The Dutch auction has one other advantage: it conveys no information about the number of bidders (see Section 11.6.2).

Because the reverse Dutch auction seems most suitable to the demands of the environment, the vendor might attempt to remove the asymmetry through other means. Unfortunately, any means of remedying the asymmetric distribution of values would first require that detectors with different probability distributions be distinguished; this requirement seems overwhelming in an environment in which anonymity is required and one in which any individual is a potential participant.[5] The vendor may thus have to accept the inefficiencies that result from asymmetric bidders.

We have so far considered only those auctions used to sell a single item. Sellers with more than one item to sell can employ sequential auctions or simultaneous multi-unit auctions. Although not found in the initial exposition of the concept, multi-unit auctions can still be revenue equivalent if the goods to be sold are homogeneous [MR89]. However, unlike single-unit auctions, designing multi-unit auctions to achieve efficient outcomes is difficult [Kle04a]. The bug auction is a sequential auction, in which the auctioneer will sequentially purchase vulnerabilities from the bidders.

The bidders in the bug auction must pay for the costs they incur in finding vulnerabilities (or attempting to find vulnerabilities), whether or not they actually win the auction. These expenses initially seem to resemble the all-pay auction, in which bidders pay an amount to the seller based on their bid, whether or not they win the auction: *e.g.* government procurement [KLSW02], political lobbying [Kru74] [BKdV93], and waiting lines [JS82].

However, this initial similarity is misleading. In all-pay auctions, the bidders must pay according to the magnitude of their bids. By dropping out early, a bidder can alter the amount he must pay. In the bug auction, the costs are incurred by each bidder whether or not he is even able to bid; a bidder can incur costs from

[5]Any user may by chance discover a vulnerability and then report it; in effect, some bidders may have zero costs for discovery.

attempting to enter the auction (searching for a vulnerability) but fail in his attempt. Furthermore, a bidder's costs are not based on his bid but on the effort required for him to identify the vulnerability. A better mechanism for modeling these costs in a bug auction is to consider them to be entry costs.

The VM can be thus characterized as an auction: a sequential open first-price ascending auction with asymmetric independent private value bidders, high entry costs, and minimal bid costs.

11.5 Efficiency enhancements

When the VM is considered as an auction, rather than a market, auction theory provides a number of useful tools for optimizing its structure. In addition, other enhancements become more clear when the VM is considered from this perspective. The alterations suggested below are intended to improve the efficiency of the auction; those alterations intended to cope with attacks are elaborated upon in Section 11.6 and 11.7.

11.5.1 Endogenous entry

Most of the standard auction models assume that the number of bidders is fixed and known. However, this assumption does not fit many real world auctions, in which potential bidders first decide whether or not to participate in an auction at all. For real world auctions, the importance of attracting a sufficient number of bidders cannot be overemphasized: the addition of a single bidder can benefit the auctioneer more than optimizations using reservation prices and entry fees [BK96].

The bug auction's high entry cost is a significant disincentive to potential bidders.[6] A further disincentive exists for those detectors who do not participate from the start of the sequence. Learning the product in order to test it is a non-trivial entry barrier, but it is one that must be incurred only once. Detectors who incur this sunk cost thus face lower entry costs for the remaining auctions of the sequence, and they can then amortize the initial learning cost over the entire sequence.

This area may be one in which speed is more important to vendors than value. Vendors can benefit from being first to market and thus may prefer as short of a pre-release testing phase as is prudent. The post-release life of a product is also usually limited (*e.g.* by the arrival of future versions of the product). If speed is of import to the vendor, then it can adjust its spending accordingly to induce high initial participation without entirely sacrificing the cost savings derived from the continuously increasing reward function:

[6]Flavio Menezes & Paulo Monteiro argue that entry costs simply discourage low-valuation bidders and are thus actually optimal from the seller's perspective [MM96]. However, in their model, bidders pay a fixed entry cost at a time when they already know their valuation: in the bug auction, the entry cost is dependent upon each bidder and may influence their valuation.

Enhancement 1 *For the first auction or auctions in the sequence, set the initial value of the reward to a high level. For the following auctions, the vendor should commit to offering a reasonably high minimum initial value for the reward.*

Alternatively, a vendor might have the reward increase at a more rapid rate or with discontinuous jumps instead of offering a high starting reward value. Creating a significant initial incentive is of particular import in order to 'jump-start' the first few auctions in the series and draw a large pool of detectors. Once these detectors have invested in learning the product, they will have lower entry costs for participating in future auctions in the sequence. The vendor expends its budget more rapidly in the beginning of the sequence in return for rapidly increasing the number of bidders. Increased participation should increase the rate with which vulnerabilities are identified and thus may well result in more vulnerabilities reported, at a lower cost, in the long run (because the reward will be claimed when it is at a low level). Of course the lower costs that could result from this increased competition might decrease the incentive for bidders to continue searching for vulnerabilities; eventually a stable participation equilibrium would be reached.

Even without the inducement of a VM, the vulnerability detectors already discover vulnerabilities and report them directly to vendors, although they often receive no direct financial reward. Instead, their reward seems to be an increase in their reputation, an incentive that can be used in combination with the monetary reward to induce entry.

Enhancement 2 *Combine the monetary reward with a reputation reward.*

When the vendor releases a patch to remediate the vulnerability, it should also highlight the pseudonym of the detector that reported the vulnerability. The auctioneer, acting as TTP, could reveal the true identity of a pseudonym at that individual's request (for example to use that reputation to gain employment).

11.5.2 Variable demand

Another important attribute of the bug auction is that it exhibits variable demand (or variable supply in the terminology of the standard monopolistic auction model). Each detector has a vulnerability he has discovered. If he is not the first to claim the reward at this auction, he can claim it during the next auction or the one after. However, at any time a second detector may identify the same vulnerability and claim the reward before the first detector. If this situation occurs, the first detector (who has not yet claimed the reward) discovers that the demand for his vulnerability has disappeared.

Tibor Neugebauer & Paul Pezanis-Christou perform an extensive analysis and experimental examination of monopolistic sequential first-price auctions both with and without demand uncertainty. They find that demand uncertainty causes high-value bidders to bid less aggressively at first (*i.e.* they 'wait and see') and then with increasing aggressiveness; low-value bidders, on the other hand, initially bid aggressively in a seeming attempt to take advantage of early uncertainty. Casting their results in the terms of a monopsonistic action like the bug auction, they find that average prices are lower with uncertainty than without and that prices tend to decline over the series of auctions [NPC03]. (The latter trend may be countered by the generally increasing difficulty of finding each new vulnerability as the most obvious or common vulnerabilities are remediated.) In the bug auction, demand uncertainty thus seems to benefit the vendor, resulting in lower costs.

However, from the perspective of the detector, two types of demand uncertainty exist in the bug auction: the anticipated number of auctions that will occur before another detector discovers the same vulnerability and the total remaining number of auctions. The latter type of demand uncertainty can have a strongly negative impact on entry. As noted in Section 11.5.1, each detector pays some one-time cost in order to familiarize herself with the product; this cost can, to some degree, be separated from the cost of finding a specific vulnerability. The longer the sequence of auctions (chronologically), the more opportunity for the detector to amortize the cost of learning the product.[7]

However, the VM is designed in such a way as to make forecasting its cost relatively easy. Although a sequence of infinite length would undoubtedly result in the most secure product, all products have official lifetimes after which the vendor is no longer interested in selling, supporting, or enhancing them. An approximate budget can be created using this maximum duration $\bar{t}$, the rate of increase r, and an estimation of the total number of vulnerabilities v: the budget equals $r\bar{t} + vR_0$. The vendor's knowledge and approximate budget can be used to eliminate the negative effect on entry of sequence-length demand uncertainty.

Enhancement 3 *The vendor commits to the bug auction until a certain minimum amount of reward money has been claimed or until a minimum chronological period has elapsed, whichever occurs first.*

11.6 Attacks by bidders

In his exposition of VMs, Schechter ignores or minimizes the impact of two important attacks: resale and collusion. This section considers each attack in turn and suggests alterations to the structure of the bug auction to minimize their potential impact. It is important to note, however, that neither of these attacks is fully defended against:

[7]The former type of demand uncertainty will also undoubtedly affect entry; however, it is necessary to provide the competition incentive that ensures the vendor obtains value for its testing budget.

their existence must be weighed against the benefits provided by the auction in order to ascertain its utility to any given vendor.

11.6.1 Resale

Although it exists without the VM, the VM is likely to exacerbate the problem of resale: the potential for detectors to sell vulnerability reports on the black market, perhaps to malicious hackers. The vendor's willingness to pay the reward will ensure that it is eventually informed of vulnerabilities, but it does not ensure that no other entity is so informed. A smart detector might sell the vulnerability after reporting it to the vendor. She could truthfully advertise it as having been reported, but as-yet unremediated. Buyers could then pay a price commensurate with their utility for a short-lived vulnerability. The VM does not have any real solution to this dilemma. The potential result of a vendor offering a reward for vulnerabilities found in a commercially available software product could thus be a temporary *decrease* in the security of that product. The reward could induce more detectors to locate vulnerabilities which would then each be unremediated for some period of time and available on the black market. This problem cannot be entirely solved, and Schechter too readily dismisses it [Sch02a, p. 11]. However, an alteration to the bug auction could decrease the likelihood of resale.

Enhancement 4 *If a vulnerability is found to be exploited before the vendor can release a patch, the reward is reduced by some fraction.*

In order to reduce the likelihood of the detector reselling a vulnerability, the vendor pays its reward conditionally: some fraction of the reward is withheld if attacks using this vulnerability are found 'in the wild' before the patch is released. This is an incentive for the detector to not resell the vulnerability, but only if the resale value is less than the amount withheld from the reward. Some portion of the reward must always be paid to ensure that vulnerabilities available on the black market are reported to the vendor.

11.6.2 Collusion

Klemperer argues that the two most important aspects of practical auction design are the encouragement of entry and the deterrence of collusion [Kle04b]. The former goal has already been addressed. This section considers the problems of both employee-detector collusion and detector-detector collusion.

Schechter largely ignores the problem of employee-detector collusion (*e.g.* an engineer involved with creating the product could team up with a detector not employed by the vendor in order to create subtle vulnerabilities and then report them for mutual gain). This problem exists even without the market. A competitor could pay an engineer to insert a vulnerability into a product. However, the VM creates more opportunities (with lower transaction costs) for this attack. Schechter does suggest, in a different context, that the feedback from the VM be used to drive an incentive system for engineers; the more rapid feedback provided by the VM could enable rewards based on the quality of that engineer's work. A quality incentive system of this type might reduce the incentive for 'cheating' by colluding with external detectors. Enforcing legal or contractual penalties for those engineers found to be collaborating with external detectors is another means of reducing this risk. Sting operations, in which security officers employed by the vendor pretend to be detectors and attempt to entice engineers would also increase the risk to engineers of cheating. Nonetheless, it seems unlikely that this risk could be completely ameliorated, and at least to the extent that it is a risk without the VM, it remains a risk with it.

Even if employee-detector collusion is limited, the vendor must still concern itself with collusion between detectors (who could, for example, agree not to submit reports until the price reached some minimum). One important tool for preventing such collusion is to keep the number of detectors (bidders) unknown [Kle04b]. No group of colluding detectors can be sure that they will be the only detectors, thus limiting their ability to control the auction. Furthermore, because the participants for any single auction in the series are unknown, no colluding group can identify and punish a defecting member through retaliatory bidding [CS02].

Enhancement 5 *The exact number of detectors will not be public, although when the number is small an approximate count may be published to entice participation.*

The vendor may choose to indicate the approximate number of bidders when that number is small, because broadcasting this fact can induce entry by other bidders.[8] Because the number will be approximate, it need not remove the collusion deterrence effect of having an unknown number of bidders.

However, in a VM the vendor has no way of determining the number of potential detectors: it does not even know how many active detectors exist. In addition to its use in inducing entry, the vendor could obtain knowledge of this number and use it to tune the starting point and rate of reward change.

Enhancement 6 *Detectors benefit by registering in advance with the trusted third party.*

This enhancement must be implemented carefully. Detectors should not be *required* to register, because a user (who is not already a detector) who discovers a vulnerability must be able to report it for the reward. Moreover, the benefit should be targeted only at detectors: otherwise, if individuals not interested in testing can receive the benefit, then they will register and the count will be inaccurate. Although

[8]Menezes and Monteiro assert that if the bidders are risk neutral, knowledge of the number of bidders does not affect expected revenue from the auction [MM96]. See note 6 for a discussion of how their model differs from this one.

the benefit can vary with the situation, an example would be to reduce the transaction fee charged to detectors that have been registered for at least three months. This benefit provides an incentive for individuals who are actively planning to test the product and claim the reward; on the other hand, individuals who discover a vulnerability by chance still have an incentive to submit that vulnerability (because R_0 is greater than the transaction fee). Detectors register with the TTP, not the vendor, so that they can maintain their (pseudo)anonymity.

Even if the vendor chooses not to attempt to register detectors, Enhancement 5 highlights the importance of maintaining (pseudo)anonymity and keeping secret any knowledge of participation numbers. The vendor should keep the potential for collusion firmly in mind. Many incidental enhancements that might be considered to improve detectors' efficiency could also lead to collusion: for example, supplying an official newsgroup for communication between detectors.

11.7 Attacks by vendors

Schechter largely fails to consider that the vendor may be motivated to 'cheat' in order to save money; it has five primary avenues through which to attack the bug auction: not paying for vulnerability reports, releasing an exploit into the wild to reduce the payout to a bidder, abbreviating the length of the auction sequence, resetting a large reward, and falsely employing a large R_0 to jump-start the auction sequence. However, it is important to note that all of these attacks are against the detectors; the vendor has no viable means of altering the auction results in a way that cheats its *customers*.

The first attack is the simplest against which to defend: bidders refuse to participate unless the vendor employs a TTP tasked with verifying vulnerabilities, ranking their severity, and paying the reward [Sch04].

The second opportunity to cheat is to decrease a claimed reward by planting an exploit (to take advantage of Enhancement 4). However, the existence of exploits in the wild can result in significant damage to the product's reputation. This disincentive to cheating would likely dominate any potential savings, with the possible exception of a vulnerability that is almost trivial (at which point the savings are also diminished, because the reward spectrum guarantees that the payout is also trivial).

The third attack, abbreviating the length of the auction sequence, is more problematic. Enhancement 3 proposes that vendors commit to a minimum amount of reward money and a minimum chronological duration. The vendor can then legitimately halt the auction once *either* of these minima has been surpassed. This attack has an obvious defense:

Enhancement 7 *The trusted third party should hold in escrow those funds that the vendor has publicly committed to the auction.*

Of course, if no vulnerabilities are found in the product or the last reward is unclaimed at the end of the chronological limit, the vendor can be refunded the remaining money. (Although bidders cannot require a vendor to implement this enhancement, they can adjust their strategies if it is not implemented.)

The vendor may attempt to cheat if it is no longer interested in spending those funds previously committed to the auction. It is prevented from arbitrarily halting the sequence because of the reputation damage it would sustain. The alternative is for the vendor to employ an intermediary to report vulnerabilities of which it was already aware until the auction funds are exhausted. A number of disincentives reduce the risk of this attack. It requires that the vendor introduce these vulnerabilities, either in patches or before the product's release. Introducing the vulnerabilities in patches creates a risk that detectors may recognize that the vendor is cheating (due to the extraordinarily high rate of vulnerability introduction in patches) and cause the vendor to suffer a reputation penalty. Introducing vulnerabilities before the product's release (or leaving unfixed some of the vulnerabilities identified by internal detectors) would require sufficient foresight. Either of these scenarios creates a risk that the vulnerabilities will be identified and reported by an independent detector, thus causing the vendor to pay a penalty (the reward). Even if the vendor succeeds in ending the auction without arousing suspicion, it suffers a reputation penalty with respect to the number of vulnerabilities identified and reported in order to end the auction—unless the vendor is confident that the full amount of money committed to rewards will be claimed before the chronological end of the sequence. Essentially, the risk of the vendor cheating to end the auction early is significant only when the product is of poor quality, an attribute that can be observed by detectors. Detectors can thus factor this information into their strategies in order to compensate for the risk of a premature end to the auction.

The fourth attack is for the vendor to report (via a proxy) previously planted vulnerabilities in order to reset the reward when it reaches too large a value. In essence, the vendor is implementing a reservation price (a cap on the reward value). Although Schechter designed the VM with a reservation price, the vendor has a disincentive to employ an explicit reservation price because such caps discourage entry [LS94]. However, several disincentives seem likely to discourage the vendor from cheating. Pre-release, if the reward has gone unclaimed for a long period, then the software must be reasonably secure and vulnerability-free; the vendor should simply end that phase of the auction in anticipation of releasing the product. The vendor can then legitimately reset the reward. If the vendor cheats to reset the reward in the post-release phase, then it suffers a double blow to its reputation: it decreases the security assurance and also further suffers the reputation penalty of a vulnerability having been found (although the latter disincentive could be somewhat decreased if the reported vulnerability is minor).

A further disincentive to resetting the reward lies in the tension between value and speed. Resetting the reward maximizes value at the expense of speed, because detectors who have identified high-value vulnerabilities will wait for the reward to reach a high level again before reporting them. However, as noted in Section 1, vendors also value speed: they are motivated to rapidly release their products and to create the impression of security at as early of a stage in the product's life as possible. In addition, resetting the reward may not result in any real savings for the vendor (unless the vendor is attempting to prematurely end the auction). If detectors are waiting for the reward to reach a certain level before they will report vulnerabilities, then they are unlikely to dramatically alter their reserve price because the vendor has reset the reward. The vendor could continuously reset the reward, but only at the cost of an increased perception of product insecurity.

The fifth opportunity to cheat arises if the vendor employs Enhancement 1. It could advertise a large R_0 in order to create a high level of initial participation and then cheat (by having a proxy report a previously planted vulnerability) to avoid paying the cost of inducing such participation. However, it might thus fail to achieve the high level of participation that is its goal, and it further suffers the reputation penalty of a vulnerability having been discovered. (Again, unless it is confident that the product is of such poor quality that all of the funds committed to the auction will be claimed before the end of the sequence; however, if it already possesses this knowledge, then it should simply decrease the funds committed to the auction or not employ a bug auction at all.)

If the vendor employs Enhancement 7, a final disincentive for the latter three of these methods for the vendor to attack the auction is legal and financial. If it uses a proxy to cheat and claim back some of the funds already given to the TTP, then general accounting practices and taxation laws may make it difficult for the vendor to explain the reclaimed funds. Furthermore, the proxy still has to identify himself to the TTP, which may identify a link between him and the vendor.

In general, vendors that are concerned about their reputation have incentives that discourage cheating. If the vendor is no longer concerned with its reputation, there exists no real defense against its cheating. However, detectors would likely have other indications of this disregard and factor it into their strategies.

11.8 Fundamental problems

The previously mentioned attacks can be defended against—through the structural suggestions made above and through legal, cultural, and managerial tools employed during the course of the auction sequence. However, Schechter's market suffers from four fundamental shortcomings: its expense, the potentially harmful effect on the vendor's reputation, the loss of free testing, and the potential for copyright infringement.

11.8.1 Expense

Any vendor can employ the VM to enhance its pre-release testing: it may find that the VM, in combination with limited in-house 'white-box' testing, is more cost effective than relying entirely on in-house vendor detectors. Deriving the competitive benefit of the security assurance is more difficult, although Schechter does provide for bootstrapping the market to force your competitor to also use a VM (see Section 11.3).

However, in order to effectively advertise a quantitative level of security (*e.g.* a constant stable reward at $20,000), the vendor must always be willing to pay at least that amount for vulnerabilities reported after the product is made commercially available; if a rash of unique vulnerabilities are reported, this guarantee could be rather costly to the vendor.

Schechter implies that few vulnerabilities will be reported after the product is made commercially available, because he assumes that the VM has been used extensively during the pre-release testing of the product. As a result, Schechter assumes that, post-release, the reward will not be claimed with any great frequency.

One problem with these assumptions is the value most vendors see in being first to market. While Microsoft claims to be delaying product releases to ensure a high enough level of security [Bek02], they are also dominant in these markets. Vendors that lack the luxury of dominating their markets may be unwilling to employ the VM in testing for an open-ended period (*i.e.* until detectors stop claiming the reward).

Another problem lies in the nature of testing. As testing continues, the effort and time required to find new vulnerabilities probably increases, *provided* that the environment is constant. After a complex product is commercially released, the vendor may receive a flurry of new vulnerability reports because the commercial users employ the product in a wide variety of new environments. The vendor's detectors were unable to anticipate or replicate all of these environments and so did not find these vulnerabilities [BAB99]. Again, this factor implies that the vendor may find the reward it offers after the product is commercially released to be more costly than expected.

11.8.2 Reputation

The problems described in Section 11.8.1 may impact more than just the amount of money the vendor spends on rewards. If a number of vulnerabilities are reported in the period immediately after the product is released, the patches released to fix those vulnerabilities may create the appearance of insecurity.

Schechter notes that, for *detectors* considering whether or not to participate in the VM, the *perception* of security can be more important than the reality [Sch04, p. 47]. However, he does not fully address the impact of perception on the *customers* of the vendor. Although a product may be more secure if the VM is used during testing and after release, customers might still base their perception of security on the number of patches disseminated. Reputation may thus be as important as the three properties already mentioned: value, speed, and order.

One area of future research would be to identify an auction structure that can better account for reputation while still minimizing the costs to the vendor. The vendor could perhaps 'cluster' its fixes, waiting until is has corrected more than one vulnerability before issuing a patch. However, it would need to be conscious that delays in patching any single vulnerability might enable that vulnerability to be sold on the black market and exploited.

11.8.3 Loss of free testing

The existence of Schechter's market also alters the interaction that occurs between the vendor, vulnerability detectors, full-disclosure fora, and the black market that was described in Section 2.3. One negative effect is that the vendor will have to pay for the vulnerability reports that it previously received for free from vulnerability detectors. One the other hand, the VM will potentially draw more analysts and thus the product may more rapidly approach a secure state, in which most vulnerabilities, and particularly the most obvious ones, have been identified and remediated. The existence of resale means that the full-disclosure model of identification will not be significantly impacted, although again, the vendor will be paying for reports that it previously received for free.

11.8.4 Copyright infringement

Schechter specifies that all detectors must have "full and complete" access to the product [Sch04, 66]. This stipulation is necessary to ensure that the product is tested by the maximum possible number of detectors and that their costs are minimized. For open-source products, full and complete access is not problematic. For closed-source products, full and complete access will realistically be interpreted as access to the product in its executable form, not its source code. Even so, this requirement may be problematic for closed-source products. If implemented poorly, it could be used as a means of copyright infringement: individuals who have no intention of testing could acquire a free copy of the product for personal use. Schechter notes the possibility of this infringement occurring; to prevent it, he suggests that the testing copies could have some functionality removed or disabled so that they would be less useful to a would-be infringer. He argues that illegal copies of most software are widely available; as a result, those copies put into distribution for testing would not greatly increase the availability of free copies for those unwilling to pay. The effectiveness and importance of this attack (and the countermeasure of disabling functionality) will differ according to the product. If a product is not already widely available from illegal sources, the vendor may choose to disable functionality. If that solution is also not practical, the vendor may choose not to employ the VM with that product.

11.9 Other economic approaches

The benefits that could be derived from a bug auction, combined with the fact that commercial entities are now paying for vulnerabilities, suggest that this area is worth further investigation. Another reasonable question is whether or not there are other economic tools with which to approach information security. Unfortunately, two alternative approaches—creating a market for negative results and relying on insurance—both have severe practical shortcomings.

The first approach is to use auctions to also pay for negative results. In other words, vendors might also find it useful to know that an entity has run the software for ten thousand person-hours and not encountered any failures. However, this method suffers from an information asymmetry: the vendor does not have a reliable means of learning whether or not the seller has performed the claimed testing. Likewise, the seller cannot prove to the vendor that it has performed the testing. The asymmetry creates an economic incentive to fake the testing. A seller that does not perform the testing as it claims will have minimal costs; as a result, it can always underbid anybody who has actually performed the testing. This situation is similar to security certification: the solution there has been to use a trusted third party (*e.g.* a government) to either perform the testing or to certify the tester.

Another approach is to rely on insurers. The argument for insurance is that cyber-insurance underwriters assign premiums based upon a firm's IT infrastructure and the processes by which it is managed. This assessment results in both detailed best practices and, over the long run, a pool of data by which the insurance company can accurately assign a monetary value to the risks associated with certain practices or software. At the moment, however, the cyber-insurance market is both underdeveloped and underutilized. This market failure is caused by interdependent risk and inadequate volume.[9]

The problem of interdependent risk takes at least two forms. Firms are 'physically interdependent' because their IT infrastructure is connected via the Internet to other entities, which implies that the work a firm performs to secure itself may be undermined by failures at other firms. Firms are 'logically interdependent' because cyber attacks often exploit a vulnerability in a system used by many firms. For example, viruses or worms may have a global impact upon a specific software platform. This interdependence makes certain cyber-risks unattractive to insurers—particularly those where the risk is globally rather than locally correlated, such as worm and virus attacks, and systemic risks such as Y2K [B05; BK06].

[9]This paragraph and the two that follow are based on joint work with Ross Anderson, Tyler Moore, and Shishir Nagaraja [AMNO07].

Because a firm's security depends in part on the efforts of others, firms underinvest in both security technology and in cyber insurance. At the same time, insurance companies must charge a higher premium because the risks against which they are insuring are highly correlated: this higher premium may prevent the vast majority of firms from adequately insuring themselves. As a result, cyber insurance markets may lack the volume and liquidity to become economically efficient [B05].

Perhaps surprisingly, moral hazard has not proven to be a significant disincentive to the offering of insurance plans. (The moral hazard in this situation is that the insured have less incentive to invest in security.) This moral hazard is equivalent to insuring automobile drivers who may then drive less carefully: only the scale and complexity differ. However, insurance adjusters are accustomed to false claims and have experience in detecting them. They work to prevent moral hazard by establishing best practices and ensuring that the insured comply with those best practices. In addition, the use of large deductibles is another disincentive.

11.10 Conclusion

Schechter's VM is an innovative solution to a vexing problem in software security; although his proposal is daring, the existence of firms already purchasing vulnerabilities demonstrates that it is not inconceivable for software vendors to employ this idea. However, the expense of implementing the VM is not trivial: a robust theoretical understanding of the concept will encourage those vendors who might consider implementing it.

This chapter has identified areas where auction theory can improve the efficiency of the VM. It has highlighted attacks that were not considered or were too readily dismissed by Schechter. Finally, it has noted four fundamental weaknesses of the VM that can not readily be corrected. I argue that the VM is an important concept, but one that can be better considered as a bug auction. Considering the VM as an auction provides a more firm theoretical foundation for understanding the idea. Further, it enables the vendor to fine-tune the auction to increase its effectiveness.

A bug auction would provide vendors with: the assurance that it will learn about those vulnerabilities that are available on the black market, the ability to significantly increase scrutiny of a product (particularly early in the development life cycle when it is otherwise difficult to achieve this goal), and the ability to provide a qualitative assurance of security to their customers.

The arguments in favor of using a bug auction in the pre-release phase seem clear: the vendor benefits from both more testing and, more importantly, from testing in a wide variety of environments. This benefit is gained at the expense of a potentially more costly testing program and also a potential delay in releasing the product. While neither of these drawbacks are trivial, the current market emphasis on security seems likely to make them bearable for some vendors. Assessing the use of the bug auction in the post-release phase of a product's life is more difficult, but it has the potential to revolutionize the market for secure products. Although the implementation of a bug auction would require significant changes to the management culture of vendors and consumers, it nonetheless seems uniquely suited to providing a relatively strong security assurance and an efficient means of improving product quality and security.

Chapter 12

Future Work

Both the engineering and the economic approach to software security show promise: however, neither is yet fully realized. A great deal of future work is necessary before they can be utilized with confidence.

12.1 Engineering approach

Software engineers have frequently yearned for a large, high quality database of faults. In the security field, we are fortunate enough to have several such databases, some of which have government sponsorship (*e.g.* the US NVD). In their current form, these databases offer tantalizing yet ultimately inadequate information for the purposes of vulnerability discovery modeling. This work has sketched a design for a next generation of public vulnerability databases: the fleshing out and implementation of that design remain for future work. That task is perhaps the single most important one for achieving high quality, useful estimations on vulnerability discovery.

Katerina Goševa-Popstojanova & Kishore Trivedi have proposed a framework for a software reliability model which incorporates the possibility of dependent failures [GPT00]. A useful next step in the VDM field is to extend that work to provide for estimation and then to apply it to vulnerability data.

The Common Vulnerability Scoring System (CVSS) standard for classifying the severity of vulnerabilities has recently been proposed [FIR]. As it becomes widely applied, that standard will enable vulnerability discovery to be analyzed with respect to severity. For example, has the *severity* of discovered vulnerabilities changed over time? A decrease in the number of discoveries could be balanced by an increase in the severity of those vulnerabilities discovered.

12.2 Economic approach

Further work on economic approaches includes formally modeling the bug auction. The bug auction would be modeled as a sequential open first-price ascending auction with an unknown number of asymmetric independent private-value bidders, variable demand, high entry costs, and minimal bid costs. A formal model could also compare the value to the vendor of an asymmetric open first-price ascending auction with an asymmetric closed second-price auction. This chapter has assumed that the drawbacks of a closed second-price auction are more significant than those of an open first-price auction, but formal verification would be useful.

In addition, discovering a means of compensating for bidder asymmetry would increase the value of the bug auction to vendors.

Also, the costs and efficiency of current quality assurance methods might provide an interesting comparison with the bug auction. How much do companies currently spend for their quality assurance detectors to identify a single vulnerability? Do those detectors typically identify vulnerabilities of only a certain magnitude (*e.g.* are user interface faults found and remote vulnerabilities not found)?

Finally, a company named WabiSabiLabi has just been founded to serve as a trusted third party host for vulnerability auctions [Wab07]. The auctions enable detectors to sell the vulnerabilities they have discovered. The prices of these auctions will thus be a fruitful source of information for future investigations of vulnerability detection.

Chapter 13

Conclusion

A better means of measuring software security is an important tool for improving information security in general. While we have not yet advanced to a real measurement, I have proposed engineering and economic approaches to better understand software security.

13.1 Summary

The engineering approach builds upon existing work; however, it notes that the foundation for that existing work is weak. One critical component of a strong research foundation is a set of precise, widely accepted definitions. I proposed developer- and detector-perspective definitions for: vulnerability, vulnerability discovery models, detection event, software security, detectors, and the events and phases of the vulnerability detection life cycle. Using those definitions, I examined the widely-used National Vulnerability Database and the CVE list upon which it relies. I found that the maintainers of the CVE list are inconsistent in their definition of a vulnerability, which has resulted in wide variation in the content of NVD entries.

I also found that the existing literature on vulnerability discovery models cannot yet be trusted. The assumptions upon which these models are based have not been proved, and the NVD data to which they are applied is inaccurate. I thus compiled my own data set of every vulnerability patched by OpenBSD over an eight year period.

I used the NVD taxonomy and a custom taxonomy to categorize these vulnerabilities, and I found that the discovery of vulnerabilities in some of these categories was dependent. However, this result may be driven by the weakness of the available taxonomies, which tend to rely too heavily on a few, broad categories.

My examination of the OpenBSD vulnerabilities produced interesting results, although they are only representative of this one system. 63% of the 155 detection events that occurred during the study included a foundational vulnerability: one present in the source code of the first version in the study, OpenBSD 2.3. Moreover, even after almost eight years, 61% of the source code remained unchanged since the foundation version, 2.3.

Because the discovery of vulnerabilities was found to be dependent, I was unable to rely upon the VDMs commonly used in the literature. However, a graphical, non-significant, examination of the rate of vulnerability detection in OpenBSD indicated a decline over time.

Vulnerability disclosure policies are an important part of the vulnerability detection process. One aspect in the debate about various policies is whether or not the same vulnerability is likely to be independently discovered by different detectors. I found strong anecdotal evidence that this situation does occur with non-trivial frequency. This evidence validates an assumption debated in the literature on vulnerability disclosure policies.

The accuracy and effectiveness of VDMs depends upon the data to which they are applied. Before we can measure software vulnerabilities, we first need to collect better data on them. A consensus on key definitions is one step towards accomplishing that goal. Another step would be the creation of a next-generation vulnerability database that would contain more detailed and better data on the events of interest to those who would measure software security.

To complement the engineering approach, I proposed an economic one. The 'bug auction' would enable software vendors to ascertain the monetary value of vulnerabilities in their own product and that of their competitors. Moreover, it would enable a minimum guarantee by ensuring that the vendor will always pay a minimum reward to be informed of new vulnerabilities. This economic measure may be more a accurate and useful representation of software security than a software engineering approach.

13.2 Results

Software vulnerabilities and their discovery provide an important means of understanding software security, and a better understanding of software security is an important component of increasing that security and of measuring organizational risk. This work has illustrated two approaches to analyzing software vulnerabilities and their discovery: engineering and economic.

An engineering analysis of vulnerabilities can provide vendors with estimates of the number of vulnerabilities that will be found and the time to the discovery of the next vulnerability. These two estimates, in turn, will allow vendors to allocate resources for patching and to compare the effectiveness of different development methodologies that have been applied to similar products. While previous work on vulnerability discovery models has been flawed, its shortcomings have been identified here and can be remedied in future work. This work also proposes a set of data that can be incorporated into existing vulnerability databases. Over time, this data will lead to an improved understanding of vulnerabilities.

The estimates provided by the engineering analysis are dependent upon the effort invested in vulnerability discovery. A complementary economic approach can use bug auctions to normalize the effort invested in vulnerability detection and thus provide a more accurate overall measure of the difficulty in finding vulnerabilities in a system.

Abbreviations & Symbols

Ω_{known}	total number of vulnerabilities known
R	Reward
R_0	initial Reward
AB	Average Bias
AE	Average Error
AIC	Akaike Information Criteria
AME	Alhazmi-Malaiya Effort-based
AML-C	Alhazmi-Malaiya Logistic - Constrained
AML	Alhazmi-Malaiya Logistic
AVE	Access Validation Error
BCE	Boundary Condition Error
BIND	Berkeley Internet Name Daemon
BO	Buffer Overflow
CERT/CC	see: Chapter 10, Footnote 2
CERT	Computer Emergency Response Team
CE	Configuration Error
CM	Coding Mistake
CVE	Common Vulnerability and Exposure
CVS	Concurrent Versions System
DEI	Design Error, Interaction
DE	Design Error
ECH	Exceptional Condition Handling error
EE	Environmental Error
FD	File Descriptor
FS	Format String
F	File handling / file system
HC	Heap Corruption
HPP	Homogenous Poisson Process
IEEE	Institute of Electrical and Electronics Engineers
IE	Internet Explorer
IIS	Internet Information Services
IOU	Integer Overflow/Underflow

ISS	Internet Security Systems
IVE	Input Validation Error
KLOC	one thousand Lines Of Code
LVD	Linear Vulnerability Discovery
MLOC	Million Lines Of Code
MTTNV	Mean Time To Next Vulnerability
NC	Not Classified
NHPP	Non-Homogenous Poisson Process
NPD	Null Pointer Dereference
NVD	National Vulnerability Database
OSVDB	Open Source Vulnerability DataBase
PLR	Prequential Likelihood Ratio
POSIX	Portable Operating System Interface
RH	RedHat Linux
RPC	Remote Procedure Call
R	Race
SNMP	Simple Network Management Protocol
SRM	Software Reliability Model
TTNV	Time To Next Vulnerability
TTP	Trusted Third Party
URL	Uniform Resource Locator
US	United States
US/CERT	United States / Computer Emergency Response Team
VDM	Vulnerability Disclosure Model
VM	Vulnerability Market

Bibliography

[3Co05] 3Com, 2005. Zero day initiative — 3com — tippingpoint, a division of 3com — program details. http://www.zerodayinitiative.com/details.html. Last accessed on 26 June 2007. (Cited on pages 21 and 97.)

[AFM00] William A. Arbaugh, William L. Fithen, and John McHugh, December 2000. Windows of vulnerability: A case study analysis. *IEEE Computer*, 33(12):52–59. ISSN 0018-9162. doi:10.1109/2.889093. URL http://www.cs.umd.edu/~waa/pubs/Windows_of_Vulnerability.pdf. (Cited on pages 17, 20, and 23.)

[AIA93] AIAA/ANSI, 1993. *Recommended Practice: Software Reliability*. ANSI. ISBN 1-56347-024-1. R-013-1992. (Cited on pages 26, 27, 37, and 67.)

[Ake70] George A. Akerlof, 1970. The market for 'lemons': Quality uncertainty and the market mechanism. *The Quarterly Journal of Economics*, 84(3):488–500. (Cited on page 12.)

[AKN+04] Ahish Arora, Ramayya Krishnan, Anand Nadkumar, Rahul Telang, and Yubao Yang, May 2004. Impact of vulnerability disclosure and patch availability - an emprical analysis. In *Workshop on Economics and Information Security (WEIS)*. Minneapolis, MN, USA. URL http://www.dtc.umn.edu/weis2004/telang.pdf. (Cited on pages 20, 22, 88, and 89.)

[AKS96] Taimur Aslam, Ivan Krsul, and Eugene H. Spafford, 1996. Use of a taxonomy of security faults. In *Proceedings of the 19th NIST-NCSC National Information Systems Security Conference*, pages 551–560. Baltimore, MD, USA. URL http://csrc.nist.gov/nissc/1996/papers/NISSC96/paper057/PAPER.PDF. (Cited on pages 36 and 50.)

[ALRL04] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr, Jan-Mar 2004. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions On Dependable And Secure Computing*, 1(1):11–33. ISSN 1545-5971. doi:10.1109/TDSC.

2004.2. URL http://se2c.uni.lu/tiki/se2c-bib_download.php?id=2433. (Cited on page 20.)

[AM05a] Omar H. Alhazmi and Yashwant K. Malaiya, 2005. 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: IEEE Computer Society. ISBN 0-7695-2482-6. doi:10.1109/ISSRE.2005.30. URL http://www.cs.colostate.edu/~malaiya/pub/issre05.pdf. (Cited on pages 26, 27, 31, 34, 37, 38, and 40.)

[AM05b] Omar H. Alhazmi and Yashwant K. Malaiya, 2005. Quantitative vulnerability assessment of systems software. In *Proceedings of the IEEE Reliability and Maintainability Symposium (RAMS'05)*, pages 615–620. Alexandria, VA, USA. URL http://www.cs.colostate.edu/~malaiya/530/rams05.pdf. (Cited on pages 32, 33, 35, 37, 38, 39, 40, 73, and 91.)

[AM06a] Omar H. Alhazmi and Yashwant K. Malaiya, 2006. 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: IEEE Computer Society. ISBN 0-7695-2684-5. doi:10.1109/ISSRE.2006.26. URL http://www.cs.colostate.edu/~malaiya/pub/alhazmio-measuring.pdf. (Cited on pages 33, 34, 37, 38, and 39.)

[AM06b] Omar H. Alhazmi and Yashwant K. Malaiya, January 2006. Prediction capabilities of vulnerability discovery models. In *Proceedings of the IEEE Reliability and Maintainability Symposium (RAMS '06)*, pages 86–91. ISBN 1-4244-0007-4. ISSN 0149-144X. doi:10.1109/RAMS.2006.1677355. URL http://www.cs.colostate.edu/~malaiya/pub/2006RM-151.pdf. (Cited on pages 30, 33, 34, 37, 38, 39, and 40.)

[AMNO07] Ross Anderson, Tyler Moore, Shishir Nagaraja, and Andy Ozment, 2007. Incentives and information security. In Noam Nisan, Tim Roughgarden, Eva Tardos, and Vijay Vazirani, eds., *Algorithmic Game Theory*, chapter 25. Cambridge University Press. ISBN 9780521872829. URL http://www.cambridge.org/uk/catalogue/catalogue.asp?isbn=9780521872829. (Cited on pages 16 and 116.)

[Amo94] Edward G. Amoroso, 1994. *Fundamentals of Computer Security Technology*. Upper Saddle River, NJ, USA: Prentice-Hall PTR. (Cited on pages 49 and 58.)

[AMR05] Omar H. Alhazmi, Yashwant K. Malaiya, and Indrajit Ray, August 2005. Security vulnerabilities in software systems: A quantitative perspective. In *Proceedings of the IFIP WG 11.3 Working Conference on Data and Applications Security*, pages 281–294. URL http://www.cs.colostate.edu/~malaiya/635/IFIP-10.pdf. (Cited on pages 33, 38, and 39.)

[And01] Ross Anderson, December 2001. Why information security is hard - an economic perspective. In *17th Annual Computer Security Applications Conference*. URL http://www.cl.cam.ac.uk/ftp/users/rja14/econ.pdf. New Orleans, LA, USA. (Cited on page 12.)

[And02] Ross Anderson, June 20-21 2002. Security in open versus closed systems - the dance of Boltzmann, Coase and Moore. In *Open Source Software: Economics, Law, and Policy*. Toulouse, France. URL http://www.ftp.cl.cam.ac.uk/ftp/users/rja14/toulouse.pdf. (Cited on page 34.)

[ATX04] Ahish Arora, Rahul Telang, and Hao Xu, May 2004. Optimal policy for software vulnerability disclosure. In *Workshop on Economics and Information Security (WEIS)*. URL http://www.dtc.umn.edu/weis2004/xu.pdf. Minneapolis, MN, USA. (Cited on pages 90 and 91.)

[B05] Rainer Böhme, June 2005. Cyber-insurance revisited. In *Workshop on the Economics of Information Security*. Cambridge, MA, USA. URL http://www.infosecon.net/workshop/pdf/15.pdf. (Cited on pages 116 and 117.)

[BAB99] Robert M. Brady, Ross J. Anderson, and Robin C. Ball, September 1999. Murphy's law, the fitness of evolving species, and the limits of software reliability. Technical Report 471, University of Cambridge Computer Laboratory. URL http://www.york.ac.uk/inst/exec/pdf/SequA.pdf. (Cited on pages 34, 91, 98, 99, and 114.)

[BAMF00] Hilary K. Browne, William A. Arbaugh, John McHugh, and William L. Fithen, November 2000. A trend analysis of exploitations. Technical Report CS-TR-4200, University of Maryland. URL http://www.cs.umd.edu/~waa/pubs/CS-TR-4200.pdf. (Cited on page 23.)

[BB96] Matt Bishop and David Bailey, September 1996. A critical analysis of vulnerability taxonomies. Technical Report CSE-96-11, University of California at Davis. URL http://seclab.cs.ucdavis.edu/projects/vulnerabilities/scriv/ucd-ecs-96-11.pdf. (Cited on page 50.)

[BCHM01] David W. Baker, Steven M. Christey, William H. Hill, and David E. Mann, September 7–9 2001. The development of a common enumeration of vulnerabilities and exposures. In *Proceedings of the Second International Workshop on Recent Advances in Intrusion Detection*. URL http://www.raid-symposium.org/raid99/PAPERS/Hill.pdf. (Cited on page 51.)

[BCLS90] Sarah Brocklehurst, P. Y. Chan, Bev Littlewood, and John Snell, 1990. Recalibrating software reliability models. *IEEE Transactions on Software Engineering*, 16(4):458–470. ISSN 0098-5589. doi:10.1109/32.54297. URL http://csdl.computer.org/dl/trans/ts/1990/04/e0458.pdf. (Cited on page 33.)

[Bek02] Scott Bekker, April 2002. Windows .net server delays complicate longhorn schedule. URL http://www.entmag.com/news/article.asp?EditorialsID=5316. (Cited on page 114.)

[Bis95] Matt Bishop, May 1995. A taxonomy of UNIX system and network vulnerabilities. Technical Report CSE-95-10, University of California at Davis. URL http://seclab.cs.ucdavis.edu/projects/vulnerabilities/scriv/ucd-ecs-95-10.pdf. (Cited on page 50.)

[BK96] Jeremy Bulow and Paul Klemperer, 1996. Auctions versus negotiations. *The American Economic Review*, 86(1):180–194. (Cited on page 106.)

[BK06] Rainer Böhme and Gaurav Kataria, June 2006. Models and measures for correlation in cyber-insurance. In *Workshop on the Economics of Information Security*. Cambridge, UK. URL http://weis2006.econinfosec.org/docs/16.pdf. (Cited on page 116.)

[BKdV93] Michael R. Baye, Dan Kovenock, and Casper G. de Vries, 1993. Rigging the lobbying process: An application of the all-pay auction. *The American Economic Review*, 83(1):289–294. (Cited on page 105.)

[BL96] Sarah Brocklehurst and Bev Littlewood, 1996. Techniques for prediction analysis and recalibration. In Michael R. Lyu, ed., *Handbook of Software Reliability Engineering*, chapter 4, pages 119–164. McGraw-Hill. ISBN 0-07-039400-8. URL http://www.cse.cuhk.edu.hk/~lyu/book/reliability/. (Cited on pages 30 and 31.)

[BLOJ94] Sarah Brocklehurst, Bev Littlewood, Tomas Olovsson, and Erland Jonsson, April 1994. On measurement of operational security. Technical Report 160, Predictably Dependable Computing Systems. URL http://www.newcastle.research.ec.org/pdcs/trs/papers/160.pdf. (Cited on page 18.)

[CCR05] Hasan Cavusoglu, Huseyin Cavusoglu, and Srinivasan Raghunathan, June 2005. Emerging issues in responsible vulnerability disclosure.

In *Workshop on the Economics of Information Security (WEIS)*. Cambridge, MA, USA. URL http://infosecon.net/workshop/pdf/cavusoglu.pdf. (Cited on page 90.)

[CER05] CERT/CC, 2005. CERT/CC vulnerability disclosure policy. http://www.cert.org/kb/vul_disclosure.html. As viewed on 20 May 2007. (Cited on pages 22 and 89.)

[CKI03] Shuo Chen, Zbigniew Kalbarczyk, and Ravishankar K. Iyer, 2003. Analysis of security vulnerabilities using multiple data sources. (Cited on page 51.)

[CKXI03] Shuo Chen, Zbigniew Kalbarczyk, Jun Xu, and Ravishankar K. Iyer, 2003. A data-driven finite state machine model for analyzing security vulnerabilities. In *International Conference on Dependable Systems and Networks*, pages 605–614. URL http://ase.csc.ncsu.edu/junxu/Papers/DSN2003_VulnModel.pdf. (Cited on page 51.)

[CS02] Peter Cramton and Jesse A. Schwartz, 2002. Collusive bidding in the FCC spectrum auctions. *Contributions to Economic Analysis and Policy*, 1(1). URL http://www.cramton.umd.edu/papers2000-2004/cramton-schwartz-collusive-bidding.pdf. (Cited on page 110.)

[CSTB01] Computer Science and Telecommunications Board, 2001. *Computers at Risk: Safe Computing In the Information Age*. Washington, DC: National Academy Press. ISBN 0-309-04388-3. URL http://www.nap.edu/books/0309043883/html/index.html. (Cited on page 19.)

[CW00] LJ Camp and C Wolfram, October 2000. Pricing security. In *Proceedings of the CERT Information Survivability Workshop*, pages 31–39. URL http://www.ljean.com/files/isw.pdf. Boston, MA, USA. (Cited on page 97.)

[CW02] Steve Christey and Chris Wysopal, February 2002. Responsible vulnerability disclosure process. URL http://www.whitehats.ca/main/about_us/policies/draft-christey-wysopal-vuln-disclosure-00.txt. (Cited on page 89.)

[dR05] Theo de Raadt, November 2005. Exploit mitigation techniques (in OpenBSD, of course). In *OpenCON 2005*. URL http://www.openbsd.org/papers/ven05-deraadt/index.html. Venice, Italy. (Cited on page 86.)

[Ess04a] Stefan Esser, May 2004. Advisory 07/2004: CVS remote vulnerability. URL http://marc.theaimsgroup.com/?l=bugtraq&m=108498454829020&w=2. (Cited on page 60.)

[Ess04b] Stefan Esser, June 2004. Advisory 09/2004: More CVS remote vulnerabilities. URL http://lists.grok.org.uk/pipermail/full-disclosure/2004-June/022441.html. (Cited on page 60.)

[Far96] William Farr, 1996. Software reliability modeling survey. In Michael R. Lyu, ed., *Handbook of Software Reliability Engineering*, chapter 3, pages 71–117. McGraw-Hill. ISBN 0-07-039400-8. URL http://www.cse.cuhk.edu.hk/~lyu/book/reliability/. (Cited on pages 27, 28, 29, and 67.)

[FIR] FIRST. Common vulnerability scoring system (CVSS-SIG). URL http://www.first.org/cvss/. Viewed on February 12, 2007. (Cited on page 119.)

[Gau92] Olivier Gaudoin, December 1992. Optimal properties of the Laplace trend test for software reliability models. *IEEE Transactions on Reliability*, 41(4):525–532. ISSN 0018-9529/92. (Cited on page 83.)

[GLO+04] Rupert Gatti, Stephen Lewis, Andy Ozment, Thierry Rayna, and Andrei Serjantov, May 2004. Sufficiently secure peer-to-peer networks. In *Workshop on Economics and Information Security (WEIS)*. URL http://www.cl.cam.ac.uk/~jo262/papers/weis04-lewis_etal-suffsec.pdf. Minneapolis, MN, USA. (Cited on page 16.)

[GO79] Amrit L. Goel and K. Okumoto, August 1979. Time-dependent error-detection rate model for software and other performance measures. *IEEE Transactions on Reliability*, 28(3):206–211. ISSN 0018-9529. (Cited on page 31.)

[Gol04] M. Corey Goldman, January 2004. Code that can't be cracked. *The Star*. (Cited on page 96.)

[Gou03] Stephen Jay Gould, 2003. *The Hedgehog, the Fox, and the Magister's Pox*. New York, NY, USA: Harmony Books. ISBN 978-0-609-60140-2. (Cited on page 16.)

[GPT00] Katerina Goševa-Popstonjanova and Kishor S. Trivedi, 2000. Failure correlation in software reliability models. Technical Report 00/04, Center for Advanced Computing and Communication (CACC). URL http://www.ece.ncsu.edu/cacc/show_techreport.php?id=22. (Cited on pages 37 and 119.)

[GS05] Rajeev Gopalakrishna and Eugene H. Spafford, May 2005. A trend analysis of vulnerabilities. Technical Report 2005-05, CERIAS, Purdue University. URL https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive/archive/2005-05.pdf. (Cited on pages 27, 32, 36, 39, 41, 52, 60, and 63.)

[GSV05] Rajeev Gopalakrishna, Eugene H. Spafford, and Jan Vitek, 2005. Vulnerability likelihood: A probabilistic approach to software assurance. Technical Report 2005-06, CERIAS, Purdue University. URL http://homes.cerias.purdue.edu/~rgk/papers/2005-06.pdf. 2005-06. (Cited on pages 18, 27, 32, and 39.)

[Hat97] Les Hatton, 1997. Re-examining the fault density - component size connection. *IEEE Software*, 14(2):89–97. ISSN 0740-7459. URL http://ieeexplore.ieee.org/iel1/52/12658/00582978.pdf?isnumber=&arnumber=582978. (Cited on pages 73 and 75.)

[HL98] John D. Howard and Thomas A. Longstaff, October 1998. A common language for computer security incidents. Technical Report SAND98-8667, Sandia National Laboratories. URL http://www.cert.org/research/taxonomy_988667.pdf. (Cited on page 50.)

[ICA02] ICAT, December 2002. ICAT metabase documentation. This page has been removed. However, a copy was obtained from the Internet Archive at: http://web.archive.org/web/20021201184650/http://icat.nist.gov/icat_documentation.htm. (Cited on pages 40, 41, 42, and 52.)

[iDe07] iDefense Labs, 2007. Vulnerability contributor program // iDefense Labs. URL http://labs.idefense.com/vcp/. Last accessed on 26 June 2007. (Cited on pages 21 and 97.)

[IEE90] IEEE, September 1990. IEEE standard glossary of software engineering terminology. (Cited on pages 17 and 18.)

[JS82] Charles A. Holt Jr. and Roger Sherman, 1982. Waiting-line auctions. *The Journal of Political Economy*, 90(2):280–294. (Cited on page 105.)

[KL96] Karama Kanoun and Jean-Claude Laprie, 1996. Trend analysis. In Michael R. Lyu, ed., *Handbook of Software Reliability Engineering*, chapter 10, pages 401–437. McGraw-Hill. ISBN 0-07-039400-8. URL http://www.cse.cuhk.edu.hk/~lyu/book/reliability/. (Cited on page 83.)

[Kle04a] Paul Klemperer, 2004. A survey of auction theory. In Paul Klemperer, ed., *Auctions: Theory and Practice*, chapter 1A. Princeton University Press. URL http://www.nuff.ox.ac.uk/users/klemperer/VirtualBook/Survey.pdf. (Cited on page 105.)

[Kle04b] Paul Klemperer, 2004. What really matters in auction design. In Paul Klemperer, ed., *Auctions: Theory and Practice*, chapter 3A. Princeton University Press. URL http://www.nuff.ox.ac.uk/users/klemperer/VirtualBook/wrm6.pdf. (Cited on pages 109 and 110.)

[KLSW02] Todd Kaplan, Israel Luski, Aner Sela, and David Wettstein, 2002. All-pay auctions with variable rewards. *The Journal of Industrial Economics*, L(4):417–430. URL http://www.econ.bgu.ac.il/facultym/wettstn/sp15.pdf. (Cited on page 105.)

[Krs98] Ivan Victor Krsul, May 1998. *Software Vulnerability Analysis*. Ph.D. thesis, Purdue University. (Cited on page 18.)

[Kru74] Anne O. Krueger, 1974. The political economy of the rent-seeking society. *The American Economic Review*, 64(3):291–303. (Cited on page 105.)

[KT04] Karthik Kannan and Rahul Telang, May 2004. Economic analysis of market for software vulnerabilities. In *Workshop on Economics and Information Security (WEIS)*. URL http://www.dtc.umn.edu/weis2004/kannan-telang.pdf. Minneapolis, MN, USA. (Cited on page 97.)

[LBMC94] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi, 1994. A taxonomy of computer program security flaws. *ACM Computing Surveys*, 26(3):211–254. ISSN 0360-0300. doi:10.1145/185403.185412. URL http://doi.acm.org/10.1145/185403.185412. (Cited on page 50.)

[Liv03] Brian Livingston, November 2003. Microsoft's patch-a-month club. http://www.eweek.com/article2/0,1895,1490665,00.asp. (Cited on page 29.)

[Lon03] Tom Longstaff, June 2003. CERT experience with security problems in software. http://research.microsoft.com/projects/SWSecInstitute/slides/Longstaff.pdf. (Cited on page 19.)

[LS94] Dan Levin and James L. Smith, 1994. Equilibrium in auctions with entry. *The American Economic Review*, 84(3):585–599. (Cited on page 112.)

[Lyu96] Michael R. Lyu, 1996. Introduction. In Michael R. Lyu, ed., *Handbook of Software Reliability Engineering*, chapter 1, pages 3–22. McGraw-Hill. ISBN 0-07-039400-8. URL http://www.cse.cuhk.edu.hk/~lyu/book/reliability/. (Cited on pages 18, 26, 27, and 35.)

[Mic05] Microsoft, February 2005. Revamping the security bulletin release process. http://www.microsoft.com/technet/security/bulletin/revsbwp.mspx. (Cited on page 28.)

[Mil07] Charlie Miller, May 2007. The legitimate vulnerability market: Inside the secretive world of 0-day exploit sales. In *Workshop on the Economics of Information Security*. Pittsburgh, PA, USA. URL http://weis2007.econinfosec.org/papers/29.pdf. Pittsburgh, PA, USA. (Cited on page 97.)

[MIO87] John D. Musa, Anthony Iannino, and Kazuhira Okumoto, 1987. *Software Reliability: Measurement, Prediction, Application*. New York: McGraw-Hill Book Company. ISBN 0-07-044093-X. (Cited on pages 27, 36, and 67.)

[MIT05] MITRE, February 2005. Common vulnerabilities and exposures. http://www.cve.mitre.org/. (Cited on page 41.)

[MM87] R. Preston McAfee and John McMillan, 1987. Auctions and bidding. *Journal of Economic Literature*, 25(2):699–738. (Cited on pages 103 and 104.)

[MM96] Flavio M. Menezes and Paulo K. Monteiro, September 1996. A note on auctions with endogenous participation. Technical Report 9610003, Washington University in St. Louis. URL http://econwpa.wustl.edu:80/eps/mic/papers/9610/9610003.pdf. Economics Working Paper Archive, Microeconomics. (Cited on pages 106 and 110.)

[MMSM03] L. Ma, S. Mandujano, G. Song, and P. Meunier, February 2003. Sharing vulnerability information using a taxonomically-correct, web-based cooperative database. Technical Report 2001-03, Purdue University. URL https://www.cerias.purdue.edu/tools_and_resources/bibtex_archive//archive/2001-03.pdf. (Cited on page 51.)

[MR85] Eric S. Maskin and John G. Riley, 1985. Auction theory with private values. *The American Economic Review*, 75(2):150–155. (Cited on page 104.)

[MR89] Eric S. Maskin and John G. Riley, 1989. Optimal multi-unit auctions. In Frank Hahn, ed., *The Economics of Missing Markets, Information, and Games*, pages 312–335. Oxford University Press. (Cited on page 105.)

[NISa] NIST. National vulneraiblity database FAQ. http://nvd.nist.gov/faq.cfm. Viewed on March 20, 2007. (Cited on page 39.)

[NISb] NIST. National vulneraiblity database (NVD). http://nvd.nist.gov/. Viewed on February 12, 2007. (Cited on page 45.)

[NPC03] Tibor Neugebauer and Paul Pezanis-Christou, February 2003. Bidding at sequential first-price auctions with(out) supply uncertainty: A laboratory analysis. Technical Report 558.03, Unitat de Fonaments de l'Anlisi Econmica (UAB) and Institut d'Anlisi Econmica (CSIC). URL http://pareto.uab.es/wp/2003/55803.pdf. (Cited on page 108.)

[Ope98] OpenBSD, February 1998. CVS – OpenBSD security page, revision 1.12. URL http://www.openbsd.org/cgi-bin/cvsweb/~checkout~/www/security.html?rev=1.12&content-type=text/html. (Cited on page 44.)

[OS06a] Andy Ozment and Stuart E. Schechter, July 2006. Bootstrapping the adoption of internet security protocols. In *Workshop on the Economics of Information Security (WEIS)*. URL http://www.cl.cam.ac.uk/~jo262/papers/Ozment_Schechter-Bootstrapping_the_Adoption_of_Internet_Security_Protocols.pdf. Cambridge, UK. (Cited on page 16.)

[OS06b] Andy Ozment and Stuart E. Schechter, 2006. Milk or wine: Does software security improve with age? In *Proceedings of the 15th Usenix Security Symposium*, pages 93–104. URL http://www.cl.cam.ac.uk/~jo262/papers/Ozment_and_Schechter-Milk_Or_Wine-Usenix06.pdf. (Cited on pages 16, 20, 32, 37, 38, 67, 68, 70, 73, and 77.)

[OSD06] Andy Ozment, Stuart E. Schechter, and Rachna Dhamija, March 2006. Web sites should not need to rely on users to secure communications. In *The W3C Workshop on Transparency and Usability of Web Authentication*. URL http://www.cl.cam.ac.uk/~jo262/papers/24-ozment-dont-rely.pdf. New York, NY, USA. (Cited on page 16.)

[OSVed] OSVDB, undated. The open source vulnerability database. URL http://www.osvdb.org/. http://www.osvdb.org/. (Cited on pages 42 and 45.)

[Ozm04] Andy Ozment, May 2004. Bug auctions: Vulnerability markets reconsidered. In *Workshop on Economics and Information Security (WEIS)*. URL http://www.cl.cam.ac.uk/~jo262/papers/

weis04-ozment-bugauc.pdf. Minneapolis, MN, USA. (Cited on pages 16 and 97.)

[Ozm05] Andy Ozment, June 2005. The likelihood of vulnerability rediscovery and the social utility of vulnerability hunting. In *Workshop on the Economics of Information Security (WEIS)*. URL http://www.cl.cam.ac.uk/~jo262/papers/weis05-ozment-vulnrediscovery.pdf. Cambridge, MA, USA. (Cited on pages 16, 20, 32, 37, 38, 41, and 68.)

[Ozm06] Andy Ozment, 2006. Software security growth modeling: Examining vulnerabilities with reliability growth models. In Dieter Gollmann, Fabio Massacci, and Artsiom Yautsiukhin, eds., *Quality Of Protection: Security Measurements and Metrics*. Milan, Italy: Springer. ISBN 978-0-387-29016-4. URL http://www.cl.cam.ac.uk/~jo262/papers/qop2005-ozment-security_growth_modeling.pdf. (Cited on pages 16, 20, 26, 29, 32, 37, 38, and 68.)

[Ozm07] Andy Ozment, 2007. *Vulnerability Discovery & Software Security*. Ph.D. thesis, University of Cambridge. (Cited on page 16.)

[Pie02] Frank Piessens, 2002. A taxonomy of causes of software vulnerabilities. In M. Vouk, ed., *Supp. Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE'02*, pages 47–52. URL http://www.cs.kuleuven.ac.be/cwis/research/distrinet/resources/publications/39426.pdf. (Cited on page 51.)

[PP07] Charles P. Pfleeger and Shari Lawrence Pfleeger, 2007. *Security in Computing*. Upper Saddle River, NJ, USA: Prentice Hall. ISBN 0-13-239077-9. 4th ed. (Cited on page 19.)

[R] R. The R project for statistical computing. URL http://www.r-project.org/. (Cited on pages 62 and 74.)

[Rai01] Rain Forest Puppy, 2001. Full disclosure policy (RFPolicy) v2.0. http://www.wiretrip.net/rfp/policy.html. (Cited on page 89.)

[Res03] Eric Rescorla, August 2003. Security holes... who cares? In *Proceedings of the 13th Usenix Security Symposium*. (Cited on page 77.)

[Res04] Eric Rescorla, May 2004. Is finding security holes a good idea? In *Workshop on Economics and Information Security (WEIS)*. URL http://www.dtc.umn.edu/weis2004/rescorla.pdf. Minneapolis, Minnesota. (Cited on pages 31, 38, 45, 77, 79, and 83.)

[Res05] Eric Rescorla, Jan-Feb 2005. Is finding security holes a good idea? *IEEE Security & Privacy*, 3(1):14–19. doi:10.1109/MSP.2005.17. URL http:

//doi.ieeecomputersociety.org/10.1109/MSP.2005.17. (Cited on pages 31, 37, 39, and 41.)

[RS81] John G. Riley and William F. Samuelson, 1981. Optimal auctions. *The American Economic Review*, 71(3):381–392. (Cited on page 103.)

[RSA04] RSA, April 2004. URL http://web.archive.org/web/20040404053235/http://www.rsasecurity.com/rsalabs/challenges/. This page has been removed. However, a copy was obtained from the Internet Archive. (Cited on page 96.)

[Sch02a] Stuart E. Schechter, October 2002. How to buy better testing: Using competition to get the most security and robustness for your dollar. In *Infrastructure Security Conference*. URL http://www.eecs.harvard.edu/~stuart/papers/isc2002.pdf. Bristol, UK. (Cited on pages 97, 99, 100, 104, and 109.)

[Sch02b] Stuart E. Schechter, May 2002. Quantitatively differentiating system security. In *Workshop on Economics and Information Security (WEIS)*. URL http://www.sims.berkeley.edu/resources/affiliates/workshops/econsecurity/econws/31.pdf. Berkeley, CA, USA. (Cited on pages 97 and 102.)

[Sch04] Stuart E. Schechter, May 2004. *Computer Security Strength & Risk: A Quantitative Approach*. Ph.D. thesis, Harvard University. URL http://www.eecs.harvard.edu/~stuart/papers/thesis.pdf. (Cited on pages 97, 111, 114, and 115.)

[SDOF07] Stuart E. Schechter, Rachna Dhamija, Andy Ozment, and Ian Fischer, May 2007. The emperor's new security indicators: An evaluation of website authentication and the effect of role playing on usability studies. In *Proceedings of the 2007 IEEE Symposium on Security and Privacy*. URL http://www.cl.cam.ac.uk/~jo262/papers/emperor.pdf. Oakland, CA, USA. (Cited on page 16.)

[Seca] Secunia. The Full Disclosure mailing list. http://lists.grok.org.uk/full-disclosure-charter.html. (Cited on pages 20 and 88.)

[Secb] SecurityFocus. The Bugtraq mailing list. http://www.securityfocus.com/archive/1. (Cited on pages 20 and 88.)

[Sec05] SecurityFocus, 2005. Securityfocus HOME vulns archive: Vendor. http://www.securityfocus.com/bid/. (Cited on pages 42 and 45.)

[SNO07] SNOsoft Research Team, January 2007. SNOsoft Research Team: Exploit acquisition program. URL http://snosoft.blogspot.com/2007/

01/exploit-acquisition-program.html. Last accessed on 26 June 2007. (Cited on page 97.)

[Sto03] Walt Stoneburner, January 2003. SMERFS (Statistical Modeling and Estimation of Reliability Functions for Systems). URL http://www.slingcode.com/smerfs/. http://www.slingcode.com/smerfs/. (Cited on page 67.)

[Tre05] Trend Micro, 2005. Vulnerability exploits break records. URL http://uk.trendmicro-europe.com/global/products/collaterals/white_papers/VulnerabilityWP.pdf. (Cited on page 13.)

[Twi99] Tymm Twillman, September 1999. Bugtraq: Exploit for proFTPd 1.2.0pre6. URL http://seclists.org/bugtraq/1999/Sep/0328.html. (Cited on page 59.)

[Uni02] University of Oulu, Electrical and Information Engineering, October 2002. PROTOS test-suite: c06-snmpv1. URL http://www.ee.oulu.fi/research/ouspg/protos/testing/c06/snmpv1. (Cited on page 60.)

[Vic61] William Vickrey, 1961. Counterspeculation, auctions, and competitive sealed tenders. *The Journal of Finance*, 16(1):8–37. (Cited on page 103.)

[Wab07] WabiSabiLabi, July 2007. Finally a marketplace site for security research. URL http://www.wslabi.com/wabisabilabi/news.do? (Cited on page 120.)

[WAM06a] Sung-Whan Woo, Omar H. Alhazmi, and Yashwant K. Malaiya, November 2006. An analysis of the vulnerability discovery process in web browsers. URL http://www.cs.colostate.edu/~malaiya/pub/wooSEA_2006.pdf. (Cited on pages 33, 38, 39, and 52.)

[WAM06b] Sung-Whan Woo, Omar H. Alhazmi, and Yashwant K. Malaiya, September 2006. Assessing vulnerabilities in Apache and IIS HTTP servers. In *IEEE International Symposium on Dependable, Autonomic and Secure Computing*, pages 103–110. ISBN 0-7695-2539-3. doi:10.1109/DASC.2006.21. URL http://www.cs.colostate.edu/~malaiya/pub/178woo.pdf. (Cited on pages 32, 33, 37, 38, 39, and 52.)

[WHR98] Keith Waehrer, Ronald M Harstad, and Michael H Rothkopf, 1998. Auction form preferences of risk-averse bid takers. *The RAND Journal of Economics*, 29(1):179–192. (Cited on page 105.)

[Wik07] Wikipedia, June 2007. Format string attack. URL `http://en.wikipedia.org/wiki/Format_string_vulnerabilities`. Accessed on 18 June 2007. (Cited on page 59.)

[XFed] X-Force, undated. Internet security systems – x-force database. `http://xforce.iss.net/xforce/search.php`. (Cited on pages 42 and 45.)