2016

The Software Vulnerability Ecosystem: Software Development In The Context Of Adversarial Behavior

Saender Aren Clark
University of Pennsylvania, saender@crypto.com

The Software Vulnerability Ecosystem: Software Development In The Context Of Adversarial Behavior

Abstract
Software vulnerabilities are the root cause of many computer system security fail- ures. This dissertation addresses software vulnerabilities in the context of a software lifecycle, with a particular focus on three stages: (1) improving software quality dur- ing development; (2) pre- release bug discovery and repair; and (3) revising software as vulnerabilities are found.

The question I pose regarding software quality during development is whether long-standing software engineering principles and practices such as code reuse help or hurt with respect to vulnerabilities. Using a novel data-driven analysis of large databases of vulnerabilities, I show the surprising result that software quality and software security are distinct. Most notably, the analysis uncovered a counterintu- itive phenomenon, namely that newly introduced software enjoys a period with no vulnerability discoveries, and further that this "Honeymoon Effect" (a term I coined) is well-explained by the unfamiliarity of the code to malicious actors. An important consequence for code reuse, intended to raise software quality, is that protections inherent in delays in vulnerability discovery from new code are reduced.

The second question I pose is the predictive power of this effect. My experimental design exploited a large-scale open source software system, Mozilla Firefox, in which two development methodologies are pursued in parallel, making that the sole variable in outcomes. Comparing the methodologies using a novel synthesis of data from vulnerability databases, These results suggest that the rapid-release cycles used in agile software development (in which new software is introduced frequently) have a vulnerability discovery rate equivalent to conventional development.

Finally, I pose the question of the relationship between the intrinsic security of software, stemming from design and development, and the ecosystem into which the software is embedded and in which it operates. I use the early development

lifecycle to examine this question, and again use vulnerability data as the means of answering it. Defect discovery rates should decrease in a purely intrinsic model, with software maturity making vulnerabilities increasingly rare. The data, which show that vulnerability rates increase after a delay, contradict this. Software security therefore must be modeled including extrinsic factors, thus comprising an ecosystem.

Degree Type
Dissertation

Degree Name
Doctor of Philosophy (PhD)

Graduate Group
Computer and Information Science

First Advisor
Matt Blaze

THE SOFTWARE VULNERABILITY ECOSYSTEM: SOFTWARE DEVELOPMENT IN THE CONTEXT OF ADVERSARIAL BEHAVIOR

Saender A. Clark

A DISSERTATION

in

Computer and Information Science

Presented to the Faculties of the University of Pennsylvania in Partial Fulfillment of the Requirements for the Degree of Doctor of Philosophy

2017

Matthew Blaze
Associate Professor of Computer and Information Science
Supervisor of Dissertation

Jonathan M. Smith
Olga and Alberico Pompa Professor of Engineering and Applied Sciences
and Professor of Computer and Information Science
Co-supervisor of Dissertation

Lyle Ungar
Professor of Computer and Information Science
Graduate Group Chairperson

THE SOFTWARE VULNERABILITY ECOSYSTEM: SOFTWARE DEVELOPMENT IN THE CONTEXT OF ADVERSARIAL BEHAVIOR

Acknowledgments

"Security is mostly a superstition. It does not exist in nature, nor do the children of men as a whole experience it. Avoiding danger is no safer in the long run than outright exposure. Life is either a daring adventure, or nothing." (Helen Keller)

I never thought I would get the opportunity to pursue a Ph.D. in Computer Science. I grew up in a conservative religion that actively and subtly discouraged me from furthering my education as far as I'd hoped. I thought I would never get the chance to participate in Security research, explore my own ideas, or study for an advanced degree. It was a dream I kept buried deep until a group of mentors took an interest in me and were willing to take a risk on a non-traditional and eccentric student, opened a new world to me, and set me on this daring adventure.

First and foremost in this group, is my adviser Dr. Matt Blaze. I met Matt at a Hackers on Planet Earth conference. Kindred spirits, we became friends immediately. After hearing my dream over dinner one night, Matt invited me to hang out in his lab at Penn and gave me the status of 'Visiting Scholar'. He also assigned me sets of problems from Herstein's "Abstract Algebra" book to work through, (he says it was "to see how serious I was about doing more than hacking".) Having never taken C.S. theory, it wasn't easy, but like every challenge Matt has given me, he was certain I could do it, even though I wasn't. During my first week in the lab, we took possession of used law enforcement wiretap equipment and the subsequent brouhaha set the

tone for my entire academic career. I have been incredibly lucky and privileged to participate in a number of prominent and fascinating research projects.

Even so, it was several years before I had the courage to actually matriculate and apply to become a 'real' Ph.D. student. I was consumed with deep feelings of inadequacy, expecting to fail out or be found out as a fraud at any moment.

Championing a frightened and underprepared advisee for five or more years is inarguably a risky proposition for a professor, yet, he never wavered in his belief in me. He gave me a shoulder to cry on, masses of encouragement, and a safety net that allowed me to take risks. My academic background had several large lacunae and I could devote several chapter-length sections to enumerating the ways in which Matt supported and nurtured me, but me describe two memorable experiences that I believe are unique to being one of Matt's students:

I was part of a team Matt led in Ohio's EVEREST Review of electronic voting machines. Although the Ohio study required me to spend 12-15 hours a day in a tight and uncomfortable office-cum-SCIF, working with Matt and the other teammates on such an important and politically relevant study was one of the highlights of my academic career. It was the unanswered questions I was left with after this study, that lead to the work presented in this dissertation.

It also led to my being invited to Norway to speak about the problems with e-voting, at what I *thought* was a normal hackercon. It wasn't. The audience contained the Minister and members of Parliament in charge of elections and led the next day to my (along with a diverse group of hacker friends, who were also invited to speak at the conference), pwning [1] the entire University of Oslo computer network (where the new internet voting system was to be tested), in an attempt to demonstrate why e-voting and internet voting were a *bad idea*. We made the cover of "Computer World" Norway that week. (see Figure 1)

[1] in hacker parlance, taking over a network or computer system by means of exploiting vulnerabilities.

- E-valg er ikke trygt Av Leif Kirknes (Computerworld) 24.02.2009 kl. 12:54 Kilde: VG NETT

SKEPTISKE TIL E-VALG: T.v. Ryan "1057" Clarke; Brad "Renderman" Haines; Victor Teissler; Luke "Purø" McOmic; Guy Martin; Sandy Clark; Michael "Theprez98" Schearer og Deviant Ollam. Foran sitter Mike "Dragom" Kenshaw. (Foto: Leif Kirknes)

* Forsøk med elektronisk valg i 2011

Elektroniske valg er ikke trygt, heller ikke den norske varianten. Det mener i alle fall en gruppe amerikanske «snille» hackere, som har besøkt Norge for å snakke om sikkerhet på konferansen Hackcon. Listen over feilkilder er på flere A4-sider.
En av disse, Ph.d-kandidat ved University of Pennsylvania, Sandy Clark, har konkret erfaring med å hacke slike valgsystemer. Hun har vært med å teste sikkerheten til e-valg-systemet i Ohio, på lovlig vis.
-Uansett hvordan vi forsøkte og uansett hvor vi så, fant vi problemer, sier hun.
Forskerne måtte jobbe i en lukket bunkers og låse inn arbeidet i en safe
hver gang de var ferdige for dagen. Til slutt ble det laget to rapporter,en med og en uten detaljert beskrivelse av hvordan angrepene ble gjennomført.

Figure 1: Cover of ComputerWorld Magazine-Norway Edition

The second experience also received a lot of media attention. Matt and I are both radio Hams. While setting up a radio receiver in the lab, Matt was randomly spinning the tuner dial and suddenly we were listening to a law enforcement surveillance operation. This led to us finding serious vulnerabilities in the protocols, specifications, UI, implementation and usage of the P25 Radios used by all government agencies from the Secret Service to the Postal Service. I spent countless hours listening to unintentional clear-text collected from our listening posts, helped develop a law enforcement version of the Marauder's Map from "Harry Potter", and met with numerous FBI, DHS and other government agents to help them mitigate the problems we discovered. It was worth it, our research won the "Best Paper" award from

the Usenix Security Conference, one of the preeminent conferences in my field.

Whatever I may accomplish in the future as a computer scientist, some part of the credit will always be due to Matt. He is more than a mentor. He is an inspiration, my ally, my co-conspirator, and one of my best friends.

I owe a tremendous amount to my co-adviser, Jonathan M. Smith. I'm pretty sure he isn't aware of how much I have learned from him, and how much of his writing style I have copied from him. From him, I learned to ask perceptive questions, and to add depth to my ideas. He taught me to critique my thoughts, and to add rigor to my research. His advice is almost prescient (In fact, he is *always* right). Jonathan involved himself in my success early on of his own volition, teaching me, providing me with direction, and counseling me, even though he had no responsibility towards me. I began referring to him as my co-adviser well before I knew that title involved actual formal commitments. I was extremely fortunate (and thrilled) when he agreed to serve that role officially. Jonathan's ability to see to the heart of problems and find solutions, his brilliance, and his gentle guidance have been invaluable to me. I have learned an enormous amount from him. Working with him on the DARPA SAFEST project led to some amazing opportunities, (including a paper co-authored with the hackers who were hired to Red Team our software, and an invitation to present our research in Tel Aviv.) Throughout my academic career he has provided me invaluable advice. I am deeply indebted to him.

My colleagues along the way have challenged, motivated, and inspired me, teased me, and made my tenure as a graduate student the best job (to use the word loosely) I have ever had. In particular, I would like to acknowledge my extraordinarily over-competent ex-minion Michael Collis and "the team", Eric Cronin, Travis Goodspeed, Perry Metzger, Andrew Righter, Micah Sherr and Gaurav Shah. Bouncing ideas off of them was a joy and resulted in making work into play.

During the process of writing this dissertation, I received enormously helpful feedback from my thesis committee. I thank Professors Steve Zdancewic, Nadia

Heninger, Chris Murphy, and Kevin Fall for their many helpful suggestions and comments and editing assistance. Of course, this work could not have been possible without the tireless help of my co-advisors, Matt Blaze and Jonathan M. Smith.

No CIS graduate student completes a thesis without significant support from the department's staff members. In particular, Mike Felker, the amazingly multitasking Graduate Coordinator, and Mark West the long suffering Manager of the SEAS Accounting and Finance department, helped me successfully navigate Penn's numerous bureaucracies and never appeared exasperated even though my unique circumstances often required their direct intervention.

I owe a great deal of thanks to my friends Bill and Lorette Cheswick, for their mentorship and for the introduction to the original 'Hackers' group, a community of the most creative and amazing people, (many of whom invented the tech that's commonplace today) who accepted me immediately and wholeheartedly as one of them, helping me believe in myself. And to my white/grey/blackhat hacker 'family' who I meet at cons all over the world, especially Mudge, Noise, l0sTboy, Deviant Ollam, Zac/Fab, Renderman, Jon, Tamzen, Cas, Walker, Q, Heidi, Spikey, Tarah and many others too numerous to mention. As one, this hugely diverse group welcomed a scared little neo-hacker and gave me the freedom to learn and explore, and let me be me. Every day, this community continues to provide me the opportunity to have the childhood I always wanted.

Lastly, especial thanks to my brother Brett Nemeroff, for his unconditional love, and his unbreakable confidence in me. Brett, you and I are so alike though our talents expressed themselves so differently. You are an inspiration and keeping up with you a challenge. Your faith in me, your encouragement and your calm support have been my anchor while finishing this dissertation. I couldn't have made it without you.

To anyone reading this, my advice is to:

- **Fail well**
- **Fail often**
- **Stay a beginner forever**
- **And above all, Don't waste time being afraid.**

-sandy [2]

"But leave the Wise to wrangle, and with me The Quarrel of the Universe let be: And, in some corner of the Hubbub couch'd, Make Game of that which makes as much of Thee."

(Omar Khayyam, The Rubaiyat of Omar Khayyam)

[2]This work is partially supported by Project EVEREST, office of the Secretary of State, State of Ohio, the Olga and Alberico Pompa Chair, Google Advanced Technology and Projects , MURI grant FA9550-12-1-0400 "Science of Cyber Security: Modeling, Composition, and Measurement", administered by the U.S. Air Force under Grant FA9550-08-1-0352. This work is also partially supported by the Defense Advanced Research Project Agency (DARPA) and Space and Naval Warfare Systems Center Pacific under Contract No. N66001-11-C-4020 and DARPA LASSES - (LADS) Program (DARPA-BAA-15-61). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the Defense Advanced Research Project Agency and Space and Naval Warfare Systems Center Pacific.

Professors Blaze and Smith's work was supported by the Office of Naval Research under N00014-07-1-907, Foundational and Systems Support for Quantitative Trust Management; Professor Smith received additional support from the Office of Naval Research under the Networks Opposing Botnets effort N00014-09-1-0770, and from the National Science Foundation under CCD-0810947, Blue Chip: Security Defenses for Misbehaving Hardware. Professor Blaze received additional support from the National Science Foundation under CNS-0905434 TC: Medium: Collaborative: Security Services in Open Telecommunications

ABSTRACT

THE SOFTWARE VULNERABILITY ECOSYSTEM: SOFTWARE DEVELOPMENT IN THE CONTEXT OF ADVERSARIAL BEHAVIOR

Saender A. Clark

Matthew Blaze

Jonathan M. Smith

Abstract

Software vulnerabilities are the root cause of many computer system security failures. This dissertation addresses software vulnerabilities in the context of a software lifecycle, with a particular focus on three stages: (1) improving software quality during development; (2) pre-release bug discovery and repair; and (3) revising software as vulnerabilities are found.

The question I pose regarding software quality during development is whether long-standing software engineering principles and practices such as code reuse help or hurt with respect to vulnerabilities. Using a novel data-driven analysis of large databases of vulnerabilities, I show the surprising result that software quality and software security are distinct. Most notably, the analysis uncovered a counterintuitive phenomenon, namely that newly introduced software enjoys a period with no vulnerability discoveries, and further that this "Honeymoon Effect" (a term I coined) is well-explained by the unfamiliarity of the code to malicious actors. An important consequence for code reuse, intended to raise software quality, is that protections inherent in delays in vulnerability discovery from new code are reduced.

The second question I pose is the predictive power of this effect. My experimental design exploited a large-scale open source software system, Mozilla Firefox, in which two development methodologies are pursued in parallel, making that the sole variable in outcomes. Comparing the methodologies using a novel synthesis of data from vulnerability databases, These results suggest that the rapid-release cycles used in agile software development (in which new software is introduced frequently) have a vulnerability discovery rate equivalent to conventional development.

Finally, I pose the question of the relationship between the intrinsic security of software, stemming from design and development, and the ecosystem into which the software is embedded and in which it operates. I use the early development

lifecycle to examine this question, and again use vulnerability data as the means of answering it. Defect discovery rates should decrease in a purely intrinsic model, with software maturity making vulnerabilities increasingly rare. The data, which show that vulnerability rates increase after a delay, contradict this. Software security therefore must be modeled including extrinsic factors, thus comprising an ecosystem.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

'In theory there is no difference between theory and practice. In practice there is.' (Yogi Berra)

1.0.1 Is Software Reliability The Same As Software Security?

The software making up today's computer systems does not exist in a vacuum. Whether the purpose is to configure an FPGA, stream video on a smart-phone, or provide virtual high-performance computation in the cloud, software is dependent on and interacts with its environment in ways which are not well understood.

The root cause of many current computer and network security threats can be traced to errors in software. As software is an engineered artifact, the discipline of software engineering has emerged to model and manage such factors as cost and time estimates, feature selection, code maturity and software quality (which has grown to include software security). [BCH+95, Bro95b, CS97].

The software engineering community has devoted over three decades to designing models and testing development methodologies for improving the quality of software,

particularly for finding and removing software defects before release and for accurately predicting runtime reliability.

This research has resulted in a widely accepted software quality standard, ISO/IEC 25010:2011 [fS11] with well tested Software Reliability Models (SRM) and recommended best practices, including strong support for software reuse and multi-system portability. [Ram12]

A major assumption made by Software Reliability Models (SRM) is that software is released with some number of defects that can be categorized on the basis of how easy each is to find. A further assumption is made that the easy-to-find defects are discovered and fixed *early* in the software life-cycle, quickly leading to a state where only difficult-to-find vulnerabilities are left and the software can be considered reliable, and its quality is judged to be high. In fact, while software quality has come to include such diverse elements as customer satisfaction and usability, nearly all models and metrics for measuring software quality revolve around defect discovery and removal. [IOfS01, ISO11, Gre01, Kan02]

This method for determining software reliability has proven to work so well that the world has come to depend on it; digitizing everything from personal cars and smart homes to national power grids and transport systems. Through the use of SRMs, the software engineering community has been able to answer questions regarding functionality, such as (will my software dependably do X?) and reliability (providing Mean Time To Failure (MTTF) guarantees). Today, ubiquitous interconnectivity, the trend toward virtualization, network controlled systems management, and the propensity to store even frequently accessed data remotely in 'the cloud' has massively increased the complexity and the scale of damage that might result from failure. This means that models for predicting reliability will continue to be important for some time to come.

Reliability and functionality are not the only software engineering concerns. The security of these systems is now also of paramount importance. When the software

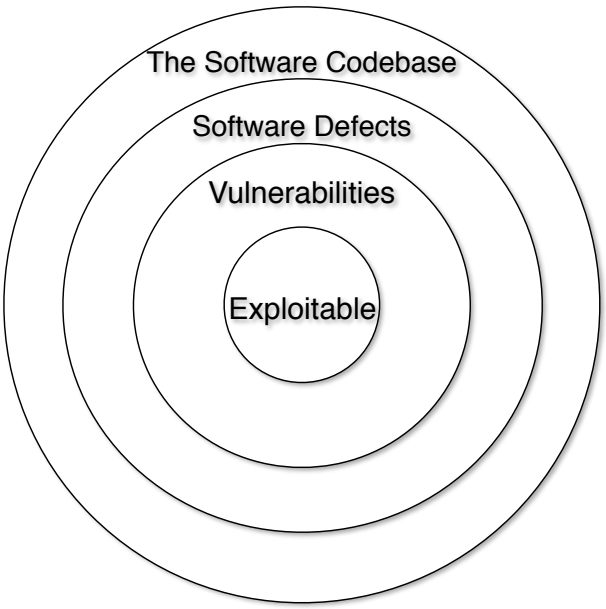


Figure 1.1: In a software codebase, some subset of the defects may be vulnerabilities, which may be found and exploited

ships with defects or *bugs*, some may be *vulnerabilities*, and some subset of these vulnerabilities will be discovered and further engineered into *exploits* (see Figure 1.1) which may then be used, sold or saved for later use. Predicting, finding and removing defects before or shortly after release has therefore, also been considered essential for the security of the product. Unlike software reliability, software security is much harder to measure. While there have been many attempts to adapt the proven reliability models and methodology to predict and quantify software security, they have met with limited success and there is, at present, no widely accepted model. Vulnerability Discovery Models (VDM)s, the equivalent of SRMs for security vulnerabilities, are unable to give answers to simple security questions such as one

of the most frequently asked questions of software developers: “Is my code secure?”, and the question most often asked by software users: “Is my system vulnerable to a software exploit?” Likewise, while Software Engineering (SWE) community has methodologies which provide confidence that a significant portion of functional defects have been removed, and a timeframe of when the remaining defects are likely to be discovered and even guarantee of reliability, software *security* has no methodology which provides an equivalent metric.

1.1 Secure Software Development Theory vs. Practice

Mainstream software engineering practice has development procedures intended to produce secure systems. Examples of Secure Development Models (SDM)s include Process Improvement Models, used in the ISO/IEC 21827 Secure Systems Engineering-Capability Maturity Model SSE-CMM [Jel00], (originated by the U.S. National Security Agency, but now an international standard), Microsoft’s Secure Development Lifecycle (SDL) [HL06], Oracle’s Software Security Assurance Process [Har14] and the Comprehensive, Lightweight Application Security Process (CLASP) [Vie05]. The goal of these models

is:

> "*To design, build, and deploy secure applications, [...] integrate security into your application development life cycle and adapt your current software engineering practices and methodologies to include specific security-related activities*”.

[MMW+05]

A major characteristic of these models is an emphasis on a significant investment of resources devoted to security at each stage of the development life-cycle before

initial product release. The expectation is that these models will help developers create a high quality, i.e., highly secure finished product.

For today's developers, the need to survive in a dynamic, rapidly changing, highly competitive marketplace has forced many software vendors to shift their focus from the high initial investment resource intensive, slow-moving secure development models to new highly adaptive, rapid-release cycle models and feature-driven development strategies.

Even though security experts have long chastised software developers for favoring adding new features over writing less vulnerable code, [MC09] the survival of a product in competitive software markets *requires* the frequent introduction of new features particularly for user-facing software systems such as web browsers embroiled in features arms races.

It comes as no surprise then, that major web browser developers, Google (Chrome) and Mozilla (Firefox), Apple (Safari), and Microsoft (Internet Explorer), have overhauled their development lifecycle, moving from large-scale, infrequent releases of new versions with later patches as needed, to releases with new features at much shorter, regular intervals. While software developed and released through these Rapid Release Cycles (RRC)s may also include bug fixes in a release, *Agile* approaches to software development such as Extreme Programming (XP) [Con04], Adaptive Software Development (ASD) [Hig13], and Feature Driven Development (FDD) [CLDL99] are primarily intended to ensure customer satisfaction via rapid feature delivery [BBvB+01] rather than to produce secure code [BK04], or improve the quality of existing code. [Nig11, Laf10] [1]

[1]New releases of Chrome and Firefox versions occur every six weeks. The primary intent of each RRC iteration is to get new features to users as rapidly as possible.

This practice stands in stark contrast to those recommended by the secure development models. More importantly, several characteristics of the RRC programming model strongly conflict with those considered necessary for developing secure software. In fact, the U.S. Department of Homeland Security [oHS06] assessed each of the fourteen core principles of the Agile Manifesto [BBvB+01] and found six principles to have negative implications for security, with only two having possible positive implications. Attempts to reconcile security with Agile development [SBK05, WBB04, KMH08] have noted that many of the practices recommended for security actually undermine the rapid iterations espoused by the Agile Manifesto [BBvB+01] (see Section 2.4.1).

1.2 Software Security Models Theory vs. Practice

Software Vulnerability Discovery Models (VDMs) resemble Software Reliability Models (SRMs), but instead of providing metrics from which to determine software quality before release, VDMs focus predominantly on predicting attacks against mature software systems.

Unfortunately, VDMs do not adequately provide insight into the number of remaining exploitable vulnerabilities, or accurately predict the time to next exploit. Problems with existing VDMs are discussed in detail in the next chapter.

I postulate several factors contributing to VDMs poor performance in predicting number of attacks and time to exploit.

First, VDMs rely exclusively on the *intrinsic qualities* of the software for a measure of its initial security. Consequently, one expectation common to users of VDMs is that the low-hanging fruit vulnerabilities are found quickly and patched. The remaining vulnerabilities (which are increasingly difficult to find) are presumed to take

much longer to discover, which leads one to consider the software "secure". [2] A VDM with those expectations would predict that vulnerabilities are found fastest shortly after the release of a product, with the rate of discovery decreasing thereafter.

The implications of such a VDM are significant for software security. It would suggest, for example, that once the rate of vulnerability discovery was sufficiently small, that the software is "safe" and needs little attention. It also suggests that software modules or components that have stood this "test of time" are appropriate candidates for reuse in other software systems. If this VDM model is wrong, these implications will be false and may have undesirable consequences for software security.

Second, the majority of VDMs assume that the software is *a finished product* at the time of release, and it is expected that it will remain unchanged throughout its lifetime. Thus the VDMs only consider *static* systems.

Third, VDMs assume that the security of the software they model is independent of the larger system to which it belongs. The models assume that the operating system (OS), architecture, hardware, network and other concurrent applications that are part of the ecosystem in which it functions are static (unchanging), that problems once solved never return *and* that problems (even security related issues) in other parts of the ecosystem are outside the scope of the security of the software being modeled.

1.3 Early In The Secure Software Lifecycle, Theory Differs From Practice

The VDMs and the SDMs mentioned above and discussed in detail in Chapter 2 have been in use for many years. In theory, they should provide developers with insight into their software's vulnerability life-cycle as well as a metric for determining the

[2] Similar to how software with its easy to find defects is considered 'reliable'.

quantity or expected time to discovery of the vulnerabilities remaining to be found. They are unable to provide any such assurance. The VDMs proposed to measure or predict software security apply their attention to one of two areas of the software lifecycle, focusing either on the period right before the product is released, or on the stable period in the main part of its lifecycle. In both cases, all three assumptions apply.

This dissertation is the first to investigate the relationship between the intrinsic security of software and the environment in which it operates. I show that properties of the ecosystem in which the software functions may positively or negatively affect the security of a member of that ecosystem, and further, the surprising result, that the early vulnerability lifecycle is not captured when modeling the security of software.

I present the results of an empirical analysis of vulnerability discovery in the early lifecycle of a software release. I demonstrate a surprising and counter-intuitive finding, that the likelihood of vulnerability discovery in the period immediately following the release of new software is contrary to what the models would predict. Instead of easy-to-find vulnerabilities being found and fixed quickly resulting in software becoming more and more secure a few months after release, the opposite appears to be the case. Early in a release lifecycle, vulnerabilities appear to take time to find and exploit. I call this newly discovered phenomenon the *Honeymoon Effect* and I discuss this in detail in Chapter 3.

To validate the Honeymoon Effect, I analyze the early vulnerability lifecycle of a single software product (Mozilla Firefox) developed under two distinct software development processes. I show that even software developed using a methodology inconsistent with secure software design best practises still experiences this phenomenon.

My evidence suggests that the Honeymoon Effect is related to the attacker's learning curve. This characteristic is a property of the *software's ecosystem*, not

intrinsic to the software itself. It is *extrinsic* to the security properties of the software that can be controlled by the developers.

I propose a new model to describe this relationship.

1.4 Research Questions

This dissertation explores the following research questions:

- **Is Software More Vulnerable When It Is New?** In Chapter 3, I present the results of a large-scale data analysis of vulnerability discovery rates early in the software lifecycle. I correlate the results with the vulnerability discovery rates of later versions of the same software to determine when early in the lifecycle software is the most vulnerable.

- **Do Frequent Code Changes Result in Less or More Secure Software?** In Chapter 4 I analyze the likelihood of vulnerability discovery in a single software product developed using the traditional, slow *Waterfall* development methodology with the likelihood of vulnerability discovery in the same software developed using an *Agile* methodology.

- **What Role does Legacy Code Play in the Exploit Lifecycle** In *Section 3.4* of Chapter 3, I analyze the density of vulnerabilities per lines of code resulting from code carried over from a previous version compared to the density of vulnerabilities found in code new to that version. I also analyze the number of legacy vulnerabilities that experience the Honeymoon Effect, and compare the length of the Honeymoon Period of the vulnerabilities found in legacy code to the length of the Honeymoon Period of vulnerabilities found in new code.

- **Do Simple Changes to Software From Version To Version Provide Insight into the Early Vulnerability Lifecycle?** I analyze the effect of a

single intrinsic property, the magnitude of code changes, as a predictive metric for the Honeymoon Effect. I present the results of this analysis in Chapter 5.

- **Does Software Quality Equate to Software Security?** In Chapters 3 and 4, I analyze the vulnerability discovery rates over several versions of different software products and demonstrate that early in the software lifecycle, the results are not consistent with expectations of widely-regarded software quality models.

1.5 Contributions

This dissertation makes the following contributions:

1. I discovered the *Honeymoon Effect*. I provide evidence that early in the software vulnerability lifecycle the delay in the attacker's learning curve appears to benefit the defender. I explore this discovery in depth in Chapter 3.

2. I analyzed the rate and magnitude of vulnerability discovery of software developed using two contrasting design models. One methodology, the *Waterfall Model*, adheres to traditional secure software development best practices, while the other, the *Rapid-Release Cycle or Agile Model*, conflicts with many of the recommended secure software development best practices. This analysis resulted in several surprising findings:

    - Frequent code changes do not result in an increase in easy-to-find vulnerabilities, (i.e., low-hanging fruit). Rapid feature-driven development does not benefit the attacker.

    - Software designed according to Agile methodologies considered incompatible with secure software design best practises, still benefits from the Honeymoon Effect

- For code developed using Agile methodologies as well as for code developed using Waterfall methods, code the majority of vulnerabilities are found in legacy code. Code reuse is *bad* for security.

These findings are presented in Chapter 4.

3. I correlated code change properties with subsequent vulnerability discovery and discovered, surprisingly, that in the Mozilla Firefox software lifecycle, the magnitude of coarse-grained code changes *does not* correlate with the length of time before which a vulnerability was discovered. This suggests that existing vulnerability discovery models that consider only intrinsic software quality are not sufficient for predicting vulnerability discovery. I present these results in Chapter 5.

4. I provide evidence that software security depends in part on properties extrinsic to those that the developer can directly control. I demonstrate that current software engineering security models fail to consider these extrinsic properties. I explore this discovery throughout Chapters 2, 3, 4, and 5.

5. I propose a model of the intrinsic and extrinsic properties that influence whether vulnerabilities will be found. Such a model could provide a framework for building richer predictive models for evaluation of the interactions and environmental factors that affect the security of software systems in deployed environments. In the concluding chapter, Chapter 6, I present this model, demonstrate that it is both necessary and sufficient to describe the software security ecosystem and further, provide a use case to illustrate its effectiveness.

Some of the research presented in this dissertation has been previously published in peer-reviewed academic computer security conferences. The results presented in Chapter 3 were published as *Familiarity Breeds Contempt: The Honeymoon Effect and the Role of Legacy Code* and appear in the Proceedings of the 26th Annual

Computer Security Applications Conference, ACSAC 2010. The results presented in Chapter 4 were published as *Moving targets: Security and rapid-release in firefox.* and appear in the Proceedings of the 2014 ACM SIGSAC Conference On Computer And Communications Security, ACM, 2014. I first proposed the ecosystem model as a framework for dynamic and adaptive security strategies at the Shmoocon Conference 2012 in a presentation called *Inside the OODA Loop: Towards an Agressive Defense.* In all of these works, the ideas, research, direction and focus are my own.

1.6 Organization

The remainder of this dissertation is organized as follows. In Chapter 2, I survey prior work in software security metrics. I introduce *the Honeymoon Effect* in Chapter 3. I analyze the *Honeymoon Effect* in different software development processes in Chapter 4. I analyze the efficacy of simple intrinsic properties as metrics for the Honeymoon Effect in Chapter 5. In Chapter 6, I propose the Dynamic/Adaptive Security Ecosystem Model (DASEM). I conclude with a discussion of promising future directions for developing predictive and adaptive models for securing software in a hostile ecosystem.

Chapter 2

Related Work

"A science is as mature as its measurement tools" (Louis Pasteur)

2.1 Background

This chapter explores previous attempts to adapt software engineering models, such as models for defect discovery and for quantifying system reliability, to software security. Such attempts looked closely at the life-cycle of exploited vulnerabilities, testing whether intrusion discovery provides any predictive benefit, attempted to determine what effect vulnerability density has on overall software quality, whether newer additions to code had more or fewer vulnerabilities, and compared vulnerability discovery models to see which, if any, best described the entire vulnerability discovery process. I will show, none provide satisfactory metrics for predicting vulnerability discovery. This chapter also examines other different types of software security models and their limitations.

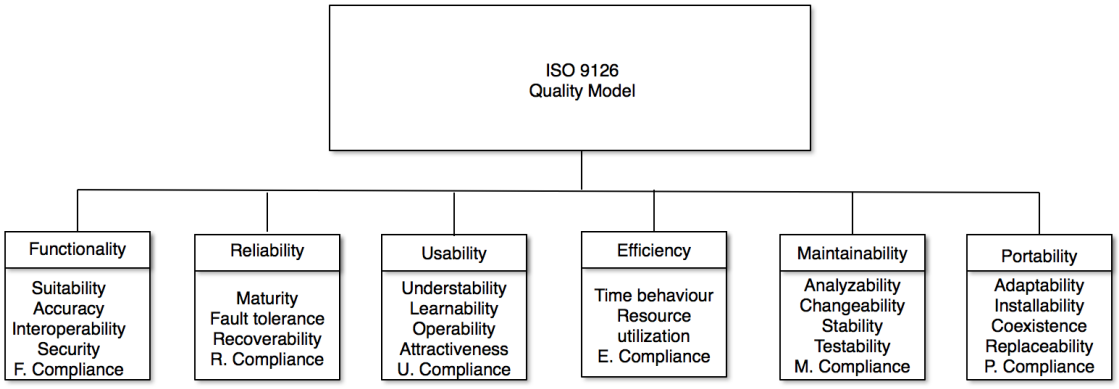


Figure 2.1: ISO-9126: The ISO/IEC 9126 Software Quality Model Categories and Subcategories [IOfS01]

2.2 The Science of Software Quality Engineering

"One of the challenges in defining quality is that everyone believes they understand it. "(Quality is Free, by P. Crosby) [Cro79]

The Journal *Software Engineering Insider* considers software reliability an essential characteristic of software quality. "Software engineering differs from other branches of engineering in that professionals are building an intangible structure and not a tangible one. Since software is embedded in the machines used in various industries, though, malfunctioning software can actually have tangible effects. With software used in everything from medical equipment to airplanes, the end result of faulty software can indeed be loss of life." [Ins11]

While software quality has been described as: "Quality consists of those product features which meet the need of customers and thereby provide product satisfaction." [Jur98] It has also been described as "Quality consists of freedom from deficiencies." [Jur98] It is because of this need to build functional, reliable, dependable and safe *intangible* systems that software engineering science has devoted much effort

Figure 2.2: ISO-25010: The ISO/IEC 25010 Software Product Quality Model Categories and Subcategories [ISO11]

to understanding this aspect of software quality.

The question is, how does one recognize high or low quality software? In order for software quality to be more than an abstract idea, there must be an agreement on what properties constitute high quality software, and standardized methodologies by which to measure them. In 1999, the International Standards Organization (ISO) published ISO-9126. [IOfS01] The standard set out a framework for evaluating software products (see Figure 2.1). It defined six categories for software quality. In 2001 recognizing that ISO-9126 did not adequately cover the complexity of modern computer systems, the ISO observed: "Software products and software-intensive computer systems are increasingly used to perform a wide variety of business and personal functions. Realization of goals and objectives for personal satisfaction, business success and/or human safety relies on high-quality software and systems. High-quality software products and software-intensive computer systems are essential to provide value, and avoid potential negative consequences, for the stakeholders."(see [fS11]) and withdrew ISO-9126, and replacing it with ISO-25010. [ISO11] As it can be seen in Figure 2.2, the new model covers a larger scope. Importantly, ISO-25010 includes a separate category for software security.

The adoption of these software quality standards by the software engineering community resulted in the development of software quality models and ways of measuring their effectiveness. These metrics are predominantly concerned with software

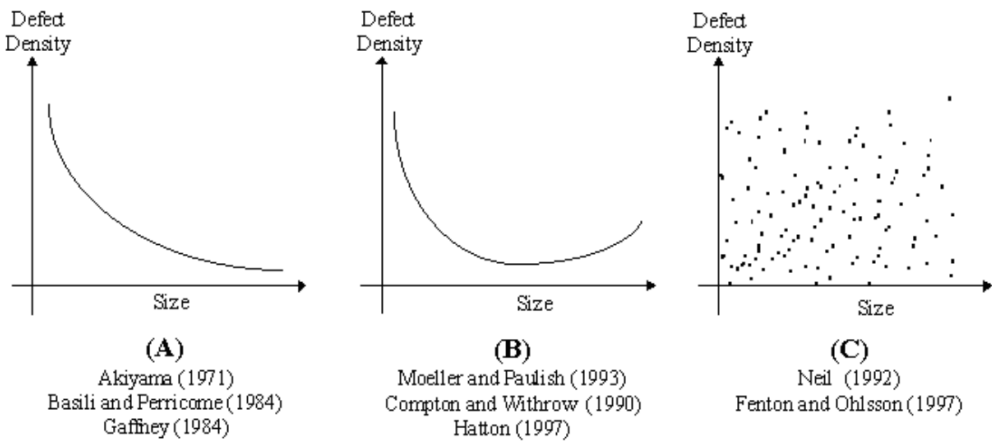


Figure 2.3: Estimated defect density from 3 different classes of defect prediction models (reproduced from [FNSS99])

defects in a product before and after release and with accounting for its subsequent software reliability. Although using defects as a measure for software quality predates the standards (see below), today's models and best practices for the software quality lifecycle continue to focus on finding and removing software defects as the key to producing high quality software. [Kan02, Ada08]

There are two defect metrics most commonly employed by SRMs to reason about software quality. The first measures the *defect density* (number of defects per lines of code) and the second measures *defect discovery rates.* The focus of the former is on measuring the software defect density to help eliminate defects before release and is measure in number of bugs per lines of code (LOC), and the latter focus is on developing models to predict software defect discovery rates to try to understand when a product maybe considered "safe", "reliable", or "ready" i.e., how many bugs are in the code, and how quickly can one depend on them being discovered. Discovery rates are most often measured by number of defects found per defined time period.

2.2.1 Defect Density Models

2.2.1.1 Defect Density and Module Size

In 1971 Akiyama published the first attempt to quantify software quality proposing a regression-based model for defect density prediction in terms of module size. [Aki71] Akiyama's model used defects discovered during testing as a measure of system complexity. This approach was later shown to be insufficient when N. Fenton [FNSS99] compared various defect prediction models and demonstrated that some complex systems have lower defect densities (see Figure 2.3). Fenton observed that the definition of defects differed from study to study and that for models to accurately predict defects, in addition to size, the models must also take into consideration key factors such as:

- The unknown relationship between defects and failures.
- Problems of using size and complexity metrics as sole "predictors" of defects.
- False claims about software decomposition.

Fenton also suggested that factors such as code maturity, software reuse and optimal size of modules may affect the defect densities differently at various points in a product's lifecycle and noted that "most defects in a system are benign in the sense that in the same given period of time they will not lead to failures" and therefore, despite their usefulness from a developer's perspective, (i.e., improving the quality of software before release), "defect counts cannot be used to predict reliability because,...it does not measure the quality of the system as a user is likely to experience it". [FNSS99] (i.e., pre-release defect removal may not translate into post-release reliability).

Number	Hypothesis	Case study	evidence?
1a	a small number of modules contain most of the total faults discovered during pre-release testing	Yes	evidence of 20-60 rule
1b	if a small number of modules contain most of the faults discovered during pre-release testing then this is simply because those modules constitute most of the code size	No	-
2a	a small number of modules contain most of the operational faults	Yes	evidence of 20-80 rule
2b	if a small number of modules contain most of the operational faults then this is simply because those modules constitute most of the code size	No	strong evidence of a converse hypothesis
3	Modules with higher incidence of faults in early pre-release likely to have higher incidence of faults in system testing	Weak support	-
4	Modules with higher incidence of faults in all pre-release testing likely to have higher incidence of faults in post-release operation	No	strongly rejected
5a	Smaller modules are less likely to be failure prone than larger ones	No	-
5b	Size metrics (such as LOC) are good predictors of number of prerelease faults in a module	Weak support	-
5c	Size metrics (such as LOC) are good predictors of number of postrelease faults in a module	No	-
5d	Size metrics (such as LOC) are good predictors of a moduleâĂŹs (pre-release) fault-density	No	-
5e	Size metrics (such as LOC) are good predictors of a moduleâĂŹs (post-release) fault-density	No	-
6	Complexity metrics are better predictors than simple size metrics of fault and failure-prone modules	No	No (for cyclomatic complexity), but some weak support for metrics based on SigFF
7	Fault densities at corresponding phases of testing and operation remain roughly constant between subsequent major releases of a software system	Yes	-
8	Software systems produced in similar environments have broadly similar fault densities at similar testing and operational phases	Yes	-

Table 2.1: Results from testing fault metrics hypothesis (reproduced from [FNSS99])

2.2.1.2 Defect Density Over the Lifetime of the Code

Emphasis on reliability as a characteristic of overall quality led to the acknowledgement of the need to distinguish between defects discovered at different life-cycle phases. Table 2.1 lists the results of a case study testing the validity of many of the hypotheses at the root of Software Reliability Engineering (SRE) metrics. Hypothesis number 4 is particularly interesting, because the result from this study suggests no clear evidence for the relationship between module complexity, pre-release discovered defects, and post-release faults (resulting in failure). This and other evidence [MD99, MCKS04] led to the current understanding that defect density as a metric for software quality must be measured "holistically" [HR06] i.e., over the entire lifetime of the software.

Studies looking at defect density over the code lifetime show that the most successful models for predicting failures from defects are those which measure contributions from LOC changes to a software module. These models show that large and/or recent changes to a module or code-base result in the *highest* fault potential. [NB05, EGK$^+$01]

2.2.1.3 Defect Density and Software Reuse

The recognition that releasing new software or changing old software resulted in more defects being discovered (presumably as a result of more defects being added), led to one of the most widely recommended software quality maxims: Reuse software whenever possible. The belief that old, already in use software is relatively free of defects and the practice of reusing old code (whether it be lines, sections or entire modules) soon became widespread, and there is much software engineering evidence to support this belief. In fact, a meta-analysis looking at software reuse from 1994-2005 found that "Systematic software reuse is significantly related to lower problem (defect, fault, or error) density" [MCKS04]. As a result, even today the current software engineering models, such as the Capability Maturity Model [Jel00], strongly recommend software reuse, and the consensus is that the longer software remains unchanged (the older it is) the fewer defects are likely to be found. The longer software *has stood the test of time the higher its apparent quality.*

2.2.2 Defect Discovery Rate Models

Defect density is not the only way software quality is measured. An equally functional metric is the rate at which defects are discovered. Defect discovery rates(DDR) are used industry wide for two distinct purposes. Often this is a key indicator of when to release software. The assumption is, that a sufficiently low defect discovery rate indicates that (1) either few defects remain to be found, or (2) that what defects do remain are harder to find, and therefore less likely to be found.

2.2.2.1 Defect Discovery Rates Before Release

Several models have been developed to try predicting the point when the percentage of defects is likely to be under a required level (set by the developer).

The most commonly used model for DDR today is the Quantitative Software

Management (QSM) Reliability Model, or Putnam model [Put78]. The QSM model uses a probability density function [1] to predict the number of defects discovered over time.

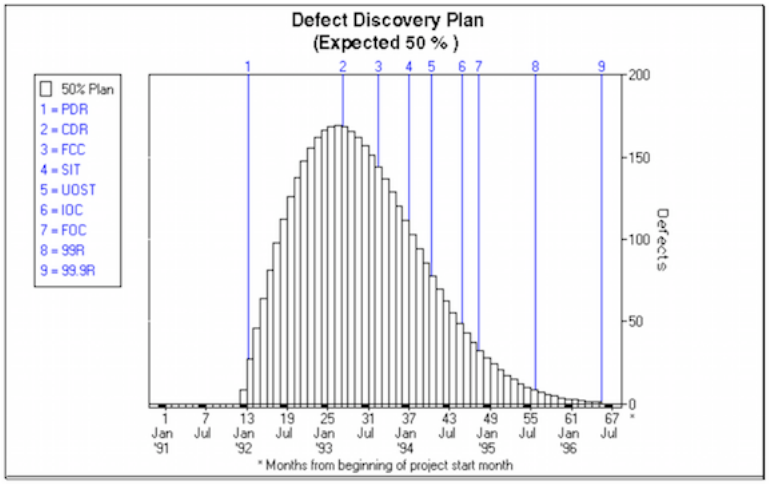


Figure 2.4: Sample Rayleigh Defect Estimate. This defect estimate is based on project of 350,000 SLOC, a PI of 12 and a peak staffing of 85 people. From the QSM Reliability Model [Put78]

Empirical research has shown that the concave curve resulting from this model (see Figure 2.4) closely approximates the actual profile of defect data collected from software development efforts. [BD01] In fact, an entire industry has sprung up around using this model to help developers and vendors predict, find and remove defects before release [Put78, Inc16]

2.2.2.2 Defect Discovery Rates After Release

The second purpose for which defect discovery rates are used is to predict failures after a product has been released. This is particularly important for determining the

[1] Rayleigh distributions are used to model scattered signals that reach a receiver by multiple paths. Depending on the density of the scatter, the signal will display different fading characteristics. In software this scatter is the predicted number of defects likely to occur in different phases of development.

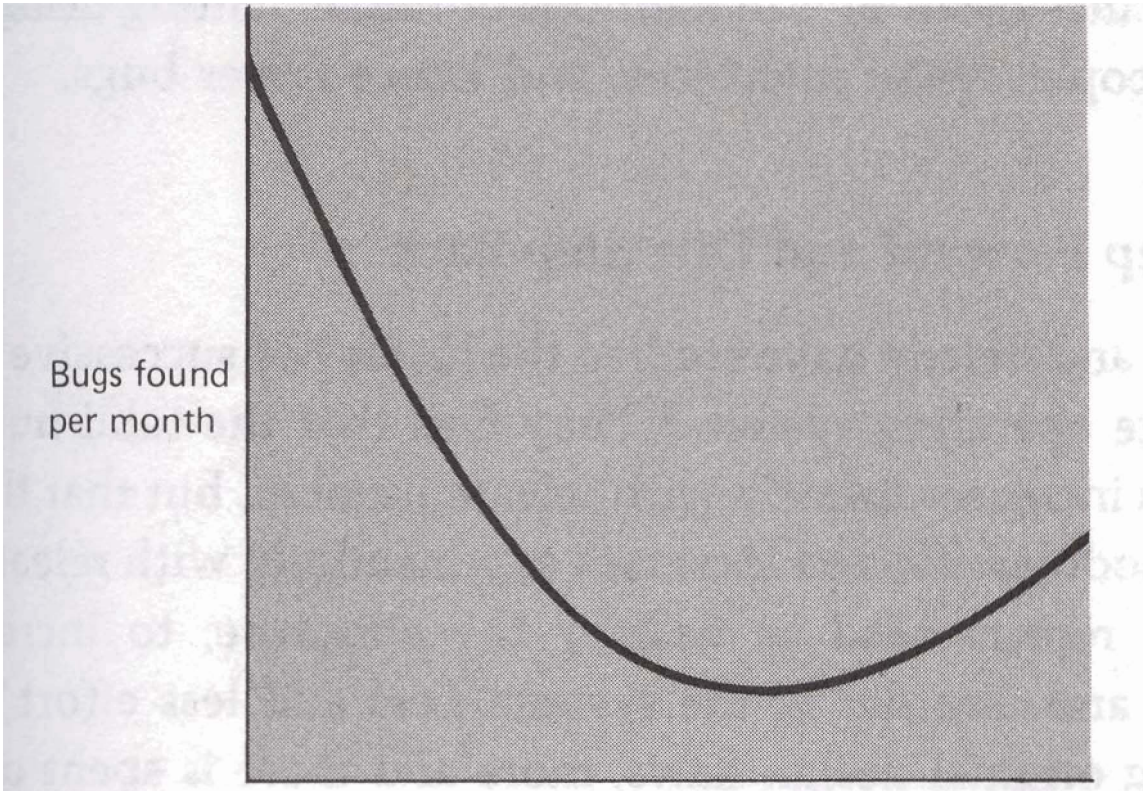


Figure 2.5: Brooks Curve anticipating defect discovery rates after product release from [Bro95a]

quality of safety critical systems where accurately predicting Mean Time To Failure (MTTF) is needed for certification and resource allocation. [ISO07, JB11]

Brooks, in the Mythical Man Month, first surmised how a curve representing the post-release DDR might look (see Figure 2.5). High numbers of defects are found early on (the low-hanging fruit), and then as time goes by, fewer defects are discovered over longer periods until the rate either reaches a constant state, or some change to the system such as the addition of new features might cause the rate to climb [2].

These models have proven to be surprisingly useful predictors over time. Figure 3.1 shows a graph from 2008 (30 years after the Mythical Man Month was

[2]Brooks notes that it was suggested that new users unfamiliar with software might break it in new ways

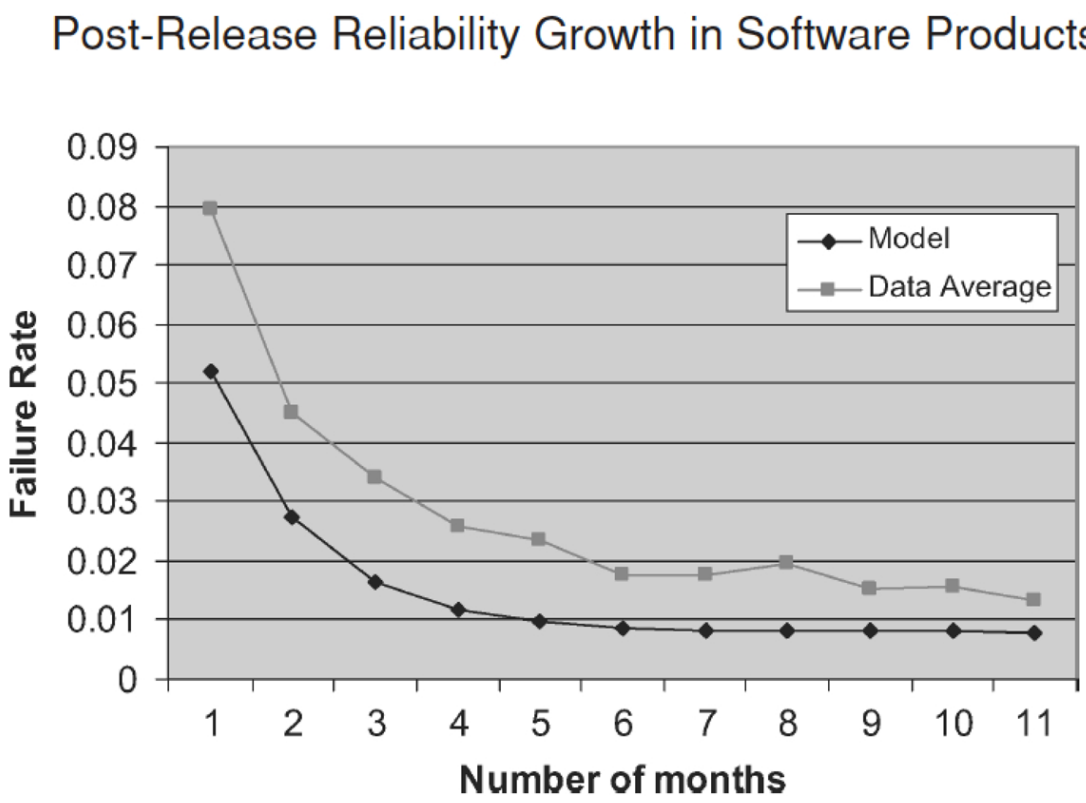


Figure 2.6: Graph comparing DDR models to defect discovery rates in popular software. Courtesy of [JMS08]

published) comparing the model predictions to actual defect discovery rates in 3 widely used, popular software products. [JMS08] Though the actual numbers of defects discovered were higher than the models predicted, the curves measuring the defect discovery rates are extremely similar.

2.2.3 Removing Software Defects Improves Software Quality

The world is much more dependent on software today than when software reliability models and metrics were first developed. Software is a necessary part of most mainstream devices and activities, from self-driving cars to streaming media. The success

of these tools in the SRE community is demonstrated daily, embedded medical devices keep people alive, communications flow through networks, and e-commerce works reliably.

Indeed, the quality of the transactions is considered so high and the failure rate so low that much of the world's industry and finance depend on software. The SRE metrics proposed in the late 1970's, which are still in use today, have aided in engineering this dependability.

2.3 Software Security as a Characteristic of Software Quality

"However beautiful the strategy, you should occasionally look at the results" (Winston Churchill)

Many of the problems relating to software security result from the exploitation of vulnerabilities in the system. To the extent that these vulnerabilities are a result of defects in software, it is natural to assume that higher quality software, i.e., software with fewer defects, must also be *more secure*. This leads, naturally, to the assumption, that the methods used so successfully to assure software quality (defect density and defect discovery rates), apply equally to software security. A further assumption is that the same metrics apply. This section examines some of the attempts to quantify software security by adapting the models and metrics that work so well for software quality, (that is defect density and defect discovery rates), for vulnerability density (V_{DD}) and vulnerability discovery rate (V_{DR}).

2.3.1 Vulnerability Discovery Rate Models

In late 2000 Arbaugh, *et al.*, explored the merits of enumerating and modeling vulnerability discovery rates as a metric for software security [AFM00]. [3] Previously, attempts had been made to estimate numbers of machines at risk of intrusion from a known but unpatched vulnerability, [How97, Ken99], but this was the first study attempting to determine when in the lifecycle intrusions occurred. *Windows of Vulnerability* focused on software *flaws*, i.e., defects. Using three case studies, the authors measured the rates of intrusion reporting for well-known vulnerabilities. From these intrusion reporting rates, they proposed a model for predicting vulnerability discovery rates and the severity of infection.

2.3.1.1 Case Studies

Arbaugh *et al.*, compiled a data set of all intrusion incidents submitted to the incident report repository of the Computer Engineering Response Team (CERT) Coordination Center. They chose the three case studies from the vulnerabilities with the highest incidence rate. These were the Phf vulnerability [Mye96], the IMAP vulnerability [UC97] and the BIND vulnerability [UC98]. From each incident, they included only those reports resulting from a successful intrusion.

Phf

The first case study examined the intrusion reports surrounding the vulnerability in Phf. Jennifer Myers found and disclosed the vulnerability on February 2nd 1996, then March of that same year, CERT issued a security advisory. [Mye96, Ins96] Figure 2.7 shows the rate of intrusions reported. The first known scripted attempts to

[3] A more detailed paper was published a few months later under the title: "A Trend Analysis of Exploitations". [WAMF01]

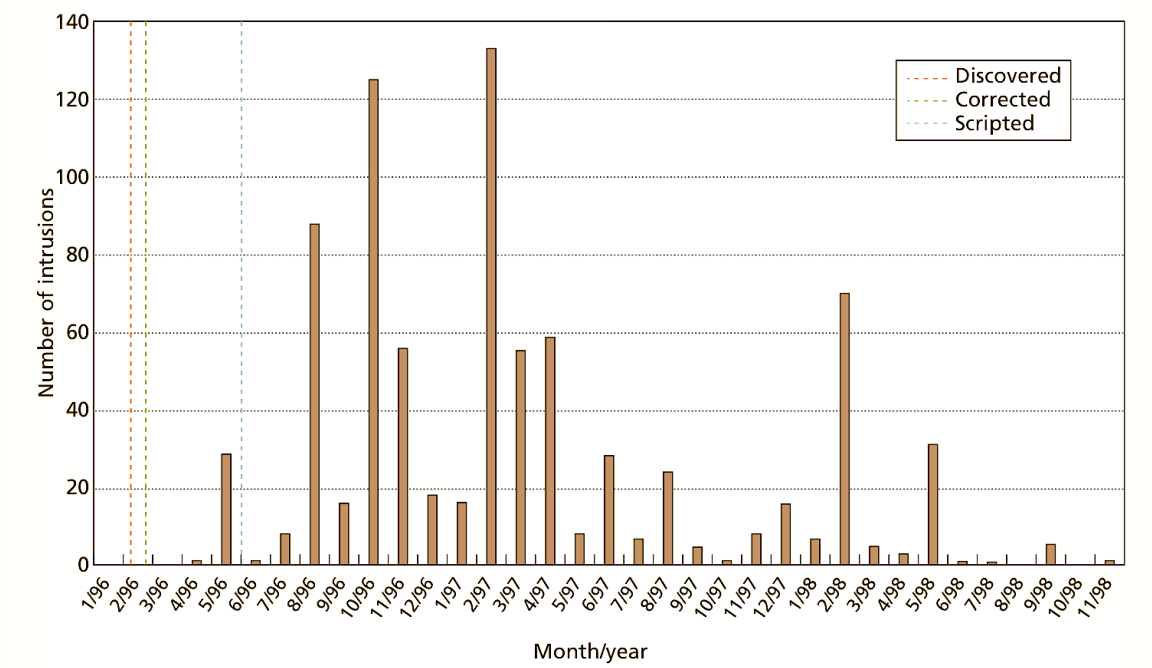


Figure 2.7: Phf Incident histogram. The rate of intrusions increased significantly six months after the correction became available, then persisted for another two years. - from [AFM00]

exploit this vulnerability appeared a few months later, in June, 1996. [4] The authors observed that while some exploitation occurred prior to the vulnerability's scripting, the vast majority of the intrusions reported took place after scripting. They also noticed that the rate of intrusion increased significantly in August of 1996. What the authors did not mention, was that complete instructions on how to script this vulnerability were published in PHRACK issue 49, in August of 1996 [One96]. This tutorial no doubt contributed to the proliferation of incidents reported at that time.

One very surprising thing that can be seen in Figure 2.7 is the length of time a vulnerability lived. Even though a patch was available before any intrusions were reported to CERT, the effective lifetime of this exploit was more than two and one-half years.

[4]This script only attempted to download the password file. It did not allow for arbitrary command execution.

IMAP

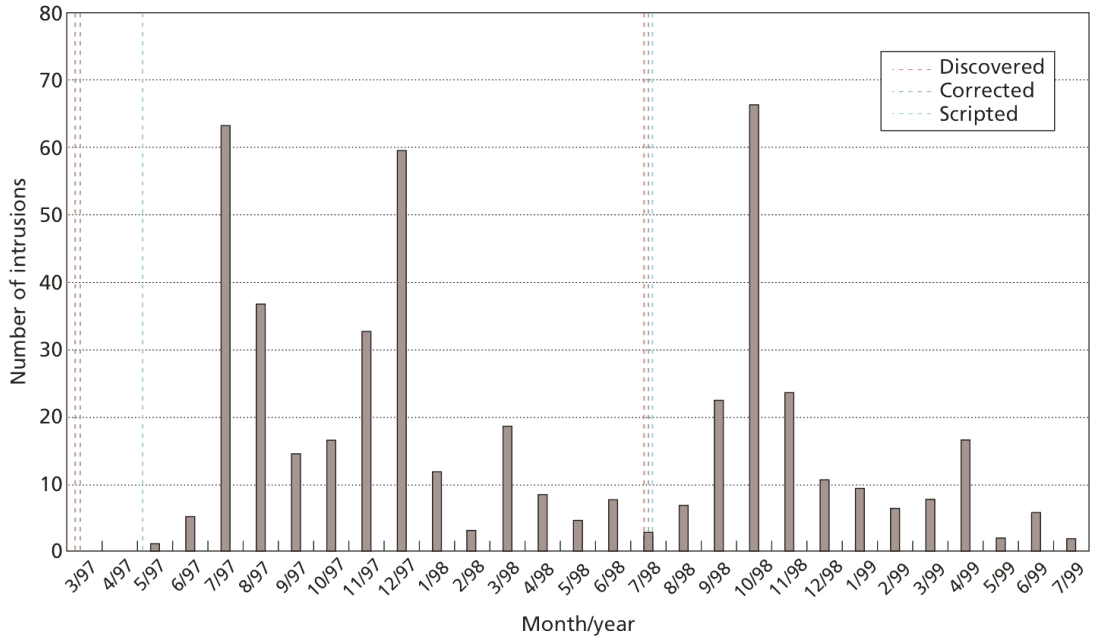


Figure 2.8: IMAP incident histogram. This study tracks two flaws, both of which exploited buffer overflow vulnerabilities in the IMAP server. - from [AFM00]

The second case study looked at two buffer overflow vulnerabilities in the IMAP mail server software. David Sacerdote of Secure Networks posted the first flaw to Bugtraq on March 2, 1997. CERT issued an advisory with links to patches a month later, and the first known scripted exploit appeared a month after that. However, the software also contained a second buffer overflow that wasn't discovered until almost a full year later. This second flaw was posted, along with a link to the patch, to the *pine-announce* email list in July of 1998, but the details of the vulnerability weren't disclosed. Six days later, an anonymous posting on Bugtraq provided the details and included a scripted exploit.

Figure 2.8 shows the reported intrusions from these two vulnerabilities. The authors stated that they combined the data of both vulnerabilities, because in most cases the

incident report did not specifically list which flaw was exploited, making it difficult to differentiate between them. However, the authors made a point of noticing that the graph has two separate curves, which they claimed represents the intrusion rates for each vulnerability. They also pointed out that both of the curves have the same general shape as the Phf graph.

The authors also observed something interesting about the behavior of the attackers using the IMAP vulnerability scripts. To a much larger degree than previously reported, "attackers used scanning or probing to identify potentially vulnerable hosts. In several cases, incidents reported to CERT/CC involved large subnet scanning, with some scans encompassing an entire Class A network, or several million hosts." The authors did not explore this any further, to see if such an increase in vulnerability scanning resulted in a change to the V_{DR}.

BIND

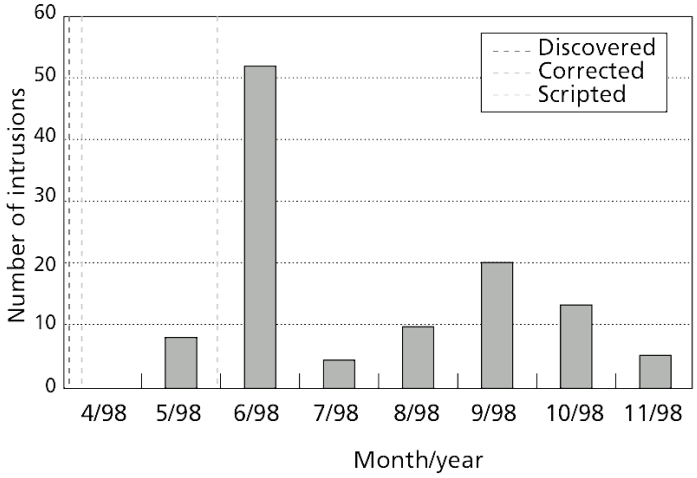


Figure 2.9: BIND Incident histogram. Part of the Internet's infrastructure BIND suffered attacks for a much shorter period of time than did phf and IMAP, thanks to aggressive countermeasures. from [AFM00]

The final case study examined a vulnerability in the BIND domain name system implementation.

CERT disclosed the flaw on April 8, 1998 and the exploit was automated nearly two months later. The authors claimed that though there were rumors that the flaw had been known for months, they (the authors) were unable to substantiate any of them. Figure 2.9 shows the histogram for the reported incidents exploiting this vulnerability. The authors found it surprising that given how integral BIND was to the Internet's infrastructure, there were still reports of incidents six months later. Yet, the authors also noted that the response to the BIND vulnerability was much more "aggressive" than the responses IMAP and phf incidents. The lifetime of BIND was six months, compared to a year or more for the others.

2.3.1.2 $\mathbf{V}_{DR}$ Model

In their paper "A Trend Analysis of Exploitations" [WAMF01], the authors claimed that all three exploits they studied could be modeled using the formula $C = I + Sx\sqrt{M}$, where C is the cumulative total of incidents, M the time since the first known exploit (the start of the exploit cycle) and I and S regression coefficients determined by analysis of the report data. Regression analysis testing of their model on their dataset, led to two conclusions: First, the model supported the hypothesis that there was a relationship between the cumulative counts per month for individual incidents, and therefore appeared to provide "very good predictive power for the accumulation of security vulnerability incidents", and second, there appeared to be no similarities in the shapes of the slopes across incidents. They concluded that no one formula allowed for the prediction of future incidents based on past incident behavior.

2.3.1.3 Discussion

The analysis presented here is one of the earliest attempts to model the behavior of software vulnerabilities as separate from software defects. The study was limited

in scope. It looked at only three vulnerabilities, and for each, they restricted their dataset to self-reported breaches (intrusions). Though this data gave the authors insight into factors affecting the lifetime of a vulnerability, such as automation (scripting) on the attackers' side, and patching behavior on the defenders' side, the authors could say nothing about the overall quality of the software, about the vulnerabilities that might remain to be found (quantity or severity), or about the rate at which new vulnerabilities might be discovered in software. Nor could their model be applied across incidents (new exploits). This limited their model's overall applicability. However, this paper made some significant contributions to the field of security metrics by providing new definitions for software and hardware vulnerabilities. They defined security vulnerabilities as software flaws with distinct characteristics differentiating them from functional defects, e.g., "A flaw in an information technology product that could allow violations of security policy" [[WAMF01], p. 52], as "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" [[WAMF01], p. 54], and considered for their analysis, a vulnerability to be a flaw that has been *discovered, deployed and "available for widespread use"*. These definitions go beyond typifying a flaw as a mistake in coding; by including deployment as a risk factor, and by recognizing that the behavior of the technology and the behavior of attackers each play a role in successful exploitation. Although, later work by Ozment claimed that these definitions were too broad, since they failed to account for multiple vulnerabilities, and their definitions didn't include the entire software lifecycle."... a single defect or flaw could result in multiple different vulnerabilities or "exploit instances" and that a vulnerability could occur in any part of the development and deployment process." [Ozm07]

The authors claimed a major purpose of this study was to investigate the hypothesis: Does poor system administration, specifically the failure to apply patches in a "timely fashion" result in an "excessive window of vulnerability". The authors

stated that they "expected the rate at which exploits occur to be fairly small in the period immediately following discovery and to increase as the vulnerability and its associated exploit become more widely known". Importantly, the authors also hypothesized that an exploit would become *"passe"* as patches became available and were widely deployed. Figure 2.10 shows the curve the authors assumed they would see, a slow start, followed by a rapid increase in intrusions (accelerating as news of the vulnerability reached a wider audience), followed by a steep decline as soon as a patch was released. [5] This expectation wasn't a new one, it had been proposed before by Kendall and Schneier [Ken99, Sch00], but Arbaugh, *et al.*, were among the first to test it.

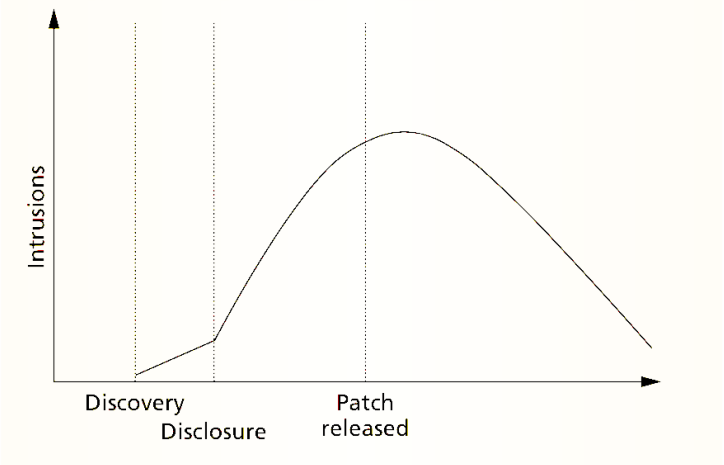


Figure 2.10: Intuitive life cycle of a system-security vulnerability. Intrusions increase once users discover a vulnerability, and the rate continues to increase until the system administrator releases a patch or workaround. (from [AFM00])

In their examination, they were surprised to find this assumption was wrong. While it was commonly assumed that most attackers choose well known vulnerabilities to exploit, it was also assumed that attackers would choose vulnerabilities for which no patch existed. Surprisingly, their results demonstrated that the most commonly compromised vulnerabilities were for flaws for which patches *were* available.

[5] Notice the similarity in the expected model and the most popular of the SRMs, the QSM model.

Their research implied that "deployment of corrections is woefully inadequate". Thus, this paper not only contributed one of the first vulnerability lifecycle models, and provided hints that the vulnerability lifetime might be much longer than expected, but it demonstrated that identification of a vulnerability and its exploitation are both distinct actions and are separated by a window of time.

At the time this paper was written, there was much debate regarding the merits of publicly disclosing vulnerabilities. [Sch04, Spa89], the authors concluded that open disclosure "obviously works", because patches for the vulnerabilities were available *before* the rise in intrusions. Moreover, they were able to show that automation was the key for mass intrusions. Particularly, since, in all cases they studied, patches were so quickly available. Furthermore, their suggestion that active systems management combining intrusion detection and patching would be the most cost-effective means of securing a system has been proven correct. Today, nearly all major software developers provide automated patching capability and automated patch management is a thriving business [AKTY06]. "Windows of Vulnerability" was the first work that demonstrated patterns of attacker behavior inside individual exploits and by measuring this behavior they discovered that the defender's actions may determine the length of that vulnerable "window".

2.3.2 Empirical Analysis of Vulnerability Density

A few years after "Windows of Vulnerability" was published, Andy Ozment and Stuart Schechter examined a different metric for measuring the quality of software. They looked at the number of vulnerabilities per lines of code. This was an attempt to determine whether older software was more or less secure than newer software. In their paper *Milk or Wine: Does Software Security Improve With Age?* [OS06], the authors analyzed the code base of the OpenBSD operating system over a period of 7.5 years and attempted to answer five questions:

1. Has there been a decline in the rate at which vulnerabilities created in the

originating version of the software (Foundational Vulnerabilities) in OpenBSD are reported?

2. Do larger code changes have more vulnerabilities?

3. Does newer code contain fewer vulnerabilities per line of code than older code?

4. How much does legacy code influence security today?

5. What is the median lifetime of a vulnerability?

The authors chose OpenBSD for their analysis, because the entire source code and all subsequent changes were readily available, and because they wanted to test a system whose "developers focused on finding and removing vulnerabilities". From the public repository, the authors obtained the source code for versions 2.3-3.7. From the CVS database they found each reported vulnerability. They then attempted to determine the lifetime of each vulnerability. To do so, they found the earliest reported dates for each, which they referred to as the date the vulnerability was *born*, and correlated that birth date with the earliest time that a patch was available, calling this the date the vulnerability *died*. Vulnerabilities that were remediated with the same patch were grouped together as one. The authors noted that their decision to bundle vulnerabilities was a result of their inability to obtain data to differentiate between them, and that this "may result in an inflated perception of security for the system", particularly since it might cause models to assume fewer vulnerabilities and demonstrate a rapid diminishing trend. Figure 2.11 show the number of vulnerabilities reported and patched per OpenBSD version along with total lines of code.

2.3.2.1 Q1: Are Vulnerability Discovery Rates declining?

Ozment and Schechter considered vulnerability discovery rates analogous to reliability engineering's MTBF (mean time between failures) metric. Thus, they felt they

Version in which the vulnerability died	Version in which the vulnerability was born															
	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	Total
2.3	5															5
2.4	11	0														11
2.5	6	0	1													7
2.6	5	1	0	0												6
2.7	12	4	2	2	2											22
2.8	12	1	0	1	2	0										16
2.9	4	0	0	2	0	0	0									6
3.0	3	1	0	0	1	0	2	0								7
3.1	8	2	1	2	0	0	0	1	1							15
3.2	6	2	0	0	0	0	1	2	0	1						12
3.3	2	1	0	2	0	0	0	0	0	0	2					7
3.4	2	0	0	0	1	0	1	0	1	0	0	0				5
3.5	7	1	1	0	0	0	0	2	0	1	0	0	1			13
3.6	3	0	1	0	0	0	0	0	0	0	0	0	0	0		4
3.7	1	1	0	0	1	0	0	0	0	0	0	0	1	0	0	4
Total	87	14	6	9	7	0	4	5	2	2	2	0	2	0	0	140
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	

Figure 2.11: The OpenBSD version in which vulnerabilities were introduced into the source code (born) and the version in which they were repaired (died). The final row, at the very bottom of the table, shows the count in millions of lines of code altered/introduced in that version. from [OS06]

could use the rate of vulnerability reporting to measure whether OpenBSD software quality was improving.

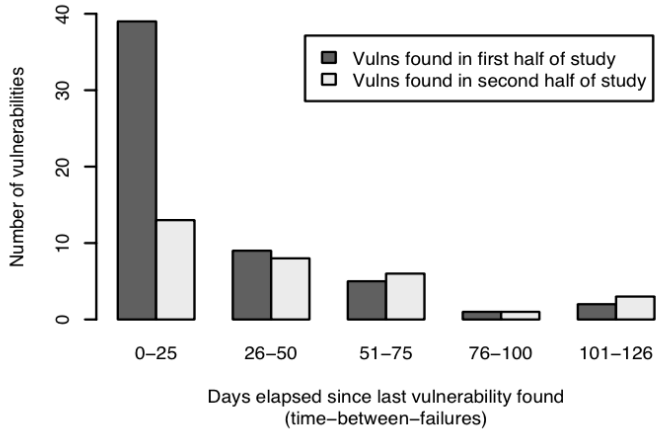


Figure 2.12: The number of days between reports of foundational vulnerabilities. from [OS06]

Figure 2.12 shows the number of days between foundational vulnerability reports in the first and second halves of the study. The authors claimed that their statistical analysis on the direction of trend in the rate vulnerability reporting indicated a clear decrease over time.

2.3.2.2 Q2: Does more code mean more vulnerabilities?

Noting that measuring defect density rates was proving useful for software engineers, the authors looked at the number of vulnerabilities per millions of lines of code in each version of the code to see if vulnerability density rates (VDR) would prove beneficial for software security. Interestingly, they found that the vulnerability density of the foundational version was "right in the middle of the pack" compared to the vulnerability densities of all of the versions studied. But, they did report that vulnerability density was higher in versions that introduced new code *if* that code provided security functionality. They gave as an example, version 2.4, which introduced a new key management daemon and OpenSSL. The also authors reported that they were unable find a significant correlation between new lines of code added and the number of reported vulnerabilities.

2.3.2.3 Q3: Has software quality improved?

The authors then asked whether programmers had gotten better at writing code, i.e., by the time the study ended compared to when the first version was released, were programmers producing code with *fewer* vulnerabilities? Looking at the vulnerability density per 1000 LOC, they found that the density of *all* reported vulnerabilities fell in a very narrow range [6] averaging 0.00657 across all versions. While they did find that the addition of new security functionality, e.g., the addition of OpenSSL and key management, did result in the introduction of new vulnerabilities, the authors did not attribute this to problems of software quality, instead stating the belief that the

[6] 0-0.033

new code added "may have drawn particular attention from vulnerability hunters". Thus, the authors drew no conclusions about improvements in the quality of the later software.

2.3.2.4 Q4: The influence of legacy code on security

One of the most striking results of this study, was the effect of legacy code on vulnerability discovery, and vulnerability density. During the period of study, the authors found that 62% of the vulnerabilities reported had existed in the code since its very first version. They proposed two possible explanations for this; that the foundational code was of a lower quality than more recent code, or that foundational code made up most of the code base regardless of version.

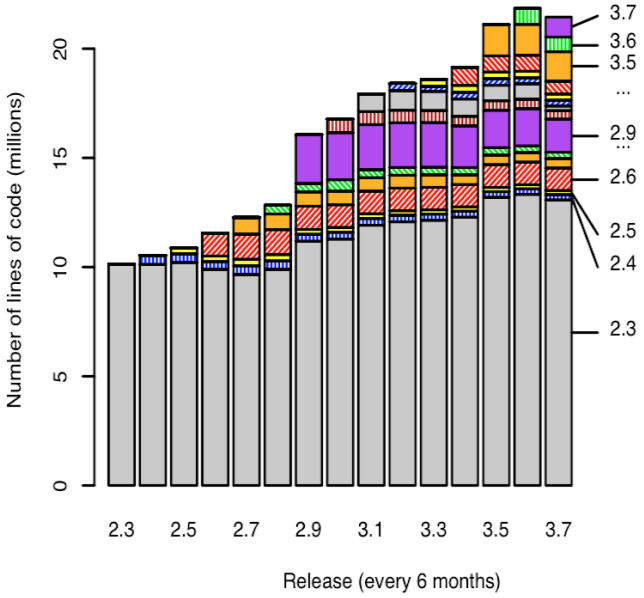


Figure 2.13: The composition of the full source code. The composition of each version is broken-down into the lines of code originating from that version and from each prior version. [OS06]

Figure 2.13 shows support for the latter hypothesis. The authors did note, that they were surprised to see the amount of foundational code increased in some of

the later versions. They attributed this to developers cutting and pasting legacy code into new modules. They also noted that the largest numbers of reported vulnerabilities were in the sys/kern directory of which, 88% of the vulnerabilities were foundational.

2.3.2.5 Q5: What is the lifetime of a vulnerability?

To calculate the median lifetime of a vulnerability, the authors used the time elapsed between the release of a version and the death of half of the reported vulnerabilities. Acknowledging that there was no way of determining whether all vulnerabilities in any particular version had been found, (especially in the foundational version), they presented their results as a lower-bound for this metric. Their analysis calculated 2.6 years as the median lifetime of foundational vulnerabilities (see Figure 2.14).

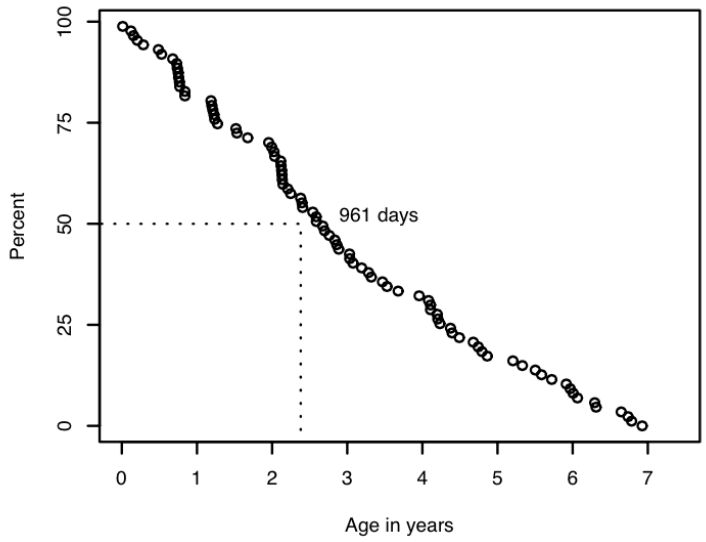


Figure 2.14: The median lifetime of foundational vulnerabilities during the study period. from [OS06]

However, Ozment and Schecter admitted that because their analysis relied on a "gross simplifying assumption", the assumption that all vulnerabilities, (i.e., the total sum of vulnerabilities contained in that software), were found within a 6 year

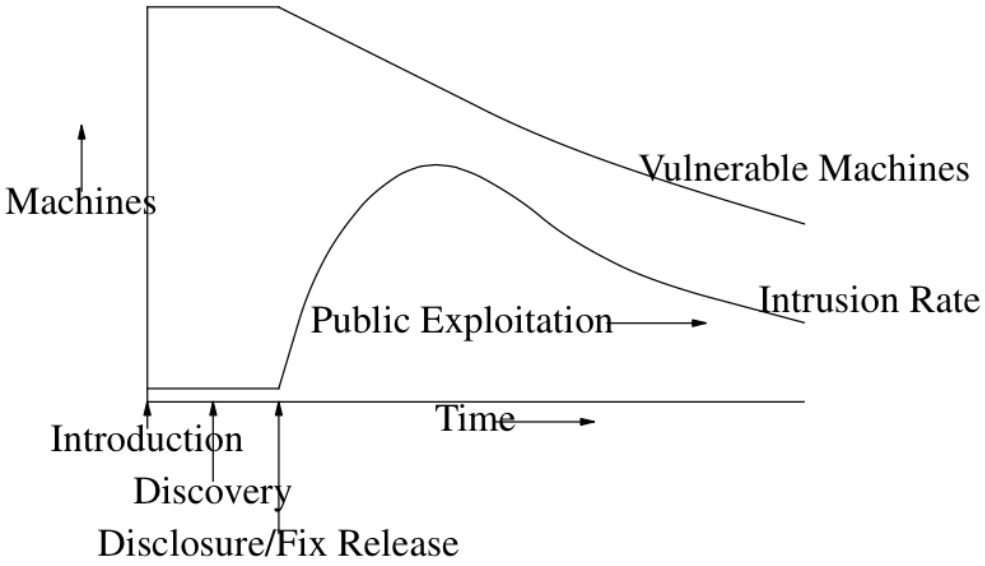


Figure 2.15: Rescorla's Whitehat vulnerability discovery process [Res05]

period, the fact that their data had examples of vulnerabilities reported outside that period meant that their analysis was limited.

Still, this does not detract from the surprising discovery that the lifetime of a vulnerability was so long.

The authors concluded that the rate of vulnerability reports decreased during the period of their study. Further they estimated that by the end of their analysis, 67.6% of the vulnerabilities that originated in the foundational version of OpenBSD had been found. [7]

2.3.2.6 Discussion

The research reported in *Milk or Wine* was carried out largely in response to work done by Eric Rescorla [Res05]. In *Is finding security holes a good idea?*, Rescorla asked whether it was "... better for vulnerabilities to be found and fixed by good guys

[7] They base this estimation on the fact that the expected number of vulnerabilities reported per day decreased from 0.051 at the start of the study, to 0.024.

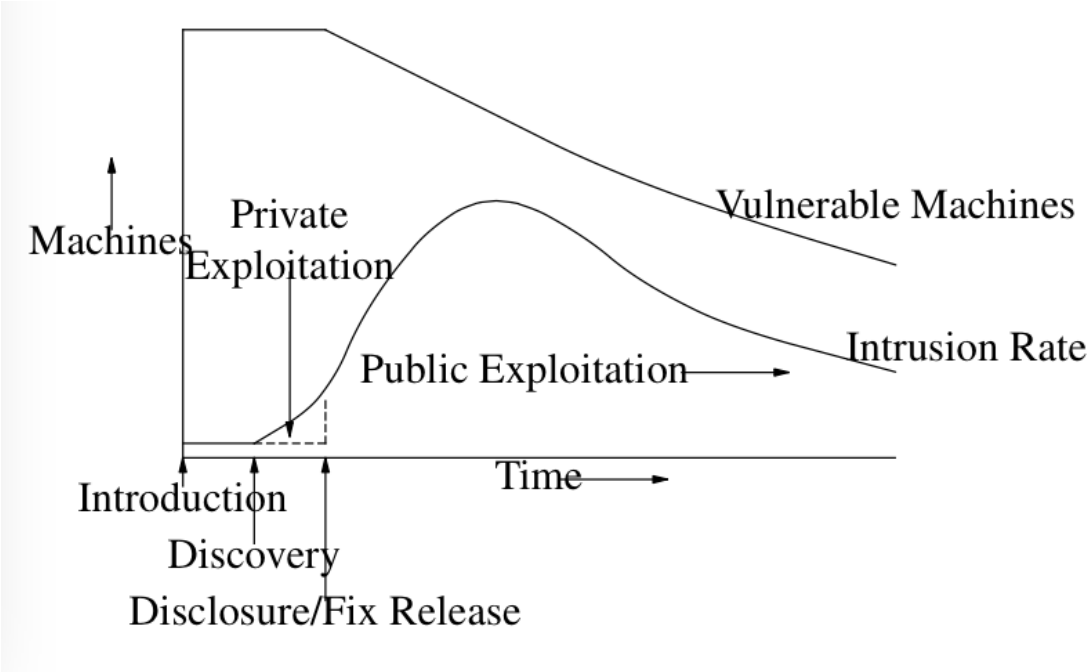


Figure 2.16: Rescorla's Blackhat vulnerability discovery model [Res05]

than for them to be found and exploited by bad guys". Rescorla presented models for measuring the effect of disclosure by 'good guys' or 'bad guys' on overall software quality, see Figure 2.15 and Figure 2.16, and argued that unless a vulnerability was already being exploited, public disclosure of vulnerabilities was not cost effective and further was actually dangerous. [8] Using data from ICAT [Rei02], and comparing several statistical analysis methods, he was unable to show any "significant trend towards increasing reliability" in the cohort data. He claimed to show that there was little evidence to support the claim that active vulnerability discovery depletes the pool of vulnerabilities and therefore, it was not cost effective for the good guys to waste resources on finding vulnerabilities. He also claimed that vulnerability disclosure did not provide an increase in security sufficient to offset the cost.

Ozment and Schechter believed his data was limited because his dataset did not

[8] Rescorla claimed slow patch rates would greatly magnify the damage malware developed after disclosure could cause.

reliably report the "birth" date of vulnerabilities, and so the subject warranted further investigation. In looking at vulnerability discovery rates over time, Ozment and Schechter claimed their results contradicted Resorla's, showing a clear decrease in the discovery rate. It is interesting to note, while history has proven most of Rescorla's conclusions about the benefits and costs of vulnerability discovery and disclosure to be wrong [Mil07, FAW13], recent research has shown that vulnerability discovery rates have actually increased [FM06], suggesting that Ozment and Schechter's conclusion cannot be widely applied.

This paper was one of the first to demonstrate that new code which involves the addition of new security functionality increases the number of vulnerabilities discovered. The authors attributed this to increased attacker attention. While they did not consider the possibility that the new security functionality itself would add new complexity and an increased possibility of unexpected interactions, this is still one of the first glimpses that extrinsic properties (such as attacker interest) affect software security.

Ozment and Schechter also demonstrated the security hazard resulting from legacy code, noting in particular that many times code was *copied* from one part of the operating system to another. However, they made no attempt to discover whether any vulnerabilities in the code were also copied. As it will be seen in Chapter 4, the high percentage of vulnerabilities found in legacy code suggests that the weaknesses in code that might result in a vulnerability being exploited are also copied.

The chief limitation of this work is that the authors chose to look at OpenBSD, an OS which was never widely adopted. [9] This small userbase makes their model a poor fit for other systems, precisely for the reasons they use to justify their results. Moreover, their decision to group similar vulnerabilities reported closely in time together into one, greatly reduced the size of their dataset. They failed to consider

[9] At the time of publication OpenBSD had fewer than 1500 servers instances, approximately 0.003% share of the server market.

the possibility that OpenBSD usage declined as Microsoft Windows and Linux and Apple OS adoption hugely increased during this period. So while they did entertain the idea of attacker interest in vulnerability discovery, they did not recognize that OpenBSD's limited adoption would mean correspondingly little attacker interest and result in a very small vulnerability dataset, making their contribution less valuable in the long term.

2.3.3 Modeling the Vulnerability Discovery Process

The same year that *Milk or Wine* was published, another group attempted to determine whether models similar to those used for software reliability engineering could be used to provide software security metrics [CA05, AMR05, AMR07]. Like Ozment and Schechter, Alhazmi, *et al.*, were interested in seeing whether vulnerability density was a useful metric for software security, and like Arbaugh, *et al.*, the authors looked at vulnerability discovery rates to determine whether models could be used to predict trends. In addition, they also compared the ratio of known vulnerabilities to known defects. Two years earlier, Anderson [And02], had proposed this as a metric for software security, and guessed that the value might be around 1%, while similarly, McGraw [McG03] suggested this ratio was probably higher, around 5%, but neither actually measured it. Alhazmi, *et al.*, hoped to determine which (if either) estimate was correct, and hypothesized that if one were correct, this ratio could be used to estimate the number of remaining undiscovered vulnerabilities. Their goal was to develop a model for the entire vulnerability discovery process.

2.3.3.1 Windows $\mathbf{V}_{DD}$ and $\mathbf{V}_{DR}$

For their analysis, the authors looked at different versions of the Microsoft Windows and the Redhat Linux operating systems.

Table 2.2 shows their results for Microsoft WIndows. It displays the known defect

Table 1: Vulnerability density versus defect density measured for some software systems

Systems	***Msloc***	***Known Defects***	***Known Defect Density (per Ksloc)***	***Known Vulnera-bilities***	***V_{KD} (per Ksloc)***	***V_{KD}/ D_{KD} Ratio (%)***	***Release Date***
Windows 95	15	5	0.3333	50	.0033	1.00%	Aug 1995
Windows 98	18	10	0.5556	66	.0037	0.66%	Jun 1998
Windows XP	40	106.5	2.6625	88	.0022	0.08%	Oct 2001
Windows NT 4.0	16	10	0.625	179	.0112	1.79%	Jul 1996
Win 2000	35	63	1.80	170	.0049	0.27%	Feb 2000

Table 2.2: Vulnerability density vs. defect density for several versions of the Microsoft Windows operating system. - from [CA05]

density (D_{KD}) and vulnerability density (here labeled V_{KD}) and the ratio between the two, for Microsoft Windows client operating systems Windows 95, 98 and XP and Windows server operating systems Windows NT and 2000. Looking at the client operating systems they noted that while the defect densities and vulnerability densities were quite close for versions 95 and 98. For Windows XP the values were much lower. They attributed this difference to the fact that their dataset included the defects reported in the beta version, as well as the final release, resulting in a much larger defect total. They also stated that their numbers represented only a fraction of XP's overall vulnerability density and therefore they expected this value to "go up significantly, perhaps to a value more comparable to the two previous versions." (See Table 2.2) Interpreting their results, the authors observed that there were several vulnerabilities shared between Win 98 and XP, and that the slope of the XP graph shows almost no learning rate.

They also compared the vulnerability and defect densities for two versions of Microsoft Windows Server: Windows NT and 2000. They were surprised to find that the V_{KD} is around three times higher for the server versions than for the client versions. The authors offered two possible explanations: First, that a larger portion of a server's software is involved with functions requiring external access, which they claimed made it more vulnerable, and Second, they asserted that server software

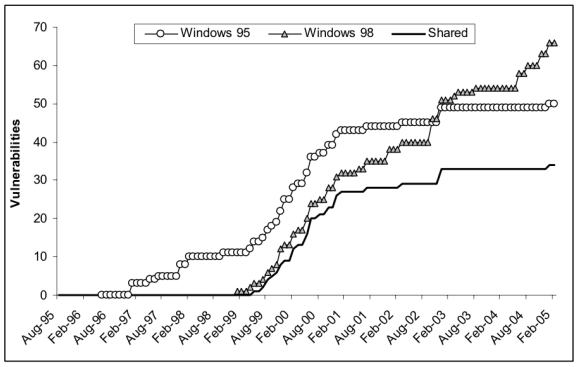


Figure 2.17: Cumulative and Shared vulnerabilities between Windows 95 and Windows 98. [CA05]

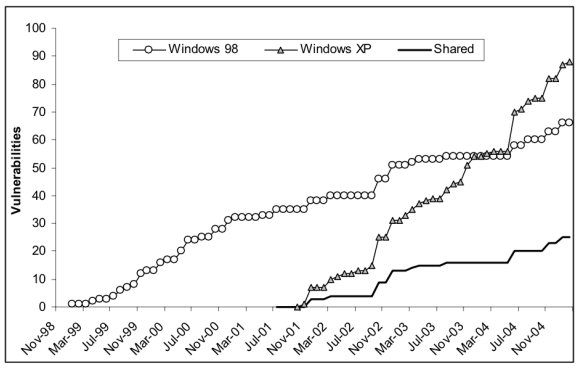


Figure 2.18: Cumulative and Shared vulnerabilities between Windows 98 and Windows XP. [CA05]

must have undergone more stringent testing and therefore more vulnerabilities were found and reported.

Figure 2.17 shows the cumulative vulnerabilities for Windows 95 and 98 as well as the shared vulnerabilities. Figure 2.18 compares Windows 98 and XP and Figure 2.19 shows the same for Windows NT and 2000.

Systems	*Msloc*	*Known Defects*	*Known Defect Density (per Ksloc)*	*Known Vulnera-bilities*	*V_{KD} (per Ksloc)*	*V_{KD}/ D_{KD} Ratio (%)*	*Release Date*
R H Linux 6.2	17	2096	0.12329	118	.00694	5.63%	Mar 2000
R H Linux 7.1	30	3779	0.12597	164	.00547	4.34%	Apr 2001

Table 2.3: Vulnerability density vs. defect density for two versions of the Redhat operating system. - from [CA05]

2.3.3.2 Linux V_{DD} and V_{DR}

After examining the various MS Windows operating systems, the authors were curious to see if an open source operating system displayed the same characteristics as the closed source systems. They chose two versions of Redhat Linux for comparison. Table 2.3 shows their results for Redhat version 6.2 and 7.1. Figure 2.20 shows the plot of cumulative vulnerabilities for both versions as well as the vulnerabilities shared between them. Looking at the graph, they made the following observations: While the code size for version 7.1 is twice as large as version 6.2, the V_{KD} and D_{KD}) are similar. Additionally, the V_{KD} for Red Hat Linux is in the same range as that of Windows 2000. Looking at the ratio of V_{KD} to D_{KD} in Red Hat 7.1, the authors state that they expected the V_{KD} "to rise significantly in the near future" and note that the value of the ratios for both Linux versions are close to the 5% proposed by McGraw. [McG03] Here, as well as with MS Windows, they noted the shared vulnerabilities between versions.

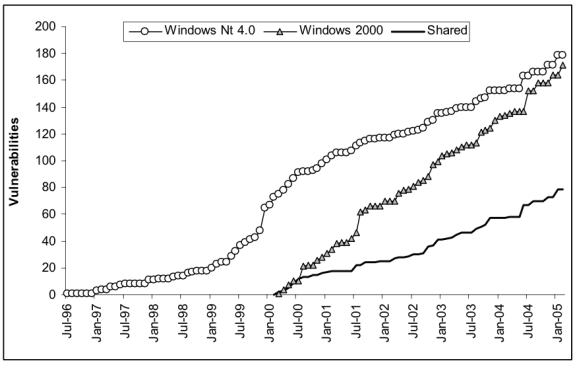


Figure 2.19: Cumulative and Shared vulnerabilities between Windows NT and Windows 2000. [CA05]

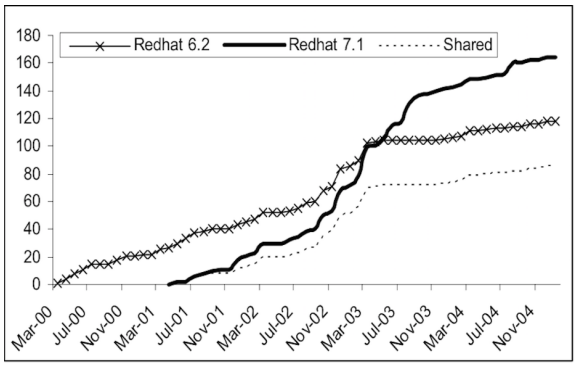


Figure 2.20: Cumulative and Shared vulnerabilities between Redhat Linux versions 6 and 7. [CA05]

2.3.3.3 A proposed model

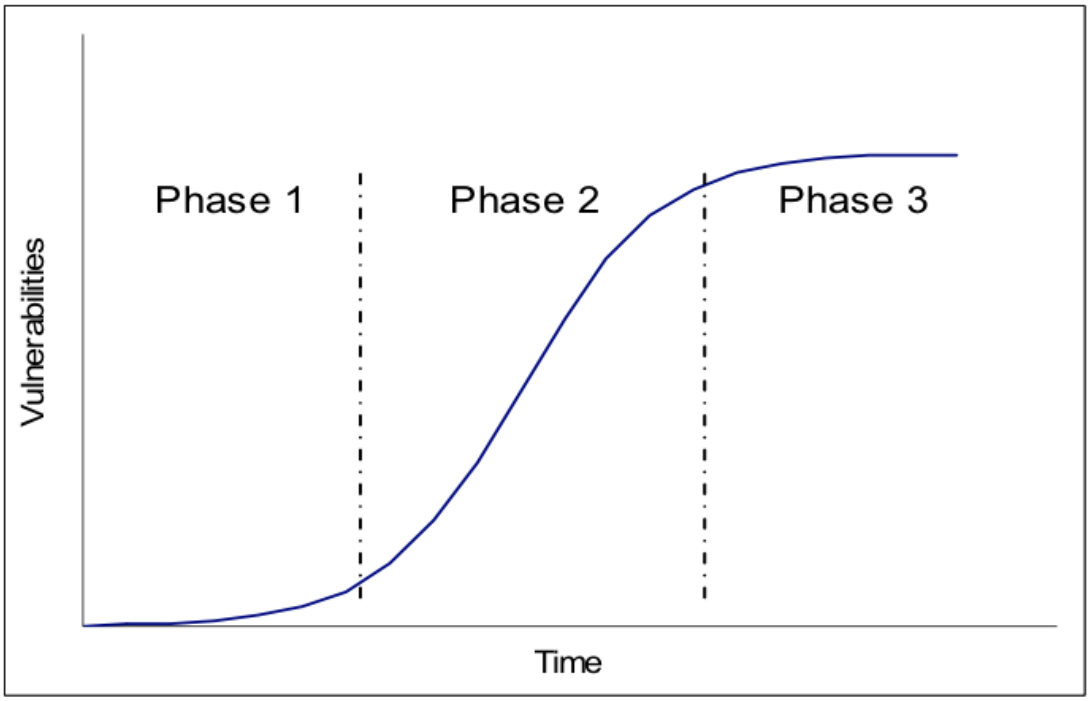


Figure 2.21: Proposed 3-phase model. See [CA05]

Alhazmi, *et al.*, found a common pattern across all the operating systems they examined. Their plots of vulnerabilities discovered over time tended to show three

phases. They claimed that these phases follow the s-shaped model they had proposed in their earlier work. [CA05]

Figure 2.21 describes the proposed three phase model. According to their definitions, Phase 1 is the phase where users begin to switch to the new operating system and testers (both good and bad) gather knowledge about how to break it. In Phase 2, the time when the operating system usage gathers momentum and it reaches its peak usage. The authors claimed that most vulnerabilities would be found in this phase. Phase 3 begins as the system is replaced by a newer release and attention shifts to the newer system. From this model, the authors claimed that the vulnerability discovery rate is controlled by two-factors, the momentum gained by market acceptance, and saturation (defined as total vulnerabilities minus the cumulative number of discovered vulnerabilities), and that the vulnerability discovery process could be modeled by the following equation:

$$dy/dt = Ay(B - y) \tag{2.1}$$

where t = calendar time, A = a constant of proportionality, y is the cumulative discovered vulnerabilities and B = total number of vulnerabilities.

Fitting their data to the model, the authors applied a chi-squared goodness of fit test to if this model applied. Figure 2.22 and Figure 2.23 show the results of this fit and their corresponding P-values for Windows NT 4.0 and Figure 2.24 and Figure 2.25 show the same for Red Hat Linux 7.1.

For most of the operating systems tested, the fit does appear to be statistically significant and the authors concluded that like defect densities, vulnerability densities fall seem to fall within a range, and that range appears to support the 1%-5% values proposed by McGraw and Anderson. They claimed that vulnerability density is a "significant and useful metric", and further surmised that the ratio of V_{KD} to V_{DD} over time could be used to predict remaining vulnerabilities. They also noted that

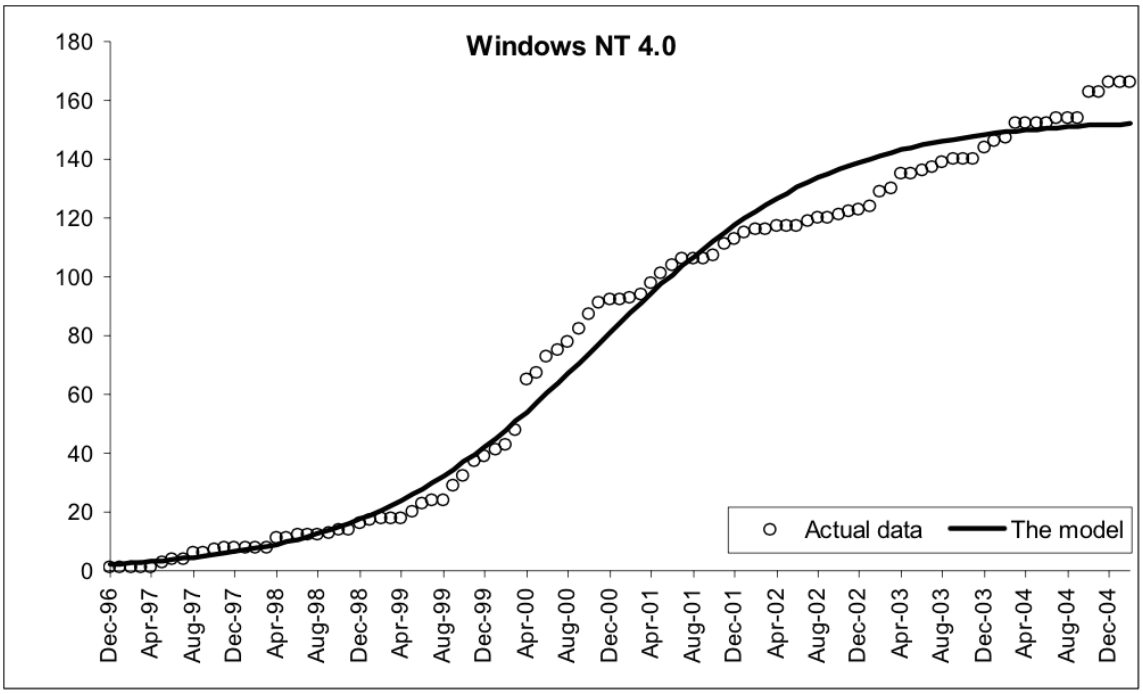


Figure 2.22: Chi-squared test of the Alhazmi vulnerability discovery equation for Windows NT. See [CA05]

Systems	***A***	***B***	***C***	χ^2	$\chi^2_{critial}$ (5%)	***P-value***
Windows 95	0.001938	49.5	1.170154	40.72	119.87	0.9999998
Windows 98	0.001049031	66	0.140462	64.79	96.2	0.742
Windows XP	0.001391	88	0.190847	25.75	56.94	0.961
Windows NT 4.0	0.000584	153.62	0.47	82.3942	127.69	0.923
Windows 2000	0.000528	163.96	0.073187	60.91	80.23	0.444

Figure 2.23: Results of the Alhazmi model fit tests for vulnerability discovery equation for Windows NT. See [CA05]

shared code in a newer operating system can impact the V_{DR} of a previous version and stated that further research was warranted to model this impact.

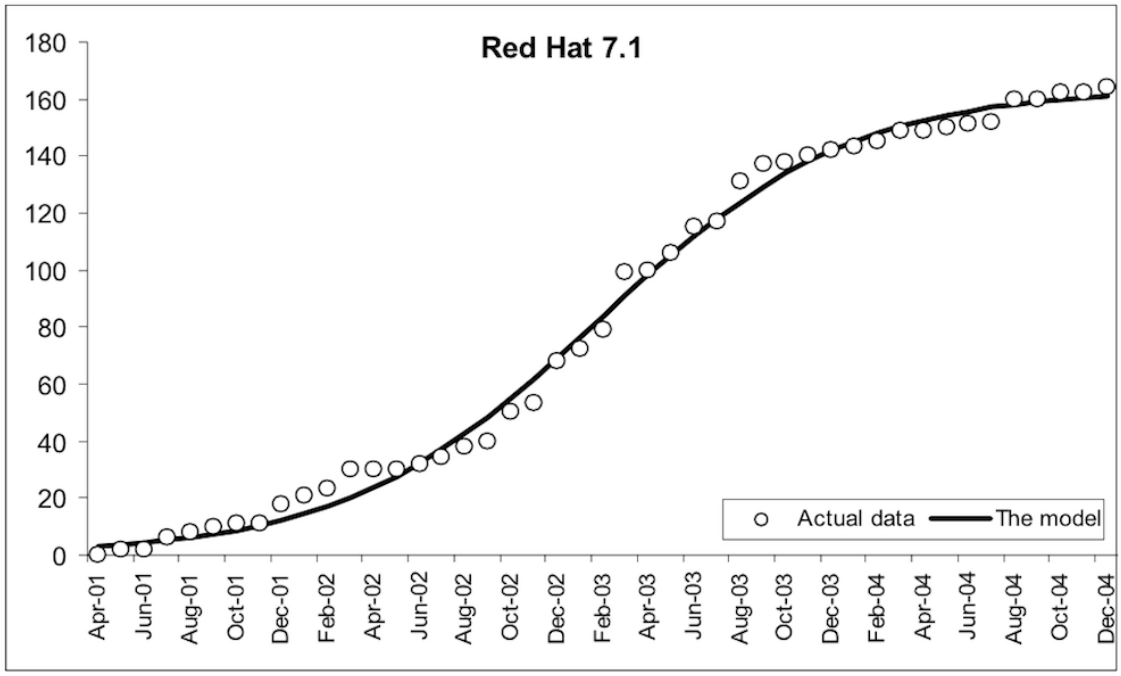


Figure 2.24: Chi-squared test of the Alhazmi vulnerability discovery equation for RedHat Linux. See [CA05]

Systems	***A***	***B***	***C***	χ^2	$\chi^2_{critial}$ (5%)	***P-value***
Red Hat Linux 6.2	0.000829	123.9393	0.129678	34.62	76.78	0.999974
Red Hat Linux 7.1	0.001106	163.9996	0.379986	27.62715	61.65623	0.989

Figure 2.25: Results of the Alhazmi model fit tests for vulnerability discovery equation for Redhat Linux. See [CA05]

2.3.3.4 Discussion

In this paper, the authors proposed a 3-phase 'S' curve model to describe vulnerability discovery over the lifetime of a software product. Later work has confirmed that the 'S' curve does appear to describe the vulnerability lifecycle, however, the authors stated that the discovery rate was governed by a finite number of vulnerabilities and their market value, (which they referred to as 'market share'), and used the ratio of vulnerabilities to defects to support their assumption. While research presented in Chapter 3 suggest that market share does appear to play a role in the number of

vulnerabilities discovered, especially in phase 2, my research also suggests that the long slow rise (here described as phase 1), followed by the steep linear rise (phase 2) is more likely the result of the attackers' learning curve. [CFBS10, CCBS14] regardless of the ratio of vulnerabilities to software defects, or the number of remaining undiscovered vulnerabilities.

Alhazmi, et al., go on to explain the shape of their model resulted from "the variability of the effort that goes into discovering vulnerabilities". They believed that the rise in phase 2 indicated a strong increase in effort devoted to finding vulnerabilities because this period was the one in which discovering vulnerabilities would be "the most rewarding". However, their only justification was that this was the period where the operating system reaches its peak of popularity. In spite of observing that legacy code carried over to a later version resulted in shared vulnerabilities between versions and that some vulnerabilities found in the later version actually affected the earlier version, they concluded that the cause of the increase resulted from increased effort.

Although the authors mentioned the attacker learning process, code involved in "external access" and the effects of shared code when discussing their results, they did not consider these as important factors affecting the vulnerability discovery process they were attempting to model. Instead, the authors considered the size of the installed base and the time to saturation the most important drivers in the vulnerability lifecycle. Since time to saturation is related to the vulnerability density and the vulnerability to defect ratio, they even claimed that measuring vulnerability density "allows us to measure the quality of programming in terms of how secure the code is." Research has shown that while vulnerability density may help determine whether software quality is improving, it can say nothing about the security of the code. [Gaf14, Bea16]

2.4 Weaknesses of Software Engineering Models

Attempts to apply software quality models to software security have not resulted in success. At the time of this writing, there is no generally accepted software security model or metric that provides any level of assurance equivalent to that provided by SREs for software reliability.

SREs were applied to security with the assumption that the factors which affect the security of software on a system are analogous to the factors that determine the reliability or quality of a software system. That these attempts met with such limited success suggests that software security may be affected by factors that are not related to software engineering.

For example, in 'Windows of Vulnerability', the authors presented a table comparing the linear regressions on the plots of each of the vulnerabilities they studied. They reported that the results "do not indicate any similarity in the shape across the incidents". It must be acknowledged that while Phf, IMAP and BIND are all software programs and as such were affected by developer choices, the programs themselves differed significantly in purpose and usage. Moreover, the characteristics of the vulnerabilities themselves differed. [10] This suggests that properties extrinsic to software affect vulnerability discovery. The Phf vulnerability affected an optional phonebook feature of web servers, while the IMAP vulnerabilities affected file system daemons integral to the server functionality, and while vulnerabilities in those two programs might adversely affect individuals or small groups of users, the BIND vulnerability affected a major part of the Internet's infrastructure. The extrinsic properties surrounding these software products are independent of each other and therefore, it is completely understandable their growth rates would be dissimilar. A focus on reported failure incidents is useful for reliability, but a similar focus on intrusions and reported incidents provides little insight to developers attempting to

[10] The Phf vulnerability was an implementation error, while the IMAP and BIND vulnerabilities were buffer overflows.

predict the next vulnerability.

A second example of the inadequacy of applying SRE models to security comes from 'Milk or Wine'. The authors claimed that SRE reliability growth models demonstrated that the rate of vulnerability reporting, particularly the reporting of foundational vulnerabilities, was declining. They also used these models to estimate that 67.6% of the total vulnerabilities in OpenBSD had been found. They claimed this demonstrated that OpenBSD was becoming 'more secure'.

However, recent discoveries of critical vulnerabilities in widely deployed legacy software has shown that a vulnerability model that relies on a decrease in the rate at which vulnerabilities are reported can say *nothing* about the security of the software. A closer look at one of these vulnerabilities serves to illustrate this point. Shellshock, the name given to a class of Bash [11] vulnerabilities discovered on September 24, 2014, was rated 10 out of 10 for severity, impact and exploitability, by NIST [NIS14]. Within hours there were compromise incidents reported. After two days, more than 17,400 attacks had been reported and after one week, attacks were averaging more than 1800 per hour. [Gaf14]. These vulnerabilities affected all vendor implementations, all versions after 1.09 and multiple platforms and operating systems. It is interesting to note that the NIST NVD database lists only one Gnu Bash vulnerability that is not related to Shellshock. Moreover, it was reported two years earlier and it was considered far less dangerous. [NIS08] [12]. Even more importantly, the vulnerable code was foundational code (It had been part of the software since 1989). The vulnerabilities that resulted in Shellshock went undiscovered for nearly *twenty five years*, thus the MTTF reliability growth models employed by Ozment and Schechter, would not have been of any help in predicting their discovery, their severity or their world-wide impact.

[11] A widely used Unix command-line interpreter.
[12] CVE-2012-3410 was rated 6.4 for impact, but only 3.9 for exploitability

In *Security Vulnerabilities in Software Systems: A Quantitative Perspective*, Alhazmi, et al. claimed, "Vulnerability density is analogous to defect density. Vulnerability density may enable us to compare the maturity of the software and understand risks associated with its residual undiscovered vulnerabilities. We can presume that for systems that have been in deployment for a sufficient time, the vulnerabilities that have been discovered represent a major fraction of all vulnerabilities initially present."

They went even further, stating, "In the same manner, vulnerability density allows us to compare the quality of programming in terms of how secure the code is."

In fact, it does no such thing. The recent discovery of severe vulnerabilities such as Heartbleed [Ltd14], Shellshock [Gaf14] and Poodle [TA14] are in widely deployed systems, comprised of mature code as well as in shared code libraries. By the accepted software quality metrics, based on the defect density and MTTF the software could be considered of high quality, yet the severity of these long dormant vulnerabilities had considerable impact. [13]

It is clear from these studies that by adapting software quality models to security, we can gain some insight into the lifetime of vulnerabilities after discovery, into the benefits of automated patching and into measuring possible damage resulting from exploit automation. It is equally clear that these models can not be used to determine the security of a software product, or to provide any means to measure expectation of risk (as measured by NIST's severity, impact and exploitability metric) from the remaining undiscovered vulnerabilities. The software security community needs new models that consider the extrinsic properties such as market share, attacker interest, exploit value, lifetime of the product, shared code (both between versions and between products), repurposed code, reverse engineering and automated fuzzing

[13]OpenSSL is at the heart of much of the world's internet communication. Heartbleed.com reported that "the open source web servers like Apache and nginx. The combined market share of just those two out of the active sites on the Internet was over 66% according to Netcraft́s April 2014 Web Server Survey." And that doesn't include VPNs and Email servers that depend on OpenSSL for protection.

that affect vulnerability discovery in addition to defects in the code.

2.4.1 Software Engineering Models and New Software Development Strategies

Programming strategies have changed considerably in the last few years from the traditional design and requirements heavy methodology, (often referred to as the *Waterfall Method*) [BBK78], to a strategy focused on rapidly programming new features in the software and releasing them as quickly as possible.

This new methodology was proposed by a group of software developers unhappy with the traditional software development methodology's inability to respond to market changes, meet customer demands, and the tendency of large scale software projects to get mired in the requirements, architecture and design phases of development, instead of producing working code. In 2001, they released *The Agile Manifesto* [BBvB+01]. This interest in new approaches to software development, (Extreme Programming, Crystal Methodologies, SCRUM, Adaptive Software Development, Feature-Driven Development and Dynamic Systems Development Methodology, Rapid Release Cycles, etc.) formed the basis of a new *Agile Software Development Alliance* [Bro14]

Since the manifesto, rapid release development lifecycles have become standard for many of the major developers including Apple, Google, Facebook and Microsoft. [Sad13, Alm13] With its focus on making "early and continuous delivery of valuable software" on shorter timescales and on flexibility and swiftly incorporating new design ideas rather than implementing pre-vetted, formalized top-down requirements, this methodology is incompatible with many of the well-tested software quality best practices discussed earlier.

Bessey *et al.*, discussed the prevailing attitudes towards software upgrades in terms of the number of bugs generated by each release. [BBC+10] They asserted that users want:

"different input (modified code base) + different function (tool version) = same result."

highlighting the delicate balancing act in traditional models of software development between the users' desire for new features and the impulse to squash as many bugs as possible in existing code.

The mainstream movement of the software engineering community to these iterative development models, raised concerns about the quality and reliability of the code being produced. Mnkandla *et al.,* introduced an innovative technique for evaluating agile methodologies and determined which factors of software quality were improved. [MD06] Kunz *et al.*, described a quality model, distinct metrics and their implementation into a measurement tool for quality management [KDZ08]. Olaqgue *et al,.* discussed the fault-proneness of object-oriented classes of highly iterative processes [OEGQ07]. Roden *et al.*, performed empirical studies and examined several quality factor models. [RVEM07] A recent study [Rod08] focuses on the metrics and maturity of iterative development. These studies provided a new viewpoint for evaluating software quality and the advantages of agile methodologies were shown in their experiment. However, noting that the results from applying these techniques did not allow for comparison with earlier products developed by traditional methodologies, Jinzenji,*et al.*, introduced a methodology for applying traditional SWE metrics to evaluate rapid release methodology. [JHWT13]

After Mozilla implemented a rapid release cycle development strategy for Firefox, Almossawi used a similar method when he analyzed the effects of the change on Firefox's code quality. [Alm13] He found that despite high file interconnectivity, actual internal and external complexity decreased and he concluded that the switch to rapid release had a positive impact on software quality.

2.4.1.1 Secure Software Development Models Conflict With Agile Methodologies

Mainstream software engineering practice resulted in development models intended to produce secure systems. Examples include the Process Improvement Model from ISO/IEC 21827, the Secure Systems Engineering-Capability Maturity Model (SSE-CMM) [Jel00], originated by the U.S. National Security Agency, but now an international standard), Microsoft's Secure Development Lifecycle (SDL) [HL06], Oracle's Software Security Assurance Process [Har14] and the Comprehensive, Lightweight Application Security Process (CLASP) [Vie05].

The goal of these models is:

> "*To design, build, and deploy secure applications, [...] integrate security into your application development life cycle and adapt your current software engineering practices and methodologies to include specific security-related activities*".

[MMW+05] In contrast, the *Agile* approaches to software development such as Extreme Programming (XP) [Con04], Adaptive Software Development (ASD) [Hig13], and Feature Driven Development (FDD) [CLDL99] are primarily intended to ensure customer satisfaction via rapid feature delivery [BBvB+01] rather than to produce secure code [BK04]. The U.S. Department of Homeland Security [oHS06] assessed each of the 14 core principles of the Agile Manifesto [BBvB+01] and found 6 to have *negative implications* for security, with only 2 having possible positive implications. Attempts to reconcile security with Agile development [SBK05, WBB04, KMH08] have noted that many of the practices recommended for security undermine the rapid iterations espoused by the Agile Manifesto. These conflicts are clearly stated as part of its intended purpose:

> "*We are uncovering better ways of developing software by doing it and helping others do it. We value:*"

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

> "*That is, while there is value in the items on the right, we value the items on the left more.*" [BBvB+01]

Woody [Woo13] surveyed Agile developers about the impact of security engineering activities on software development within an Agile approach, notably, the survey found that many industry-standard frameworks, including:

1. Design Requirements
2. Threat Modeling
3. Code Review

recommended practices in Microsoft's SDL's [Cor09], the risk analyses and external review recommended in Cigital Touchpoints [McG10], and the risk analyses, critical assets and UMLSec in the Common Criteria for Information Technology Security [Cri12], are at least partially incompatible with the rapid delivery approach.

Seacord notes that the traditional model of patch-and-install is problematic as "patches themselves contain security defects. The strategy of responding to security defects is not working. There is a need for a prevention and early security defect removal strategy." [Sea08]

2.4.1.2 Secure Software Design has Significant Upfront Costs

Secure Software Engineering models, such as SSE-CMM [Jel00] and Microsoft's SDL [HL06], presume that heavy investment in preventing vulnerabilities early in the software lifecycle is more cost effective, and that finding and removing vulnerabilities early in the development cycle produces more secure code over its lifetime. In 2010, Aberdeen Group published research [Bri] confirming that the total annual cost of application security initiatives is far outweighed by the accrued benefits organizations implementing structured programs for security development and found that they realized a 4x return on their annual investments in applications security.

Although security experts chastise software developers for favoring adding new features over writing less vulnerable code, It is well understood that, the survival

of a product in competitive software markets *requires* frequent introduction of new features. [MC09] This is especially important for user-facing software systems such as web browsers embroiled in features arms races. As a consequence, three major web browser developers, Google (Chrome), Mozilla (Firefox) and Internet Explorer (Microsoft), overhauled their development lifecycle, moving from large-scale, infrequent releases of new versions with later patches as needed, to releases with new features at much shorter, regular intervals. [Cor13].

New releases of Chrome and Firefox versions occur every six weeks. The primary intent of each RRC iteration is to get new features to users as rapidly as possible, though they may also include bug fixes in the release. [Nig11, Laf10]

In contrast to SSE models, 'Agile' programming models, with their focus on frequent change and rapid delivery of software, cannot afford to spend the extensive time required to do risk analysis, threat modeling and external review [Woo13] in the development phase. Therefore, with Chrome, Firefox and Microsoft [Cor13] releasing new features at a much faster rate, one might expect to see increases, both in the number of vulnerabilities and the rate at which they are discovered and disclosed.

2.5 An Alternative Software Security Metric: Attack Surface Models

A different method for measuring software and system security which does not consider vulnerabilities resulting from software defects has been proposed as an alternative to VDMs. This model focuses on enumerating attack vectors or attack surfaces. An *Attack Surface* is any combination of methods, channels, ports, interfaces, system calls, etc. by which the software, (or individual processes within software) communicates outside itself. The first attack surface models attempted to model the path an exploit might take and measure the likelihood of success for each attack

vector. [How03, HPW05] After applying this methodology to measure the attack surfaces of four versions of linux [MW04], Manadhata and Wing, found that the model while promising was too informal and undependable, since it relied on the subjective analysis of a security expert to determine the attack vectors. They proposed a more formal approach, with a formal model and a standard Attack Surface Metric (ASM) to address these issues. Their model defined an attack surface as a triple composed of a system's set of entry and exit points for each method, communication channels and untrusted data items. It defined the value measured as the ratio between a to-be-determined *damage potential* value and *damage effort* value. The final attack surface metric is the sum of these ratios. In *An Attack Surface Metric* [MW08] the authors recommended using call graphs to define entry and exit points, and privilege and access rights as parameters for the damage potential and effort ratio. Testing their model on Firefox, they analyzed the source code of vulnerability patches to quantify changes in attack surface measurements. After identifying seven types of vulnerabilities as relevant to attack surface measurements [14] they found 12 of the 48 vulnerability patches analyzed to be relevant, and of those relevant patches 8 reduced the attack surface. Interestingly, they found that 3 out 4 of the Cross-Site scripting patches did not reduce the attack surface at all. [15]

Attack surface metrics are being used by developers to prioritize testing, code analysis resource deployment and patch management, i.e., ensuring that patches do not increase the attack surface. However, the damage potential to effort ratio classification used is often subjective, the attack surface enumeration is not automated, and tends to be prohibitively time consuming on large codebases. This model also cannot account for side channel, covert channel and multi-layered attacks and stack

[14] the relevant types are: Authentication Issues; Permissions, Privileges, and Access Control; Cross-Site Scripting; Format String Vulnerability; SQL Injection; OS Command Injection; Information Disclosure.

[15] The authors note that they 'do not expect XSS patches to always' reduce the attack surface', but do not explain why.

pivoting attacks. [LS13] [16] This seems to negate the authors claim that their model "entirely avoids the need to identify the attack vectors" and that it "does not require a security expert". [MW04] [MW08] Chapter 6 addresses the limitations of this model.

2.6 Defensive Models From Outside Of Software Engineering

Computer Scientists use many real world analogies to describe patterns in computer security. Scientists often use epidemiological and biological terminology, e.g.,: "computer viruses" and "anti-viruses", etc and use the mathematics of infectious disease to describe virus propagation. [SZ03, Som04] At the same time, they also talk about computer security in military terms, referring to the security arms race, and offensive and defensive strategies "cyber warfare", etc. [MMRJ+05, And01]

These analogies not only give people familiar ways to frame and discuss security problems, the strategies used by these two groups to solve real world problems and try to develop the means to apply analogous real world solutions to computer security problems.

2.6.1 Dynamic Defensive Strategies from Nature:

One area of the biological sciences from which software security may draw is the field of evolutionary biology. Evolutionary biology models describe the growth and interactions between competing and cooperating organisms within a defined environment.

To adapt such a model to software security, the "environment" would be the system in which the software operates, including the hardware and firmware, and

[16] A *pivot* attack is a common technique used in Return-oriented Programming (ROP) exploitation [Ros11]. By pointing the stack pointer to an attacker-owned buffer, such as the heap, pivoting can provide more flexibility for the attacker to carry out a complex ROP exploit.

the "organisms" the programs, libraries, processes and interfaces running on the system. Similar to the behavior of living organisms in biological models, software organisms compete for system resources, share space and communicate with each other. In such a model, defects could be considered detrimental to the health of the software and vulnerabilities especially harmful to the 'survival' of the software. It is important to recognize, that evolutionary biology models cannot be strictly applied to software security. In the real world, evolution is dependent on forces of nature and random mutations. In the computer security ecosystem, mutations are not random, but directed by intelligence. [SLF08].

That said, two popular evolutionary models appear useful for thinking about the behaviors of attackers finding and exploiting vulnerabilities and developers creating and maintaining software. The first model describes the behaviors of predator vs. prey and the second that of parasite vs. host. A major component of these models is that change on the part of one of the organisms results in a corresponding change in the competitor. In fact, these models portray cycles of adaptation. This factor seems particularly apt for software security. One has only to look at the history of exploitation techniques and corresponding mitigations in Microsoft Windows to see the patterns[Sot09]. Moreover, we can look to competing malware development platforms Zeus and Spy-eye or the enhancements of Duqu that came out of the Stuxnet virus to see competition for resources driving evolution. [Wil10, Ula10]

Looking first at predator vs. prey, one finds definite analogues to computer security. Attackers are predators, programs are prey. Attackers actively search for new weaknesses similar to predators searching for the weakest animal in the herd. Natural selection favors more effective predators and stronger defenses in prey. Weak, easy to exploit prey is quickly exploited. Very secure software demands much greater investment in time and resources of the predator, and so may cause predators to ply their attacks elsewhere.

However, there are some aspects of the common predator vs. prey models that

seem to have no software security analogues. In evolutionary biology predator vs. prey models, the size of population of one side is dependent on the size of the population of the other. Weaknesses can be fatal. Yet, an increase in the number of attackers doesn't kill off a software product, and vendors do not go out of business just because vulnerabilities in their code are exploited. The software security ecosystem does not see the same rise and fall of interdependent population cycles that is seen in the biological ecosystem. Instead, attempts to measure the attacks and corresponding defenses show a steady increase in both rate of exploitation and corresponding increase in patch availability rate, (though the latter continues to lag behind the former) [Fre09].

If we look closely at the patterns in attacks and defenses, we see a cycle of vulnerability discovery, exploit released into the wild, and subsequent vulnerability patch released. However many vulnerabilities are found and exploited, the host is never actually destroyed, so the prey population density doesn't change, and unlike an environment where prey is unable to fight back, big changes on the defenders side, such as stack randomization, have measurable adverse affects on attackers' ability to successfully develop exploits [Mil08]. The victim also has the ability to actively force the attackers to adapt in order to survive. This is a life-cycle that follows much more closely the model of parasites and their hosts. In evolutionary biology, the parasite vs. host hypothesis that most closely describes what we see in computer security is called the *Red Queen hypothesis*.

The Red Queen hypothesis, first proposed by biologist L. van Valen in 1973, is a model that tries to explain the evolutionary dynamics between competing species and the interactions of tightly co-evolving species. It proposes that an evolutionary change by one species in a parasite/host relationship results in corresponding change in the other. The name comes from Lewis Carroll's "Through the Looking Glass" where the Red Queen says "It takes all the running you can do, to keep in the same

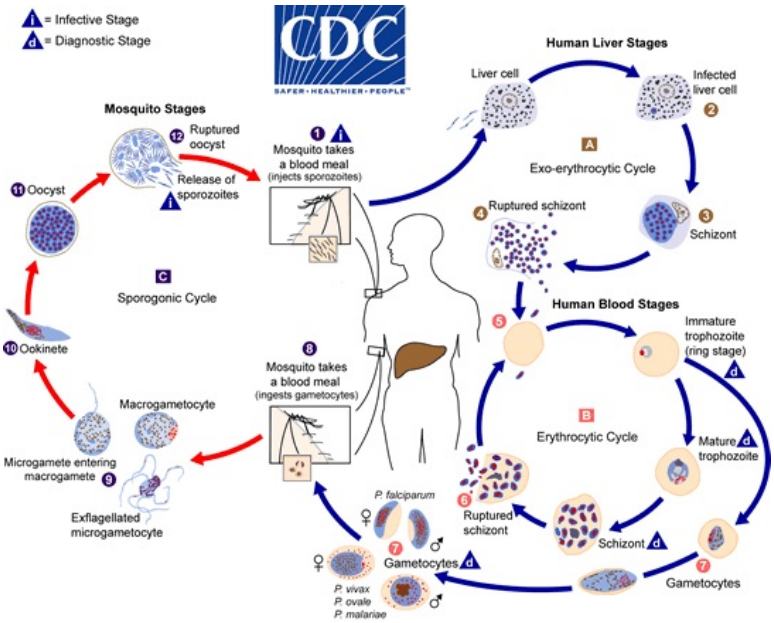


Figure 2.26: Center for Disease Control Model of the Malaria Parasite Lifecycle [fDCP16]

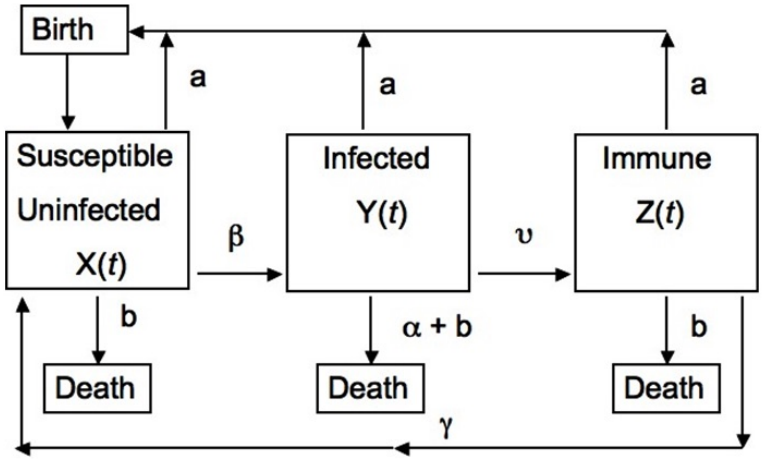


Figure 2.27: Anderson and May Model: Microparsitic infections as regulators of natural populations. Taken from [AG82]

place". Current research seems to support this hypothesis. For example, the Red Queen hypothesis applied to a predator vs. prey model, research shows that exposure to parasites makes organisms results in changes to the hosts (in particular, it makes them more resilient). [SKVL14].

Models of this relationship between parasite host show the feedback loop and changes to the defender (host) and the attacker (parasite) systems as new information is received and processed. For example, Figure 2.26 shows the Center for Disease Control's model of the lifecycle of a very common parasite, the organism which causes malaria, while Figure 2.27 shows a more general model for microparasites.

This model has been shown to explain escalations of insurgent activity in Faluja, [JCB+11], economic and political choices in China [BM11], and the success or failure of biotech companies [Oli00]

This perspective was formally acknowledged by the Department of Homeland Security (DHS) in 2011. In the paper *Enabling Distributed Security in Cyberspace Building a Healthy and Resilient Cyber Ecosystem with Automated Collective Action* [oHS11b] and in their publication *Blueprint for a Secure Cyber Future* [oHS11a] DHS identified what they consider the fundamental elements of the cyber security ecosystem; describing it as analogous to natural ecosystems. Their proposed defensive strategies for a "healthy ecosystem" were explained in terms of the human immune system and the public health system (the Centers for Disease Control and Prevention (CDC)).

2.6.2 Dynamic Defensive Strategies from the Military:

Military terminology has become one of the most common ways to talk about vulnerabilities in computer systems. It seems appropriate that military strategy models might prove equally useful for thinking about software security. One common model is known as the OODA Loop (see Figure 2.28) OODA stands for Observe, Orient, Decide and Act. [Boy76] The OODA Loop is a decision-making model developed by U.S. Air Force pilot and military strategist John Boyd that has become important in the design of government, military, corporate and even courtroom strategies. [Lin03, Kot02, Ric04, Dre04]

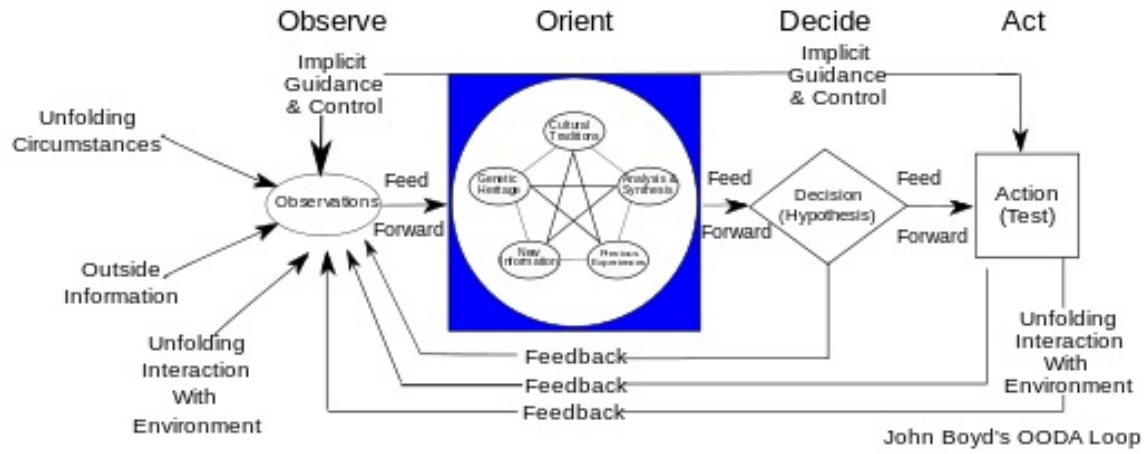


Figure 2.28: The OODA Loop, John Boyd
[Boy95]

The loop is a continuous feedback cycle with 4 primary stages, Observe, Orient, Decide and Act. Boyd first proposed the concept as a means to clarify his "Energy-Maneuverability Theory" [BP66] for achieving success in air-to-air combat. Referring to Goedel's incompleteness theorem, Boyd believed that "One cannot determine the character or nature of a system within itself. Moreover, attempts to do so lead to confusion and disorder." He claimed that "The ability to shift or transition from one maneuver to another more rapidly than the adversary enables one to win in air-to-air combat" and the key to success were the ability to "diminish the adversary's capacity for independent action", and also "diminish the adversary's ability to communicate or interact with his environment while sustaining or improving ours." The game Boyd maintained was "a see-saw of analysis and synthesis across a variety of domains, or across competing/independent channels of information." [Boy06]

In developing his theory, Boyd drew deeply from a number of scientific fields. In particular, evolutionary biology, complexity theory and the science of how people learn. Boyd looked toward evolutionary biology for models of active defense in nature. Organisms competing for resources drive changes in the ecosystem and the result is survival of the fittest. He was extremely interested in adaptation for

survival and emergent behaviors in complex systems. He believed that successful organisms were those that were most aware and best able to adapt to changes in their environment. [Osi06] As he distilled this concept into the OODA Loop, he understood that tying his ideas together was a central theme. Fast and correct processing of information was key to winning. In a letter to his wife he wrote, "I may be on the trail of a theory of learning quite different and - it appear now more powerful than methods or theories currently in use." [Boy72]

While the common perception of this model is that it represents the 'need for speed', that is, the one who moves that fastest wins, [17] [Hil15] Boyd preferred the term "tempo" and to him it meant more than speed. It meant processing the new information and synthesizing the correct choices so that the adversary would be forced to react instead of acting. "In order to win, we should operate at a faster tempo or rhythm than our adversaries–or, better yet, get inside [the] adversary's Observation-Orientation-Decision-Action time cycle or loop." [Cor04] Correct tempo required the correct strategic decision making process. In fact, in his presentations he often tried to bring his listeners through the steps themselves so that like he, their thought process would be to "...observe, analysis, synthesis, hypothesis and test", and he strongly emphasized, that the first two must lead to the most imporant synthesis. [Boy95]

This emphasis is strongly apparent in his drawing of the OODA Loop (see Figure 2.28). Most of the detail is in the Observe and Orient stages, with by far the most attention spent on Orientation. Boyd frequently repeated, "we must effectively and efficiently orient ourselves; that is, we must quickly and accurately develop mental images, or schema, to help comprehend and cope with the vast array of threatening and non-threatening events we face.", "Adaptability is the power to adjust or change in order to cope with new and unforeseen circumstances." [Boy95] and later stated that, "Orientation is the schwerpunkt. It shapes the way we interact with the

[17] *"Time is the dominant parameter.*

environment[...]." [Boy87]

According to Boyd, "variety/rapidity/harmony/initiative (and their interaction) seem to be key qualities that permit one to shape and adapt to an ever changing environment"[Ang86].

2.6.3 Dynamic Models From Industry

(b)reaking a whole into its parts is analysis. You gain knowledge by analysis. Building parts into wholes is synthesis. You gain understanding through synthesis. When you take a system apart and analyze it, it loses its properties. To understand systems you need to look at them as wholes. (John Boyd, expanded by O'Connor & McDermott) [OM97]

A key characteristic of both the Red Queen hypothesis and the OODA Loop models is that they describe a learning process which ultimately results in an adaptation. In *Learning in Action* Garwin delineates the characteristics necessary for learning to take place in complex systems, stating: "learning is defined as the process by which knowledge about action-outcome and relationships between the organization and the environment is developed" [Gar00]

Modern business strategies intent on benefiting from this process in the corporate environment focus primarily on enhancing learning in the *early part of the cycle*, and success is commonly measured by a *Learning Curve.* [Wri36, Gro70]

2.6.3.1 The Learning Curve

Learning curves, also called experience curves, have long been used to measure the increase in learning that results from repeated experience, or from increased knowledge over time. First employed to measure production costs in the aircraft industry, their usage has expanded to multiple industries including economics, machine learning and software development. [Rac96, HT81, Gal86] In these fields, the focus is on

'development', where development is defined as "whole system learning process with varying rates of progression." [Ger91]

When these learning processes are modeled, most complex systems exhibit a "Sigmoid" or "S" shaped curve, slower at the start, accelerating and finally plateauing. [HM95, San95, Mit97]

The "S" learning curve shape is a measure of the increase in proficiency as result of repeated exposure. As I will show in chapters 3 and 4, this characteristic can also be found in the early vulnerability lifecycle.

2.6.4 The Learning Curve in Vulnerability Discovery

Jonsson and Olovsson [JO97] tested the effect an attacker's knowledge and experience had on successfully compromising a system. Assigning students to attack a University computer system, they measured number of successful breaches, rate of breach and experience level. They concluded that there appears to be a learning curve that disadvantages the less experienced attacker.

Gopalakrishna and Spafford [GS05] presented a trend analysis of vulnerabilities reported on Bugtraq, CVE and ICAT. They speculated that the increased rate of discovery of vulnerabilities of the same type in a piece of software was the result of a learning period. They reasoned that this 'learning' was the period of time required for a given piece of software to gain a "critical-mass" of users before bugs are discovered.

However, as Ozment [Ozm07] points out, this incorrectly assumes that some fixed proportion of the total user population are looking for vulnerabilities. Ozment conjectured scenarios in which an attacker discovers a vulnerability or reads about the details of one, and applies these "lessons learned" to a similar domain by attempting an attack of a similar type. This observation is the contrapositive to the benefits of rapid releases we have proposed in this dissertation: the usefulness of these "lessons

learned" is minimized as the section of the codebase relevant to the type of vulnerability in question may have already been deprecated by the time the attacker applies this learning. Indeed, this is further supported by the Bug Bounty findings presented by Coates [Coa11], wherein the vast majority of flaws reported fall into a small set of classes (*e.g.*, CSRF and XSS bugs account for 70% of those reported).

Chapter 3

The Honeymoon Effect

"Vitality shows in not only the ability to persist but the ability to start over." (F. Scott Fitzgerald)

3.1 Properties of the Early Vulnerability Life Cycle

Existing approaches to understanding the vulnerability life cycle focus on attempting to measure the lifetime of vulnerabilities in long standing systems. These are systems that are assumed to have 'stood the test of time' in that their easy to find defects (and the resulting vulnerabilities) have been discovered and patched. [OS06, AFM00, Ran10]

The focus of these studies is on the vulnerability of software relatively late in its life cycle. In this chapter, I examine the characteristics of the first vulnerabilities found in software while it is new, and I present the results of the first study to look at the intervals from genesis to initial vulnerability discovery and from initial vulnerability discovery to second.

To understand the early vulnerability life cycle, I analyzed the vulnerabilities of several versions of the most popular software products, Operating Systems, server

applications and user applications, I measured the time between the official release date of the version and the disclosure dates of the vulnerabilities. Surprisingly I found that, in the majority of cases, the average period between release date of a software product and its very first vulnerability, (often referred to as a *Zero-day or 0-day*), is considerably longer than the mean time between first vulnerability and second or between the second and the third. A similar, although slightly less pronounced effect is present when minor version releases are considered.

I call this unexpected grace period the *Honeymoon Effect*, alluding to the blissful state newlyweds experience in the first days of marriage, and believe it to be important, because this new quantitative analysis challenges the expectations and intuition of the software engineering community about the effect of software quality on security. The results suggest that *early* in the system life cycle, when new software is first deployed, factors other than intrinsic quality of the software can dominate the question of how likely a system is to be attacked. For the purpose of this discussion, I define an *Intrinsic Property* as any property or characteristic of a software product that can directly controlled by the developer, such as, programming language, addition of features, patch release rate, etc. An *Extrinsic Property* is defined as a characteristic of the environment in which the software operates (lives). Extrinsic properties are properties of components that may be essential to the program's functionality, such as the operating system needed by the software to function, or ancillary such as other applications on the system, firmware, networking protocols, shared libraries, etc., that are not under the direct control of the software program developer.

Interestingly, I found that on newly deployed systems, that is, those that have not yet had the "easy" bugs fixed and patches made available, often enjoy a longer "honeymoon" period (before the first zero-day attack occurs) than they will enjoy later in their life.

The Honeymoon Effect also illustrates a tension between current software engineering practices and security: the effect of code reuse. "Good programmers write code, Great programmers reuse " is an often quoted aphorism. [Ray99] An implicit assumption made is that reusing code not saves effort, but as the code has been deployed and is in service, it is both more reliable and more secure. While reliability in the absence of an adversary may result from code reuse, the addition of an adversary completely changes the observables as I will show in subsequent sections of this chapter of the thesis.

3.2 Methodology and Dataset

I began by compiling an empirical dataset of more than 30,000 vulnerabilities disclosed between January 1999 and January 2008. The analysis focused on the number and time of vulnerability disclosures on a per vendor, per product and per version basis. Only publicly available information from Secunia [Seca], the National Vulnerability Database (NVD) [NIS08] and the Common Vulnerabilities and Exposures (CVE) [CVE08] initiative that feeds NVD was used. For every vulnerability NVD provides the publication date, a short description, a risk rating, references to original sources, and information on the vendor, version and name of the product affected. Defining the disclosure date as the earliest calendar day on which information on a specific vulnerability is made freely available to the public in a consistent format by a trusted source. [Fre09] The information of over 200,000 individual security bulletins from several Security Information Providers (SIP) was downloaded, parsed, and correlated. [1] Thus, *all* security advisories from the following eight SIPs: Secunia, US-CERT, SecurityFocus, IBM ISS X-Force, Vupen, SecurityTracker, iDefense's (VPC), and TippingPoint's Zero Day Initiative (ZDI) were processed. [Seca, UC, Sec08, XF, Vup16, Secb, iDe, Tip] To ensure accuracy,

[1] The set of SIPs was chosen based on criteria such as independence, accessibility, and available history of information

over 3,000 instances of software version information for the products subject to this analysis were manually checked to account for inconsistencies in NVD's vulnerability to product mapping.

The majority of the existing vulnerability lifecycle and VDM research which makes use of the NVD dataset focused primarily on a small number of operating systems or a few server applications and only examined a single version or compared a small set of versions (e.g., Windows NT, Solaris 2.5.1, FreeBSD 4.0 and Redhat 6.2, or IIS and Apache).

As I was concerned with understanding the properties of vulnerability discovery early in the post-release vulnerability lifecycle, this dataset contained many types of mass market software, including operating systems, web clients and servers, text and graphics processors, server software, and so on.

My analysis focused on publicly distributed software released between 1999 and 2007. (2007 is the latest date for which complete vulnerability information was reliably available from various published data sources during the time-frame of this analysis). I included both open and closed source software.

To encompass the most comprehensive possible range of relevant software releases, I collected data about all released versions of the major operating systems (Windows, OS X, Redhat Linux, Solaris, FreeBSD), all released versions of the major web browsers (Internet Explorer, Firefox, Safari), and all released versions of various server and end user applications, both open and closed source. The server and user applications were based on the top 25 downloaded, purchased, or favorite applications identified in lists published by ZDNet, CNet, and Amazon, excluding only those applications for which accurate release date information was unavailable or that were not included in the vulnerability data sources described below. In total, I was able to compile data about 38 of the most popular and important software packages.[Ama08, CNE08]

For each software package and version during the period of our study, I examined

public databases, product announcements, and published press releases to assign each version a release date. For the period of versions (1990-2007) and for the period of vulnerabilities (1999-2008), I identified 700 distinct released versions ('major' and 'minor') of the 38 different software packages.

While it is not possible to measure the amounts of legacy code from version to version in closed source products in contrast with open source products, it is possible to measure the numbers of *legacy* vulnerabilities. Vulnerabilities from legacy code are those bugs which are not found through normal regression testing and may lie dormant through more than one version release. By comparing the disclosure date of a vulnerability with the release dates and product version affected, it is possible to determine which vulnerabilities result from earlier versions. For example, a vulnerability which affects versions (k,...N) (0<k<N) of a product, but not versions (1,...,k-1) and was disclosed after the release date of version N, indicates that the vulnerability was introduced into the product with version k, and that it stayed dormant until its discovery after the release of version N. On the other hand, a vulnerability only affecting version N but not any earlier versions indicates that the vulnerability was introduced with the new version N. I used this method to find the legacy vulnerabilities for all the versions of the products in our analysis.

Next, I determined which vulnerabilities resulted in the Honeymoon Effect by finding the very first vulnerability disclosed (hereafter referred to as the *Foundational* vulnerability). For each version of each product in the analysis the number of days from the release of that version to the disclosure date of its foundational vulnerability were measured. Where possible, I also calculated the number of days from foundational vulnerability until the disclosure date of the second earliest vulnerability and from second to third earliest and from third to fourth earliest vulnerability. A Honeymoon vulnerability is defined as either *Regressive* if it results from a vulnerability in legacy code or *Progressive* if it was found in code new to this version. Regressive vulnerabilities are those vulnerabilities which are discovered and disclosed

in code *after* the version in which it was introduced has been obsoleted by a more recent version. For example, a vulnerability disclosed in version 13 that also affects versions 10, 11 and 12 would be classified as regressive.

Finally, to ascertain whether regressive vulnerabilities could be the result of code reuse rather than configuration or implementation errors, I manually checked the NVD database description and the original disclosure sources for information regarding the type of vulnerability. I found that 92% of the regressive vulnerabilities were the result of code errors (buffer overflows, input validation errors, exception handling errors) which strongly indicates that a vulnerability that affects more than one version of a product is most likely a result of legacy code shared between versions. I removed the vulnerabilities which are the result of implementation or configuration errors from the dataset and focused exclusively on code errors.

For this chapter, I define the following terms. A *zero-day or 0-day vulnerability* [2] is a security threat that is known to an attacker which may or may not be known to a defender and for which no patch or security fix has been made available. A *zero-day or 0-day attack* is an attack which exploits a zero-day vulnerability. A *window of vulnerability* [AFM00] exists during the period of time between the discovery of the zero-day vulnerability and the release of the security fix.

3.3 The Early Vulnerability Life Cycle

Virtually all mass-market software systems undergo a lengthy post-release period, during which users discover and report bugs and other deficiencies. Most software suppliers (whether closed-source or open-source) build into their life-cycle planning a mechanism for reacting to bug reports, repairing defects, and releasing patched versions at regular intervals. The number of latent bugs in a particular release of a given piece of software thus tends to decrease over time, with the initial, unpatched,

[2] The terms *zero-day or 0-day will be used interchangeably*

release suffering from the largest number of defects. In systems where bugs are fixed in response to user reports, the most serious and easily triggered bugs would be expected to be reported early, with increasingly esoteric defects accounting for a greater fraction of bug reports as time goes on.

As was discussed in Chapter 2, empirical studies in both the classic [Bro95b] and the current [JMS08] software engineering literature have shown that, indeed, this intuition reflects the software life-cycle well (see Figure 3.1). Invariably, these and other software engineering studies have shown that the rate of bug discovery is at its highest immediately after software release, with the rate (measured either as inter-arrival time of bug reports or as number of bugs per interval) slowing over time.

Note that some (but not all) of the bugs discovered and repaired in this process represent *security vulnerabilities;* in security parlance a vulnerability that allows an attacker to exploit a newly discovered, previously unknown bug is called a *0-day* vulnerability. Virtually all software vendors give high priority to repairing defects once a 0-day exploit is discovered.

It seems reasonable, then, to presume that users of software are at their most vulnerable, with software suffering from the most serious latent vulnerabilities, immediately after a new release. That is, one would expect attackers (and legitimate security researchers) who are looking for bugs to exploit to have the easiest time of it early in the life cycle. This, after all, is when the software is most intrinsically weak, with the highest density of "low hanging fruit" bugs still unpatched and vulnerable to attack. As time goes on, after all, the number of undiscovered bugs will only go down, and those that remain will presumably require increasing effort to find and exploit.

In other words, to the extent that security vulnerabilities are a consequence of software bugs, our intuition, based on conventional software engineering wisdom tells us to expect the discovery of 0-day exploits to follow the same pattern as other

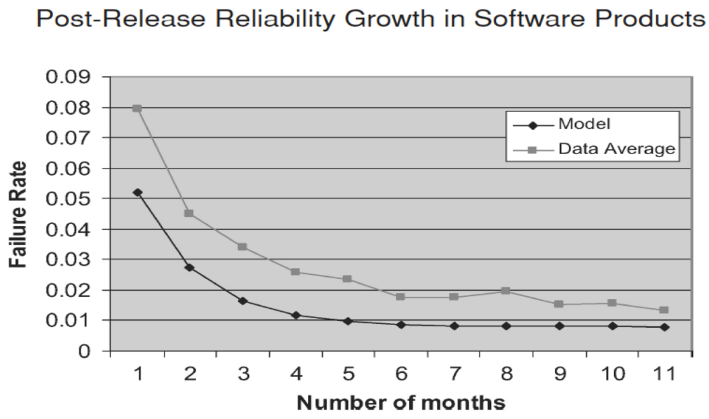


Figure 3.1: The highly regarded Brooks software engineering defect predictive model and actual defect discovery metrics thirty years later. [JMS08]

reported bugs. The pace of exploit discovery should be at its most rapid early on, and slowing down as the software quality improves and the "easiest" vulnerabilities are repaired.

But my analysis of the rate of the discovery of exploitable bugs in widely-used commercial and open-source software, tells a very different story than what the conventional software engineering wisdom leads us to expect. In fact, new software overwhelmingly enjoys a *honeymoon* from attack for a period after it is released. The time between release and the first 0-day vulnerability in a given software release tends to be markedly longer than the interval between the first and the second vulnerability discovered, which in turn tends to be longer than the time between the second and the third. That is, when the software should be at its *weakest*, with the "easiest" exploitable vulnerabilities still unpatched, there is a *lower* risk that this will be discovered by an actual attacker on a given day than there will be *after the vulnerability is fixed!* We expect to see something like the top graph in Figure 3.2 and instead we find something much more similar to the bottom graph in that same Figure 3.2.

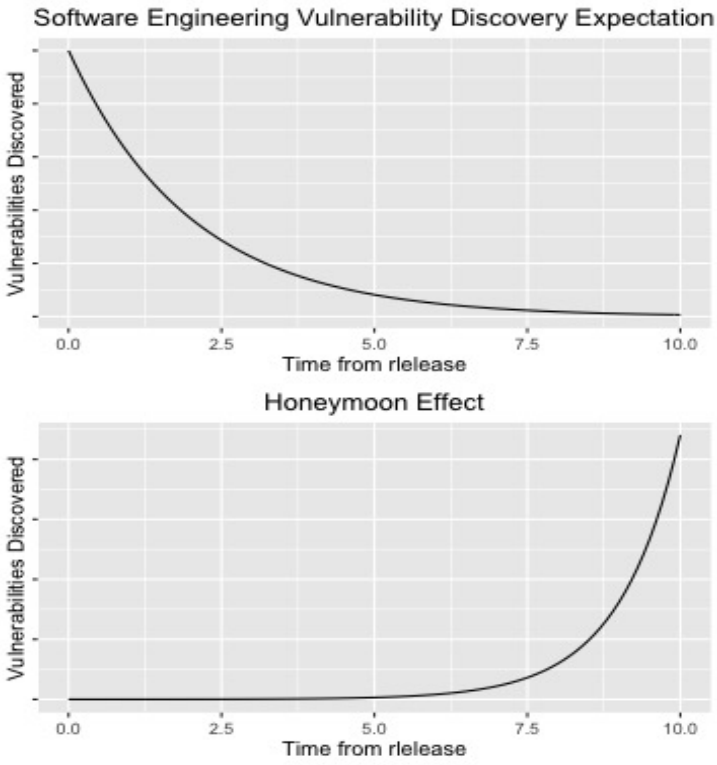


Figure 3.2: Toy Graph: The top graph displays an expected vulnerability discovery timeline according to software engineering models. The bottom graph displays an expected vulnerability discovery timeline resulting from the Honeymoon Effect.

3.3.1 The Honeymoon Effect and Mass-Market Software

Remember, the first (publicly reported) exploitable vulnerability is defined as the *Foundational* vulnerability, and a software release experiences a *Positive Honeymoon* if the interval p_0 between the (public) release of the software and the foundational vulnerability in the software is greater than the interval p_{0+1} between the foundational vulnerability and the second(publicly reported) vulnerability.(see Figure 3.3) We will refer here to the interval p_0 as the *Honeymoon Period* and the ratio p_0/p_{0+1} as the *Honeymoon Ratio*. By definition, a software release has experienced a positive honeymoon when its honeymoon ratio > 1.

For this analysis, I examined 700 software releases of the most popular recent

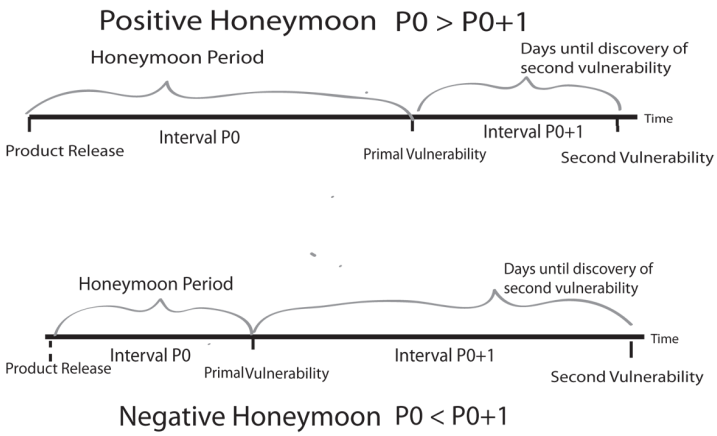


Figure 3.3: The Honeymoon Period, both Positive and Negative time-lines

mass-market software packages for which release dates and vulnerability reports were available. In 431 of 700 (62%) of releases, the Honeymoon Effect was positive. Most notably, the median overall honeymoon ratio (including both positive and negative honeymoons) p_0/p_{0+1} was 1.54. That is, the median time from initial release to the discovery of the foundational vulnerability is 1 1/2 times greater than the time from the discovery of the foundational vulnerability to the discovery of the second vulnerability. The Honeymoon Effect is not only present, it is quite pronounced, and the effect is even more pronounced when the minor version updates are excluded and the analysis is limited the set of major releases. For major releases only, the honeymoon ratio rises to 1.8. [3]

Remarkably, positive honeymoons occur across the entire dataset for all classes of software and across the entire period under analysis. The Honeymoon Effect is strong whether the software is open- or closed- source, whether it is an operating system, web client, server, text processor, or other application, and regardless of the year in which the release occurred.(see Table 3.1)

Although the Honeymoon Effect is pervasive across the entire dataset, one factor appears to influence its length more than any other: the re-use of code from previous

[3]This includes both positive and negative honeymoons.

Table 3.1: Percentages of Honeymoons by Year

Year	Honeymoons
1999	56%
2000	62%
2001	50%
2002	71%
2003	53%
2004	49%
2005	66%
2006	58%
2007	71%

releases, which, counter-intuitively, *shortens* the honeymoon. Software releases based on "new" code have longer honeymoons than those that re-use old code.

3.3.2 Honeymoons in Different Software Environments

The number of days in the honeymoon period varies widely from software release to software release, ranging from a single day to over three years in the dataset. The length of the honeymoon presumably varies due to many factors, including the intrinsic quality of the software and extrinsic factors such as attacker interest, familiarity with the system, and so on.

To "normalize" the length of the honeymoon for these factors in order to enable meaningful comparisons between different software packages, the *honeymoon ratio* which is defined as the ratio of the time between the product or version release and the discovery of the first exploit and the time between the discovery of the first vulnerability and the second, may be more revealing. This is because time to the second vulnerability discovery occurs in exactly the same software and this analysis was interested in understanding the mechanism behind the rate of discovery within an individual release and not between products. Moreover, because this analysis

comprises an extremely diverse set of software packages, with widely differing development methodologies, marketshare, attacker interest, size of code-bases, etc. The use of the honeymoon ratio gives a relative or self-normalized value for comparison. To minimize the effect of skew and to see overall trend in the data, the graphs in this chapter are presented in log-scale. For reference, the same data plotted in linear-scale can be found in appendix B.

The median number of days in the honeymoon period across all 700 releases in the dataset was 110. The median honeymoon ratio across all releases is 1.54. [4]

The honeymoon ratio remained positive in virtually all software packages and types. The effect is weaker, but also occurred, between the foundational and second and second and third reported vulnerabilities, depending on the particular software package.

Figure 3.4 shows the median honeymoon ratio (and the median ratios for the intervals between the second, third and fourth vulnerabilities) for each operating system in the dataset. Figure 3.5 shows the median honeymoon ratio of servers, and Figure 3.6 shows end-user applications.

3.3.3 Open vs. Closed Source

The Honeymoon Effect is strong in both open- and closed-source software, but it manifests itself somewhat differently.

Of the 38 software systems we analyzed, 13 are open-source and 25 are closed-source. But of the 700 software releases in the dataset 171 were for closed-source systems and 508 were for open source. Open-source packages in the dataset issued new release versions at a much more rapid rate compared to their closed source counterparts.

[4]There was a high variance in the time to first vulnerability between products. Additional

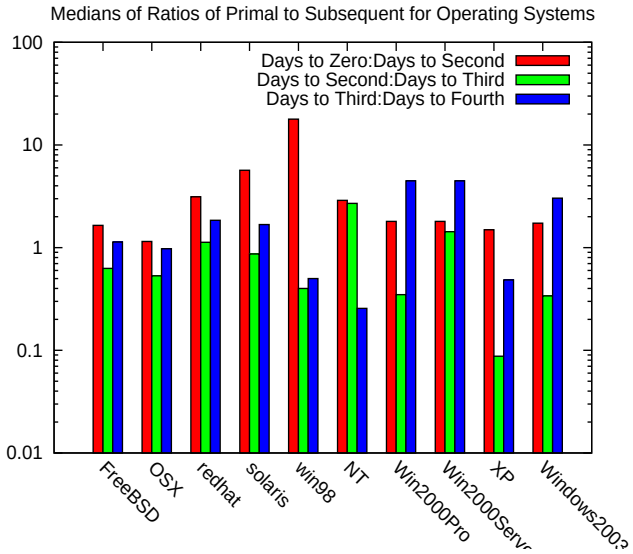


Figure 3.4: Honeymoon ratios of p_0/p_{0+1}, p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for major operating systems. (Log scale. Note that a figure over 1.0 indicates a positive honeymoon).

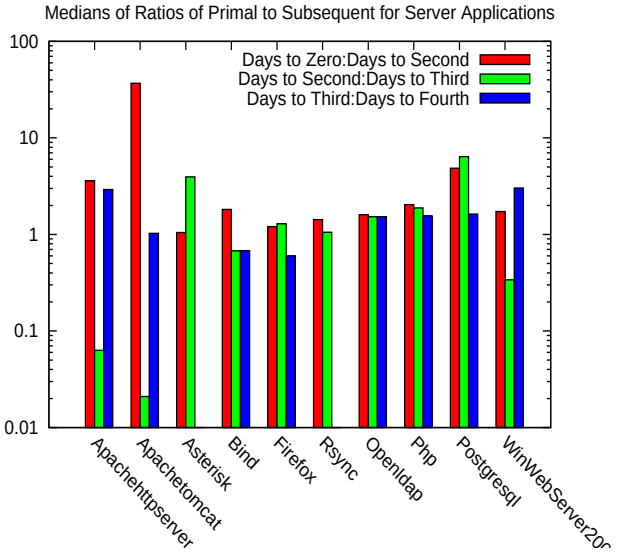


Figure 3.5: Honeymoon ratio of p_0/p_{0+1}, p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for common server applications

information can be found in Appendix B.

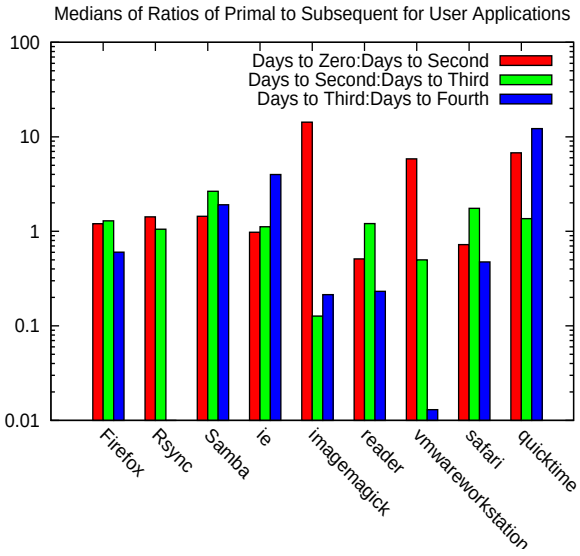


Figure 3.6: Honeymoon ratios of p_0/p_{0+1}, p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for common user applications

Table 3.2: Median Honeymoon Ratio for Open and Closed Source Code

Type	Honeymoon Days	Ratios
Open Source	115	1.23
Closed Source	98	1.69

Yet in spite of its more rapid pace of new releases, open source software releases enjoyed a significantly longer median honeymoon before the first publicly exploitable vulnerability was discovered: 115 days, vs. 98 days for closed-source releases.(see Table 3.2)

The median honeymoon ratio, however, is shorter in open-source than in closed. The median ratio for all open-source releases was 1.23, but for closed source it was 1.69. Figure 3.7 shows the median honeymoon ratios for various open-source systems, and Figure 3.8 shows the median ratios for closed-source systems.

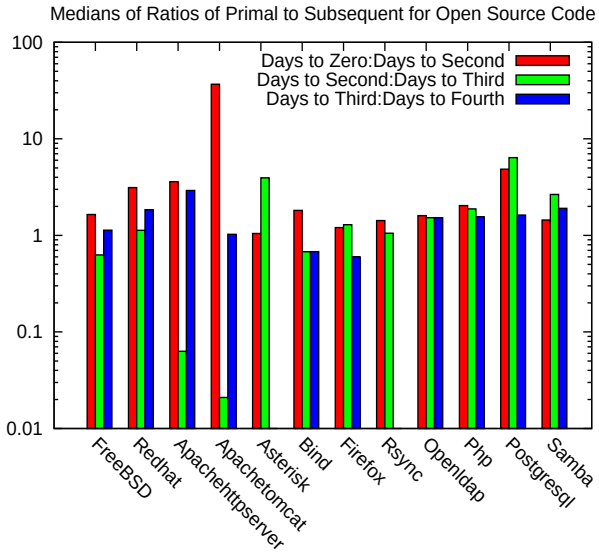


Figure 3.7: Ratios of p_0/p_{0+1} to p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for open source applications

The longer honeymoon period with a shorter honeymoon ratio for open-source software suggests that it not only takes longer for attackers to find the initial bugs in open-source software, but that the rate at which they "climb the learning curve" does not accelerate as much over time as it does in closed-source systems. This may be a surprising result, given that attackers do not have the opportunity to study the source code in closed-source systems, and suggests that familiarity with the system is related to properties *extrinsic to the system* and not simply access to source code.

3.4 The Honeymoon Effect and Foundational Vulnerabilities

To more fully understand the factors responsible for the Honeymoon Effect, I analyzed the attributes of a particular set of *foundational* vulnerabilities. I compared the duration of the Honeymoon Periods of the software in the data set and found

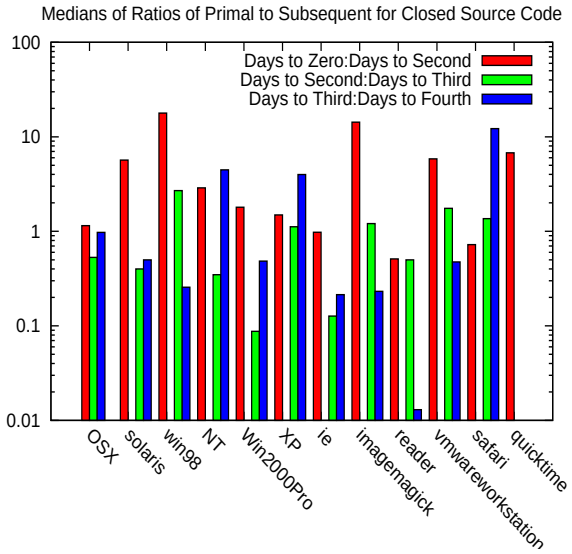


Figure 3.8: Ratios of p_0/p_{0+1} to p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for closed source applications

that foundational vulnerabilities are not a result of the first vulnerabilities being easy to find, i.e., "low-hanging fruit", and that other extrinsic property or properties must apply.

It is well known that as complex software evolves from one version to the next, new features are added, old ones deprecated and changes are made, but throughout its evolution much of the standard code base of a piece of software remains the same. One reason for this is to maintain backward compatibility, but an even more prevalent reason is that code re-use is a primary principle of software engineering [McI68, Bro95b].

As discussed in Chapter 3, in "Milk or Wine" [OS06] Ozment *et al.*, measured the portion of legacy code in several versions of OpenBSD and found that 61% of legacy (their term is 'foundational') code was still present 15 releases (and 7.5 years) later. This legacy code accounted for 62% of the total vulnerabilities found. While it is not possible to measure the amounts of legacy code from version to version in closed

source products as it is for open source, it is well known that the major vendors strongly encourage code re-use among their collaborating developers [Mic10], and more importantly, it *is* possible to measure the numbers of legacy vulnerabilities. By comparing the disclosure date of a vulnerability with the release dates and product version affected, it is possible to determine which vulnerabilities discovered in the current release result from earlier versions. For example, if a vulnerability V affects versions (k,...N) (0<k<N) of a product, but not versions (1,...,k-1) and was disclosed *after* the release date of version N, we know that the vulnerability was introduced into the product with version k, and that it stayed hidden until its discovery after the release of version N. These *regressive* vulnerabilities are those vulnerabilities which are not found through normal regression testing and may lie dormant through more than one version release (sometimes for years).[1] Remember, a *regressive* vulnerability is defined as a foundational vulnerability that was discovered to affect not only version N in which it was found, but also affect one or more earlier versions (versions N-1, N-2,...,1.0)

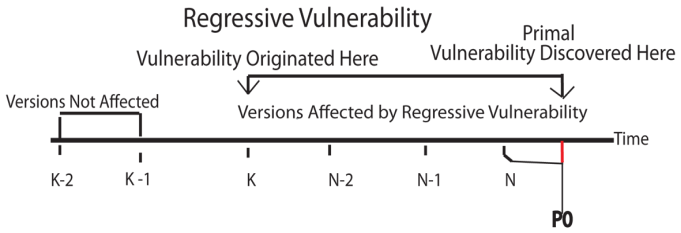


Figure 3.9: Regressive Vulnerability timeline

On the other hand, a *progressive* vulnerability is defined as a foundational vulnerability which is discovered in version N and does not affect version N-1 or any earlier versions. A progressive vulnerability indicates that the vulnerability was introduced with the new version N. (see Figure 3.9)

[1] In OpenBSD, Ozment *et al* states "It took more than two and a half years for the first half of these ... vulnerabilities to be reported." [OS06].

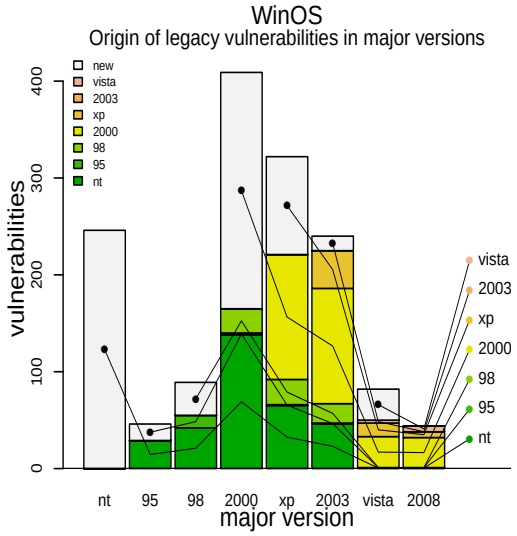


Figure 3.10: Proportion of legacy vulnerabilities in Windows OS

Figure 3.10 shows that legacy vulnerabilities[2] make up a significant percentage of vulnerabilities across all products, e.g. 61% of the Windows Vista vulnerabilities originate in earlier versions of the OS, 40% of which originate in Windows 2000 released seven years earlier. This analysis shows that vulnerabilities are typically long-lived and can survive over many years and many product versions until discovered.

3.4.1 Regressive Vulnerabilities

If factors such as code reuse or an attacker's familiarity with the system has an effect on the rate of vulnerability discovery, then when upon analysis of the *foundational* vulnerabilities, one would expect to see that regressive vulnerabilities make up a significant percentage of them. And indeed, after examining all the foundational vulnerabilities in the data set, I found that 77% of them are regressive. (ie, 77% of the foundational vulnerabilities were found to also affect earlier versions). Table 3.3

[2]including both regressives and progressives

lists the percentages of regressives for all, open source, closed source foundational vulnerabilities. Table 3.3 also shows that the percentage of regressive vulnerabilities is even higher for the foundational vulnerabilities found in open source software (rising up to 83%), and lower for those found in closed source software (59%). The high percentage of regressive vulnerabilities is surprising, because it shows that the majority of foundational vulnerabilities, (the first vulnerability found after a product is released), are not the easy to find "low-hanging fruit" one would expect from conventional software engineering defects, instead these regressive vulnerabilities lay dormant throughout the lifetime of their originating release (and possibly several subsequent releases). If these vulnerabilities had been easy to find, then presumably, *they would have been found in the version in which they originated.*

Table 3.3: Percentages of Regressives and Regressive Honeymoons for all Foundational Vulnerabilities

Type	Total Regressives	Total Regr. Honeymoons
ALL	77%	63.4%
Open Source	83%	62%
Closed Source	59%	66%

3.4.2 The Honeymoon Effect and Regressive Vulnerabilities

Another unexpected finding is that regressive vulnerabilities also experience the Honeymoon Effect. Because regressive vulnerabilities have been lying dormant in the code for more than one release, and because the attackers have had more time to familiarize themselves with the product, it seems reasonable to presume that the first of these vulnerabilities would be found in a shorter amount of time than time to find the second vulnerability (whether regressive or progressive). But, our analysis shows this isn't the case. The second column of Table 3.3 lists the percentages of regressive vulnerabilities that were also positive Honeymoons. In each case whether

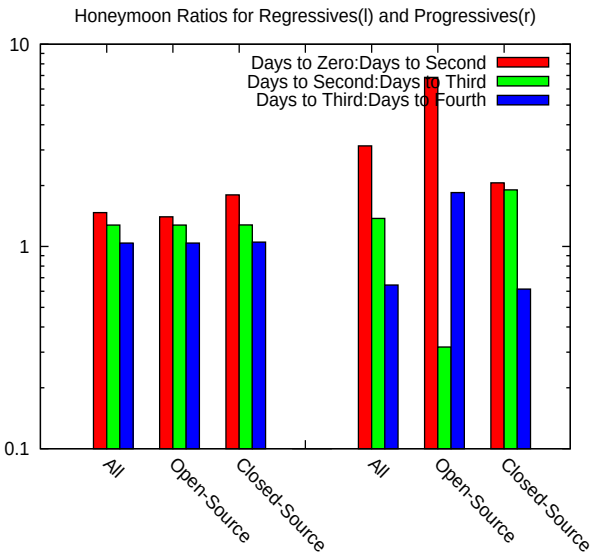


Figure 3.11: Honeymoon Ratios of p_0/p_{0+1}, p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for common user applications

my analysis looked at all regressive vulnerabilities combined, at only open source regressive vulnerabilities or those only in closed source software, the percentages of positive Honeymoons is in the low to mid 60th percentile - almost the same as the total Honeymoon Effect for all regressive and progressive vulnerabilities combined. Closed source software does exhibit a slightly longer Honeymoon Effect, but not significantly so. The existence of regressive positive Honeymoons, especially in such high proportions indicates that properties extrinsic to the quality of the code, in particular an attacker's familiarity with the system may play a much greater role early on in the life-cycle of a release than previously expected.

3.4.3 Regressives Vulnerabilities Experience Shorter Honeymoons

The strong presence of the Honeymoon Effect even among regressive vulnerabilities led me to wonder what if any effect regressives might have on the length of the Honeymoon Period. Yes, regressive vulnerabilities experience a positive Honeymoon, but is the time interval for a regressive Honeymoon longer or shorter than the honeymoon for progressive vulnerabilities? The Honeymoon Ratio provides insight into the length of the Honeymoon Period. Figure 3.11 shows the median Honeymoon Ratios for regressives (all, open and closed), progressives (all, open and closed), for the vulnerabilities p_0/p_{0+1}, through p_{0+2}/p_{0+3}. The median Honeymoon Ratio for regressive vulnerabilities is lower than that for progressives. In fact, the Honeymoon Ratio for progressive vulnerabilities is almost twice as long. This strongly suggests that familiarity with the system is a major contributor to the time to first vulnerability discovery. Interestingly, it doesn't seem to have a significant effect on open source code, but closed source does seem to have a longer Honeymoon Period, even for regressives. In other words, these results suggest that *familiarity shortens the honeymoon.*

3.4.4 Less than Zero Days

Table 3.4: Percentages of foundational vulnerabilities that are Less-than-Zero (released vulnerable to an already existing exploit) and the new expected median time to first exploit, for all products, Open source and Closed Source

Type	Percentages	Median Honeymoon Period
ALL	21%	83
Open Source	18%	89
Closed Source	34%	60

Dormant vulnerabilities are not the only cause of zero-days. Legacy vulnerabilities result in a second category of regressive vulnerabilities for which there can be no Honeymoon Period. These *Less-than-Zero* days occur when a new version of a product is released vulnerable to a previously disclosed vulnerability. For example, the day Windows 7 was officially released, it was discovered that it was vulnerable to several current prominent viruses, in the form of widely circulated malware which had originally been crafted for Windows XP. [Wis09] My research shows that less-than-zero days account for approximately 21% of the total legacy vulnerabilities found, with closed source code containing the most (34%)(see Table 3.4). In all cases the median number of days to first exploit is reduced by approximately 1/3 and the median Honeymoon Ratio drops from 1.54 to 1.0. This leads one to the obvious conclusion that not patching known vulnerabilities has a significant negative effect on the Honeymoon Period. Of course there is no way to measure exactly when an attacker is likely to test an existing exploit against a newly released product however, the Sophos Labs report is indicative of how quickly a vendor might expect attackers to act.

3.5 Discussion and Analysis

The software lifecycle has been repeatedly examined, with the intent of understanding the dynamics of software production processes, most particularly the arrival rate of software faults and failures. These rates decrease with time as updates gradually repair the errors as they are found, until an acceptable error rate is achieved. There is an interesting dynamic at work in finding and patching software defects versus finding and patching software vulnerabilities. With non-security bugs, there is little or no learn time required to find them. The software simply doesn't work as it is expected to. With regression testing, automated defect discovery and patch generation the developers tend to be intimately involved with finding

as well as fixing defects, while in the case of vulnerability discovery, the developers are not the same people finding vulnerabilities, crafting exploits and attacking their code. [WFLGN10, SLP+09, Sch09, OCJ09, SIK+13]

The software vulnerability lifecycle has been less extensively studied, with most attention paid to the period after an exploit has been discovered. In attempting to understand the properties of vulnerability discovery, there are two approaches I might have taken. One approach would have been to study a single software system in depth, over an extended period, draw detailed conclusions, and perhaps generalize from them. Indeed, several of the related works mentioned in Chapter 2 try to do just that for the middle and end phases of the lifecycle. But, another approach is to examine a large set of software systems and try to find properties that are true over the entire set and over an extended period.

I chose the latter approach for an number of reasons, which include the following: This approach allowed me to incorporate both open and closed source systems in my analysis, this approach also allowed me to analyze several different classes of software (Operating Systems, Web Browsers User applications, Server applications, etc), and this approach allowed me to discover general vulnerability properties, e.g. the Honeymoon Period, independent of the type of software, and without requiring a detailed analysis of the properties of each specific, individual vulnerability.

It might appear that given so many changes in tools, utilities, methodologies and goals used by both attackers and defenders over the last decade, a long term analysis would be inconsistent. To mitigate this each analysis was broken down by year and from version to version. These are much shorter time intervals, and the results have demonstrated the consistency of this approach over time.

I also analyzed the role of legacy code in vulnerability discovery and found surprisingly, based on a detailed study of a large database of software vulnerabilities, that software reuse may be a significant source of new vulnerabilities. I determined that the standard practice of reusing code offers unexpected security challenges. The

very fact that this software is mature means that there has been ample opportunity to study it in sufficient detail to turn vulnerabilities into exploits.

There are multiple potential causal mechanisms that might explain the existence of the Honeymoon Effect and the role played by familiarity. One possibility is that a second vulnerability might be of similar type to the first, so that finding it is facilitated by knowledge derived from finding the first one. A second possibility is that the methodology or tools developed to find the first vulnerability lowers the effort required to find a subsequent ones. A third possible cause might be that a discovered vulnerability would signal weakness to other attackers (ie, blood in the water), causing them to focus more attention on that area. [CBS10]

The first two possible causes require familiarity with the system, while the third is an example of properties *extrinsic* to the quality of the source code that might affect the length of the Honeymoon period.

The dynamics of the Honeymoon Effect suggest an interesting tradeoff between decreasing error rate necessary for software reliability and increasing familiarity with the software by attackers. This basic result has important implications for the arms race between defenders and attackers.

First, it suggests that a new release of a software system can enjoy a substantial *Honeymoon Period* without discovered vulnerabilities once it is stable, *independent of security practices*. Second, this Honeymoon Period appears to be a strong predictor of the approximate upper bound of the vulnerability arrival rate. Third, it suggests that attacker familiarity is a key element of the software process dynamics, and this is a contraindication for software reuse, as the greater the fraction of software reuse, the smaller the amount of study required by an attacker. Fourth, it suggests the need for new approaches to securing software systems than simply trying to create defect-free code.

In particular, research into alternative architectures or execution models which focuses on properties extrinsic to software, such as automated diversity, redundant

execution, software design diversity might be used to extend the Honeymoon period of newly released software, or even give old software a *second honeymoon*. [CEF+06, WFLGN10]

Chapter 4

Exploring The Honeymoon Effect in Different Development Methodologies

"In art there are only fast or slow developments. Essentially it is a matter of evolution, not revolution." (Bela Bartok)

4.1 Introduction

Secure Software Development Models such as those discussed in Chapter 2 have long been the recommended best practices for improving security in software. At the time these models were created the predominant methodology for designing secure software was a highly structured, top down process requiring intensive upfront resource investment, particularly in the conception, specifications and requirements stages. Today, however, the most common software development method is one which spends minimal time in pre-coding stages. The specifications and requirements evolve as the software is being written with the goal of delivering new features to the customer as quickly as possible. In this Chapter, I present the results of

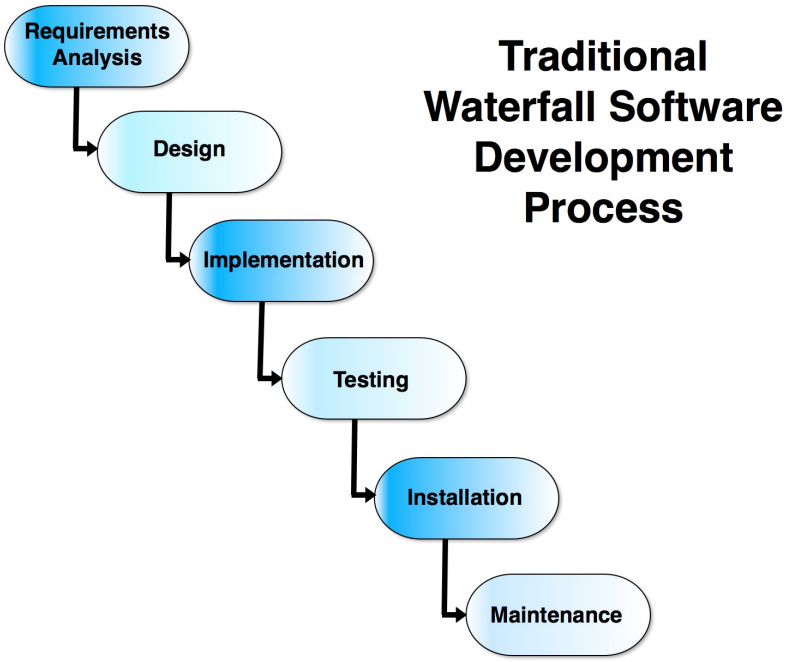
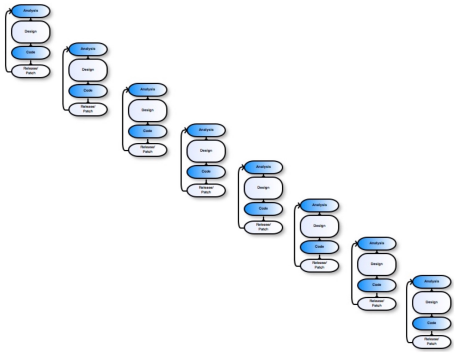


Figure 4.1: The Waterfall Software Development Model. [Hau09]

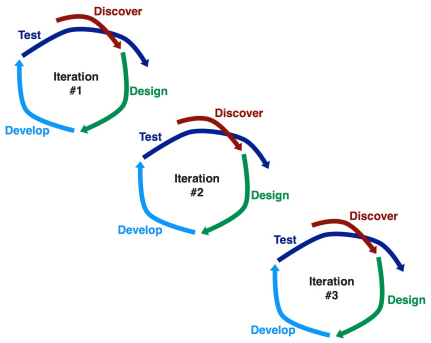
an analysis of the effect of this rapid software development methodology on the likelihood of experiencing the Honeymoon Effect compared to that same product developed under the more traditional design process.

4.1.1 Secure Software Development Best Practices and Popular Development Methodologies

The most common traditional software development model is known as the *Waterfall Model* (see Figure 4.1), because the process flows in one direction from initial to final phase. The recommended secure software design models, such as those discussed in Chapter 2 are based on these waterfall models. The process begins with the definition of requirements and specifications, which are expected to be strictly adhered to, continues through to exhaustive testing, ending with product release. There is even



(a) Iterative Software Development Methodology



(b) Individual Iterations in the Agile Methodology

Figure 4.2: Models of the Agile Software Development Process.

a stage for formalizing documentation. [1] This highly structured methodology is considered an essential part of the development of secure system. Software developed under a traditional SSDMs requires an extensive investment of resources upfront, particularly as the product *must* pass through a strict testing phase which must include all aspects of the specifications and requirements as well as the security infrastructure before a product can be considered ready for release. [Cor08, oHS06, Woo13] For products developed under this process, both the development lifecycle and the post-release lifetime tend to be quite long (often lasting years).

Over the last decade however, the approach of the software development community has changed radically. Overwhelmingly, developers large and small, including such companies as Apple, Facebook, Google, Microsoft, Firefox, and Amazon, have moved away from traditional Waterfall models to rapid release cycles and Agile methodologies. The Waterfall model could be thought of as a monolithic approach to software development, while the Agile approach could be considered iterative.

Such Agile development processes are not sequential, but cyclic, (see Figure 4.2). The requirements evolve in conjunction with feature development, readying code for

[1] In fact, for mission critical software it has been noted that "within some Traditional Methods, writing documentation is considered paramount to the quality, maintainability, reliability, and safety of mission critical systems such as aviation electronics." [Ric08]

release, incorporating customer feedback and integrating with previous iterations. Rapid Release Cycle (RRC) development models have no formalized set of initial requirements, and the only requirement that must be strictly adhered to are release deadlines. These models also promote a much shorter development lifecycle (usually 6 weeks) as well as a much shorter effective lifetime.

The focus of Agile methods on customer collaboration, feature implementation and rapid delivery of working software is intended to create timely business value, not to meet initial design requirements or provide long term maintainability and reliability. These goals often conflict with the recommended security best practices. [Cor09] As stated in Chapter 2, the DHS found six of Agile programming's core principles to have *"negative implications"* [oHS06] for security. Moreover, of the fifteen security touch points listed as necessary in Microsoft's SDL, a survey of Agile developers found seven of them to be *"Detrimental"* or *"Very Detrimental"* to their development process.

4.1.2 Evaluating the Early Vulnerability Lifecycle of Agile Programming Models

The Honeymoon Effect described in the previous chapter, was discovered while analyzing the early vulnerability lifecycle of software that was largely developed before the wide-scale adoption of Agile methods, i.e., software that was developed using the traditional method designed to be compatible with the recommended security best practices.

It is worthwhile exploring whether this phenomenon is also present in software developed using a process that finds certain of those recommended security activities "detrimental". Is the Honeymoon Effect a product of the software development methodology?

To test this, an experiment would need to compare vulnerability discovery rates in the early lifecycle of both programming methods. One way might be to assign

a software project to two separate groups of programmers (perhaps as a classroom exercise). One group would complete the assignment using the traditional SSDL methodology, and the other would complete it using an Agile methodology.

Unfortunately, there are a number of issues with this scenario that make it unsuitable for studying the early vulnerability lifecycle. As, the goal of this experiment is to compare the likelihood of software vulnerability discovery in the period immediately following a release, across several releases for both development strategies. To justify generalizing the results to real world software, the complexity of the assigned project would need to be sufficiently large and several versions would need to be developed. The scale and timeframe necessary to accomplish this is unrealistic for a classroom exercise.

Another possibility would be to compare the early vulnerability discovery lifecycle of two publicly available commercial software products, where one is developed using the traditional model and the other using the Agile model. Again, such an analysis is less than ideal. To justify comparison, the products would need to fulfill the same purpose, have similar usage, and similar features including such properties as market-share and attacker interest, yet come from completely different development models. This could prove very difficult to find, as most companies in competitive markets tend to use the same development methodologies as their competitors. For example, between 2010 and 2011, all of the main web-browser companies had committed to some form of Agile development. [Sad13, Cor09, Bak]

Comparing two different types of software products such as a web-browser to a word processor would introduce too many variables to be certain that any differences found were the result of the development process.

I chose a third possibility, examining the effects of both development models within a single specific software product. While this approach is also limited, in that it only looks at one software product, it has the advantage of narrowing the experiment to the analysis of a single variable; the software development methodology.

For this analysis, I chose Mozilla's Firefox web-browser.

4.1.3 Why Firefox?

Desiderata for a system to study include:

1. Open source
2. A frequent target of attack
3. A broad user base, and
4. A statistically significant population of publicly disclosed vulnerabilities.

The Firefox web browser proved to be an ideal system for analysis, for four primary reasons. First, and most important, Firefox was originally designed using a traditional development model. Released in November of 2004, it proved to be extremely popular, with over 100 million downloads in less than a year. Mozilla released a new version approximately once a year. Each new version included significant design changes, the addition of numerous new features and major bug fixes. [Moz04]

Then, in June of 2011, Mozilla Firefox underwent a significant change in its development process. In a post by Mozilla Chief Mitchell Baker arguing for the change, [Bak] Baker said "If we want the browser to be the interface for the Internet, we need to make it more like the Internet," Baker wrote. "That means delivering capabilities when they are ready. That means a rapid release process." and further added: "Before Mozilla instituted the rapid release process, we would sometimes have new capabilities ready for nearly a year before we could deliver them to people. Web developers would have to wait that year to be able to make their applications better... Philosophically, I do not believe a product that moves at the speed of traditional desktop software can be effective at enabling an Internet where things happen in real time." [Bak]

Mozilla's developers acknowledged the switch to RRC "*involved changing a number of our processes. It's also raised some new issues.*". [Bak] The midstream introduction of RRC provides the basis for a *sui generis* analysis of the effect of changing a single variable. That is, a "before and after" comparison of security properties in light of a *significant change in software development practices.* Because of the need to support large organizations, such as corporations and governments Mozilla also maintains a second non-Agile Firefox development track for its Extended Support Release (ESR) version which continued to be developed and maintained according to the traditional model. So, Firefox has documented history using *both* development models *at the same time.* The concurrent release processes for RRC and ESR (discussed below and displayed in Tables 4.1 and 4.2), effectively tab provide two versions of the same software differing only by a *single* variable, the release cycle. Thus, this dual-track Firefox release strategy provides a unique analytic framework for a data-driven examination of RRC methodologies. Moreover, Mozilla because syncs the two platforms approximately once a year, so code developed in RRC could adversely affect the ESR versions as well. This created the opportunity to also study the effects of code reuse.

Second, since its initial release, all Firefox source code has been open source and freely available. Pre-RRC source code is available in a CVS repository. To prepare for the switch to RRC, Mozilla moved what was then the current source 3.6.2 and 4.0a to be the foundation of the first RRC version (5.0) into a new Mercurial repository. Since then all changes for each subsequent new RRC version have been added to this repository. This analysis, used Firefox version 4.0 as the 'baseline' version of all subsequent RRC versions, and covered versions 5-20. Version 20 had just been released when the data collection was complete and the analysis begun; version 23 was available by the time the analysis was complete.

Third, Firefox has a well maintained and freely available bug database, Bugzilla [Moz13a], containing detailed information on all bugs, including patched vulnerabilities. Mozilla

does not openly list the details of the most recent, unpatched security vulnerabilities in Bugzilla, but they do publish timely and somewhat detailed references to the latest security bugs on the Mozilla Foundation Security Advisory (MFSA) site [Moz13b] and the relevant details are made public in Bugzilla sometime thereafter.

It is important to note that all acknowledged bugs (defects and vulnerabilities) reported in Firefox are given a Bug ID before being assigned to be patched, so all known vulnerabilities are associated with some Bugzilla Bug ID. In addition, the MFSAs are linked to relevant references in the NIST National Vulnerability Database (NVD) [NIS08] which contains an entry for each known vulnerability, including versions affected, criticality and date released. For this dissertation, all Firefox vulnerabilities from the NVD database were collected and each vulnerability disclosed was cross-referenced with its corresponding MFSA to find each Bug ID issued. There is some overlap, as a single NVD Common Vulnerability Enumeration (CVE) entry may contain several Firefox Bugzilla Bug IDs, and a single Bug ID may link to multiple NVD (CVE) entries. [CVE08]

Fourth, Firefox is a frequent target of attackers. As such, Mozilla recognized very early on the benefits of the "many eyes" approach to vulnerability discovery, and in 2004 Mozilla started the first 'Bug Bounty' program. Mozilla has a long history of purchasing vulnerability information from researchers and rewarding those who find and report vulnerability. Recent research [FAW13] on the efficacy of Bug Bounty programs suggests that 25% of Firefox's vulnerabilities are discovered through its bug bounty program. Mozilla does not announce each purchase, but Coates [Coa11] showed that, on average, Mozilla purchases six new vulnerabilities per month. While one must recognize that it is impossible to know anything about the number of private or undisclosed vulnerabilities that may have been discovered in Firefox, and that this is a potential source of error, this comparison found the number of Bug Bounty purchases consistent with the dataset compiled from MFSAs, Bugzilla, and the CVE database. Also, while it may be argued that some RRC code might lack critical

functionality, the data shows that RRC code does contain critical vulnerabilities, and as Mozilla synchronizes the two Firefox development tracks annually, new features added to the RRC versions become part of the next ESR version. Thus RRC does modify some of the core code base. Further, this data set was cross-referenced with the MFSA [Moz13b] security advisories, vulnerabilities it contains references for those vulnerabilities which Mozilla considers important enough to issue a public advisory. Using this standard as a measurement for severity avoids any risk of bias in the results due to a bespoke metric for severity.

Mozilla's change in Firefox software development and release strategy raised three security research questions that are addressed in detail in Section 4.3:

1. Does a switch to Agile RRC development introduce large numbers of new vulnerabilities into software, given that the focus is on new features, rather than on improving existing code?
2. Where in the code base are vulnerabilities being discovered? (*i.e.*, are they in code written prior to the switch to RRC, are they in code introduced in previous iterations of RRC or are they in code added in the current version?)
3. Are vulnerabilities being discovered more quickly since the switch to RRC?

This investigation gave some surprising results:

1. Quantitative evidence that:
    - The rate of vulnerability disclosure has not increased substantially since the start of Firefox RRC
    - The overwhelming majority of vulnerabilities discovered and disclosed are *not* in the new code
    - Vulnerabilities originating in Firefox RRC versions are almost all not disclosed until that version has been obsoleted by newer versions
    - Firefox RRC does *not* appear to produce demonstrably more vulnerable software

2. A data-inspired observation that frequent releases of high volumes of new code, due its relative unfamiliarity to attackers, may provide some protection for frequently targeted software; and
3. Further supporting evidence for an exploit-free "honeymoon" or "grace period"[CFBS10] provided by the attacker's learning curve.

4.2 Methodology

4.2.1 Assumptions

When analyzing the security of software over its lifecycle, there are three assumptions, researchers commonly make:

- First, that with each addition of new code, a number of new software defects are also added;
- Second, that (to the extent that security vulnerabilities are a consequence of software defects), that new vulnerabilities are also introduced and will be discovered and disclosed; and
- Third, that attackers are analyzing code bases searching for weaknesses in both old and new code.

4.2.2 Vulnerability Taxonomy

For the purposes of this analysis it is necessary to differentiate among three types of vulnerabilities:

1. *Foundational* vulnerabilities: Vulnerabilities that affect the original codebase on which RRC was based.

2. *Regressive* vulnerabilities: Defined in Chapter 3,

3. *New* vulnerabilities: Vulnerabilities that affect the current version of code at the time of disclosure but that do not affect previous versions.

There are also two different *states* of vulnerabilities:

1. *Active* vulnerabilities: Vulnerabilities that affect a given version of software while that version is the most current available,

2. *Inactive* vulnerabilities: Vulnerabilities which once the most recent version which it affects has been obsoleted by a more recent version, are no longer exploitable.

For example, a regressive vulnerability disclosed in version 20, but introduced in version 18, is said to be *active* until it is patched in either version 20 or some later version.

Finally, there are *Unknown* vulnerabilities: Vulnerabilities in a given version of software that have not yet been publicly found or disclosed.

4.2.3 Firefox RRC

Mozilla began the Firefox RRC development process in June of 2011 with the release of version 5.0 as the first rapid release version. The Firefox RRC is structured such that new code actually goes through three 6-week phases before being released. The code spends 6 weeks in development, 6 weeks being stabilized in what is referred to as the *Aurora* phase and 6 weeks being beta-tested. The code is freely available at any of these phases. Thus, at the time of release of version n, versions n+1 through n+3 are in the Beta, Aurora and development phases, respectively. This schedule allows Mozilla to release a new version regularly every 6 weeks. Prior to the inception of RRC, a version of Firefox would spend as long as a year in an alpha *pre-beta* phase, and a further year in *beta*, undergoing several revisions. Meanwhile,

the current release would be patched as needed. Between major version releases, Mozilla introduces point releases only if a critical vulnerability has been found to affect it. In practice, there are only one or two of these between each version.

At the start of RRC, the then current stable version of Firefox code, which had been developed using the traditional Waterfall methodology, (version 3.6.2) was cloned to become the base of the new RRC code. That same code became the first Extended Support Release (ESR). The ESR software is intended for mass deployment in organizations. ESR releases are maintained for a full year or longer, with point releases containing security updates coinciding with new RRC Firefox releases. A new ESR version is released (essentially) by rolling the features of the current RRC version into the new ESR version. At the time this analysis was performed, Mozilla had done this twice: for versions 10 and 17, both of which were released at the same time as the corresponding RRC versions. This analysis looked at RRC versions 5 through 20, but compared RRC to ESR, it was important to be careful to compare only concurrent versions: RRC versions 10.0-20.0 to ESR versions 10.0-10.0.12 and 17.0-17.0.6 (See Table 1).

4.2.4 Data collection

This analysis was concerned specifically with vulnerabilities disclosed in software developed and released under a 6-week Rapid Release Cycle (RRC).

From the inception of RRC up to the time of this writing, 617 new Bug IDs were issued, corresponding to new vulnerabilities reported in the MFSAs [Moz13b] and CVE [CVE08] database, providing sufficient volume of data for empirical study.

Line of code (LOC) and file counts in this dissertation are derived from the Mercurial repositories hosted by Mozilla and are filtered to account for a subset of file types that account for almost all of the code relevant to this dissertation [Alm13]. Specifically, I included files with the extensions: `.c`, `.C`, `.cc`, `.cpp`, `.css`, `.cxx`, `.h`, `.H`, `.hpp`, `.htm`, `.html`, `.hxx`, `.inl`, `.js`, `.jsm`, `.py`, `.s`, and `.xml`; test cases and

harness code have been excluded, as well as code comments and whitespace. LOC counting for this dissertation is conservative and may understate changes to the Firefox codebase between versions.

For this dissertation I also examined Firefox's Extended Support Releases (ESR). These long-term support releases still follow essentially the traditional release-and-patch model that preceded the transition to RRC, with the same code base as RRC versions 10 and 17 covered by in this analysis; the next ESR release was version 24. ESR is an effective point of comparison when examining the impact of RRC on security.

4.2.5 Limitations

As noted earlier, unknown vulnerabilities exist, and this makes the date that any given vulnerability was initially discovered hard to obtain. One can only know with certainty when the vulnerability was first *reported*. For the purposes of this analysis I used the disclosure date as an approximation for the discovery date. The disclosure date, while later than the discovery date, is workable for these purposes, since the analysis is concerned with large-scale phenomena and inter-arrival times for vulnerability discoveries.

Notably, as Firefox is a frequent attack target and Mozilla responds quickly, by issuing inter-cycle point releases for critical and severe vulnerabilities, this error is as small as it can be without omniscience of undisclosed vulnerabilities attackers might have "on the shelf".

4.3 Security Properties of RRC

4.3.1 Code Bases: RRC versus ESR

Mozilla's RRC reflects the principles of Agile programming. For example, Nightingale states:

> "*Rapid release advances our mission in important ways. We get features and improvements to users faster. We get new APIs and standards out to web developers faster.*" [Nig11]

While security patches are also included in each new release, the focus of the program is to deliver new features, not patch vulnerabilities. In contrast, ESR versions are intended to remain stable and unchanged after release, except for required security patches:

> "*Maintenance of each ESR, through point releases, is limited to high-risk/high-impact security vulnerabilities and in rare cases may also include off-schedule releases that address live security vulnerabilities.*" [Fou14]

Table 4.1 lists the release dates of the RRC versions. Table 4.2 lists the corresponding ESR point releases. While both RRC and ESR started from the same codebase, they soon differ substantially.

New features and new APIs mean new code, which can affect functionality and maintainability, as well as security. How many lines of new code are pushed out in Firefox's RRC? In Table 4.1 are listed the number of lines of code added, and removed, the total numbers of files changed between versions, and the total number of LOC per version since RRC was instituted. Since the start of RRC, Firefox has added a minimum of 100k LOC per version, and averages 290k LOC added, 160k

LOC removed, and 3,475 files changed per version. There is a wide variance, but the median LOC added is 249k. This amounts to an average of 10% of the code-base changing in some way every 42 days.

These changes are not isolated, but rather appear to have wide-reaching effects. In a study to determine the maintainability of the Firefox codebase since RRC, Almossawi [Alm13], found that 12% of files in Firefox are highly interconnected. Almossawi also found that making any change to a randomly selected file can, on average, directly impact eight files and indirectly impact over 1,500 files. This means that on average each new RRC version could potentially impact as many as 30,000 files.

The difference between this and the ESR versions is substantial. Only two of the point releases add more than 10k LOC and only changes to version 17.0.5 reach anywhere near the average of the RRC versions (see Tables 4.1 and 4.2).

Does the modification of such large amounts of new code result in a less secure product? If so, there are three things we would expect to see:

1. An *increase* in the number of vulnerabilities affecting each new release (active vulnerabilities);
2. The *scope* of vulnerabilities should change. That is, the vulnerabilities discovered should be primarily new and should affect only the current (and possibly subsequent) versions; and
3. The regular introduction of such code should *increase* the rate of vulnerability discovery and disclosure.

In other words, if the current models for general software defects apply here, the new code should be more vulnerable than the old code.

Table 4.1: RRC changes from the previous version

RRC						
Version	Release Date	LOC Added	LOC Removed	LOC Δ	Total LOC	Files Δ
4	-	157.4k	710k	230k	362.1k	5300
5	-	164k	161k	325k	362.4k	1700
6	-	142k	164k	306k	360.6k	2100
7	-	124k	120k	243k	361.0k	2000
8	-	109k	90k	199k	363k	1700
9	-	159k	90k	250k	368.7k	2100
10	1/31/12	491k	282k	773k	386k	4000
10.0.1	2/10/12	-	-	-	-	-
10.0.2	2/16/12	-	-	-	-	-
11	3/13/12	254k	203k	457k	390.2k	2000
12	4/24/12	245k	190k	436k	395k	2500
13	6/5/12	133k	85k	218k	399.1k	2300
13.0.1	6/15/12	-	-	-	-	-
14	7/17/12	265k	88k	354k	414.6k	2200
14.0.1	7/17/12	-	-	-	-	-
15	8/28/12	383k	280k	664k	422.6k	9000
15.0.1	9/6/12	-	-	-	-	-
16	10/9/12	608k	85k	693k	467.5k	2800
16.0.1	10/11/12	-	-	-	-	-
16.0.2	10/26/12	-	-	-	-	-
17	11/20/12	271k	177k	448k	475.3k	5400
17.0.1	11/30/12	-	-	-	-	-
18	1/8/13	820k	385k	120.4k	512k	7700
18.0.1	1/18/13	-	-	-	-	-
18.0.2	2/5/13	-	-	-	-	-
19	2/19/13	193k	146k	339k	515.6k	3700
19.0.1	2/27/13	-	-	-	-	-
19.0.2	3/7/13	-	-	-	-	-
20	4/2/13	252k	163k	415k	523.2k	2700

Table 4.2: Total LOC changes per version

ESR					
Version	Release Date	LOC Added	LOC Removed	LOC Δ	Total LOC
4	-	-	-	-	-
5	-	-	-	-	-
6	-	-	-	-	-
7	-	-	-	-	-
8	-	-	-	-	-
9	-	-	-	-	-
10	1/31/12	-	-	-	386k
10.0.1	2/10/12	29	7	36	386k
10.0.2	2/16/12	7	3	10	386k
10.0.3	3/13/12	2,510	1,782	4,292	386k
10.0.4	4/24/12	12,314	7,066	19,380	386.4k
10.0.5	6/5/12	1,070	528	1,598	386.4k
10.0.6	7/17/12	1,182	514	1,696	386.5k
10.0.7	8/28/12	605	216	821	386.5k
10.0.8	10/9/12	535	165	700	386.6k
10.0.9	10/12/12	23	10	33	386.6k
10.0.10	10/26/12	124	20	144	386.6k
10.0.11	11/20/12	1,151	316	1,467	386.7k
17.0	11/20/12	-	-	-	475.3k
17.0.1	11/30/12	126	27	153	475.4k
10.0.12	1/8/13	2,585	260	2,845	386.8k
17.0.2	1/8/13	2,092	1,076	3,168	475.4k
17.0.3	2/19/13	1,204	440	1,644	475.5k
17.0.4	3/7/13	4	4	8	475.5k
17.0.5	4/2/13	67,142	61,198	128,340	475.7k

4.3.2 Rapid Release and Software Quality

In this section, I address the three questions on software quality raised in Section 4.1.

4.3.2.1 Does the addition of 250K+ lines of code every 42 days markedly increase the number of vulnerabilities discovered and disclosed?

Almossawi's research [Alm13] indicated that the defect density remains constant for releases 5-9 and then rises by a factor of two in release 12. This finding is consistent with the current defect discovery models: the new code does indeed result in more defects, but not out of proportion to the LOC added. Certainly this means that the quality of the code is not getting worse. But what about vulnerabilities? Has the switch to RRC increased the vulnerability density? How does the vulnerability density compare to the defect density?

Figure 4.3 is a plot of the vulnerability density against the defect density for RRC versions 5-20. Looking at the different distributions, it can be seen that there is no noticeable or significantly out of proportion increase in the vulnerability density. Figure 4.4 is a plot of the ratio of the vulnerability and defect densities shown in Figure 4.3. Looking at it, it is clear that the proportion of vulnerabilities to defects for the different releases is well within the 1%-5% values proposed by McGraw and Anderson and confirmed by Alhazmi, *et al.*, in their analysis of the Windows and Linux operating systems developed which were developed according to traditional Waterfall methods.(See Chapter 2)

It is interesting to compare the number of vulnerabilities discovered over time before the switch to RRC to those found after. Figure 4.5 shows the number of vulnerabilities disclosed in buckets of 16 6-week periods preceding the switch to RRC and 16 6-week periods following the switch. There is no predictable pattern to be seen for values from pre-RRC to post-RRC. In both categories, the totals for

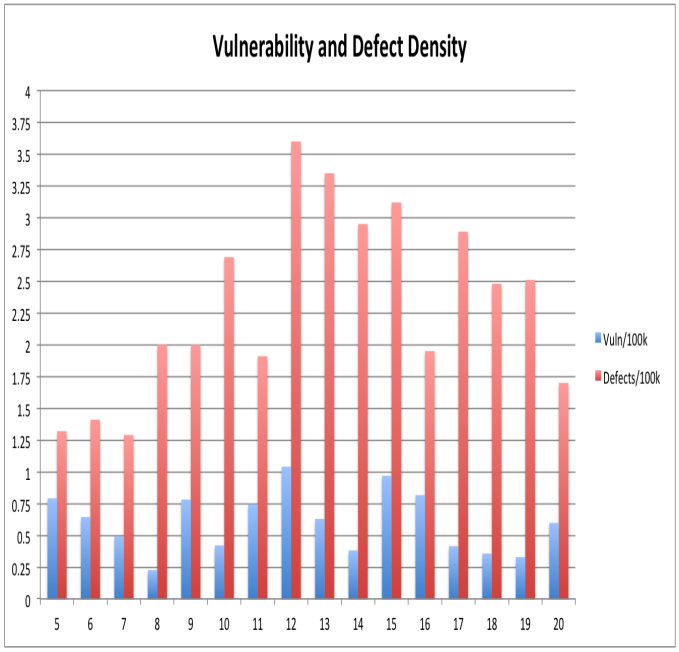


Figure 4.3: The Density of Defects and Vulnerabilities per 100,000 LOC in Firefox RRC Versions.

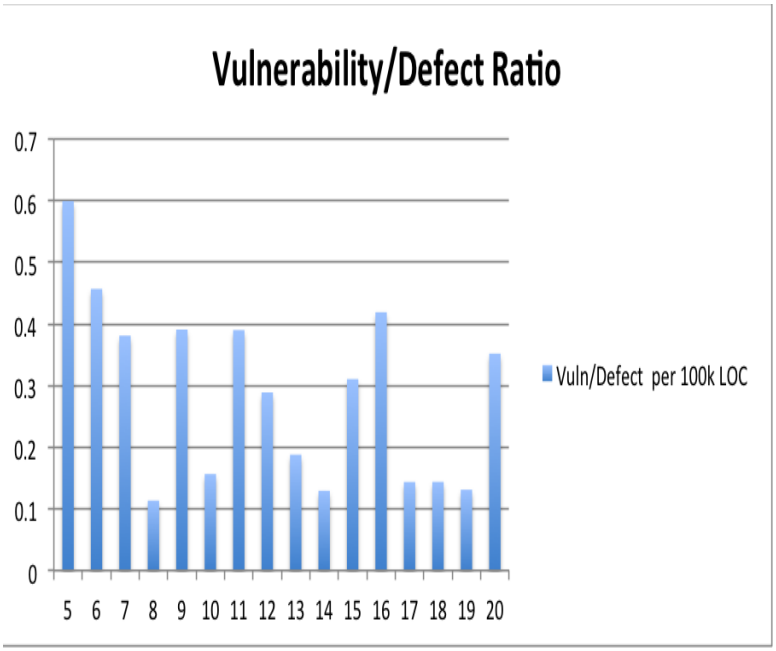


Figure 4.4: The Ratio of Vulnerabilities to Defects per 100,000 LOC in Firefox RRC Versions.

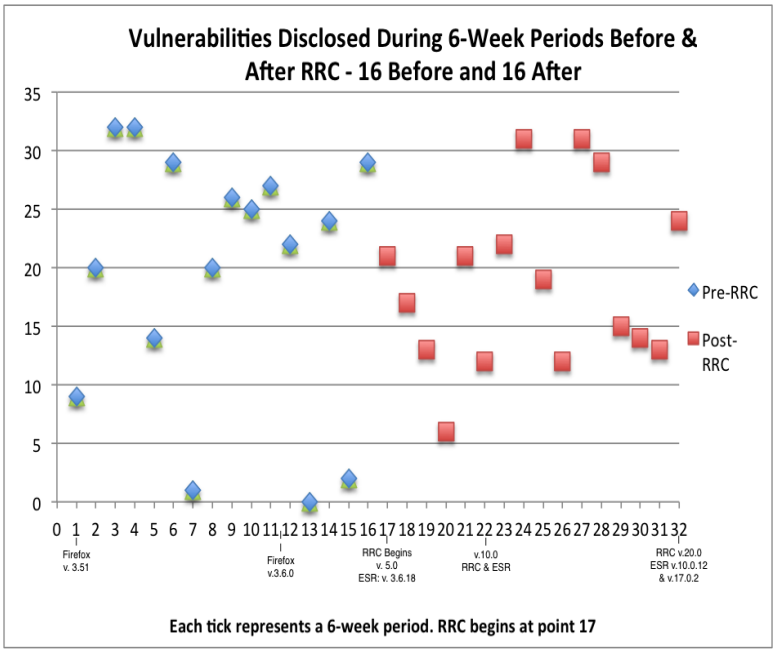


Figure 4.5: Plot of the total vulnerabilities disclosed during the 16 6-week periods preceding RRC and the 16 6-week periods following RRC.

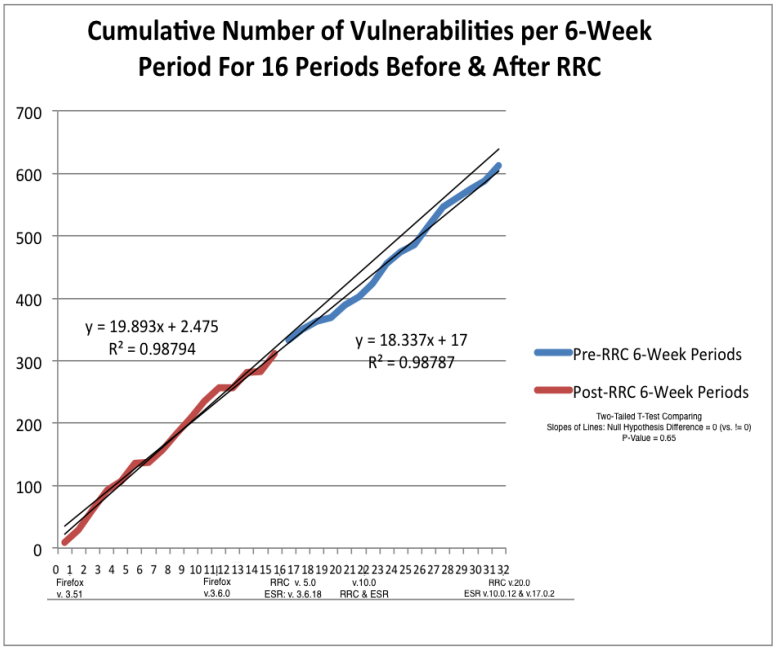


Figure 4.6: Cumulative total vulnerabilities binned into an equal number of 6-week periods preceding and following Mozilla's implementation of RRC in Firefox.

Compare Means of Slopes of Pre-RRC and Post-RRC Regression Lines.					
Descriptive Statistics					
VAR		*Sample size*	*Mean*	*Standard Deviation*	*Variance*
9		15	20.2	11.02076	121.45714
21		15	18.6	7.59511	57.68571
t-test assuming unequal variances (heteroscedastic)					
Degrees of Freedom		25			
Hypothesized Mean Difference		0.			
Pooled Variance		89.57143			
Test Statistics		0.46298			
Two-tailed distribution					
p-level		0.64738	*Critical Value (5%)*	2.05954	
One-tailed distribution					
p-level		0.32369	*Critical Value (5%)*	1.70814	
G-criterion					
Test Statistics		0.05614	*Critical Value (5%)*	0.179	
p-level		0.13301			
Pagurova criterion					
Ratio of variances parameter		0.67799			
Test Statistics		0.46298	*Critical Value (5%)*	0.06335	
p-level		0.35253			

Figure 4.7: Result of T-Test Comparing the Slopes of the Cumulative Totals of Vulnerabilities Disclosed During 16 6-Week Periods Leading Up To And After Mozilla's Implementation of RRC For Firefox

one release appear to show no correlation to the earlier releases or the later ones. Moreover, the plots show no correlation between the values for releases before and after RRC.

Figure 4.6 provides even more insight. It displays the cumulative total of the same vulnerabilities seen in Figure 4.5. A two-tailed T-test (also known as a Student's T-test) was performed to determine if the slopes of the regression lines fitted to the cumulative plots were similar. The results of the test can be found in Figure 4.7. The null hypothesis for this test was that the means of the slopes are equal. Since the test resulted in a t-value of 0.46 which is far below critical value of 2.05, one must fail to reject the null hypothesis. Therefore, one must conclude that there

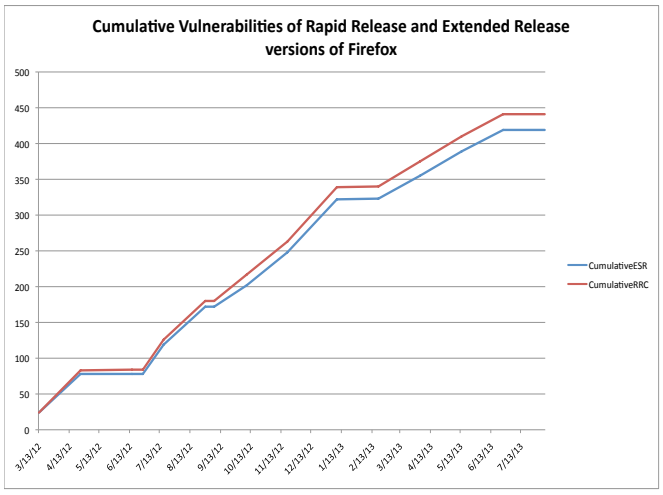


Figure 4.8: Cumulative total vulnerabilities affecting RRC and corresponding ESR versions during the same 6-week period

is not sufficient evidence to support a conclusion that the slopes of lines fitted to the pre-RRC and post-RRC cumulative plots differ significantly. Similar to what Almazawi found with regard to the software defects, there is no significant jump in the number of vulnerabilities disclosed.

Additionally, looking at the ratio of total vulnerabilities between versions (see Figure 4.9) for RRC one can see that much of the graph is nearly flat and it is only going up by less than a factor of two at its maximum. Overall, the total number of active vulnerabilities disclosed per LOC in each Firefox version since the advent of rapid release mirrors the defect discovery.

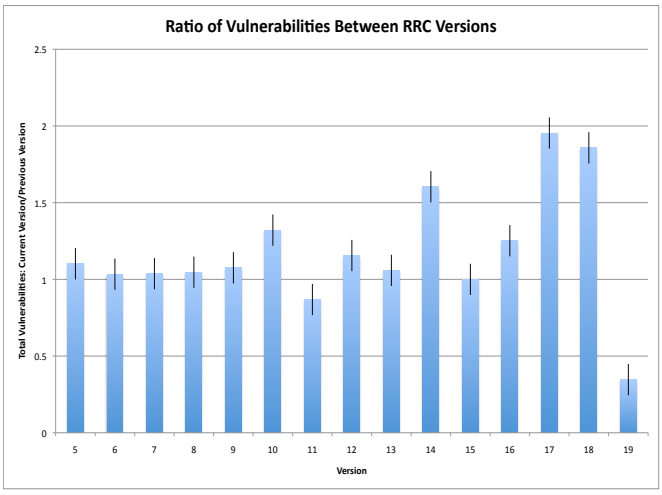


Figure 4.9: The ratio of vulnerabilities from version to version

Table 4.3: Counts of vulnerabilities by type affecting RRC (correspondence between RRC versions 10+ and ESR versions 10 and 17 is given in Table 4.4). Totals are not unique, as a single vulnerability may affect multiple versions

Version	Total	Foundational	Regressive	New
4	476	292	184	-
5	432	261	171	-
6	418	255	149	14
7	403	246	146	11
8	385	240	144	1
9	358	240	118	-
10	271	223	47	1
11	312	223	89	-
12	270	213	55	2
13	255	213	42	-
14	159	159	-	-
15	159	159	-	-
16	127	126	-	1
17	65	65	-	-
18	35	35	-	-
19	101	1	65	35
20	65	-	65	-

Table 4.4: Counts of vulnerabilities by type affecting ESR. Totals are not unique, as a single vulnerability may affect multiple versions

Version	Total	Foundational	Regressive	New
10	351	286	64	1
10.0.1	318	318	-	-
10.0.2	360	360	-	-
10.0.3	335	335	-	-
10.0.4	294	294	-	-
10.0.5	268	268	-	-
10.0.6	237	237	-	-
10.0.7	188	188	-	-
10.0.8	163	163	-	-
10.0.9	162	162	-	-
10.0.10	139	139	-	-
10.0.11	98	98	-	-
10.0.12	-	-	-	-
17	170	135	35	-
17.0.1	145	110	35	-
17.0.2	114	79	35	-
17.0.3	96	61	35	-
17.0.4	81	47	34	-
17.0.5	46	30	16	-
17.0.6	28	28	-	-

4.3.2.2 Are the RRC vulnerabilities easier to find?

With traditional defect and vulnerability discovery models, the expectation is that the 'low-hanging fruit' vulnerabilities in new code are found and patched quickly. [Ozm07, AM08]

Looking at traditional non-RRC software *et al.*, I suggested in Chapter 3 that these models do not accurately represent the early lifecycle of vulnerability disclosure. Instead, there appears to be a relatively long period before the first vulnerability in new software is disclosed, after which the rate of vulnerability disclosure in that version of code increases. I speculated that this period corresponds to the attacker's learning curve.

If new code, released without a traditionally long code review process (as is common with RRC) is bad for security, then the vulnerabilities disclosed should not only be new, but found quickly. (*i.e.*, they should be the expected low-hanging fruit.)

One would also expect to see the rate of vulnerability disclosure for vulnerabilities introduced in the new code to increase in proportion to the volume of LOC added. If, on the other hand, it takes time for an attacker to become familiar with the new code, then the vulnerabilities should take longer to find and those disclosed will be primarily from code introduced in older versions.

Table 4.3 lists the total number of vulnerabilities disclosed that affect each rapid release version. For each version, the table lists the total number of vulnerabilities that affect the foundational version, the total number that are regressive, (also affect earlier versions), and the total number newly introduced. Table 4.4 lists the corresponding data for the ESR versions.

On average, across all the RRC versions (5-20), approximately 75% of the vulnerabilities affecting each RRC version are foundational. A further 22% of them are regressive.

This means that for any version, on average, 97% of the vulnerabilities disclosed since RRC was implemented *were not found in the new code while it was the current version.*

Accounting for the fact that RRC versions 5.0-9.0 were released before the release of the first ESR version, from the release of version 5.0 up to 20.0, a total of 617 active vulnerabilities were disclosed. Only 16% of those are new vulnerabilities (ones which originate in RRC source code) *across all 15 versions studied.* Looking only at these new vulnerabilities, we see that fewer than half (41.5%) of them were disclosed during the current lifetime of the originating version. In other words, while vulnerabilities *are* found in the code that is introduced each RRC iteration, the overwhelming majority of them *are not* found during the 6-week period following their initial release. At the time this analysis was performed, even in the worst case, version 19, which had 32 vulnerabilities disclosed during its active lifetime, 70% of those were not disclosed until version 21, which was released 12 weeks later. As shown above, the number of active vulnerabilities found affecting RRC (465) and ESR (420) during the same period suggests this is not the case. The newly discovered vulnerabilities are not found in new RRC code, but in older code that has been reused from version to version and has existed at minimum, since the time of the ESR release. It is surprising to note how close the totals are, because the magnitude of code changes in RRC is so much greater than the code changes in ESR in the same time frame. Overwhelmingly, the vulnerabilities are found in the code shared between the two development platforms. This is code that is *not changing* as new RRC versions are being released.

It is important to note that, although code developed using RRC does contain its share of vulnerabilities, as Table 4.3 shows, most of the RRC vulnerabilities are either foundational or regressive, and therefore were found in code that had been available for a longer period of time than a single release cycle.

A look at the trend in the cumulative plot (see Figure 4.8) shows an S-Curve, similar to that seen by Alhazmi, *et al.* (see Chapter 2) and further supporting the hypothesis of vulnerability discovery depending in part on the learning curve of an attacker. [JO97, GS05, CFBS10].

Table 4.5: Count by type of RRC vulnerabilities that do not affect ESR (correspondence between RRC versions 10 or greater and ESR versions 10 and 17 is given in Tables 4.3 and 4.4). Totals are not unique, as a single vulnerability may affect multiple versions

Version	**Total**	**Foundational**	**Regressive**	**New**
5	38	19	19	-
5.0.1	38	19	19	-
6	38	19	19	-
6.0.1	37	18	19	-
6.0.2	37	18	19	-
7	37	18	19	-
7.0.1	15	12	3	-
8	37	18	19	-
8.0.1	36	18	18	-
9	36	18	18	-
9.0.1	14	12	2	-
10	13	12	1	-
10.0.1	13	12	1	-
10.0.2	13	12	1	-
11	14	12	2	-
12	13	10	1	2
13	11	10	1	-
13.0.1	8	8	-	-
14	9	9	-	-
14.0.1	8	8	-	-
15	9	9	-	-
15.0.1	4	4	-	-
16	2	2	-	-
16.0.1	2	2	-	-
16.0.2	2	2	-	-
17	2	2	-	-
17.0.1	1	1	-	-
18	1	1	-	-
18.0.1	1	1	-	-
18.0.2	-	-	-	-
19	4	-	1	3
19.0.1	4	-	4	-
19.0.2	1	-	1	-

These results show that Mozilla's Firefox RRC development process did not increase the rate of vulnerability discovery. Moreover, the vulnerabilities found in a particular RRC version while it was current were any easier to find than those remaining in older code. This suggests that during the RRC lifecycle, the time to find vulnerabilities and learn how to exploit them in new code compensates for the presumed increase in the density of vulnerabilities in immature code.

4.3.2.3 Is the scope of disclosed vulnerabilities confined to RRC?

If the vulnerabilities found in the current RRC version result from new code added, one ought to find that most of these are *new* vulnerabilities and therefore ones that do not affect code shared with ESR versions. However, this is not the case. As stated above, during the active lifetimes of ESR versions 10 and 17, only a few new RRC vulnerabilities were disclosed, but, more importantly, if one compares the 465 total RRC vulnerabilities to the 420 in ESR, all but 45 have the *same* BugID affecting both RRC and ESR. In other words, the overwhelming majority of active vulnerabilities disclosed in Firefox RRC *also* affect ESR. This means that the 24 ESR point releases, which average 4,700 LOC code added per release, are affected by nearly *90%* of the vulnerabilities that affect the concurrent RRC versions *which average more than 290K LOC added per release!*

This does not mean that the new code in RRC does not contain new vulnerabilities, but rather, that 90% of the vulnerabilities disclosed in the RRC versions released during the lifetime of each ESR version must be in the older, *shared* code. As we can see in Table 4.3 very few vulnerabilities affecting each RRC version actually originate in those RRC versions. Moreover, of the 617 vulnerabilities disclosed in Firefox since the inception of RRC, 32 of them do not affect *any* of the RRC versions. These vulnerabilities *only* affect the foundational code originating in or before version 4, but were not found until after RRC was adopted.

But what about those vulnerabilities that *were* introduced by new code added

each RRC cycle? That is, vulnerabilities that *do not* affect the corresponding, contemporary ESR version. These vulnerabilities, if not found and fixed, may affect a later ESR version, because the new RRC features are rolled into the next ESR release. How quickly were they discovered and disclosed? As one can see in Table 4.5, of the 45 vulnerabilities that affect only RRC versions and do not affect the corresponding ESR versions, only 5 actually originate in the newly released versions. More importantly, *only these 5* were disclosed during the 6 weeks that that version was current. [2] While the new code does indeed contain vulnerabilities *those vulnerabilities are not being found and disclosed while the version in which they originate is the current version.*

The implications of this for ESR, and software engineering more generally, are highly significant. The effective lifetime of the ESR versions is four times longer than for RRC. No new features are added after its initial release. It is only changed to patch critical security bugs. Yet, it is still vulnerable to 90% of the same vulnerabilities that affect the RRC versions. This raises concerns about the security of code over time, as well as the impact of code reuse on security. Particularly as many of the new features developed for each RRC version are eventually rolled into the next ESR version. This strongly suggests that the slower changing ESR versions may not experience any benefit from the Honeymoon Effect. By the time RRC code is rolled into ESR, the adversary may have already climbed the learning curve.

4.4 Discussion

Intuition, gleaned from decades of secure software engineering best practices, suggests a tension between the rapid deployment of new software features and the avoidance of software defects, particularly those affecting security.

[2] Two in version 12.0 and three in version 19.0

The rapid release employed by Mozilla for Firefox, in which new software releases, with new features, are rolled out on an aggressive schedule, seems as if it could only come at the expense of security. Users concerned with security, one might assume, would be better off eschewing the latest features in favor of the more mature, stable ESR releases. Agile programming methodology, particularly in an application as exposed as a web browser, *should be* a security disaster.

At least with respect to vulnerabilities disclosed during the lifecycle of each Firefox release, my results suggest that this intuition appears to be wrong. Vulnerabilities are disclosed in the older code at least as often as they are in the newer code. This is both surprising and encouraging news. It suggests that during the active lifecycle, the adversary's ability to discover security defects is *dominated less by the intrinsic quality of the code and more by the time required to familiarize themselves with it.* It suggests that the Firefox rapid-release cycles expose the software to a shorter *window of vulnerability.* [AFM00] Frequent releases of new features appear to have provided the Firefox developers with new grace periods or *second honeymoons* (using terminology I coined in [CFBS10]). While there may also be other factors affecting vulnerability discovery which are changing over the duration of software evolution I studied, it is clear that the net effect of RRC, seen in the data, has been the attenuation of the attacker.

Even while generalization remains an open question, in Firefox, the unexpected benefit of frequent large code releases appears to be a lengthening of the attacker's learning curve. The findings reported in this chapter further support the ideas that familiarity with a codebase is a useful heuristic for determining how quickly vulnerabilities will be discovered and, consequently, that software reuse (exactly because it is already familiar to attackers) can be more harmful to software security than beneficial.

These results are consistent with a "Honeymoon Effect", and suggest that the pattern exhibited by vulnerability disclosure in Firefox could result from would-be

attackers having to re-learn and re-adapt their tools in response to a rapidly changing codebase. [CFBS10] These results should lead software developers to question conventional software engineering wisdom when security is the goal.

The "S" learning curve shape seen in Figure 4.8 could be construed as a measure of the increase in proficiency as result of repeated exposure. In other words, a measure of the skill and expertise resulting from increased familiarity over time. As demonstrated by the Honeymoon Effect presented in Chapter 3, this learning curve appears to play a significant role in the software vulnerability lifecycle. As demonstrated above, rapid code changes may affect an adversary's familiarity with the codebase, resulting in a longer, or even a second honeymoon.

I chose Firefox software as the basis for analysis, as it was originally architected using the traditional development model and switched to rapid-release midstream. It will be interesting to see if other software systems, including those that have been designed and developed using *only* Agile methods share the same properties. It will also be interesting to see what effect the switch to silent auto-updates has had on the vulnerability life-cycle. The dataset that I integrated for Firefox with its large code-base, and large user-base, coupled with its prominence as an attack target is strongly suggestive that the rapid release strategy has significant and unexpected security advantages in real world systems.

Chapter 5

Testing the Effect of a Single Intrinsic Property

"The only relevant test of the validity of a hypothesis is comparison of prediction with experience." (Milton Friedman)

5.1 Introduction

Recall that each time Mozilla releases a new version of Firefox, hundreds of thousands of LOC were being added, modified or removed and thousands of files were affected.(see Tables 4.1 and 4.2)

Because these changes are primarily the result of adding new features and not merely fixing defects, it is important to recognize that each time a new version is released, its substance has altered in some way. Whether it is measured as lines of code, or numbers of files, these changes are in effect altering some of the intrinsic properties of the product as a whole.

At the same time, it was demonstrated in the previous chapter, that even after Mozilla moved from a traditional development model to a rapid release development

cycle, the Honeymoon Effect is still predominant. New Firefox versions are likely to experience a positive Honeymoon twice as often as not.

This begs the question, do releases with more changes directly correlate with an increase in the time from release until the first vulnerability is discovered and exploited?

This chapter explores whether coarse-grained changes to intrinsic software properties are the sole cause of the Honeymoon Effect and attempts to answer the following questions:

- Can the Honeymoon Effect be directly correlated with the total LOC or file changes? That is, does the addition, modification or removal of code alone account for the increased likelihood of having a positive Honeymoon?
- Is there a minimum number of files that must be changed in order for the Honeymoon Effect to appear?
- If either of the above is true, how large a change is required?
- Does one type of change matter more than another? (i.e., adding new code, modifying existing code, or removing old code)
- Do code changes in one type of programming language affect the Honeymoon more than another?
- Does the frequency of file changes matter?

5.2 Dataset and Methodology

Mozilla relies on a Mercurial repository for Firefox source code management and control. Changes to the repository are logged and each change has a unique id. Mercurial provides tools for cloning, searching, logging, and comparing changes by

revision id. Table 5.1 lists the Firefox repository revision ids that correspond to the version releases used in this analysis.

The first step was to collect the overall file changes between versions to see if there were meaningful correlations to the Honeymoon Effect. I first attempted to use Mercurial's built-in 'hg diff' tool for the analysis. Hg diff can provide users with per version totals of files added, files removed, files modified, as well as information about total LOC added and lines of removed. Each pair of consecutive major version releases, excluding all blank lines, comments and test files, (e.g. 4.0-5.0 or 19.0-20.0) was 'diffed' to find all lines added or removed, and all the files added, removed and modified from all Firefox source code files. Unfortunately, the hg diff tool does not offer fine enough granularity, for a detailed analysis of lines of cod e, as it cannot differentiate between LOC modified and LOC added or removed. Fortunately, there is a widely used open source code counter CLOC written by Al Danial which provides this feature. [Dan16] CLOC was employed to find LOC modified, added and removed, both in total for all source code files listed above, and per file by file type. CLOC was also used to calculate total LOC per file type and total LOC unchanged by file type. Similar to the analysis presented in the previous chapter, the results were filtered to include only files with the extensions: `.c`, `.C`, `.cc`, `.cpp`, `.css`, `.cxx`, `.h`, `.H`, `.hpp`, `.htm`, `.html`, `.hxx`, `.js`, `.jsm`, `.py`, `.s`; test cases and harness code, code comments and whitespace were excluded. LOC counting for the research presented here, is conservative and may understate changes to the Firefox codebase between versions.

The numbers of Firefox files unchanged, modified, added and removed, are listed in Table 5.2 and the counts of the files and LOC changes for C, JavaScript can be found in Tables 5.5, 5.3 and 5.4

In my analysis, each RRC release was considered an independent event. As in the

earlier studies, The date of the first vulnerabilities disclosed affecting each release, was determined by correlating data from Mozilla's MFSA with the CVE database and with Bugtraq. [Sac97] This date was used to determine if there was a positive or negative Honeymoon for that release.

Once the file and line counts were completed, the totals were plotted, evaluated and various statistical analysis applied to see if there were patterns in the distributions that might answer the questions listed above.

5.2.1 Limitations

The analyses discussed in the next section are high level studies, seeking only to determine whether the simple property, changes in magnitude of files or LOC, increases the likelihood of a release incurring a positive Honeymoon. Do these changes alone account for a lengthening of the time to first vulnerability discovery compared to subsequent ones? While there may be many other properties that affect the attacker's learning curve, their contribution is left for future research.

5.3 Is it Really so Simple? Answer: No

5.3.1 File Type Changes

The first property to examine was whether there is a correlation between the number of files being changed, and the honeymoon effect.

In Table 5.2 one finds the total numbers of source code files found in Firefox versions 4.0-40.0. The numbers are shown for files unchanged from the previous version, modified, added (new), and removed for each version. The types of files counted include all c, cpp, header files, objective-c, Java, JavaScript, JSM, Assembly, S, and Python files combined.

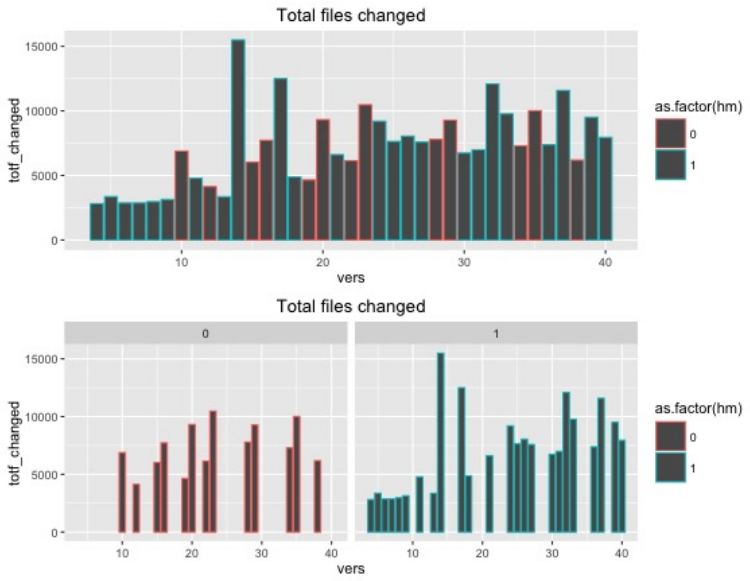


Figure 5.1: Graph of total files changed per version for Mozilla Firefox source code

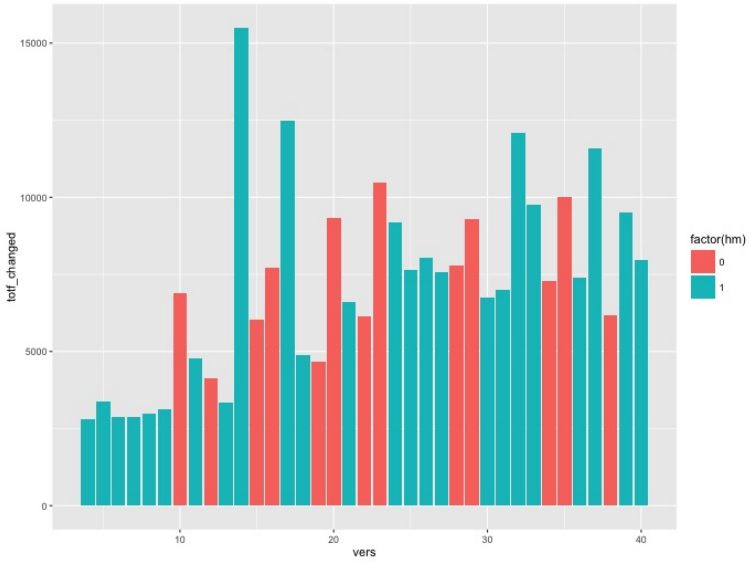


Figure 5.2: Graph of the delta between total files changed and unchanged per version for Mozilla Firefox source code

At first examination, there seems little evidence that the magnitude of file changes has any effect on whether a new release benefits from a positive honeymoon. If the simple magnitude of file changes were the sole cause of an increase in time to first exploit, one would expect to see at least a low positive correlation with the magnitude

of LOC changes and the likelihood of a positive honeymoon.

But, this isn't the case, as one can see from Figure 5.1, there is no obvious pattern where fewer file changes result in either a majority of non-honeymoons or a majority of honeymoons. There is no definitive pattern to file change totals in consecutive releases with honeymoons or without. There is no pattern to file change totals where a honeymoon release is followed by a non-honeymoon release, or vice versa. This observation is consistent for different subsets of the data as well, there appears to be no significant difference when comparing the versions with positive honeymoons to those with negative honeymoons, looking at only the numbers of files added, only the numbers of files modified, or only the numbers of files removed. Nor is there any more clarity if we examine the difference between total files changed and total files untouched.

What about for individual programming languages? Perhaps the time to first disclosed vulnerability is dependent on magnitude of one type of file being changed rather than the changes to the total numbers of files changed in the source code combined. Table 5.5 lists the total numbers of files changed for C and JavaScript files in Firefox versions 5.0-40.0. Just as with total number of files changed, separating total changes by programming language type provides little insight. In comparing releases with positive honeymoons to those without there does not appear to be any major difference seen when looking at the total numbers of file changes for individual types of code. There are wide variances from release to release, even between consecutive releases where both show a positive honeymoon, the total number files changed varies widely and the same for consecutive non-honeymoon releases. Nor does is seem that there is anything to be learned from comparing at the total files changes of one language or another. Additionally, there isn't any clear pattern evident when the file counts are binned by type of change. Analyzing the total files changed, even if separated by type of change or programming language does not provide any information that can be used to predict the likelihood of a release experiencing a positive

honeymoon.

5.3.2 LOC Changes

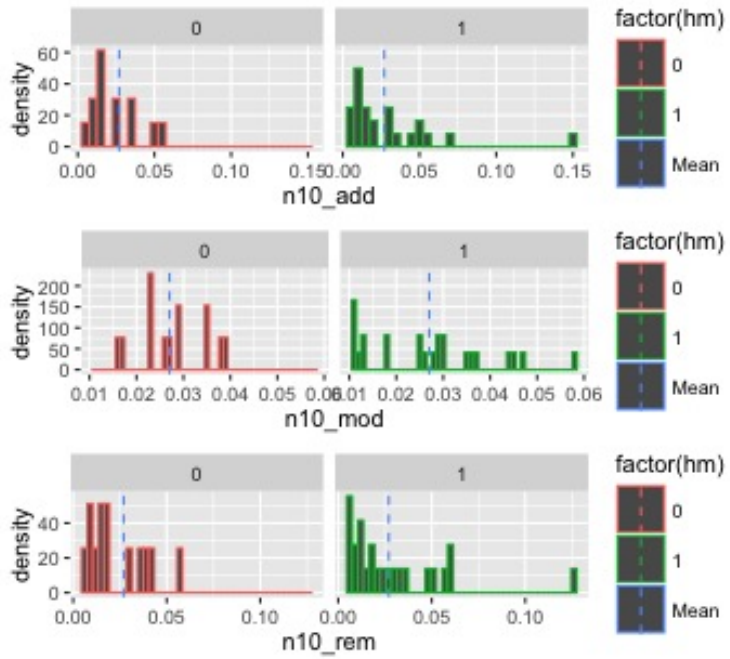


Figure 5.3: Graph of total LOC changes for C and Javascript code (normalized for readability) for Mozilla Firefox source code

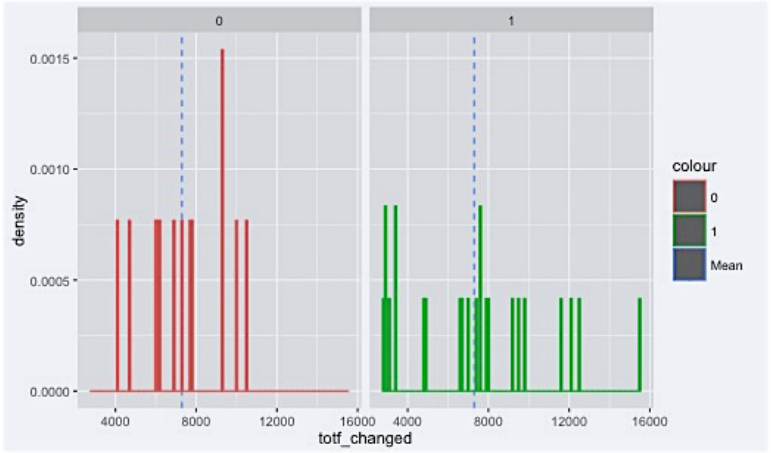


Figure 5.4: Graph of frequency of total files changed for Mozilla Firefox source code

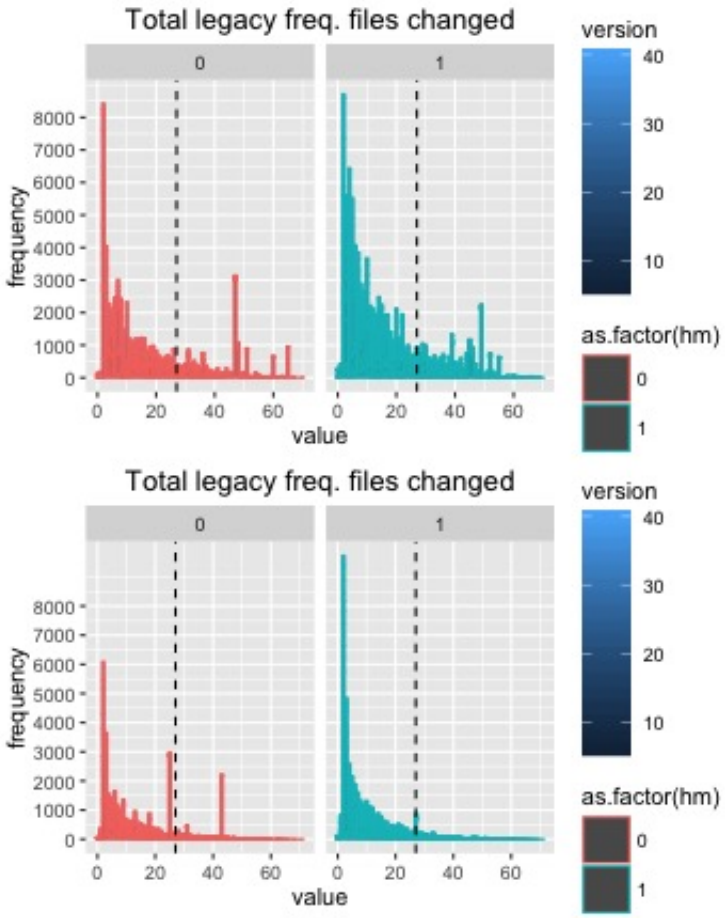


Figure 5.5: Graph of frequency of total files changed by file age for Mozilla Firefox source code

Since it appears that gross file changes do not correlate with the Honeymoon Effect, perhaps analyzing the lines of code changed from version to version may provide more information.

Here too, there doesn't seem to be any obvious set of characteristics that separate releases with positive honeymoons from those without. Looking at Tables 5.4 and 5.3 one observes the wide variations in LOC changed per release. Whether a release has a positive Honeymoon, does not appear to correlate with a positive Honeymoon in the immediately previous release (see Figure 5.6).

Neither is there any more clarity to be found from looking at the total counts of

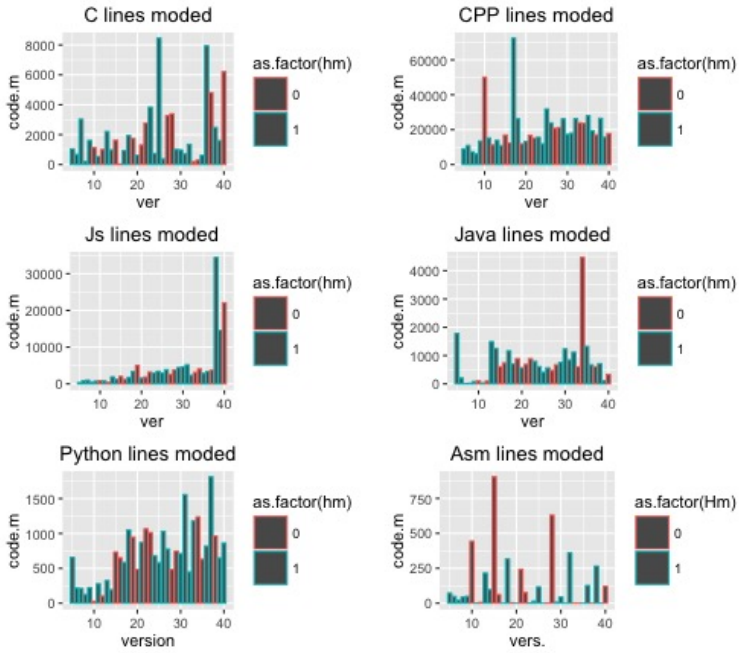


Figure 5.6: Graph of LOC modified per version for Mozilla Firefox source code

LOC by type of code changed.

5.3.3 Question: Given these results, does changing files or LOC matter at all?

Does this lack of any obvious correlation mean that magnitude, frequency or type of file and LOC changes are completely irrelevant to the Honeymoon Effect?

Further examination into the frequency of file changes, and the lifetime of legacy code may provide some insight.

5.3.4 Frequency of change and legacy code

It is only when one compares the density of the distribution of file and LOC changes for releases with positive Honeymoons to releases without that one finds any differences. Looking at the density of total files changed (see Figure 5.4) and the density of total LOC added, modified and removed (see Figure 5.3) it can be seen

that in all cases the largest changes occur more frequently in releases with a positive honeymoon. The most frequent file changes cluster around the median for both Honeymoon and non-Honeymoon releases, but there are no non-Honeymoon releases where 10,500 or more files have changed, while 18% of the Honeymoon releases have greater than 11,000 files changed. Similarly, while there isn't much to be gleaned from the LOC added or removed graphs, one sees the same longer right tail in the frequency plot of LOC modified. 21% of the releases with positive Honeymoons have LOC modified counts greater than 300k while only 7% of the non-Honeymoon releases do.

Moreover, Figure 5.5 shows that releases with positive Honeymoons contain larger numbers of younger file changes. That is, the versions of the files remain unchanged for a shorter period of time. Releases that didn't experience positive honeymoons have greater numbers of files that remained unchanged from the previous version. Thus it appears that legacy code, does indeed negatively affect the honeymoon period.

5.4 Discussion

The Honeymoon Effect is a measurement of the attacker's learning curve when subjected to unfamiliar code. A release experiences a positive honeymoon when the time it takes to discover the first vulnerability is longer than the time to discover subsequent vulnerabilities.

While it would seem, at first blush, that simply adding or changing code, regardless of any other security conditions should be sufficient to account for this effect, analysis shows that it is not that simple. However, it must be acknowledged that this analysis is limited by the number of Firefox RRC releases available for analysis. This relatively small sample size makes it impossible to truly confirm or eliminate the property of code change on the steepness of the attacker's learning curve.

Moreover, it is important to remember, that there is a definite Honeymoon Effect present in two thirds of Mozilla's Firefox RRC releases. Lack of any direct correlation between the addition or modification of files or LOC does not invalidate this, but instead appears to show that the simplest metric for measuring the attacker's learning curve, many new files, or many new unfamiliar LOC is not sufficient to predict the likelihood of a positive Honeymoon.

Consistent with observations about the Honeymoon Effect, when looking at the frequency distribution of changes overall, there is some indication that larger quantities of file or code changes, particularly with regard to legacy code does show some correlation with a positive Honeymoon. Clearly, though, this single intrinsic property is insufficient to explain the phenomenon.

If extending the attacker's learning curve was entirely dependent on throwing lots of changes (files or LOC) into each new release, the solution to making secure software would be trivial. The magnitude of file or LOC changes is an intrinsic property of the source code controlled entirely, and it might be argued, easily, by the developers. Instead, the Honeymoon Effect appears to result from more complex or subtle interactions of software's intrinsic properties, and those extrinsic properties of the larger ecosystem in which it lives.

Version	Revision:Hash
40.0	275544:84e0a4087157
39.0	267811:d3b3e57e8088
38.0	260428:4c4dc6640c7e
37.0	252119:29182ac68a26
36.0	245425:88c5342693e3
35.0	235743:32e36869f84a
34.0	227446:456394191c90
33.0	218077:1f22a8cc7aa5
32.0	209473:44234f451065
31.0	200814:32dddf30405a
30.0	193465:529a45c94e5a
29.0	184811:f60bc49e6bd5
28.0	176480:5f7c149b07ba
27.0	168007:b8896fee530d
26.0	162111:39faf812aaec
25.0	155154:d86ad7db1de3
24.0	149404:7c3b0732e765
23.0	144004:5efffaf39814
22.0	138619:0d4b9c74be55
21.0	133508:30ec6828d10e
20.0	128557:c90d44bfa96c
19.0	123486:20238b786063
18.0	118275:8efe34fa2289
17.0	110374:919435c6f654
16.0	105367:10fe550fadc6
15.0	101019:450143d2d810
14.0	96695:f0f78d96f061
13.0	92652:2b643ea8edf9
12.0	89093:a294a5b4f12d
11.0	85921:b967d9c07377
10.0	81941:baefae4b6685
9.0	79326:34852484d0ae
8.0	76675:d03b51a9b2bd
7.0	73377:273977a2c0ea
6.0	70736:218ed8178b1e
5.0	68331:7b56ff900c2a
4.0	68309:fca718600ca0
3.6a1	31196:da7fbe8a24dd

Table 5.1: Firefox versions included in this analysis, with Revision Number and Repository Tag

Version	Files Unchanged	Files Modified	Files Added	Files Removed	Total Files
4	3398	4133	3932	1146	6184
5	9607	1494	278	362	9523
6	8917	1908	257	554	8620
7	9163	1731	168	188	9143
8	9386	1498	118	178	9326
9	8966	1900	192	136	9022
10	6908	3781	1396	369	7935
11	10108	1767	1126	210	11024
12	10307	2165	635	529	10413
13	10897	2093	254	117	11034
14	11225	1887	650	132	11743
15	5626	7960	695	176	6145
16	11749	2433	1613	99	13263
17	10521	5177	378	97	10802
18	10011	5437	2068	628	11451
19	14295	3095	229	126	14398
20	14793	2391	463	435	14821
21	13144	3019	1801	1484	13461
22	14342	3237	1406	385	15363
23	14927	3865	428	193	15162
24	15834	2964	475	422	15887
25	12804	5207	1963	1262	13505
26	14589	4021	1596	1364	14821
27	15565	4006	868	635	15798
28	15799	4245	1155	395	16559
29	16450	4108	901	641	16710
30	16135	3660	2161	1664	16632
31	17195	4639	368	122	17441
32	18023	3970	729	209	18543
33	16994	4730	1350	998	17346
34	17251	4898	1589	925	17915
35	19548	3710	576	480	19644
36	18482	3540	2068	1812	18738
38	19861	3673	309	556	19614
38	16877	6215	1338	751	17464
39	20726	3663	261	41	20946
40	19098	5152	443	400	19141

Table 5.2: Firefox File Changes Per Version

JavaScript Files					
Version	Unchanged	Modified	Added	Removed	Total
4	1843041	111028	157441	710761	1289721
5	2778955	12835	73863	95171	2757647
6	2816139	14466	77077	108090	2785126
7	2820412	12271	69486	74999	2814899
8	2838015	8532	46049	55622	2828442
9	2759489	18641	76164	42397	2793256
10	2724589	64733	315079	141152	2898516
11	2991415	21929	96141	91057	2996499
12	2889768	17376	154851	126161	2918458
13	3081803	20676	64557	38928	3107432
14	3033314	14317	103481	39993	3096802
15	3057403	27245	153096	66464	3144035
16	3181652	15272	424556	40820	3565388
17	3573051	94503	91447	36385	3628113
18	3428683	35459	556319	212400	3772602
19	4029538	17710	70775	51766	4048547
20	3878192	21454	99207	70946	3906453
21	3772293	23402	401788	367113	3806968
22	4021347	22675	416017	71483	4365881
23	4468126	24468	112996	49981	4531141
24	4306513	22795	167375	193746	4280142
25	4205431	56625	438377	318753	4325055
26	4346186	31743	352658	322504	4376340
27	4548025	33567	130880	64869	4614036
28	4574197	32977	286237	105298	4755136
29	4727059	37653	171861	128699	4770221
30	4604540	24764	479633	307269	4776904
31	5030799	25150	103343	140993	4993149
32	5138201	46131	179988	63550	5254639
33	5004576	38795	287442	232359	5059659
34	5083155	32034	286487	215624	5154018
35	5310695	39734	192941	141205	5362431
36	5030940	35538	460516	386934	5104522
37	5409425	33650	114804	175538	5348691
38	5234463	53789	269840	178008	5326295
39	5603301	22272	65789	25886	5643204
40	5546367	28932	144225	116063	5574529

Table 5.3: JavaScript LOC Changed Per Firefox Version

C Files					
Version	Unchanged	Modified	Added	Removed	Total
4	99256	4777	70746	25391	144611
5	161126	343	28497	13310	176313
6	184432	841	12475	4693	192214
7	194767	1023	3536	1958	196345
8	197172	543	9582	1611	205143
9	199090	746	6514	7461	198143
10	186883	845	14886	18622	183147
11	181378	805	40902	20431	201849
12	220232	476	9064	2377	226919
13	223198	1928	11432	4646	229984
14	224003	1240	27391	11315	240079
15	231052	2040	70562	19542	282072
16	300320	1180	14861	2154	313027
17	309500	1774	12370	5087	316783
18	302641	3385	31972	17618	316995
19	328052	5041	43360	4905	366507
20	370500	1575	10578	4177	376901
21	372564	1880	40398	8209	404753
22	381117	3215	16737	30510	367344
23	379019	2935	23919	19115	383823
24	388409	3436	25018	14028	399399
25	404085	2968	23948	9810	418223
26	381117	3878	61782	46006	396893
27	422741	2603	135189	21433	536497
28	537666	3813	19466	19054	538078
29	534139	4455	33440	22351	545228
30	556747	4631	31498	10656	577589
31	528701	5228	31376	6107	553970
32	604823	2367	29806	9525	625104
33	613394	3161	407433	20441	1000386
34	683626	4093	68654	62011	690269
35	676821	2880	89898	76672	690047
36	757032	3384	17774	9020	765786
37	753065	3728	25796	21397	757464
38	707120	34435	45448	41034	711534
39	760328	14596	25106	12079	773355
40	738530	22103	30266	39397	729399

Table 5.4: C LOC Changed Per Firefox Version

JavaScript Files					C Files				
Version	Unchanged	Modified	Added	Removed	Version	Unchanged	Modified	Added	Removed
4	186	155	252	62	4	3191	3968	3644	1067
5	493	73	85	27	5	9075	1411	193	317
6	526	122	20	3	6	8375	1767	220	537
7	543	120	5	5	7	8573	1607	163	182
8	546	113	15	9	8	8797	1377	103	169
9	534	109	24	31	9	8389	1783	166	105
10	503	91	33	73	10	6359	3684	1361	295
11	449	96	130	82	11	9615	1661	287	128
12	598	71	42	6	12	8969	2071	592	523
13	581	113	25	17	13	9684	1857	178	91
14	581	121	62	17	14	9978	1636	544	105
15	161	590	94	13	15	4965	7034	570	159
16	663	177	57	5	16	10335	2143	1527	91
17	663	217	47	17	17	9114	4814	313	77
18	567	319	88	41	18	8631	5024	1951	586
19	795	179	25	0	19	12655	2831	192	120
20	864	129	21	6	20	13091	2160	423	427
21	816	180	228	18	21	11493	2727	1550	1454
22	849	279	61	96	22	12774	2752	1299	244
23	863	255	61	71	23	13198	3508	319	119
24	909	246	48	24	24	14021	2616	398	388
25	889	282	83	32	25	10981	4829	1874	1225
26	894	326	96	34	26	12759	3616	1451	1309
27	1001	263	50	52	27	14062	3663	331	101
28	1004	298	21	12	28	13883	3829	1064	344
29	944	329	97	50	29	14553	3643	669	580
30	1010	339	119	21	30	14056	3176	1978	1633
31	1120	343	55	5	31	14964	4132	291	114
32	1240	268	81	10	32	15643	3553	625	191
33	941	594	129	54	33	14870	4020	1190	931
34	1226	362	123	76	34	15284	3972	1074	824
35	1358	338	56	15	35	16830	3053	477	447
36	1297	422	59	33	36	15663	2921	1983	1776
38	1464	284	53	30	38	16774	3269	245	524
38	1316	424	105	61	38	13960	5647	1165	681
39	1490	341	62	14	39	17466	3280	158	26
40	1295	462	80	136	40	16033	4611	360	260

Table 5.5: C and JavaScript Files Changed Per Firefox Version

Chapter 6

Conclusion: Toward A New Model

"Il faut regarder la configuration ensemble pour déterminer le comportement des parties et non l'inverse."(Paul Weiss) [1]

6.1 Introduction

The results in the earlier chapters are evidence that early in the lifecycle of large scale systems such as Firefox, the likelihood of a vulnerability being discovered and exploited fits a learning curve. This appears to be the case even for software where development and release strategies may not be compatible with the recommended secure software development best practices.

It may seem surprising, but even new releases of entire computer systems such as operating systems (and consequently those applications running on them), seem to experience a learning curve early in their lifecycle. This phenomenon can be clearly seen by examining the effects of changes made to the Windows Operating System in 2012. These changes resulted in a quantifiable Honeymoon Effect for software

[1] Translation: We must look at the overall configuration to determine the behavior of the parties and not the reverse.

products that ran on the system.

In October of 2012 Microsoft released the Windows 8 Operating System. Microsoft had focused significant effort on securing the system and the OS contained several built-in exploit mitigations. [JM12] Exploit developers and security researchers analyzing the new OS simultaneously praised the new mitigations, and bemoaned the loss of their previously successful exploitation techniques.

In an analysis of the new Windows 8 Heap Internals at the BlackHat Briefings in 2012 Chris Valasek(Coverity) and Tarjei Mandt(Azimuth) reported that "All of the attacks that have been demonstrated on Windows 7 have been pretty much addressed in Windows 8" [VM12], and shortly after release, Alex Ionescu (CrowdStrike) reported: "Windows 8 was the subject of the most intensive and well-thought-out exploit mitigation and security hardening process ever attempted by Microsoft... And it delivered." [Ion12]

These new exploit mitigations did not make Windows 8 invulnerable, or even free of exploitable vulnerabilities. Shared libraries and legacy code contained vulnerabilities that were outside the new protections. [Mic12] and within a month, Vupen Security claimed to have chained together multiple vulnerabilities attacking Internet Explorer 10 to gain remote execution on Windows 8. [2]

The mitigations did make many of the widely used attacker tools and techniques unusable, and resulted in attackers having to develop new exploitation methodologies obsolete. *And, importantly, the mitigations reset the attacker's learning curve!* It took three months for a scripted attack against *legacy* kernel code to be released [OSC13], and four months for the first proof of concept demonstrating a successful bypass of some of the newly implemented protections. [Tea11, SF12, Che13, Ros11, E.10] Compare that to the release of Windows 7 which was found to be vulnerable to several known exploits the very day it was released. [Wis09]

[2]The attack was demonstrated at CanSecWest in Jan. 2013

Of course, these new attacks were addressed by Microsoft in their next few releases. Security fixes in Windows 8.1 made ASLR unique across devices eliminating predictable address space mappings, increasing the amount of entropy that exists in the address space, additionally, the ability to execute code was removed from the working set memory pages. [Mic13]. Again forcing attackers to learn new ways to bypass these new mitigations. [WRI13, Fra15, Yun15] Additionally, non-Microsoft applications appeared to receive an unexpected second honeymoon from these mitigations without having to update or alter their own code. [Sar13]

Unfortunately, while these examples provide evidence that changes to software can result in a positive Honeymoon, the research presented in Chapter 5 suggests that predicting whether a release will experience a Positive Honeymoon, i.e., that the attackers will experience a long learning curve, is not as simple as measuring the magnitude of changes in files or LOC. Rather, that changes to the system, extrinsic to any particular software product can result in a Positive Honeymoon. Overall, this is evidence that there are extrinsic properties which have an impact on the robustness of the system against attack. [3]

This leads me to make the following observations:

- **Observation:** Any individual piece of software (P1) functions as part of a complex system comprised of any number of other objects (P2..Pn) not under the control of the developers of P1.

- **Observation:** The properties of P2..Pn are extrinsic to P1 and any changes to them occur independently of P1 and at any time during the lifecycle of P1.

- **Observation:** The security of P1 appears to be dependent on *both the intrinsic and the extrinsic properties of its containing system.*

The existence of these extrinsic properties may help explain why traditional software engineering models have failed to provide insight for predicting and measuring

[3] N.B., This impact can be both positive or negative

the vulnerability lifecycle. While a software product's intrinsic properties are determined by software engineering, properties such as attacker motivation, skill level, and the attacker's learning curve, are not. Instead, these are properties of the larger software ecosystem in which that product functions.

Note that, if the security of any individual piece of a system is dependent on other pieces of the system over which it has no control, then *any changes to the extrinsic properties of P2..Pn can affect the security of P1 (positively or negatively.)* Strictly software engineering models tend to consider the environment a blackbox and thus inadequately account for the effects which may come from changes to those extrinsic properties. Additionally, strictly software engineering models are intended to describe a static products or systems and as such insufficiently explain the effects which have been observed to come from changes to those extrinsic properties.

The results in this dissertation suggest a relationship between a product and the other members of its ecosystem. One that changes dynamically throughout its entire lifecycle. This relationship is missing from exclusively secure software engineering models.

This suggests that approaches to understanding the security of an individual software product must broaden their scope beyond pure software engineering models to include the properties of the entire ecosystem and their relationship to each other. Furthermore, this strongly suggests that *the security of software systems is not a strictly software engineering problem*, and that a new conceptual framework is necessary to fully describe this relationship.

6.2 Examining The Problems With Strictly Secure Software Engineering Models

VDMs and ASMs have proven useful for thinking about weaknesses in the software ecosystem. VDMs focus on finding and removing exploitable defects. ASMs focus

on the processes and communication channels used to exploit a vulnerability. Both types of models attempt to help developers predict, find or eliminate mechanisms for exploiting software.

Yet, in spite of these improvements, the rate of vulnerability discovery in software does not appear to be decreasing. A recent report on vulnerability trends showed that the year 2015 broke the previous all-time record for the highest number of reported vulnerabilities. The report also stated that 20.5% of reported vulnerabilities received CVSS scores between 9.0 and 10.0 and the number of vulnerabilities and the CVSS scores were both trending higher over the last four years. [Sec15] Many of these vulnerabilities have been lying dormant in the code for decades. [Ltd14, Gaf14, NIS14, Bea16]

Not only are more vulnerabilities being reported now than in previous years, systems are being exploited in much greater numbers. Analysts reported record numbers of large company security breaches and referred to 2015 as "The Year of the Breach". [Gre15, Loh15, Hol16, Har15, Mar15, Whi16]. Moreover, given the trends, the expectation among the security community at the time of this writing is that 2016 will be most exploited year yet. [(RB16, Kou16, Sym16] It is apparent from these reports, that current models for developing secure software, for finding and removing vulnerabilities, and even for enumerating and minimizing attack vectors, however useful, are not sufficient to model the security of a software product as it interacts with its environment over its entire lifecycle.

The research presented in Chapters 3 - 5 suggests that these models have limitations that prevent them from recognizing and capturing the interactions between intrinsic and extrinsic properties of members of the system.

6.2.1 Current Models are Constrained by Their Assumptions

VDMs, for example, are concerned with finding vulnerabilities remaining in the software, and predicting when (or if) they are likely to be discovered. The two primary assumptions of these models, one, that the majority of the vulnerabilities are found early in the lifecycle, so that the only remaining vulnerabilities are assumed to be difficult to find, and two, that the older the software is, the higher its quality and therefore the higher its security, are assumptions based exclusively on the intrinsic properties of the software. There is no mechanism for modeling the interactions and complexity that extrinsic properties add to the security of the system.

Similarly, although ASMs do include a definition of the environment in their description, the intent of the model is to make it completely independent of any factors in its environment. The assumption is that the rest of the ecosystem is irrelevant to the model. Moreover, the model assumes that the *Damage Potential* and the *Damage Effort* variables can be calculated *without considering the attacker at all*. The Damage Effort metric is of particularly questionable value when isolated this way, because it has been frequently demonstrated that attackers are willing to expend enormous effort developing a successful exploit. E.g., since in 2015 and early 2016, every successful compromise of the browsers Chrome, Safari, Internet Explorer and Firefox combined vulnerabilities in the browser with vulnerabilities of third-party objects in the ecosystem to gain root access to the machine (either in the operating system or third-party products running on the machine, or both). See [Bea16] for a very recent example as well as [Tre16a, Tre16b].

6.2.1.1 These Models Assume a Static Product

An important limitation of both VDMs and ASMs is that the models assume that product being modeled is part of a static ecosystem. Both types of models lack any

sort of feedback loop for receiving new information from the environment. Nor do they contain any mechanism for recalculating metrics or adapting to new or outside information.

This is quite understandable, because one major assumption made by these models is that the software they model is a finished product. *i.e., complete and ready for release, or nearly so.*

Therefore, the lack of any feedback loop in VDMs and ASMs means that any defensive strategy based on them is one that relies on patching each individual vulnerability as it is discovered. Consequently, instead of taking advantage of knowledge gained from analyzing a newly discovered exploit, and prophylactically removing not yet discovered vulnerabilities that could potentially be exploited by the same method or with the same tools, the software will continue to be vulnerable to any exploitation techniques introduced after its release. See [Ltd14, Gaf14]

The assumption that legacy code has passed *"the test of time"* is also of particular concern. It means that ASMs and VDMs tend to be applied to the newly developed code, not to the old existing, already tested, legacy code. Hence, code carried over from earlier versions which may be vulnerable any newly discovered exploitation techniques, would remain untested. The recent discoveries of Heartbleed and Shellshock and others [Ltd14, NIS14, Gaf14] speak to how inimicable vulnerabilities in long forgotten libraries can be.

6.2.1.2 These Models Assume a Product in Isolation

A significant limitation of these models is their narrow scope. VDMs and ASMs focus only on the software itself and omit any mechanism for modeling the role played by the attacker(s).

Both VDMs and ASMs quantify security by measuring the intrinsic properties of the software itself. Neither VDMs or ASMs capture the complexity involved in exploiting software today wherein attackers chain together multiple vulnerabilities

from different parts of the system in order to subvert protections. [OS12a, OS12b].

VDMs try to find the software defects that may become exploitable vulnerabilities. This meant that VDMs are blind to any type of exploitation that does not manifest as a fault during execution. Testing platforms that rely on such models are likely to find violations of memory handling, such as buffer overflows and use after free errors, but are not well suited for finding weaknesses that result in hidden functionality or information leakage. [The16]

6.2.1.3 Corollary: These Models Ignore the Ecosystem

The flipside of a model's focus being limited to *intrinsic properties* is that there is little or no focus on the *extrinsic properties*. That is, those characteristics of the rest of the system that may affect whether a vulnerability is found and exploited.

With VDMs, the density of vulnerabilities, or the rate vulnerability discovery is measured entirely by examining the source code itself, there is no mechanism to account for any additional variables that may affect the security of the product (or release) that come from its interaction with its environment, (e.g., the Honeymoon Effect, exploit market value, product market share, attacker interest, scripted tools, etc.).

With ASMs, the environment is defined as external to the software being modeled. In fact, the software's attack surfaces are isolated to the subset of the system that have direct input to the software. [MW08]

Yet, by its very nature, no piece of software operates in isolation. Software by one vendor is often dependent on hardware and firmware and software developed by others, including shared libraries, APIs and other second and third party applications. Simply put, software functions as part of a complex ecosystem. Moreover, in today's increasingly interconnected world, a single software application is likely to be running on a device which is connected to any number of other devices and

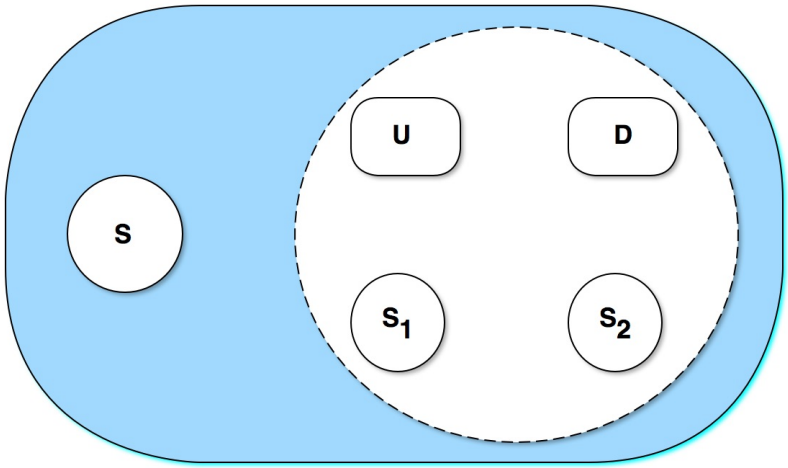


Figure 6.1: The Environment pictured as external to the system in Attack Surface Models. Courtesy of [MW04]

systems. [4] This complex relationship results in unexpected interactions that are not addressed by the models.

6.2.2 These Models Limit the Investment of Resources to Only One Stage of the Software Lifecycle

The result of these models is that limited to a particular release, their benefit is only applicable at one stage in that release's lifecycle. That is, these models expect high resource investment *either* early in the software planning and development stages *or* after completion in the software testing phase.

[4] such connections include networks, servers, printers...

VDMs are in the latter category. Like defect discovery models, VDMs can be useful for determining product readiness to release, but most can be applied to data acquired only after a vulnerability has been reported, in the hopes of determining the total number of remaining vulnerabilities or the likelihood of those remaining being exploited. Such passive strategies are not well suited for an environment with a fast moving, intelligent adversary. Additionally, these models offer little insight to help vendors protect software already released.

The ASMs fall into the first category. An ASM require significant analysis of the source code and its communication channels and call graph to determine values for its variables. Damage potential and damage effort must be estimated well in advance of product release, thus unable to incorporate new information from the constantly changing computer security environment. Additionally, ASMs high upfront resource investment conflicts with the current trend toward Agile programming methodology.

The difficulty of isolating software from its environment has long been recognized as a primary source of software compromise. [oHS06, Gur15] A model which can capture this behavior, must necessarily think of a product's security in relation to its ecosystem.

6.3 Introducing A Framework For A Dynamic/Adaptive Software Ecosystem Model (The DASEM Model)

"The excessive increase of anything causes a reaction in the opposite direction." (Plato)

6.3.1 The Requirements of a Security Ecosystem Model

If strictly software engineering models are not sufficient to describe the software security ecosystem, what characteristics must a model have to be useful in a complex and dynamic security ecosystem?

Observe that implicit in any ecosystem model, are four key concepts.

1. An organism in an ecosystem interacts in some way with its environment and with any (or all) other member organisms.

2. The system is dynamic, subject to change, (possibly disruptive), by any organism in the ecosystem, and that organisms within the ecosystem are acted upon by and affected by other members of the system.

3. Throughout its lifecycle, an organism competes for resources. This competition results in cycles. An organism can undergo any number of cycles. These cycles result in an "evolutionary arms race" between organisms in the ecosystem. We note that in the software security ecosystem, there are intelligent adversaries with competing goals, which adds complexity.

4. Changes to independent organisms in the ecosystem can result in alterations of the system that can be *harmful* or *beneficial* to any other member organism.

In order to address these concepts, a descriptive model of the software security ecosystem must have the following features:

- Mechanisms for monitoring the environment to detect and recognize attacks. Many useful tools are available such as, IDSes, Honeypots, Antivirus, some of the latest versions of these incorporate adaptive machine learning algorithms to speed response time, but as many of the extrinsic properties affecting whether a vulnerability will be found and exploited are still not well understood, much more research needs to be done.

- Mechanisms to detect extrinsic beneficial ecosystem changes.

- Analysis tools and models to understand the properties and effects of these changes. This includes, learning from new attacker techniques, determine virility(likelihood of spread), changes to attack surfaces, fuzzing for new similar vulnerabilities, criticality, cost to patch. Such tools include Attack Surface Metric models and Vulnerability Discovery models, Fuzz-testing. (N.B., although these models have limitations, they can still be valuable tools employed as part of a larger defensive strategy.)

- Mechanisms and Tools for Risk, ROI and Cost-benefit analysis. To adapt efficiently and economically to changes in the environment, it is necessary to answer questions such as where to place resources, (e.g., in fuzz-testing, patching, defect/vulnerability discovery), or patch design, e.g., should a patch fix a specific vulnerability, or should a full rewrite of modules be carried out to eliminate this class of attack from all future releases (examples of this include ASLR, Heap randomization and stack canaries.) [Mic16]

- Mechanisms for adaptation. Such methods should include Rapid-Release Cycles, and Continuous Patching or Update Cycles which operate without customer interaction. [Laf10, Cor13, Bak]

6.3.2 Is There A Model To Use As A Template?

Boyd's OODA Loop [Boy95] model appears to have many of the characteristics our model requires. Indeed, it has been shown to be quite useful when applied to the security of computer systems. Recent research in areas such as malware detection[Bra12, Pad16, Bil08], threat modeling[HAK07], metrics for security decision making[PR12] and designing dynamic security protocols[CBS12] have attempted to incorporate Boyd's active defensive strategy to preemptively predict and remove

vulnerabilities before they can be exploited, to contain an attacker, or to channel a successful attack to a lesser target to mitigate damage.

However, the OODA Loop model as specified by Boyd is not a plug-and-play fit to model the computer security ecosystem for a number of reasons.

First: Boyd applied his original model to a particular type of war fighter experience: Combat between fighter pilots. In fact, the OODA Loop model assumes scenarios where the attacker and defender can switch places., i.e., where the defender can attack back. There is no similar situation in the computer security world. Instead, *the computer security ecosystem divides into strictly offensive and strictly defensive sides.*

Second: The OODA Loop model is designed for a simple system. Combat between fighter planes is one on one or few on few. This is not the case in the computer security ecosystem. Unknown attackers, (i.e., the hostile organisms), are numerous. *The computer security ecosystem is asymmetric.*

Third: The OODA Loop models a competition between equals or at the very least against a known and well understood opponent. The attacker/defender arms race faced by any organism in the ecosystem of a software product is anything but a fight between equals. Attackers vary in resources, goals, abilities. *The computer security ecosystem is diverse and heterogeneous.*

Fortunately, with a few additions, the four stages of the OODA Loop model can be adapted to fit a dynamic computer security ecosystem. Starting with this as a framework, the top-level view of the model would seem to need only the addition of feedback loops returning to each of the intermediate stages, and the addition of a conduit for new information to pass directly into the third stage. Nonetheless, the model also needs some conceptual modifications to capture the interactions between

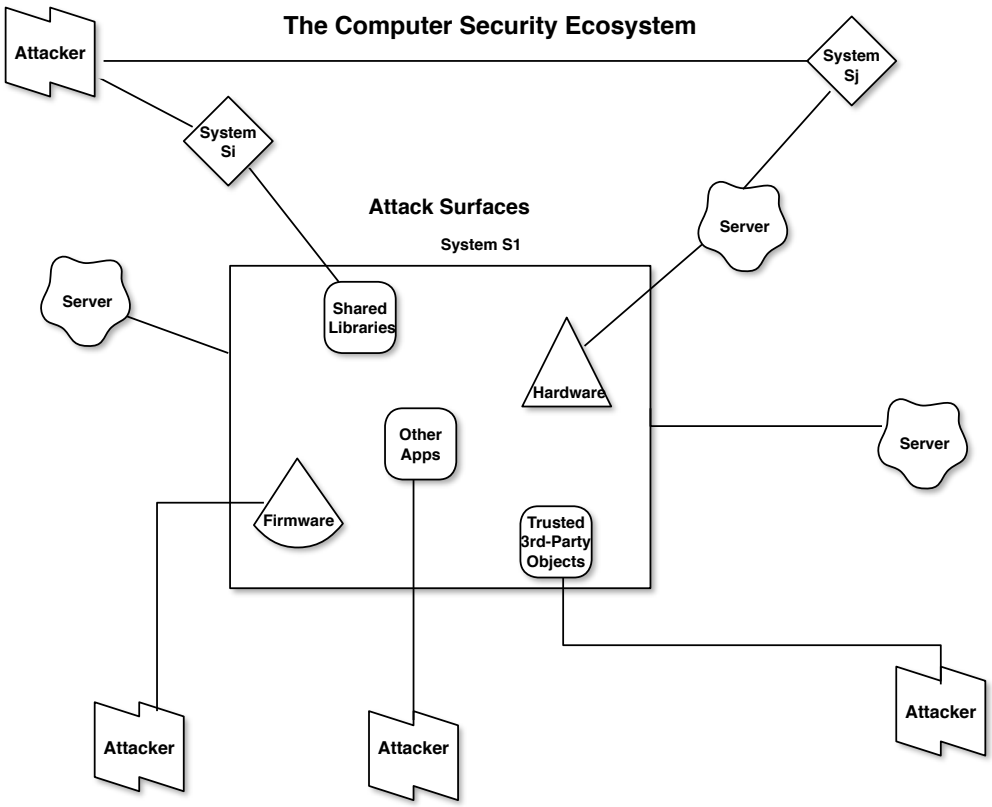


Figure 6.2: The Computer Security Ecosystem; Any system is vulnerable not just to external attack, but to attack by any compromised member of the ecosystem.

the hostile and benign organisms in the system.

Figure 6.2 describes the ecosystem of a typical software product. Note that in this model, a system is vulnerable to attack not just from external malicious organisms, but by compromised benign external organisms, and compromised third-party resources that may be considered part of the system itself and therefore allowed inside the security perimeter of the system. Figure 6.3 introduces a framework for a new type of model, a *Dynamic/Adaptive Security Ecosystem Model (DASEM)* and shows the modifications that would be necessary to make to the OODA Loop in order to capture the interactions of these *extrinsic properties* with the system. For clarity, the 4 stages have been renamed to fit their commonly used computer

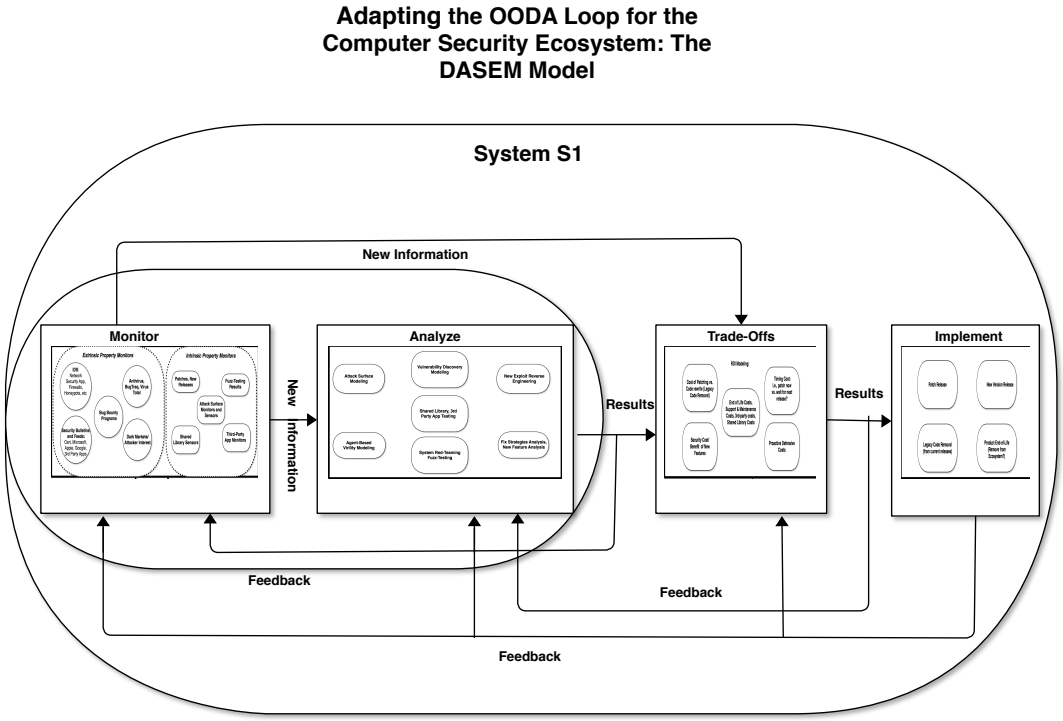


Figure 6.3: The addition of feedback loops between internal stages, the encapsulation of the Monitor and Analyze stages and the recognition that the encapsulated stages occur concurrently with the other stages in each iteration, rather than consecutively, are necessary to adapt the OODA Loop into a model to adequately describe a dynamic software security ecosystem.

security analogues. The *Monitor* (equivalent to Boyd's Observe) stage, the *Analyze* (equivalent to the Orient) stage, the *Trade-Offs* (equivalent to the Decide) stage,and the *Implement* (equivalent to the Act) stage. The additions to the model are not large, but they are significant in making it fit reality.

The first set of changes needed are the addition of feedback loops between the internal stages of the model. Specifically, these are new lines of communication necessary to provide information flow to the Analysis and Trade-Offs stages as well as the Monitor stage. This is necessary because in the security ecosystem, attacker

and defender interaction is not a one-on-one competition between relative equals. In computer security, systems are constantly under attack from varied and multiple sources simultaneously.

The next change is in how the 'loop' of the model functions. The chief characteristic of this model, unlike the OODA Loop, where the adherent moves sequentially through the model from the first stage to the last, the DASEM model does not operate as a closed loop.

Instead, the Monitor and Analyze stages actually operate *simultaneously and concurrently* with the Trade-Offs and Implement stages. After all, a software system shouldn't turn of its IDS or stop fuzz-testing for new vulnerabilities while it has reached the Trade-Off stage with regard to a vulnerability discovered in an earlier iteration.

The last important change is illustrated in Figure 6.4. In computer security, systems use many defensive tools to find vulnerabilities, and recognize attacks. The set of tools is made up of applications and devices that monitor intrinsic properties such as software testing scripts, and others that are focused on properties extrinsic to the system, such as IDSes and antivirus programs. These tools tend to be independent of each other, thus they provide an incomplete picture of the ecosystem. The increased information flow between the Monitor and Analyze stages provides for correlation between the information gleaned from disparate detection and the results obtained from the analyzation tools. The DASEM model allows for this by encapsulating the Monitor and Analysis stages together. This encapsulation signifies the biggest change to the original OODA Loop. The concurrent operation and intercommunication between the activities of these two stages increases the likelihood of recognizing whether the threat results from extrinsic or intrinsic properties. It also captures potentially malicious communication between hostile and trusted third-party members of the ecosystem that would be missed by typical detection tools.

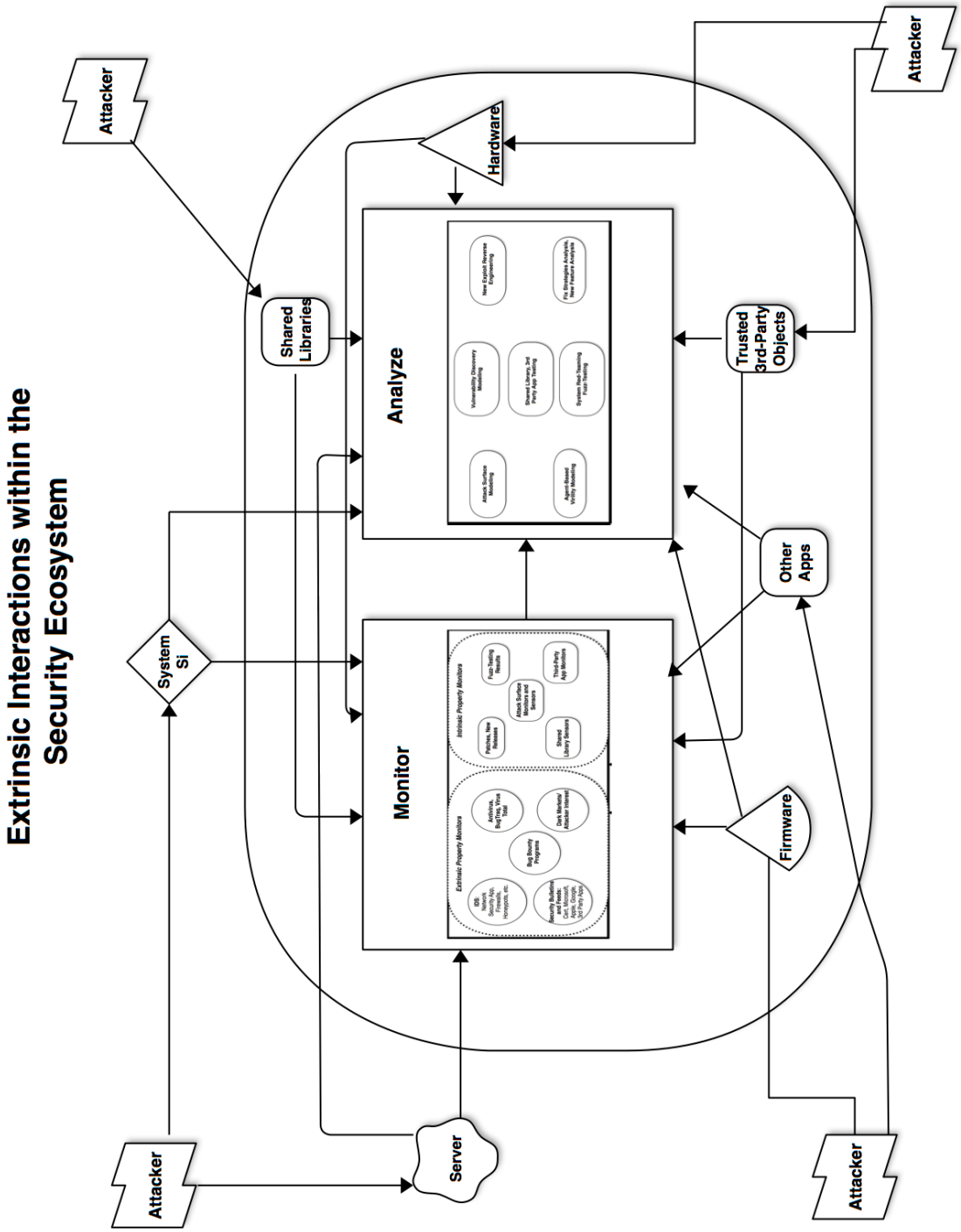


Figure 6.4: Close-up view of the Extrinsic Interactions of the Ecosystem with the Dynamic/Adaptive Security Ecosystem Model Monitor and Analysis Stages

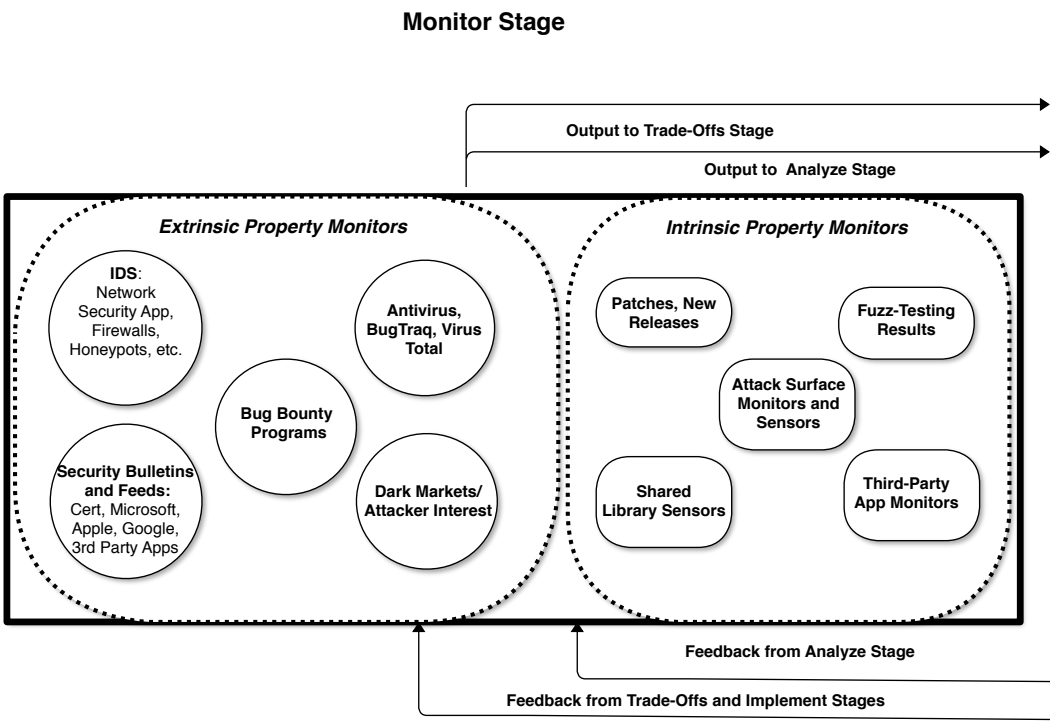


Figure 6.5: Dynamic/Adaptive Security Ecosystem Model: Monitor Stage

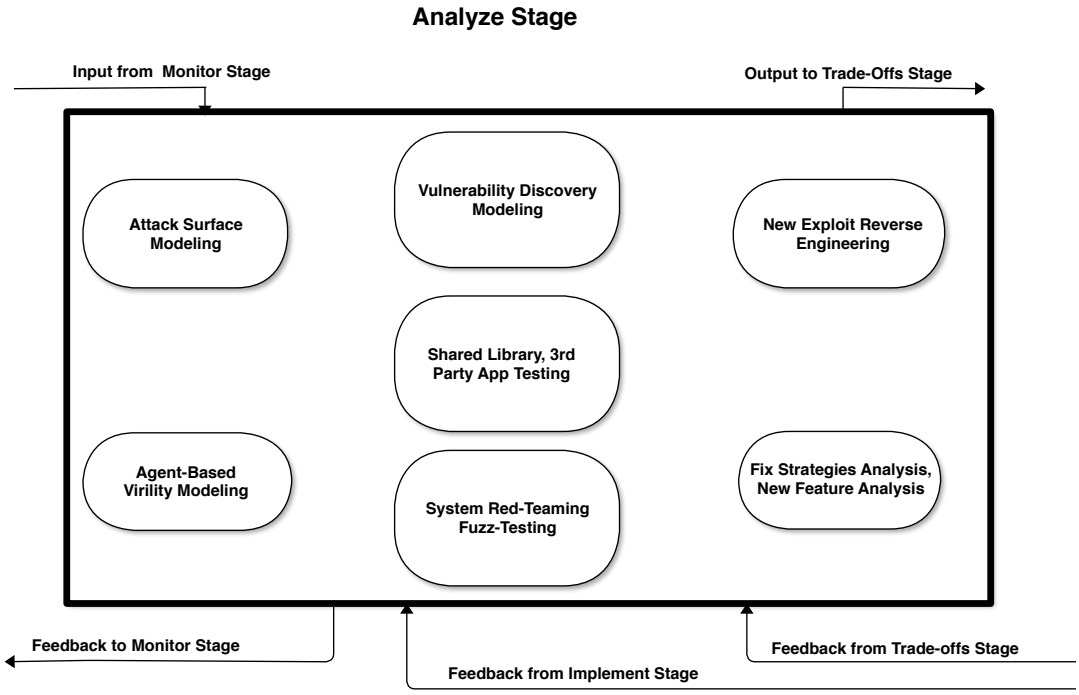


Figure 6.6: Dynamic/Adaptive Security Ecosystem Model: Analyze Stage

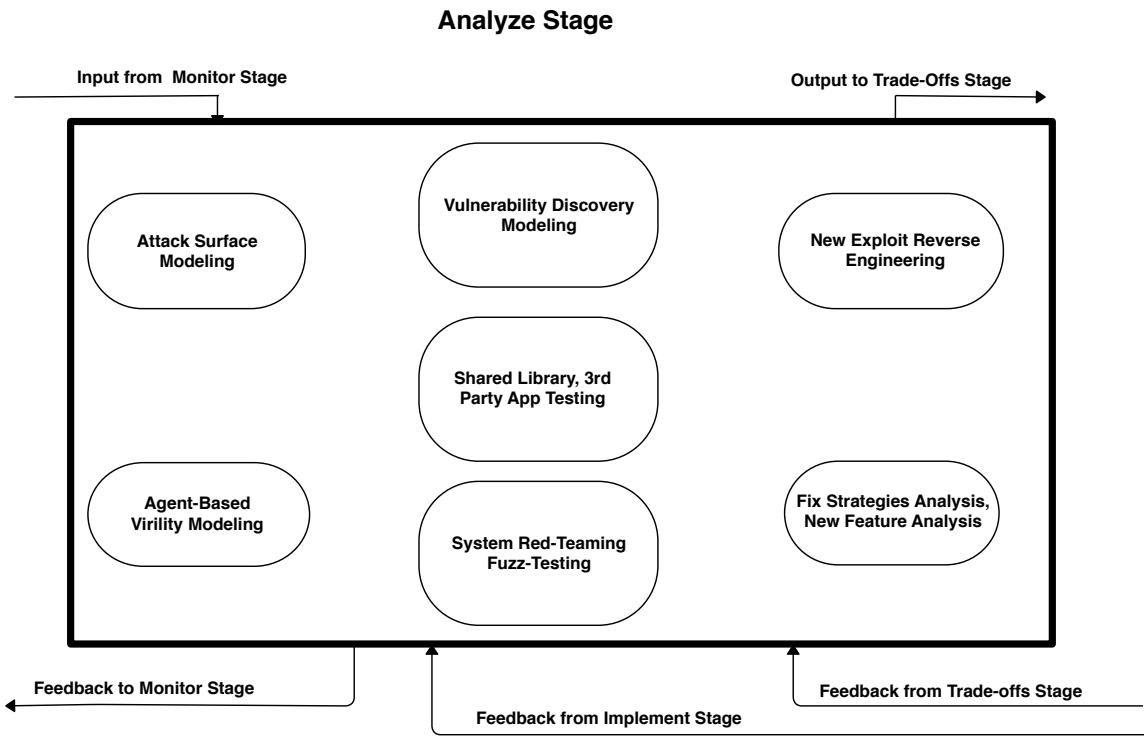


Figure 6.7: Dynamic/Adaptive Security Ecosystem Model: Trade-Offs Stage

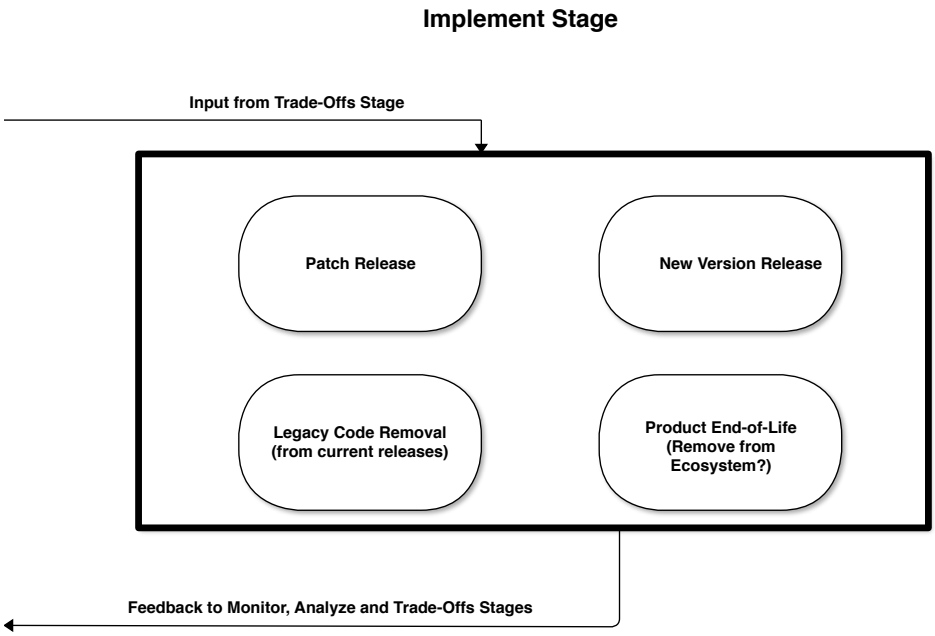


Figure 6.8: Dynamic/Adaptive Security Ecosystem Model: Implement Stage

Figures 6.5 and 6.6 illustrate the details of the individual Monitor and Analyze stages. Additional important features to note are that in both the Monitor and the Analysis stages, a developer needs to make use of as many of the current tools for attacker identification, vulnerability detection, vulnerability removal and system protection available to her. This includes keeping abreast of current research including an awareness of vulnerabilities found in other developer's products (the product modeled may be similarly vulnerable, or a compromise of that other developer's product may be used to attack this one).

In the Trade Offs stage, shown in Figure 6.7, not only should risks vs. benefits be studied, resources should be allocated, and decisions about what tools to retire or add to the toolboxes of the Monitor and Analyze stages should be made. The decisions made in this stage result in a set of change or actions to be taken.

Finally, Figure 6.8 shows the Implement stage. The important feature of this stage, is that what ever the set of actions that result from the Trade-offs stage, they should be executed with alacrity. Recognizing that an ecosystem is never static, it is obvious that adaptation needs to be done as soon after the decision to act as possible.

6.4 Discussion

"All have their worth and each contributes to the worth of the others."

(J.R.R. Tolkien, The Silmarillion)

A key concept behind DASE is not to model every single possible interaction between all members of the ecosystem for each iteration of the model. That is impossible even for any moderately complex system. The intention isn't to predict every possible vulnerability, or to identify every possible attack vector. Instead, the DASE model seeks to enhance communication and information flow to facilitate the best use of available resources. Such a framework would make it possible to decide

quickly whether a threat comes from an extrinsic or intrinsic property, to determine the characteristics of that threat, and to provide the data necessary to evaluate risk. The framework illustrated by the DASE model is designed to magnify the chances of recognizing the relationship between extrinsic and intrinsic properties, to analyze their effects singularly and in combination with other members of the ecosystem, and especially to make it possible to adapt quickly, implementing changes, making minimal and cost-effective trade-offs for negative properties and taking advantage of beneficial ones. The DASE model has not been validated, and is not expected to have predictive power, but any model capable of describing a computer ecosystem in the face of an (or possibly many) intelligent adversary, as well as providing insight into timely, low-cost adaptation must, at a minimum, possess these properties.

6.5 Final Thoughts

The limitations of prior models share a predominant characteristic, namely the presumption that the environment containing the software is static. That is, that the surrounding system in which the software functions and on which it may depend (such as hardware, firmware, communications buses or shared libraries), is fixed and unchanging, a blackbox about which the model makes a single, definite assumption. (e.g., in the case of ASMs: 'This is not a security risk', or 'This is an attack vector with a fixed potential-effort value'; in the case of VDMs: The environment is irrelevant to the model.)

This is understandable, since the software engineering models that measure functionality and reliability consider each developer's software product, even those incorporating open source and GPL licensed code in their product, independently of each other, regardless of how many of these products may be in operation on a single system at any given time. In these static models, *only intrinsic properties matter.* But

the phenomenon of the Honeymoon Effect describes an *extrinsic property.* Its existence suggests that the presence of living, intelligent adversaries in the environment means the software security ecosystem is *dynamic*, not static. The results of this dissertation provide evidence that there exists at least one extrinsic property that affects the robustness of the security of a software system. [5] This strongly suggests that the problems cannot be solved exclusively through software engineering means, and that Software Quality != Software Security.

"If you know your enemy and you know yourself you need not fear the results of a hundred battles. If you know yourself but not the enemy for every victory gained you will also suffer a defeat. If you know neither the enemy nor yourself you will succumb in every battle." (Sun Tzu, The Art of War: Chapter III - Strategic Attack)

[5] There may be many others, but one is sufficient to prove my point.

Appendix A

Glossary

Active Vulnerability: Vulnerability that affects a given version of software while that version is the most current available.

ASLR: Address Space Layout Randomization.

BUGID: Mozilla Firefox Defect Identifier.

CVE: Common Vulnerability Enumeration - Vulnerability numbering system employed by NVD.

DASEM Model: Dynamic/Adaptive Software Ecosystem Model.

DDR: Defect Density Rate.

DHS: Department of Homeland Security (USA).

DkD: Known Defect Density Measure.

ESR: Mozilla Firefox Extended Support Release

Extrinsic Property: Any characteristic of the environment in which the software operates (outside of the control of the product developer).

Foundational Version: The originating version of a software product.

Foundational Vulnerability: Vulnerability which affects the original codebase on which subsequent versions were based.

Honeymoon Effect: The unexpected grace-period wherein the time to discover and exploit vulnerabilities in new code appears to be dependent on the attacker's familiarity with the code.

Honeymoon Period: The time between release of software and the first discovery vulnerability.

Honeymoon Ratio: The ratio between the intervals of the Honeymoon Period and the period of time between first and second vulnerabilities.

Inactive Vulnerability: Vulnerability no longer exploitable as a result of being obsoleted by a new code release.

Intrinsic Property: Any property or characteristic of a software product that can directly controlled by the developer, such as, programming language, addition of features, patch release rate, etc.

ISO: International Standards Organization.

LOC: Lines of Code.

MFSA: Mozilla Firefox Security Advisory.

MTTB: Mean Time to Breakdown.

MTTF: Mean Time to Failure.

Negative Honeymoon Period: When the time to discovery of first vulnerability for a release is shorter than the time to discovery of subsequent vulnerabilities.

New Vulnerability: Vulnerability that affect the current version of code at the time of disclosure but that do not affect previous versions.

NVD: NIST National Vulnerability Database.

Positive Honeymoon Period: When the time to discovery of first vulnerability for a release exceeds the time to discovery of subsequent vulnerabilities.

Progressive Vulnerability: Vulnerability discovered and disclosed in new (non-legacy) code.

Regressive Vulnerability: Vulnerability discovered and disclosed in code after the version in which it was introduced has been obsoleted by a more recent version.

RRC: Rapid-Release Cycle Development Methodology; Also Mozilla Firefox Rapid-Release Versions.

SDL: Microsoft Security Development Lifecycle.

SDM: Software Defect Model.

SRM: Software Reliability Model.

SWE: Software Engineering (Model).

VDD: Vulnerability Density Discovery (Model).

VDR: Vulnerability Discovery Rate (Model).

VkD: Known Vulnerability Density Measure.

Vulnerability: a security flaw, glitch, or weakness that permits an attacker to reduce security assurance.

Zero-Day Vulnerability: A security threat that is known to an attacker which may or may not be known to a defender and for which no patch or security fix has been made available (also known as an 0-day).

Appendix B

Supplemental Data for the Honeymoon Effect

The results shown in chapter 3 display the data graphed on a log-scale. This appendix contains graphs of the same data displayed on a true-scale to provide readers with additional means to interpret the results. This appendix also contains additional statistical data concerning the days to first vulnerability.

Figure B.1 show OS honeymoon ratios on linear scale. Compare this to Figure 3.4.

Figure B.2 the Server applications honeymoon ratios on linear scale. Compare this to Figure 3.5.

Figure B.3 the User applications honeymoon ratios on linear scale. Compare this to Figure 3.6.

Figure B.4 the Open Source honeymoon ratios on linear scale. Compare this to Figure 3.7.

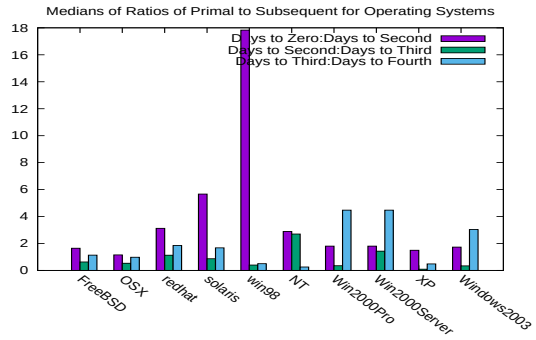


Figure B.1: Honeymoon ratios of p_0/p_{0+1}, p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for major operating systems. (Note that a figure over 1.0 indicates a positive honeymoon).

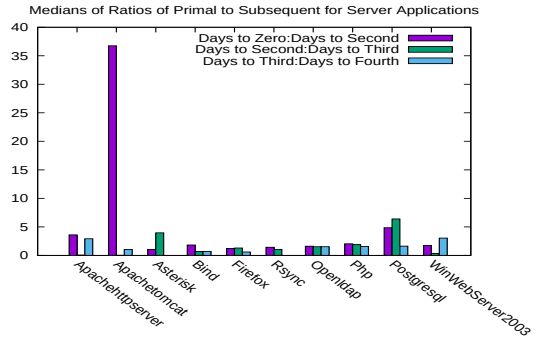


Figure B.2: Honeymoon ratio of p_0/p_{0+1}, p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for common server applications

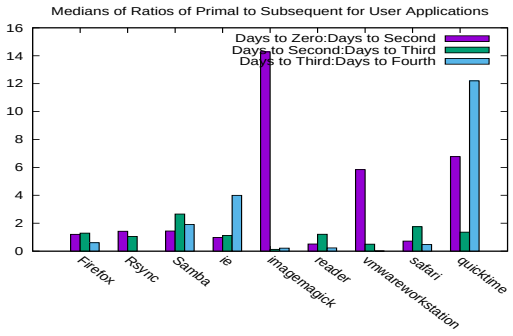


Figure B.3: Honeymoon ratios of p_0/p_{0+1}, p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for common user applications

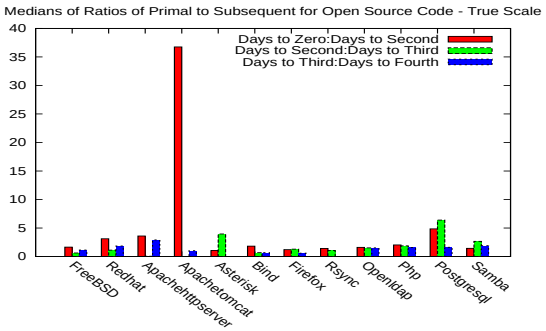


Figure B.4: Ratios of p_0/p_{0+1} to p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for open source applications

US

Figure B.5 the Closed Source honeymoon ratios on linear scale. Compare this

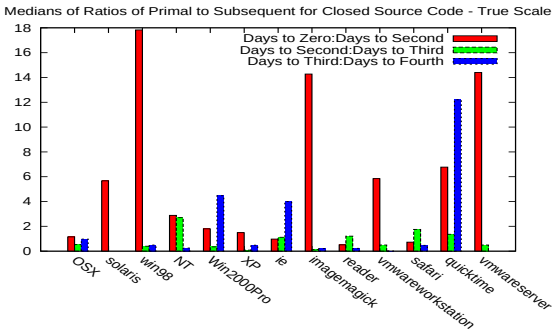


Figure B.5: Ratios of p_0/p_{0+1} to p_{0+1}/p_{0+2} and p_{0+2}/p_{0+3} for closed source applications

to Figure 3.8.

Bibliography

[Ada08] E. Adams. *Minimizing Cost Impact of Software Defects*. IBM Research Division, 2008.

[AFM00] William A. Arbaugh, William L. Fithen, and John McHugh. Windows of vulnerability: A case study analysis. *Computer*, 33(12):52–59, 2000.

[AG82] RM Anderson and DM Gordon. Processes influencing the distribution of parasite numbers within host populations with special emphasis on parasite-induced host mortalities. *Parasitology*, 85(02):373–398, 1982.

[Aki71] F Akiyama. An example of software system debugging. *Inf Processing 71*, 1971.

[AKTY06] Ashish Arora, Ramayya Krishnan, Rahul Telangand, and Yubao Yang. Empirical analysis of software vendors patching behavior, impact of vulnerability disclosure. Technical report, Carnegie Mellon University, 2006.

[Alm13] Ali Almossawi. How maintainable is the Firefox codebase?, May 2013. `http://almossawi.com/firefox/prose/`.

[AM08] O.H. Alhazmi and Y.K. Malaiya. Application of vulnerability discovery models to major operating systems. *IEEE Transactions on Reliability*, 57:14–22, 2008.

[Ama08] Amazon. Amazon software bestsellers. http://www.amazon.com/gp/bestsellers/software, September 2008.

[AMR05] O. H. Alhazmi, Y. K. Malaiya, and I. Ray. Security vulnerabilities in software systems: A quantitative perspective. In *Proc. Ann. IFIP WG11.3 Working Conference on Data and Information Security*, pages 281–294. Springer Verlag, 2005.

[AMR07] O.H. Alhazmi, Y.K. Malaiya, and I. Ray. Measuring, analyzing and predicting security vulnerabilities in software systems. *Computers & Security*, 26(3):219 – 228, 2007. http://www.sciencedirect.com/science/article/B6V8G-4MFTVD1-1/2/3956b58760fff238dd09c0526525e6d9.

[And01] Ross Anderson. Why information security is hard - an economic perspective. In *Computer Security Applications Conference, 2001. ACSAC 2001. Proceedings 17th Annual*, pages 358–365, 2001. http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=991552.

[And02] Ross Anderson. Security in open versus closed systems - the dance of Boltzmann, Coase and Moore. In *The Conference on Open Source Software Economics*, pages 1–15. MIT Press, 2002.

[Ang86] William S. Angerman. Coming full circle with Boyd's OODA loop ideas: An analysis of innovation diffusion and evolution. *AIR University Briefings*, 1986.

[Bak] Mitchell Baker. Mozilla blog. http://blog.lizardwrangler.com/2011/08/25/rapid-release-process/.

[BBC+10] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak,

and Dawson Engler. A few billion lines of code later: using static analysis to find bugs in the real world. *Communications of the ACM*, 53(2):66–75, 2010. http://www.bibsonomy.org/bibtex/29abe3708a9620e8f6513e76e5826e701/sjbutler.

[BBK78] Barry W Boehm, John R Brown, and Hans Kaspar. Characteristics of software quality. *TRW series of software technology*, 1978.

[BBvB+01] Kent Beck, Mike Beedle, Arie van Bennekum, Alistair Cockburn, Ward Cunningham, Martin Fowler, James Grenning, Jim Highsmith, Andrew Hunt, Ron Jeffries, Jon Kern, Brian Marick, Robert C. Martin, Steve Mellor, Ken Schwaber, Jeff Sutherland, and Dave Thomas. Manifesto for agile software development, 2001. http://www.agilemanifesto.org/.

[BCH+95] Barry Boehm, Bradford Clark, Ellis Horowitz, Chris Westland, Ray Madachy, and Richard Selby. Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering*, 1:57–94, 1995.

[BD01] Clark B. and Zubrow D. How good is the software: A review of defect prediction techniques. Technical report, Carnegie Mellon Software Engineering Institute, 2001. https://www.sei.cmu.edu/library/assets/defect-prediction-techniques.pdf.

[Bea16] N. Beauchesne. Own a printer, own a network with point and print drive-by, July 2016. http://blog.vectranetworks.com/blog/microsoft-windows-printer-wateringhole-attack.

[Bil08] Daniel Bilar. Noisy defenses: subverting malware's OODA loop. In *Proceedings of the 4th annual workshop on Cyber Security and Information Intelligence Research: Developing strategies to meet the Cyber*

Security and Information Intelligence challenges ahead, page 9. ACM, 2008.

[BK04] Konstantin Beznosov and Philippe Kruchten. Towards agile security assurance. In *Proceedings of the 2004 Workshop on New Security Paradigms*, pages 47–54. ACM, 2004.

[BM11] Dan Breznitz and Michael Murphree. *Run of the red queen: Government, innovation, globalization, and economic growth in China*. Yale University Press, 2011.

[Boy72] John R. Boyd. Letter to his wife, 1972.

[Boy76] John R. Boyd. *Destruction and Creation*, chapter 23, pages 317–326. Little, Brown and Company, September 1976. `http://www.belisarius.com/modern_business_strategy/boyd/destruction/destruction_and_creation.htm`.

[Boy87] John R. Boyd. Organic design for command and control. *US Air Force Briefings*, May 1987.

[Boy95] John R. Boyd. The essence of winning and losing. *US Air Force Briefings*, June 1995.

[Boy06] John R. Boyd. The strategic game of ? and ? *US Air Force Briefings*, June 2006. `http://www.dnipogo.org/boyd/strategic_game.pdf`.

[BP66] Jonh R. Boyd and Christie Thomas P. Energy-manuverability. *US Army Command and General Staff College*, 1966.

[Bra12] Branco, R.R., et al. A scientific (but non academic) study of how malware employs anti-debugging, anti-disassembly and anti-virtualization technologies, July 2012. `http://www.kernelhacking.com/rodrigo/docs/blackhat2012-paper.pdf`.

[Bri] Derek A. Brink. Security and the software development lifecycle: Secure at the source. download.microsoft.com/download/9/D/4/9D403333-C4F6-4770-A330-89661BE545CF/Aberdeen_SecureSource.pdf.

[Bro95a] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, August 1995. http://www.amazon.ca/exec/obidos/redirect?tag=citeulike09-20&path=ASIN/0201835959.

[Bro95b] Frederick P. Brooks. *The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition*. Addison-Wesley Professional, August 1995.

[Bro14] Phil Brock. The agile alliance, July 2014. https://www.agilealliance.org/.

[CA05] Omar Alhazmi Colorado and Omar H. Alhazmi. Quantitative vulnerability assessment of systems software. In *Proc. Annual Reliability and Maintainability Symposium*, pages 615–620, 2005.

[CBS10] Sandy Clark, Matt Blaze, and Jonathan Smith. Blood in the water: Are there honeymoon effects outside software? In *In Proceedings of the 18th Cambridge International Security Protocols Workshop -pending publication*. Springer, 2010.

[CBS12] Sandy Clark, Matt Blaze, and Jonathan M Smith. The casino and the ooda loop. In *International Workshop on Security Protocols*, pages 60–63. Springer, 2012.

[CCBS14] Sandy Clark, Michael Collis, Matt Blaze, and Jonathan M Smith. Moving Targets: Security and Rapid-Release in Firefox. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, pages 1256–1266. ACM, 2014.

[CEF+06] Benjamin Cox, David Evans, Adrian Filipi, Jonathan Rowanhill, Wei Hu, Jack Davidson, John Knight, Anh Nguyen-tuong, and Jason Hiser. N-variant systems: A secretless framework for security through diversity. In *In Proceedings of the 15th USENIX Security Symposium*, pages 105–120, 2006.

[CFBS10] Sandy Clark, Stefan Frei, Matt Blaze, and Jonathan Smith. Familiarity breeds contempt: the honeymoon effect and the role of legacy code in zero-day vulnerabilities. In *Proceedings of the 26th Annual Computer Security Applications Conference*, ACSAC '10, pages 251–260, New York, NY, USA, 2010. ACM. http://dx.doi.org/10.1145/1920261.1920299.

[Che13] X. Chen. Aslr bypass apocalypse in recent zero-day exploits, October 2013. https://www.fireeye.com/blog/threat-research/2013/10/aslr-bypass-apocalypse-in-lately-zero-day-exploits.html.

[CLDL99] Peter Coad, Eric LeFebrve, and Jeff De Luca. Feature-driven development. *Java Modeling in Color with UML*, pages 182–203, 1999.

[CNE08] CNET. Most popular windows downloads. *CNET*, September 2008.

[Coa11] Michael Coates. Security Evolution - Bug Bounty Programs for Web Applications, September 2011. http://www.slideshare.net/michael_coates/bug-bounty-programs-for-the-web.

[Con04] Kieran Conboy. Toward a conceptual framework of agile methods: a study of agility in different disciplines. In *Extreme Programming And Agile Methods - XP/ Agile Universe 2004, Proceedings*, pages 37–44. ACM Press, 2004.

[Cor04] Rober Coram. *Boyd: The Fighter Pilot Who Changed the Art of War*. Little, Brown and Company, 2004.

[Cor08] Microsoft Corporation. Microsoft security development lifecycle, September 2008. http://www.microsoft.com/security/sdl/benefits/measurable.aspx.

[Cor09] Microsoft Corporation. Microsoft security development lifecycle for agile. *Microsoft Technet Reports*, 2009. http://www.microsoft.com/security/sdl/discover/sdlagile-onetime.aspx.

[Cor13] Microsoft Corporation. Microsoft rapid release. http://www.microsoft.com/en-us/news/speeches/2013/06-26build2013.aspx, 2013.

[Cri12] Common Criteria. Common criteria for information technology security evaluation. Technical report, Common Criteria, September 2012.

[Cro79] P. Crosby. *Quality is Free*. McGraw-Hill, 1979.

[CS97] Michael A. Cusumano and Richard W. Selby. How Microsoft Builds Software. *Communications of the ACM*, 40:53–61, June 1997.

[CVE08] CVE. Common vulnerabilities and exposures. http://cve.mitre.org, 2008.

[Dan16] A. Danial. CLOC - Software to Count Lines of Code, July 2016. https://github.com/AlDanial/cloc.

[Dre04] A.S. Dreier. *Strategy, Planning & Litigating to Win*. Conatus Press, 2004.

[E.10] Bashar E. Pass-the-hash attacks: Tools and mitigation, January 2010. https://www.sans.org/reading-room/whitepapers/testing/pass-the-hash-attacks-tools-mitigation-33283.

[EGK+01] Stephen G Eick, Todd L Graves, Alan F Karr, J Steve Marron, and Audris Mockus. Does code decay? Assessing the Evidence From Change Management Data. *Software Engineering, IEEE Transactions on*, 27(1):1–12, 2001.

[FAW13] M. Finifter, D. Akhawe, and D. Wagner. An Empirical Study of Vulnerability Reward Programs. In *22nd USENIX Security Symposium*, 2013.

[fDCP16] Centers for Disease Control and Prevention. Malaria, July 2016.

[FM06] Stefan Frei and Martin May. The speed of (in)security. BlackHat USA, 2006. `http://www.techzoom.net/papers/blackhat_speed_of_insecurity_2006.pdf`.

[FNSS99] Norman E. Fenton, Martin Neil, Ieee Computer Society, and Ieee Computer Society. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25:675–689, 1999.

[Fou14] Mozilla Foundation. Mozilla Firefox ESR Overview, 2014. `https://www.mozilla.org/en-US/firefox/organizations/faq/`.

[Fra15] Ivan Fratric. Dude where's my heap, June 2015. `https://googleprojectzero.blogspot.com/2015/06/dude-wheres-my-heap.html`.

[Fre09] Stefan Frei. *Security Econometrics - The Dynamics of (In)Security*. Eth zurich, dissertation 18197, ETH Zurich, 2009. `http://www.techzoom.net/publications/security-econometrics/`.

[fS11] International Organization for Standardi. Iso/iec 25010:2011 îŇŘ systems and software engineering, systems and software quality

requirements and evaluation (square), system and software quality models. http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=35733, 2011.

[Gaf14] Marc Gaffan. Did shellshock hit one billion servers?, October 2014. https://www.incapsula.com/blog/shellshock-one-billion-servers.html.

[Gal86] J. Gall. *Systemantics: How Systems Work and especially How They Fail*. General Systemantics Press of Ann Arbor, 1986.

[Gar00] David A. Garwin. Learning in action. *Harvard Business School Press*, page 10, 2000.

[Ger91] Connie J. G. Gersick. Revolutionary change theories: A multilevel exploration of the punctuated equilibrium paradigm. *The Academy of Management Review*, 16(1):10–36, 1991. http://www.jstor.org/stable/258605.

[Gre01] J. Greene. Software quality assurance (sqa) of management processes using the sei core measures. Technical report, Quantitative Software Management LTD, 2001.

[Gre15] Tim Green. Biggest data breaches of 2015, December 2015. http://www.networkworld.com/article/3011103/security/biggest-data-breaches-of-2015.html.

[Gro70] Boston Consulting Group. *Perspectives on Experience*. Boston Consulting Group, 1970.

[GS05] Rajeev Gopalakrishna and Eugene H. Spafford. A Trend Analysis of Vulnerabilities. *CERIAS Tech Report 2005-05*, May 2005.

[Gur15] K. Guruswamy. Critical Vulnerability in Firefox Built-in PDF Viewer, August 2015. https://www.menlosecurity.com/blog/critical-vulnerability-in-firefox-built-in-pdf-viewer.

[HAK07] Kjetil Haslum, Ajith Abraham, and Svein Knapskog. Dips: A framework for distributed intrusion prediction and prevention using hidden markov models and online fuzzy risk assessment. In *Third International Symposium on Information Assurance and Security*, pages 183–190. IEEE, 2007.

[Har14] Duncan Harris. Oracle software security assurance. Technical report, Oracle, 2014. http://www.oracle.com/us/support/assurance/overview/index.html.

[Har15] Justin Harvey. 2015: The year of the breach, December 2015. http://www.threatgeek.com/2015/12/2015-the-year-of-the-breach.html.

[Hau09] Duncan Haughey. Project smart: Waterfall v agile: How should i approach my software development project?, September 2009. http://www.projectsmart.com/articles/waterfall-v-agile-how-should-i-approach-my-software-development-project.php.

[Hig13] Jim Highsmith. *Adaptive software development: a collaborative approach to managing complex systems.* Addison-Wesley, 2013.

[Hil15] H. Hillaker. Tribute to john r. boyd. *Code Magazine*, January 2015. http://www.codeonemagazine.com/f16_article.html?item_id=156.

[HL06] Michael Howard and Steve Lipner. *The Security Development Lifecycle.* Microsoft Press, May 2006. http://www.microsoft.com/security/sdl/resources/publications.aspx.

[HM95] Jun Han and Claudio Morag. The influence of the sigmoid function parameters on the speed of backpropagation learning. *Natural to Artificial Neural Computation*, 1995.

[Hol16] David Holmes. Was 2015 the year of breach fatigue?, January 2016. `http://www.securityweek.com/was-2015-year-breach-fatigue`.

[How97] John D Howard. An analysis of security incidents on the internet 1989-1995. Technical report, DTIC Document, 1997.

[How03] Michael Howard. Fending off future attacks by reducing attack surface. *DLB ModDigital Microsoft*, 2003. `http://dlbmodigital.microsoft.com/ppt/012711_JTaylor.pdf`.

[HPW05] Michael Howard, Jon Pincus, and Jeannette M Wing. Measuring relative attack surfaces. In *Computer Security in the 21st Century*, pages 109–137. Springer, 2005.

[HR06] Wilhelm Hasselbring and Ralf Reussner. Toward trustworthy software systems. *Computer*, 39(4):91–92, 2006.

[HT81] R. A. Harvey and D. R. Towill. Applications of learning curves and progress functions: Past, present, and future. *Industrial Applications of Learning Curves and Progress Functions*, pages 1–15, 1981.

[iDe] iDefense. Vulnerability contributor program. `http://labs.idefense.com/vcp`.

[Inc16] Synopsis Inc. Static code analysis. *Coverity*, 2016. `https://www.synopsys.com/software/coverity`.

[Ins96] Software Engineering Institute. Vulnerability in ncsa/apache cgi example code, March 1996. `https://www.cert.org/historical/advisories/CA-1996-06.cfm?`

[Ins11] Software Engineer Insider. What is software engineering, 2011. http://www.softwareengineerinsider.com/articles/what-is-software-engineering.html.

[IOfS01] ISO International Organization for Standardization. Software engineering, product quality, part 1: Quality model, June 2001. http://www.iso.org/iso/catalogue_detail.htm?csnumber=22749.

[Ion12] Alex Ionescu. Unravelling windows 8 security and the arm kernel. *BreakPoint 2012 Ruxcon*, October 2012. http://2012.ruxconbreakpoint.com/assets/Uploads/bpx/alex-breakpoint2012.pdf.

[ISO07] ISO/IEC. Software engineering, Software product Quality Requirements and Evaluation (SQuaRE): Measurement reference model and guide . Technical Report 25020(E), ISO/IEC, May 2007. http://www.iso.org.

[ISO11] ISO. Systems and software engineering: Systems and software quality requirements and evaluation (square): System and software quality models, March 2011. http://www.iso.org/iso/catalogue_detail.htm?csnumber=35733.

[JB11] Capers Jones and Olivier Bonsignour. *The economics of software quality*. Addison-Wesley Professional, 2011.

[JCB+11] Neil Johnson, Spencer Carran, Joel Botner, Kyle Fontaine, Nathan Laxague, Philip Nuetzel, Jessica Turnley, and Brian Tivnan. Pattern in escalations in insurgent and terrorist activity. *Science Magazine*, 333, July 2011.

[Jel00] George Jelen. SSE CMM Security Metrics. *NIST and CSSPAB Workshop*, 2000.

[JHWT13] Kumi Jinzenji, Takashi Hoshino, Laurie Williams, and Kenji Takahashi. Empirical study of software quality evaluation in agile methodology using traditional metrics. *International Symposium on Software Reliability Engineering (ISSRE)*, 2013.

[JM12] K. Johnson and Matt Miller. Exploit Mitigation Improvements in Windows 8, July 2012. `https://www.blackhat.com/html/bh-us-12/bh-us-12-briefings.html`.

[JMS08] Pankaj Jalote, Brendan Murphy, and Vibhu Saujanya Sharma. Post-release reliability growth in software products. *ACM Trans. Softw. Eng. Methodol.*, 17(4):1–20, 2008. `http://doi.acm.org/10.1145/13487689.13487690`.

[JO97] E. Jonsson and T. Olovsson. A quantitative model of the security intrusion process based on attacker behavior. *IEEE Transactions on Software Engineering*, 23(4):235–245, Apr 1997.

[Jur98] J.M. Juran. *Juran's Quality Control Handbook*. McGraw-Hill, 1998.

[Kan02] Stephen Kan. *Metrics and Models in SW Quality Engineering, Second Edition*. Addison-Wesley, 2002.

[KDZ08] Martin Kunz, Reiner R Dumke, and Niko Zenker. Software metrics for agile software development. In *19th Australian Conference on Software Engineering (ASWEC 2008)*, pages 673–678. IEEE, 2008.

[Ken99] Kristopher Kendall. A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems. In *DARPA OFF-LINE INTRUSION DETECTION EVALUATION, Proceedings Darpa Information Survivability Conference and Exposition (DISCEX), VOL*, pages 12–26. DARPA, 1999.

[KMH08] Hossein Keramati and S-H Mirian-Hosseinabadi. Integrating software development security activities with agile methodologies. In *Computer Systems and Applications, 2008. AICCSA 2008. IEEE/ACS International Conference on*, pages 749–754. IEEE, 2008.

[Kot02] Jim Kotnour. Leadership mechanisms for enabling learning within project teams. *Proceedings from the Third European Conference on Organizational Knowledge, Learning and Capabilities*, 2002. http://apollon1.alba.edu.gr/oklc2002/proceedings/pdf_files/id340.pdf.

[Kou16] J. Kouns. A Record Year For Vulnerabilities, March 2016. https://blog.osvdb.org/2016/03/15/2015-a-record-year-for-vulnerabilities/.

[Laf10] Anthony Laforge. Release Early, Release Often, July 2010. http://blog.chromium.org/2010/07/release-early-release-often.html.

[Lin03] Henry Linger. *Constructing the Infrastructure for the Knowledge Economy:Methods and Tools, Theory and Practice*. Springer Science and Business Media, 2003.

[Loh15] Daniel Lohrmann. 2015: The year data breaches became intimate, December 2015. http://www.govtech.com/blogs/lohrmann-on-cybersecurity/2015-the-year-data-breaches-became-intimate.html.

[LS13] Xiaoning Li and Peter Szor. Emerging stack pivoting exploits bypass common security, May 2013. https://blogs.mcafee.com/mcafee-labs/emerging-stack-pivoting-exploits-bypass-common-security/.

[Ltd14] Codenomicon Ltd. The heartbleed bug. *Codenomicon*, May 2014. `http://heartbleed.com/`.

[Mar15] Elliot Maras. 2015: The year of the breach; close to 200 million personal records exposed, December 2015. `https://hacked.com/2015-year-breach-close-200-million-personal-records-exposed/`.

[MC09] Gary McGraw and Brian Chess. The building security in maturity model(bsimm). In *Proceedings of the 18th USENIX Security Symposium (USENIX Security '09)*, Montreal, Canada, August 2009.

[McG03] Gary McGraw. From the ground up: The dimacs software security workshop. *IEEE Security & Privacy*, 1(2):59–66, 2003.

[McG10] Gary McGraw. Software security touchpoint: Architectural risk analysis. Technical report, Cigital, 2010. `http://www.cigital.com/presentations/ARA10.pdf`.

[McI68] M.C. McIlroy. Mass producted software components. *Report to Scientific Affairs Division, NATO*, October 1968.

[MCKS04] Parastoo Mohagheghi, Reidar Conradi, Ole M Killi, and Henrik Schwarz. An empirical study of software reuse vs. defect-density and stability. In *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 282–291. IEEE, 2004.

[MD99] Yashwant K Malaiya and Jason Denton. Requirements volatility and defect density. In *Software Reliability Engineering, 1999. Proceedings. 10th International Symposium on*, pages 285–294. IEEE, 1999.

[MD06] Ernest Mnkandla and Barry Dwolatzky. Defining agile software quality assurance. In *Software Engineering Advances, International Conference on*, pages 36–36. IEEE, 2006.

[Mic10] Microsoft. Code reuse in Microsoft Internet Explorer architecture. *Microsoft Internet Explorer Architecture*, 2010. http://msdn.microsoft.com/en-us/library/aa741312(VS.85).aspx.

[Mic12] Microsoft. Microsoft security bulletin ms12-074 - critical. Technical report, Microsoft, November 2012. https://technet.microsoft.com/en-us/library/security/ms12-074.aspx.

[Mic13] Microsoft. Windows 8.1 security improvements. Technical report, Microsoft, October 2013. https://technet.microsoft.com/en-us/windows/jj983723.aspx.

[Mic16] Microsoft. Exploit Mitigations in Windows, October 2016. https://www.microsoft.com/security/sir/strategy/default.aspx#!section_3_3.

[Mil07] Matt Miller. A brief history of exploitation techniques and mitigations on windows. http://www.hick.org/~mmiller/presentations/misc/exploitation_techniques_and_mitigations_on_windows.pdf, 2007.

[Mil08] Matt Miller. The evolution of microsoft's exploit mitigations, 2008. http://technet.microsoft.com/en-us/security/dd285253.aspx.

[Mit97] Tom Mitchell. *Machine Learning*. WCB-McGraw-Hill, 1997.

[MMRJ+05] Gary M McGraw, Nancy R Mead, Samuel T Redwine Jr, Ronda R Henning, Linda Ibrahim, Steven Hofmeyr, WS Harrison, Nadine Hanebutte, Paul W Oman, Jim Alves-Foss, et al. Crosstalk: The journal of defense software engineering. volume 18, number 10. Technical report, DTIC Document, 2005.

[MMW+05] J.D. Meier, Alex Mackman, Blaine Wastell, Prashant Bansode, Andy Wigley, and Kishore Gopalan. Security guidelines for .net framework version 2.0. Technical report, Microsoft, October 2005. http://msdn.microsoft.com/en-us/library/aa480477.aspx.

[Moz04] Mozilla. Mozilla Foundation releases the highly anticipated Firefox Web Browser. Technical report, Mozilla Foundation, November 2004. https://blog.mozilla.org/press/2004/11/mozilla-foundation-releases-the-highly-anticipated-mozilla-firefox-1-0-web-browser/.

[Moz13a] Mozilla. Bugzilla@Mozilla. Technical report, Mozilla Foundation, September 2013. https://bugzilla.mozilla.org/.

[Moz13b] Mozilla. Mozilla foundation security advisories, September 2013. https://www.mozilla.org/security/announce/.

[MW04] Pratyusa Manadhata and Jeannette M. Wing. Measuring a system's attack surface. Technical report, Carnegie-Mellon University, 2004.

[MW08] Pratyusa K. Manadhata and Jeannette M. Wing. An attack surface metric. Technical report, Carnegie-Mellon University, 2008.

[Mye96] J. Myers. Cgi security: Escape newlines. http://www.securityfocus.com/archive/1/6370, February 1996. http://www.securityfocus.com/archive/1/6370.

[NB05] Nachiappan Nagappan and Thomas Ball. Use of relative code churn measures to predict system defect density. In *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pages 284–292. IEEE, 2005.

[Nig11] Johnathan Nightingale. Mozilla blog post future releases, 2011. https://blog.mozilla.org/futurereleases/2011/07/19/every-six-weeks/.

[NIS08] NIST. National vulnerability database. http://nvd.nist.gov, 2008.

[NIS14] NIST. National vulnerability database. https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-7169, September 2014.

[OCJ09] Jon Oberheide, Evan Cooke, and Farnam Jahanian. If it ain't broke, don't fix it: Challenges and new directions for inferring the impact of software patches. *HotOS*, 2009.

[OEGQ07] Hector M Olague, Letha H Etzkorn, Sampson Gholston, and Stephen Quattlebaum. Empirical validation of three software metrics suites to predict fault-proneness of object-oriented classes developed using highly iterative or agile software development processes. *IEEE Transactions on software Engineering*, 33(6), 2007.

[oHS06] Department of Homeland Security. *SECURITY IN THE SOFTWARE LIFECYCLE: Making Software Development Processes– and Software Produced by Them– More Secure*. Department of Homeland Security, 2006. http://resources.sei.cmu.edu/asset_files/WhitePaper/2006_019_001_52113.pdf.

[oHS11a] Department of Homeland Security. Blueprint for a secure cyber future. *Department of Homeland Security Publications*, November 2011. https://www.dhs.gov/xlibrary/assets/nppd/blueprint-for-a-secure-cyber-future.pdf.

[oHS11b] Department of Homeland Security. *Enabling Distributed Security in Cyberspace, Building a Healthy and Resilient Cyber Ecosystem with*

Automated Collective Action. Department of Homeland Security, March 2011. https://www.dhs.gov/xlibrary/assets/nppd-cyber-ecosystem-white-paper-03-23-2011.pdf.

[Oli00] Richard W Oliver. Real time strategy: The seven laws of e-commerce strategy. *Journal of Business Strategy*, 21(5):8–10, 2000.

[OM97] OĆonnor and McDermott. *The Art of Systems Thinking*. Thorstons, 1997.

[One96] Aleph One. Smashing the stack for fun and profit. *Phrack magazine*, 7(49):14–16, 1996.

[OS06] Andy Ozment and Stuart E. Schechter. Milk or wine: does software security improve with age? In *USENIX-SS'06: Proceedings of the 15th USENIX Security Symposium*, Berkeley, CA, USA, 2006. USENIX Association.

[OS12a] J. Obes and J. Schuh. A Tale of Two Pwnies Part 1, May 2012. https://blog.chromium.org/2012/05/tale-of-two-pwnies-part-1.html.

[OS12b] J. Obes and J. Schuh. A Tale of Two Pwnies Part 2, May 2012. https://blog.chromium.org/2012/05/tale-of-two-pwnies-part-2.html.

[OSC13] T. Ormandy, A. Souchet, and B. Campbell. Ms13-005 hwnd_broadcast low to medium integrity privilege escalation. Technical report, Rapid7, February 2013. https://www.rapid7.com/db/modules/exploit/windows/local/ms13_005_hwnd_broadcast.

[Osi06] Frans Osinga. *Science, Strategy and War*. Routledge, 2006.

[Ozm07] Andy Ozment. Improving vulnerability discovery models. In *QoP '07: Proceedings of the 2007 ACM Workshop on Quality of Protection*, pages 6–11, New York, NY, USA, 2007. ACM. http://doi.acm.org/10.1145/1314257.1314261.

[Pad16] D. Padon. Hack in the box: Malware disguises itself to infiltrate your device, May 2016. http://blog.checkpoint.com/2016/05/25/hack-in-the-box-malware-disguises-itself-to-infiltrate-your-device/.

[PR12] James Pettigrew and Julie Ryan. Making successful security decisions: A qualitative evaluation. *IEEE Security & Privacy*, 1(10):60–68, 2012.

[Put78] Lawrence H. Putnam. A general empirical solution to the macro software sizing and estimating problem. *IEEE transactions on Software Engineering*, 4(4):345–361, 1978.

[Rac96] L.B.S. Raccoon. A learning curve primer for software engineers. *SIGSOFT Softw. Eng. Notes*, 21(1):77–86, January 1996. http://doi.acm.org/10.1145/381790.381805.

[Ram12] Muthu Ramachandran. Guidelines based software engineering for developing software components. *Journal of Software Engineering and Applications*, 2012.

[Ran10] Sam Ransbotham. An empirical analysis of exploitation attempts based on vulnerabilities in open source software. In *Workshop on the Economics of Information Security (WEIS)*, June 2010.

[Ray99] Eric Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999.

[(RB16] Risk Based Security (RBS). 2015, A Record Year For Vulnerabilities, March 2016. https://www.riskbasedsecurity.com/2016/03/2015-a-record-year-for-vulnerabilities/.

[Rei02] M. Reilly. Icat vulnerability database, September 2002.

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

[Ric04] Chet Richards. *Certain to Win: the Strategy of John Boyd, Applied to Business*. Xlibris Corporation, 2004.

[Ric08] David Rico. What is the roi of agile vs. traditional methods? an analysis of xp, tdd, pair programming, and scrum (using real options). *Semantic Scholar*, 2008.

[Rod08] Patricia L Roden. *An examination of stability and reusability in highly iterative software*. University of Alabama in Huntsville, 2008.

[Ros11] D. Rosenberg. Itś bugs all the way down, September 2011. http://vulnfactory.org/blog/2011/09/21/defeating-windows-8-rop-mitigation/.

[RVEM07] Patricia L Roden, Shamsnaz Virani, Letha H Etzkorn, and Sherri Messimer. An empirical study of the relationship of stability metrics and the qmood quality models over software developed using highly iterative or agile software processes. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 171–179. IEEE, 2007.

[Sac97] D. Sacerdote. Imapd and ipop3d hole. *Bugtraq Archive*, 1997. http://www.securityfocus.com/archive/1/6370.

[Sad13] P. Saddington. Apple, google or microsoft? which does agile better, February 2013. http://agilescout.com/apple-google-or-microsoft-which-does-agile-better/.

[San95] Francisco Sandoval. From natural to artificial neural computation. *International Workshop on Artificial Neural Networks*, pages 195–201, June 1995.

[Sar13] Amol Sarwate. January 13 patch tuesday. Technical report, Qualys, January 2013. https://blog.qualys.com/laws-of-vulnerabilities/2013/01/08/january-2013-patch-tuesday.

[SBK05] Mikko Siponen, Richard Baskerville, and Tapio Kuivalainen. Integrating security into agile development methods. In *System Sciences, 2005. HICSS'05. Proceedings of the 38th Annual Hawaii International Conference on*, pages 185a–185a. IEEE, 2005.

[Sch00] Bruce Schneier. Closing the window of exposure: reflections on the future of security. *SecurityFocus. com*, 200, 2000.

[Sch04] Bruce Schneier. The nonsecurity of secrecy. *Commun. ACM*, 47(10):120, 2004. http://doi.acm.org/10.1145/1022594.1022629.

[Sch09] Guido Schryen. A comprehensive and comparative analysis of the patching behavior of open source and closed source software vendors. In *IT Security Incident Management and IT Forensics, 2009. IMF'09. Fifth International Conference on*, pages 153–168. IEEE, 2009.

[Sea08] Robert C. Seacord. *Secure Coding in C and C++*. Addison-Wesley Professional, June 2008. http://www.amazon.com/Secure-Coding-Robert-C-Seacord/dp/0321335724.

[Seca] Secunia. Vulnerability intelligence provider. http://www.secunia.com.

[Secb] SecurityTracker. http://www.SecurityTracker.com. SecurityTracker.

[Sec08] SecurityFocus. Vulnerabilities database, April 2008. http://www.securityfocus.com/vulnerabilities.

[Sec15] Risk Based Security. 2015 vulnerability trends, 2015. https://www.riskbasedsecurity.com/vulndb-quickview-2015-vulnerability-trends/.

[SF12] A. Sacco and Muttis F. Html5 heap sprays, pwn all the things, September 2012. https://www.coresecurity.com/corelabs-research/publications/html5-heap-sprays-pwn-all-things.

[SIK+13] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M Ibrahim, Masao Ohira, Bram Adams, Ahmed E Hassan, and Ken-ichi Matsumoto. Studying re-opened bugs in open source software. *Empirical Software Engineering*, 18(5):1005–1042, 2013.

[SKVL14] DM Soper, KC King, D Vergara, and CM Lively. Exposure to parasites increases promiscuity in a freshwater snail. *Biology letters*, 10(4):20131091, 2014.

[SLF08] Anil Somayaji, Michael Locasto, and Jan Feyereisl. The future of biologically-inspired security: is there anything left to learn? In *Proceedings of the 2007 Workshop on New Security Paradigms*, NSPW '07, pages 49–54, New York, NY, USA, 2008. ACM. http://doi.acm.org/10.1145/1600176.1600185.

[SLP+09] Stelios Sidiroglou, Oren Laadan, Carlos Perez, Nicolas Viennot, Jason Nieh, and Angelos D Keromytis. Assure: automatic software self-healing using rescue points. *ACM SIGARCH Computer Architecture News*, 37(1):37–48, 2009.

[Som04] Anil Somayaji. How to win and evolutionary arms race. *IEEE Security and Privacy*, 2:70–72, 2004. `http://doi.ieeecomputersociety.org/10.1109/MSP.2004.100`.

[Sot09] A Sotirov. Modern exploitation and memory protection bypasses. In *SSYM'09: Proceedings of the 18th annual USENIX Security Symposium*, Berkeley, CA, USA, July 2009. USENIX Association.

[Spa89] Eugene H. Spafford. The internet worm program: An analysis. *COMPUTER COMMUNICATION REVIEW*, 19, 1989.

[Sym16] SymantecTrend. Symantec internet security threat report, 2016. `https://www.symantec.com/security-center/threat-report`.

[SZ03] Giuseppe Serazzi and Stefano Zanero. Computer virus propagation models. In *In Tutorials of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTSâĂŹ03*. Springer-Verlag, 2003.

[TA14] US-CERT Alert (TA14-290A). Ssl 3.0 protocol vulnerability and poodle attack. https://www.us-cert.gov/ncas/alerts/TA14-290A, October 2014.

[Tea11] Corelan Team. Exploit writing tutorial part 11 : Heap Spraying Demystified. *Corelan Team Tutorials*, December 2011. `https://www.corelan.be/index.php/2011/12/31/exploit-writing-tutorial-part-11-heap-spraying-demystified/`.

[The16] The MITRE Corporation (Mitre). Common weakness enumeration, 2016. https://cwe.mitre.org/index.html.

[Tip] TippingPoint. Zero day initiative (zdi). http://www.zerodayinitiative.com/.

[Tre16a] TrendMicro. Pwn2own: Day 1, March 2016. http://blog.trendmicro.com/pwn2own-day-1-recap/.

[Tre16b] TrendMicro. Pwn2own: Day 2 and event wrap-up, March 2016. http://blog.trendmicro.com/pwn2own-day-2-event-wrap/.

[UC] US-CERT. Vulnerability statistics. http://www.cert.org/stats/vulnerability_remediation.html.

[UC97] US-CERT. Vulnerability in imap and pop. http://www.cert.org/advisories/CA-1997-09.html, September 1997.

[UC98] US-CERT. Multiple vulnerabilities in bind, May 1998. http://www.cert.org/advisories/CA-1998.05.bind_problems.html.

[Ula10] Sergey Ulasen. Kaspersky lab provides its insights on stuxnet worm., September 2010. https://www.kaspersky.com/about/news/virus/2010/Kaspersky_Lab_provides_its_insights_on_Stuxnet_worm.

[Vie05] John Viega. Building security requirements with clasp. In *Proc. ACM SESS*, pages 1–7, 2005.

[VM12] C. Valasek and Tarjei Mandt. Windows 8 heap internals, July 2012. https://www.blackhat.com/html/bh-us-12/bh-us-12-briefings.html.

[Vup16] Vupen. Vupen security blog. *Vupen Security*, 2016. http://www.vupen.com.

[WAMF01] Hilary Browne William, William A. Arbaugh, John M, and William L. Fithen. A Trend Analysis of Exploitations. In *In IEEE Symposium on Security and Privacy*, pages 214–229, 2001.

[WBB04] Jaana Wäyrynen, Marine Bodén, and Gustav Boström. Security engineering and extreme programming: An impossible marriage? In *Extreme programming and agile methods-XP/Agile Universe 2004*, pages 117–128. Springer, 2004.

[WFLGN10] Westley Weimer, Stephanie Forrest, Claire Le Goues, and ThanhVu Nguyen. Automatic program repair with evolutionary computation. *Communications of the ACM*, 53(5):109–116, 2010.

[Whi16] Zack Whittaker. These companies lost your data in 2015's biggest hacks, breaches, January 2016. `http://www.zdnet.com/pictures/biggest-hacks-security-data-breaches-2015/`.

[Wil10] Tim Wilson. Holy zeus! popular botnet rules as new exploits come online, August 2010. `http://www.darkreading.com/vulnerabilities---threats/holy-zeus!-popular-botnet-rules-as-new-exploits-come-online/d/d-id/1134123`.

[Wis09] Chester Wisniewski. Windows 7 vulnerable to 8 out of 10 viruses, 2009. `http://www.sophos.com/blogs/chetw/g/2009/11/03/windows-7-vulnerable-8-10-viruses/`.

[Woo13] Carol Woody. Agile security review of current research and pilot usages. *SEI Library White Paper*, 2013. `http://resources.sei.cmu.edu/library/asset-view.cfm?assetid=70232`.

[Wri36] T.P. Wright. Factors affecting the cost of airplanes. *Journal of Aeronautical Sciences*, 3:122–128, April 1936.

[WRI13] G. Wicherski, A. Radocea, and A. Ionescu. Hacking like in the movies: Visualizing page tables for local exploitation. *BlackHat Briefings*, July 2013. https://www.blackhat.com/us-13/briefings.html.

[XF] IBM Internet Security Systems X-Force. X-force advisory. http://www.iss.net.

[Yun15] Zhang Yunhai. Bypass control flow guard comprehensively. *BlackHat Briefings USA*, August 2015. https://www.blackhat.com/docs/us-15/materials/us-15-Zhang-Bypass-Control-Flow-Guard-Comprehensively-wp.pdf.